Ibrar Ahmed, Gregory Smith

PostgreSQL 9.6 High Performance

Optimize, troubleshoot, and benchmark your PostgreSQL database

Packt

PostgreSQL 9.6 High Performance

Optimize, troubleshoot, and benchmark your PostgreSQL database

Ibrar Ahmed
Gregory Smith

BIRMINGHAM - MUMBAI

PostgreSQL 9.6 High Performance

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2017

Production reference: 1260517

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78439-297-0

www.packtpub.com

Credits

Authors
Ibrar Ahmed
Gregory Smith

Reviewer
Daniel Durante

Commissioning Editor
Amey Varangaonkar

Acquisition Editor
Denim Pinto

Content Development Editor
Lawrence Veigas

Technical Editors
Pranali Badge
Dhiraj Chandanshive

Copy Editor
Safis Editing

Project Coordinator
Prajakta Naik

Proofreader
Safis Editing

Indexer
Rekha Nair

Graphics
Abhinash Sahu

Production Coordinator
Shantanu Zagade

About the Authors

Ibrar Ahmed is a Senior Database System Architect in an Enterprise PostgreSQL company. Prior to coming to open source development, he had experience in software and embedded system development. He also possesses vast experience in open source development. He has contributed to the PostgreSQL community as well as other open source communities, such as the Google Chromium project. He is an open source evangelist, and his passion for open source development has enabled him to contribute a number of significant features to different open source communities.

Ibrar has contributed significantly to the goal of making PostgreSQL a federated RDBMS. He has developed, and maintains, a number of PostgreSQL extensions that communicate with other RDBMS's, such as MySQL; NoSQL databases, such as MongoDB; and parallel file system storage, such as HDFS. He has authored another book, *PostgreSQL Developer's Guide*, for Packt Publishing.

Gregory Smith is a Chief PostgreSQL Evangelist in Charleston, South Carolina, for Crunchy Data Solutions, Inc. He's been providing database advice to clients in industries such as manufacturing, finance, and web development for 20 years.

Writing about PostgreSQL represents his second foray into teaching database performance tuning. He wrote a small free e-book titled, *Progress Performance FAQ* in 1995, covering the basics of how to make the Progress 4GL and its associated database run faster. In 2001, he converted exclusively to using PostgreSQL 7.0 for projects, and has been watching the complexity of problems the database is capable of solving increase with every release ever since.

Greg has contributed feature additions to every PostgreSQL version since 8.3. He's also the creator of a growing set of add-on tools for the database, currently including pgtune, pgbench-tools, peg, and 2warm.

About the Reviewer

Daniel Durante is an avid coffee drinker/roaster, motorcyclist, archer, welder, and carpenter whenever he isn't programming. From the age of 12, he has been involved in web and embedded programming with PHP, Node.js, Golang, Rust, and C.

He has worked on text-based browser games that have reached over 1,000,000 active players, created bin-packing software for CNC machines, and helped contribute to one of the oldest ORMs of Node.js--SequelizeJS.

He has also reviewed *PostgresSQL Developer's Guide* for Packt.

I would like to thank my parents, my brother, and my friends, who've all put up with my insanity sitting in front of a computer day in and day out. I would not have been here today if it wasn't for their patience, guidance, and love.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/1784392979`.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface 1

Chapter 1: PostgreSQL Versions 7

- **Performance of historical PostgreSQL releases** 8
  - Choosing a version to deploy 9
  - Upgrading to a new major version 10
    - Minor version upgrades 11
- **PostgreSQL or another database?** 12
- **PostgreSQL Tools** 12
  - PostgreSQL contrib 13
    - Finding contrib modules on your system 13
    - Installing a contrib module from source 14
    - Using a contrib module 15
    - Using a PostgreSQL's Extensions 16
  - pgFoundry 16
  - Additional PostgreSQL related software 17
- **PostgreSQL application scaling life cycle** 17
- **Performance tuning as a practice** 18
- **Summary** 20

Chapter 2: Database Hardware 21

- **Balancing hardware spending** 21
  - CPUs 22
  - Memory 23
  - Disks 24
    - RAID 25
    - Drive error handling 27
    - Drive firmware and RAID 28
    - SSDs 29
  - Disk controllers 30
    - Hardware and software RAID 30
    - Recommended disk controllers 31
    - Attached storage - SAN and NAS 33
- **Reliable controller and disk setup** 35
    - Hard drive reliability studies 35
  - Write-back caches 36
    - Sources of write-back caching 37
    - Disk controller monitoring 38

Disabling drive write caches 39
Performance impact of write-through caching 40
Summary 41
Chapter 3: Database Hardware Benchmarking 42
CPU and memory benchmarking 43
Memtest86+ 43
STREAM memory testing 44
STREAM and Intel versus AMD 44
CPU benchmarking 46
Sources of slow memory and processors 47
Physical disk performance 49
Random access and I/O per second 49
Sequential access and ZCAV 50
Short stroking 51
Commit rate 51
PostgreSQL test_fsync 52
The INSERT rate 52
Windows commit rate 52
Disk benchmarking tools 52
Basic disk benchmarking using HD Tune 53
Short stroking tests 54
Measuring IOPS using HD Tune 55
Unpredictable performance and Windows 57
Disk throughput in case of sequential read and write 57
bonnie++ 58
Bonnie++ 2.0 60
Bonnie++ ZCAV 61
sysbench 63
Seek rate 65
Removing test files 66
fsync commit rate 67
Complicated disk benchmarks 67
Sample disk results 68
Disk-performance expectations 70
Sources of slow disk and array performance 71
Summary 73
Chapter 4: Disk Setup 75
Maximum filesystem sizes 75
Filesystem crash recovery 76
Journaling filesystems 77
Linux filesystems 78

ext2 78
ext3 79
ext4 81
XFS 82
Other Linux filesystems 83
Write barriers 84
Drive support for barriers 84
Filesystem support for barriers 85
General Linux filesystem tuning 86
Read-ahead 86
File access times 87
Read caching and swapping 88
Write cache sizing 89
I/O scheduler elevator 90
Solaris and FreeBSD filesystems 92
Solaris UFS 92
FreeBSD UFS2 94
ZFS 95
FAT32 97
NTFS 97
Adjusting mounting behaviour 98
Disk layout for PostgreSQL 98
Symbolic links 99
Tablespaces 99
Database directory tree 100
Temporary files 101
Disk arrays, RAID, and disk layout 101
Disk layout guidelines 104
Summary 105
Chapter 5: Memory for Database Caching 106
Memory units in the postgresql.conf file 107
Increasing UNIX shared memory parameters for larger buffer sizes 108
Kernel semaphores 109
Estimating shared memory allocation 110
Inspecting the database cache 111
Installing pg_buffercache into a database 112
Database disk layout 113
Creating a new block in a database 117
Writing dirty blocks to disk 118
Crash recovery and the buffer cache 118
Checkpoint processing basics 119

Write-ahead log and recovery processing 119
Checkpoint timing 120
Checkpoint spikes 122
Spread checkpoints 122
Database block lifecycle 122
Dirty block write paths 123
Database buffer cache versus operating system cache 124
Doubly cached data 125
Inspecting the OS cache 126
Checkpoint overhead 126
Starting size guidelines 126
Platform, version, and workload limitations 127
Analyzing buffer cache contents 128
Inspection of the buffer cache queries 128
Top relations in the cache 130
Summary by usage count 131
Buffer contents summary with percentages 131
Buffer usage count distribution 133
Using buffer cache inspection for sizing feedback 133
Summary 134
Chapter 6: Server Configuration Tuning 135
Interacting with the live configuration 135
Defaults and reset values 136
Allowed change context 136
Reloading the configuration file 138
Commented-out settings 139
Server-wide settings 140
Database connections 140
listen_addresses 140
max_connections 141
Shared memory 142
shared_buffers 142
Free space map settings 142
Logging 142
log_line_prefix 143
log_statement 143
log_min_duration_statement 144
Vacuuming and statistics 144
autovacuum 145
Enabling autovacuum 145
maintainance_work_mem 146
default_statistics_target 146
Checkpoints 147

checkpoint_timeout 147
checkpoint_completion_target 147
wal_buffers 148
wal_sync_method 148
PITR and WAL replication 150
Per-client settings 150
effective_cache_size 150
synchronous_commit 151
work_mem 152
random_page_cost 152
constraint_exclusion 153
Tunables to avoid 153
fsync 153
full_page_writes 154
commit_delay and commit_siblings 154
max_prepared_transactions 154
Query enable parameters 155
New server tuning 155
Dedicated server guidelines 155
Shared server guidelines 157
pgtune 158
Summary 159
Chapter 7: Routine Maintenance 160
Transaction visibility with multiversion concurrency control 161
Visibility computation internals 161
Updates 162
Row lock conflicts 164
Serialization 165
Deletions 166
Advantages of MVCC 167
Disadvantages of MVCC 167
Transaction ID wraparound 168
Vacuum 170
Vacuum implementation 170
Regular vacuum 170
Returning free disk space 171
Full vacuum 171
Removing tuple timeout 172
HOT 172
Cost-based vacuuming 173
Autovacuum 174
Autovacuum logging 175
Autovacuum monitoring 176
Autovacuum triggering 176

Per-table adjustments 178
Common vacuum and autovacuum problems 178
Autovacuum is running even though it was turned off 179
Autovacuum is constantly running 179
Out of memory errors 179
Not keeping up on a busy server 180
Autovacuum is too disruptive 180
Long-running transactions 181
Free space map exhaustion 181
Recovering from major problems 182
Autoanalyze 183
Index bloat 184
Measuring index bloat 185
Fixing the index bloat 186
Dump and restore 187
Vacuuming the database/table 187
CLUSTER 187
Reindexing 188
Detailed data and index page monitoring 189
Monitoring query logs 189
Basic PostgreSQL log setup 190
Log collection 190
log_line_prefix 191
Multi-line queries 192
Using syslog for log messages 192
CSV logging 193
Logging difficult queries 194
auto_explain 195
Log file analysis 196
Normalized query fingerprints 196
pg_stat_statements 197
pgFouine 198
pqa 201
eqpa 201
pgsi 201
mk-query-digest 202
Summary 202
Chapter 8: Database Benchmarking 204
pgbench default tests 204
Table definition 205
Scale detection 206
Query script definition 207
Configuring the database server for pgbench 209

Sample server configuration 210
Running pgbench manually 211
Graphing results with pgbench-tools 212
Configuring pgbench-tools 212
Sample pgbench test results 214
SELECT-only test 214
TPC-B-like test (Read/Write) 215
Latency analysis 217
Sources for bad results and variation 220
Developer PostgreSQL builds 221
Worker threads and pgbench program limitations 222
pgbench custom tests 223
Insert speed test 223
Transaction processing performance council benchmarks 224
Summary 225
Chapter 9: Database Indexing 226
Indexing example walkthrough 227
Measuring query disk and index block statistics 227
Running the example 229
Sample data setup 229
Simple index lookups 231
Full table scans 232
Index creation 233
Lookup with an inefficient index 234
Combining indexes 235
Planning for plan changes 236
Clustering against an index 237
Explain with buffer counts 239
Index creation and maintenance 239
Unique indexes 240
Concurrent index creation 240
Clustering an index 241
Fillfactor 242
Reindexing 242
Index types 242
B-tree 242
Text operator classes 243
Hash 244
GIN 244
GiST 245

SP-GiST 245
BRIN 245
Index only scans 247
count(*) 248
Visibility map 248
Advanced index use 248
Multicolumn indexes 249
Indexes for sorting 249
Partial indexes 250
Expression-based indexes 251
Indexing for full-text search 251
Summary 252
Chapter 10: Query Optimization 253
Sample datasets 254
Pagila 254
Dell Store 2 254
EXPLAIN basics 256
Timing overhead 257
Hot and cold cache behavior 257
Clearing the cache 258
Query plan node structure 259
Basic cost computation 261
Estimated costs and real-world costs 262
Visual explain 263
Verbose output 263
Machine readable explain output 264
Planning analysis tools 266
Assembling row sets 266
Tuple ID 266
Object ID 267
Sequential scan 267
Index scan 268
Index only scans 269
Bitmap heap and index scans 269
Processing nodes 271
Sort 271
Limit 273
Offsets 273
Aggregate 274

HashAggregate 275
Unique 275
WindowAgg 276
Result 276
Append 277
Group 278
Subquery scan and subplan 279
Subquery conversion and IN lists 279
Set operations 280
Materialize 281
CTE scan 282
Copy command 282
COPY TO 282
COPY FROM 283
COPY FREEZE 283
Joins 284
Nested loop 285
Nested loop with inner-index scan 286
Merge join 287
Nested Loop and Merge Join materialization 287
Hash joins 288
Hash semi and anti joins 288
Join ordering 289
Forcing join order 290
Join removal 291
Genetic query optimizer 292
Collecting statistics 292
Viewing and estimating with statistics 293
Statistics targets 297
Adjusting a column target 297
Distinct values 298
Difficult areas to estimate 298
Other query planning parameters 299
effective_cache_size 299
work_mem 300
constraint_exclusion 301
cursor_tuple_fraction 302
Executing other statement types 302
Improving queries 303
Optimizing for fully cached datasets 303
Testing for query equivalence 304

Disabling optimizer features 304
Working around optimizer bugs 308
Avoiding plan restructuring with OFFSET 309
External trouble spots 311
SQL limitations 311
Numbering rows in SQL 312
Using window functions for numbering 313
Using window functions for cumulatives 313
Summary 315
Chapter 11: Database Activity and Statistics 316
Statistics views 316
Cumulative and live views 318
Table statistics 319
Table I/O 322
Index statistics 323
Index I/O 325
Database wide totals 326
Connections and activity 326
Locks 327
Virtual transactions 328
Decoding lock information 329
Table-level Lock Modes 332
Transaction lock waits 332
Table lock waits 334
Logging lock information 335
Deadlocks 335
Disk usage 336
Monitoring Buffer, background writer, and checkpoint activity 339
Saving pg_stat_bgwriter snapshots 340
Tuning using background writer statistics 343
Summary 345
Chapter 12: Monitoring and Trending 347
UNIX monitoring tools 347
Sample setup 348
vmstat 349
iostat 352
iotop for Linux 354
Examples of good performance 354
Overloaded system samples 357

top 360
Solaris top replacements 361
htop for Linux 362
SysStat and sar 362
Enabling sysstat and its optional features 364
Graphing with kSar 364
Windows monitoring tools 365
Task manager 365
Sysinternals tools 365
Windows system monitor 366
Saving Windows system monitor data 367
Trending software 368
Types of monitoring and trending software 368
Storing historical trend data 369
Nagios 370
Nagios and PostgreSQL 371
Nagios and Windows 371
Cacti 371
Cacti and PostgreSQL 372
Cacti and Windows 372
Munin 372
Other trending packages 373
pgStatspack 373
Zenoss 374
Hyperic HQ 374
Reconnoiter 374
Staplr 375
SNMP tools 375
Summary 375
Chapter 13: Pooling and Caching 377
Connection pooling 377
Pooling connection counts 378
pgpool-II 379
Connection pooling 380
Replication 380
Watch-dog 380
Failover 380
Load balancing 381
pgBouncer 382
Application server pooling 383
Database caching 383
What is memcached 383

pgmemcache 384
Summary 386
Chapter 14: Scaling with Replication 387
Hot standby 388
Terminology 388
Setting up WAL shipping 389
Streaming replication 390
Cascading replication 390
Synchronous replication 391
Tuning Hot Standby 391
Replication queue managers 393
Slony 393
Londiste 394
Read scaling with replication queue software 395
Special application requirements 395
Bucardo 395
pglogical 396
xDB 396
pgpool-II 397
Other interesting replication projects 397
Replications solution comparison 398
Summary 399
Chapter 15: Partitioning Data 400
Table inheritance 401
dellstore2 database 401
Partitioning in PostgreSQL 402
Range partitioning 403
Determine a key field to partition over 404
Sizing the partitions 405
List partitioning 406
Redirecting INSERT statements to the partitions 406
Dynamic trigger functions 407
Partition rules 408
Empty partition query plans 409
Date change update trigger 410
Live migration of a partitioned table 410
Partitioned queries 412
Creating new partitions 414

Scheduled creation 414
Dynamic creation 415
Partitioning advantages 415
Common partitioning mistakes 416
Horizontal partitioning with PL/Proxy 417
Hash generation 417
Scaling with PL/Proxy 419
Sharding 420
Scaling with GridSQL 420
Summary 421
Chapter 16: Database Profiling 423
Profiling using gprof 423
Debugging using Valgrind 425
Visual Studio 426
Profiling using DTrace 426
DTrace on FreeBSD 427
Linux SystemTap emulation of DTrace 427
Summary 427
Chapter 17: Avoiding Common Problems 428
Bulk loading 428
Loading methods 428
External loading programs 429
Tuning for bulk loads 430
Skipping WAL acceleration 431
Recreate indexes and add constraints 432
Parallel restore 432
Post load clean up 433
Common performance issues 434
Counting rows 434
Unexplained writes 435
Slow function and prepared statement execution 436
PL/pgSQL benchmarking 437
High foreign key overhead 438
Trigger memory use 439
Heavy statistics collector overhead 439
Targeted statistics resets 440
Materialized views 440
Summary 441
Chapter 18: Performance Features by Release 442

Aggressive PostgreSQL version upgrades 443
Performance features in version 8.1 443
Performance features in version 8.2 444
Performance features in version 8.3 445
Performance features in version 8.4 446
Performance features in version 9.0 449
Replication 449
Queries and EXPLAIN 450
Database development 451
Configuration and monitoring 452
Performance features in version 9.1 454
Performance features in version 9.2 455
Performance features in version 9.3 456
Performance features in version 9.4 457
Performance features in version 9.5 458
Performance features in version 9.6 458
Summary 459
Index 460

Preface

PostgreSQL has become an increasingly viable database platform to serve as storage for applications, ranging from classic corporate database use to the latest web apps. However, getting the best performance from it has not been an easy subject to learn. You just need the right combination of rules of thumb to get started, solid monitoring and maintenance to keep your system running well, suggestions for troubleshooting, and hints for add-on tools to add the features that the core database doesn't try to handle on its own.

What this book covers

`Chapter 1`, *PostgreSQL Versions*, introduces how the PostgreSQL performance has improved in the most recent versions of the databases. It makes the case to use the most recent version feasible, in contrast to the common presumption that newer versions of any software are buggier and slower than their predecessors.

`Chapter 2`, *Database Hardware*, discusses how the main components in the server hardware, including processors, memory, and disks, need to be carefully selected for reliable database storage and a balanced budget. In particular, accidentally using volatile write-back caching in disk controllers and drives can easily introduce database corruption.

`Chapter 3`, *Database Hardware Benchmarking*, moves on to quantifying the different performance aspects of the database hardware. Just how fast is the memory and how raw are the drives in your system? Does performance scale properly as more drives are added?

`Chapter 4`, *Disk Setup*, looks at popular filesystem choices and suggests the trade-offs of various ways to layout your database on disk. Some common effective filesystem tuning tweaks are also discussed.

`Chapter 5`, *Memory for Database Caching*, digs into how the database is stored on disk, in memory, and how the checkpoint process serves to reconcile the two safely. It also suggests how you can actually look at the data being cached by the database to confirm whether what's being stored in memory matches what you'd expect to be there.

`Chapter 6`, *Server Configuration Tuning*, covers the most important settings in the postgresql.conf file, what they mean, and how you should set them. Moreover, the settings you can cause trouble to by changing are pointed out, too.

`Chapter 7`, *Routine Maintenance*, starts by explaining how PostgreSQL determines what rows are visible to which clients. The way visibility information is stored requires a cleanup process named VACUUM to reuse leftover space properly. Common issues and general tuning suggestions for it, and the always running autovacuum, are covered. Finally, there's a look at adjusting the amount of data logged by the database using a query log analyzer on the result to help find query bottlenecks.

`Chapter 8`, *Database Benchmarking*, investigates how to get useful benchmark results from the built-in pgbench testing program included with PostgreSQL.

`Chapter 9`, *Database Indexing*, introduces indexes in terms of how they can reduce the amount of data blocks read to answer a query. This approach allows thoroughly investigating common questions, such as why a query is using a sequential scan instead of an index in a robust way.

`Chapter 10`, *Query Optimization*, is a guided tour of the PostgreSQL optimizer, exposed by showing the way sample queries are executed differently based on what they are asking for and how the database parameters are set.

`Chapter 11`, *Database Activity and Statistics*, looks at the statistics collected inside the database, and which of them are useful for finding problems. The views that let you watch query activity and locking behavior are also explored.

`Chapter 12`, *Monitoring and Trending*, starts with how to use basic operating system monitoring tools to determine what the database is doing. Then, it moves onto suggestions for trending software that can be used to graph this information over time.

`Chapter 13`, *Pooling and Caching*, explains the difficulties you can encounter when large numbers of connections are made to the database at once. Two types of software package are suggested to help: connection poolers, to better queue incoming requests; and caches that can answer user requests without connecting to the database.

`Chapter 14`, *Scaling with Replication*, covers approaches for handling heavier system loads by replicating the data across multiple nodes; typically, a set of read-only nodes synchronized to a single writeable master.

`Chapter 15`, *Partitioning Data*, explores how data may be partitioned into subsets usefully so that queries can execute against a smaller portion of the database. Approaches discussed include the standard single node database table partitioning and using PL/Proxy with its associated toolset to build sharded databases across multiple nodes.

`Chapter 16`, *Database Profiling*, discusses parts of PostgreSQL that regularly seem to frustrate newcomers to the database. The focus is how to find why our code is not working, and what are the existing tools available to help us diagnose the issues.

`Chapter 17`, *Avoiding Common Problems*, covers all bulk loading; counting records and foreign key handlings are examples. It also covers the common problems people face while using PostgreSQL.

`Chapter 18`, *Performance Features by Release*, contains a detailed review of what the performance-related features changed between each version of PostgreSQL from 8.1 to 9.6. Sometimes, the best way to avoid a common problem and get better performance is to upgrade to a new version where the problems don't exist anymore.

What you need for this book

In order for this book to be useful, you need at least access to a PostgreSQL client that is allowed to execute queries on a server. Ideally, you'll also be the server administrator. Full client and server packages for PostgreSQL are available for most popular operating systems at `http://www.postgresql.org/download/`. All of the examples here are executed at a command prompt, usually running the psql program. This makes them applicable to most platforms. It's straightforward to do many of these operations instead using a GUI tool for PostgreSQL, such as the pgAdmin III program.
There are some scripts provided that are written in the bash scripting language. If you're on Windows, the cygwin software suite available from `http://www.cygwin.com/` provides a way to get common UNIX tools such as bash onto your system.

Who this book is for

This book is for intermediate to advanced database administrators and developers who use or plan to use PostgreSQL. System administrators can benefit from installing, configuring, and monitoring the database server. Developers benefit from writing optimal queries and detecting performance issues in their database design. It will benefit PostgreSQL internal architects in being able to monitor performance using benchmarking tools.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "A good practice here is to bundle them in with increasing the shared memory parameters to support larger values of `shared_buffers`, which requires editing the same `sysctl` file."

A block of code is set as follows:

```
blocks = 250,000 * (gigabytes of RAM)
```

Any command-line input or output is written as follows:

```
$ chmod +x bench.sh
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Right-click on the trace data and select **Save Settings As…;** the result will be an HTML file with a list of all the counters you're logging, along with the server name embedded in it."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit

`http://www.packtpub.com/support`and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/PostgreSQL-9.6-High-Performance`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-`errata, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support`and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
PostgreSQL Versions

PostgreSQL certainly has a reputation. It's known for having a rich feature set and very stable software releases. The secure stance, which its default configuration takes, is simultaneously praised by security fans and criticized for its learning curve. The SQL specification conformance and data integrity features allow only the strictest ways to interact with the database, surprising to those who come from a background working with looser desktop database software. All of these points have an element of truth to them.

Another part of PostgreSQL's reputation is that it's slow, but in new releases PostgreSQL's performance has improved dramatically. There are many database operations where "the right thing" just plain takes longer to do than the alternative. As the simplest example of that, if you ask a database to store February 29, 2009 into a date field, a date that's only valid as an abstract one, it can just do that, the fast approach. Or it can check whether that date is valid to store into the destination field, note there is no such date in a regular calendar, and reject your change. That's always going to be slower. PostgreSQL is designed by and for the sort of people who don't like cutting corners just to make things faster or easier, and in cases where the only way you can properly handle something takes a while, that may be the only option available.

However, once you have the correct implementation of something, you can then go back and optimize it. That's the mode PostgreSQL has been in for the last few years. PostgreSQL usually rises above these smaller issues now to give excellent database performance anyway. Parts of it have the sort of great design that outperforms simpler approaches, even after paying the overhead that complexity can introduce. This is a fairly recent phenomenon though, which explains quite a bit of the perception that PostgreSQL is a slower database than its competitors.

In this chapter, we will cover the following topics:

- Performance of historical PostgreSQL releases
- PostgreSQL or another database?
- PostgreSQL Tools
- PostgreSQL application scaling life cycle
- Performance tuning as a practice

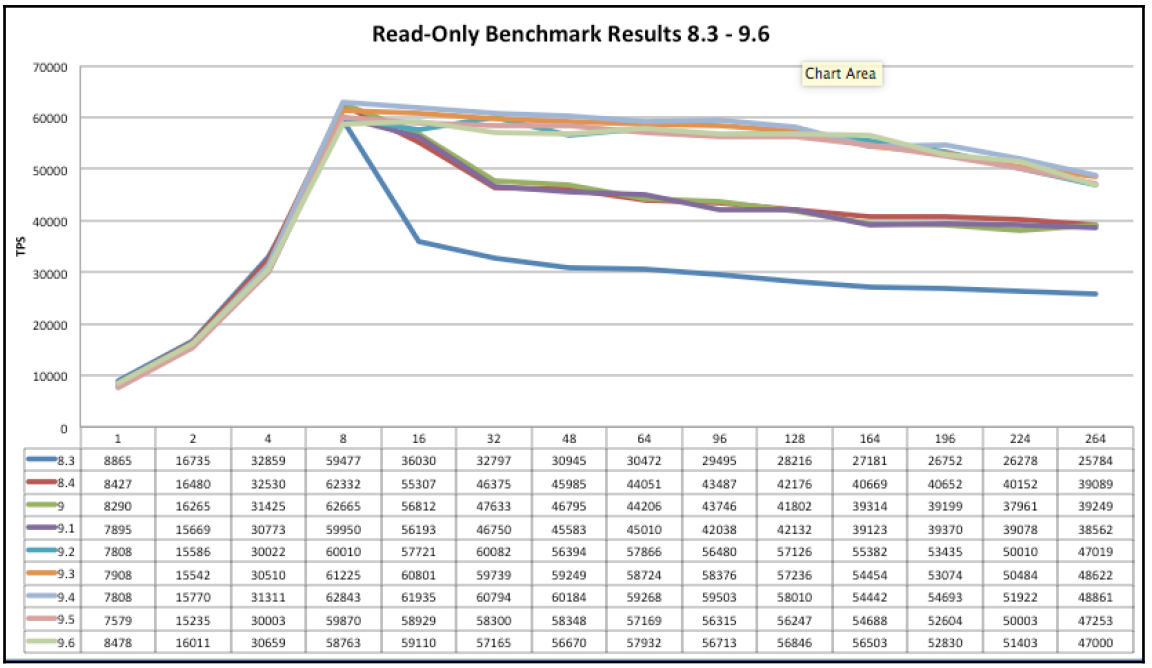
Performance of historical PostgreSQL releases

In September of 2016, PostgreSQL 9.6 was released. It included a number of internal architectural changes with significant changes in the performance such as Parallel execution of sequential scans, aggregates and joins. Specially improvement in the area of scalability on multi-CPU-socket servers. The complete change log can be found at (`https://www.postgresql.org/docs/9.6/static/release-9-6.html`).

The results were a major improvement in the ability of the database to scale upwards to handle a heavy load. Benchmarks on modern hardware really highlight just how far that version leapfrogged earlier ones. Here are the detailed benchmark results from version 8.3 to version 9.6. The PostgreSQL's tool pgbench is used to benchmark the different releases of PostgreSQL. The complete documentation can be found at (`https://www.postgresql.org/docs/current/static/pgbench.html`).

The test provide the Transactions Per Second (TPS) figure that measures total system speed in case of read-only transaction and mixed read/write transactions. Here is the mixed mode (read/write) performance comparison form version 8.3 to 9.6.

The performance can easily be seen from versions, you can observe that when increasing the number of clients increase the TPS, but at some peek value of clients TPS start decreasing. That's indicate that that are the approximately maximum no of clients that version can handle on particular hardware. Hence the rise in the number of clients at the peak load gives an idea of how well the database internals handle access to shared resources, the area 9.6 in particular included a significant upgrade to 8.4/8.3:



	1	2	4	8	16	32	48	64	96	128	164	196	224	264
8.3	8865	16735	32859	59477	36030	32797	30945	30472	29495	28216	27181	26752	26278	25784
8.4	8427	16480	32530	62332	55307	46375	45985	44051	43487	42176	40669	40652	40152	39089
9	8290	16265	31425	62665	56812	47633	46795	44206	43746	41802	39314	39199	37961	39249
9.1	7895	15669	30773	59950	56193	46750	45583	45010	42038	42132	39123	39370	39078	38562
9.2	7808	15586	30022	60010	57721	60082	56394	57866	56480	57126	55382	53435	50010	47019
9.3	7908	15542	30510	61225	60801	59739	59249	58724	58376	57236	54454	53074	50484	48622
9.4	7808	15770	31311	62843	61935	60794	60184	59268	59503	58010	54442	54693	51922	48861
9.5	7579	15235	30003	59870	58929	58300	58348	57169	56315	56247	54688	52604	50003	47253
9.6	8478	16011	30659	58763	59110	57165	56670	57932	56713	56846	56503	52830	51403	47000

These improvements have been confirmed by other benchmarking results, albeit normally not covering such a wide range of versions. It's easy to see that any conclusion about PostgreSQL performance reached before late 2005, when 8.1 shipped, is completely out of date at this point. The speed improvement in 9.0 releases was an additional large leap. Performance is not the only reason to choose version greater than 9.0, there are other reasons to prefer using that one or later too.

More detail PostgreSQL benchmark can be found at `http://blog.pgaddict.com/posts/performance-since-postgresql-7-4-to-9-4-pgbench`

Choosing a version to deploy

Due to the significant gains detailed in the previous section, if you have an older PostgreSQL system you'd like to make faster, the very first thing you should ask yourself is not how to tweak its settings, but instead if it's possible to upgrade to a newer version. If you're starting a new project, 9.6 is the earliest version you should consider. In addition to the performance improvements, there are some changes to that version that impact application coding that you'd be better off to start with, to avoid needing to retrofit later.

The final chapter of this book includes a reference guide to what the performance related features were added to each major version of PostgreSQL, from 8.4 through 9.6. You might discover that one of the features only available in a very recent version is compelling to you, and therefore have a strong preference to use that one. Many of these version specific changes are also highlighted throughout the book.

Upgrading to a new major version

The older way of to upgrade an existing PostgreSQL install to a newer major version, such as going from 9.1.X to 9.2.X, was to dump and reload. The `pg_dump` and/or `pg_dumpall` programs are used to write the entire contents of the database to a file, using the newer version of those programs. That way, if any changes need to be made to upgrade, the newer dumping program can try to handle them. Not all upgrade changes will happen automatically though. Then, depending on the format you dumped in, you can either restore that just by running the script it generates, or use the `pg_restore` program to handle that task. `pg_restore` can be a much better alternative in newer PostgreSQL versions that include a version with parallel restore capabilities.

If you are using a system that doesn't easily allow running more than one system PostgreSQL version at a time, such as the current RedHat Linux RPM packages, getting both old and new versions of PostgreSQL installed on your system at the same time can be difficult. There are some changes to improve this situation under development for PostgreSQL 9.0. Make sure to check the feasibility of running more than one version at once as part of planning an upgrade.

Dumping can take a while, and restoring even longer. While this is going on, your database likely needs to be down, so that you don't allow any changes that won't then be migrated over by the dump. For large databases, this downtime can be both large and unacceptable.

Nowadays, a real approach available for zero-downtime, 24/7 sites that need to do PostgreSQL upgrades, has been to use statement replication. Slony (`http://slony.info/`) is the most popular tool for that, and more information about it will be covered in the later replication chapter. One of Slony's features is that you don't have to be running the same version of PostgreSQL on all the nodes you are replicating to. You can bring up a new node running a newer PostgreSQL version; wait for replication to complete to it, then switch over once it matches the original.

PostgreSQL provides a new way, works without needing replication software. A program calls `pg_upgrade` (`https://www.postgresql.org/docs/9.6/static/pgupgrade.html`) shipped with PostgreSQL distribution is capable of upgrading from older version to new version without the dump and reload, a process called **in-place upgrading**. While all in-place upgrades have some risk and need careful testing, in many cases `pg_upgrade` will take you from 8.3 to the latest new version.

The PostgreSQL development community is now committed to allowing in-place upgrades to future versions. Now that terabyte and larger PostgreSQL installs are common, upgrading only via dump and reload just isn't always practical.

Minor version upgrades

A dump/reload or use of tools such as `pg_upgrade` is not needed for minor version updates, for example going from 9.4.3 to 9.4.4. These simply require stopping the server, installing the new version, and then running the newer database binary against the existing server data files. Some people avoid ever doing such upgrades once their application is running; for fear that a change in the database will cause a problem. This should never be the case for PostgreSQL. The policy of the PostgreSQL project described at `http://www.postgresql.org/support/versioning` states very clearly:

> *While upgrades always have some risk, PostgreSQL minor releases fix only frequently encountered, security, and data corruption bugs to reduce the risk of upgrading. The community considers not upgrading to be riskier than upgrading.*

You should never find an unexpected change that breaks an application in a minor PostgreSQL upgrade. Bug, security, and corruption fixes are always done in the way that minimizes the odds of introducing an externally visible behavior change, and if that's not possible the reason why and suggested workarounds will be detailed in the release notes. What you will find is that some subtle problems resulting from resolved bugs could clear up after even a minor version update. It's not uncommon to discover a report of a problem to one of the PostgreSQL mailing lists is resolved in the latest minor version update compatible with that installation, and upgrading to that version is all that's needed to make the issue go away.

PostgreSQL or another database?

There are certainly situations where other database solutions will perform better. For example, PostgreSQL is missing features needed to perform well on some of the more difficult queries in the TPC-H test suite (see the database benchmarking chapter number 8 for more details). It's correspondingly less suitable for running large data warehouse applications than many of the commercial databases. If you need queries like some of the very heavy ones TPC-H includes, you may find that databases such as Oracle, DB2, and SQL Server still have a performance advantage worth paying for. There are also several PostgreSQL derived databases that include features making them more appropriate for data warehouses and similar larger systems. Examples include Greenplum, Aster Data, and Netezza.

For some types of web applications, you can only get acceptable performance by cutting corners on the data integrity features in ways that PostgreSQL just won't allow. A less strict database such as MySQL, or even a really minimal one such as SQLite might better serve these applications. Unlike the fairly mature data warehouse market, the design of this type of application is still moving around quite a bit. Work on approaches using the key-value based NoSQL approach, including CouchDB, MongoDB, and Cassandra, are all becoming more popular at the time of writing. All of them can easily outperform a traditional database, if you have no need to run the sort of advanced queries key/value stores are slower at handling.

But for many "normal" database use cases in the middle ground between those two extremes, PostgreSQL performance in 9.6 reached a point where it's more likely you'll run into the limitations of your hardware or application design before the database is your limiting factor. And some of PostgreSQL's traditional strengths, like its ability to handle complicated queries well and its heavy programmability, are all still there.

PostgreSQL Tools

If you're used to your database vendor supplying a full tool chain with the database itself, from server management to application development, PostgreSQL may be a shock to you. Like many successful open-source projects, PostgreSQL tries to stay focused on the features it's uniquely good at. This is what the development community refers to as the PostgreSQL core: the main database server, and associated utilities that can only be developed as part of the database itself. When new features are proposed, if it's possible for them to be built and distributed "out of core", this is the preferred way to do things.

This approach keeps the database core as streamlined as possible, as well as allowing those external projects to release their own updates without needing to synchronize against the main database's release schedule.

Successful PostgreSQL deployments should recognize that a number of additional tools, each with their own specialized purpose, will need to be integrated with the database core server to build a complete system.

PostgreSQL contrib

One part of the PostgreSQL core that you may not necessarily have installed is what's called the contrib modules, after the `contrib` directory they are stored in. These are optional utilities shipped with the standard package, but that aren't necessarily installed by default on your system. The contrib code is maintained and distributed as part of the PostgreSQL core, but is not required for the server to operate.

From a code quality perspective, the contrib modules aren't held to quite as high of a standard primarily by how they're tested. The main server includes heavy regression tests for every feature, run across a large build farm of systems that look for errors. The optional contrib modules don't get that same level of testing coverage. But the same development team maintains the code itself, and some of the modules are extremely popular and well tested by users.

A listing of all the contrib modules available is at `http://www.postgresql.org/docs/current/static/contrib.html`.

Finding contrib modules on your system

One good way to check if you have contrib modules installed is to see if the `pgbench` program is available. That's one of the few contrib components that installs a full program, rather than just scripts you can use. Here's a UNIX example of checking for `pgbench`:

```
$ pgbench -V
pgbench (PostgreSQL) 9.6
```

If you're using an RPM or DEB packaged version of PostgreSQL, as would be the case on many Linux systems, the optional `postgresql-contrib` package contains all of the contrib modules and their associated installer scripts. You may have to add that package using `yum`, `apt-get`, or a similar mechanism if it isn't installed already. On Solaris, the package is named `SUNWpostgr-contrib`.

If you're not sure where your system PostgreSQL contrib modules are installed, you can use a filesystem utility to search for them. The `locate` works well for this purpose on many UNIX-like systems, as does the `find` command. The file search utilities available on the Windows Start menu will work. A sample file you could look for is `pg_buffercache.sql` which will be used in the upcoming `Chapter 5`, *Memory for Database Caching*, on memory allocation. Here's where you might find the file on some of the platforms that PostgreSQL supports:

- RHEL and CentOS Linux systems will put the main file you need into `/usr/share/pgsql/contrib/pg_buffercache.sql`
- Debian or Ubuntu Linux systems will install the file at `/usr/share/postgresql/version/contrib/pg_buffercache.sql`
- Solaris installs into `/usr/share/pgsql/contrib/pg_buffercache.sql`
- The standard Windows one-click installer with default options will always include the contrib modules, and this one will be in `C:\Program Files\PostgreSQL/version/share/contrib/pg_buffercache.sql`

Installing a contrib module from source

Building your own PostgreSQL from source code can be a straightforward exercise on some platforms, if you have the appropriate requirements already installed on the server. Details are documented at `http://www.postgresql.org/docs/current/static/install-procedure.html`.

After building the main server code, you'll also need to compile contrib modules like `pg_buffercache` yourself too. Here's an example of how that would work, presuming that your PostgreSQL destination is `/usr/local/postgresql` and there's a directory under there named that `source` you put the source code into (this is not intended to be a typical or recommended structure you should use):

```
$ cd /usr/local/postgresql/source
$ cd contrib/pg_buffercache/
$ make
$ make install
/bin/mkdir -p '/usr/local/postgresql/lib/postgresql'
/bin/mkdir -p '/usr/local/postgresql/share/postgresql/contrib'
/bin/sh ../../config/install-sh -c -m 755  pg_buffercache.so
  '/usr/local/postgresql/lib/postgresql/pg_buffercache.so'
/bin/sh ../../config/install-sh -c -m 644
  ./uninstall_pg_buffercache.sql
  '/usr/local/postgresql/share/postgresql/contrib'
/bin/sh ../../config/install-sh -c -m 644 pg_buffercache.sql
```

```
'/usr/local/postgresql/share/postgresql/contrib'
```

It's also possible to build and install all the contrib modules at once by running `make` and `make install` from the `contrib` directory. Note that some of these have more extensive source code build requirements. The `uuid-ossp` module is an example of a more challenging one to compile yourself.

Using a contrib module

While some contrib programs, such as `pgbench`, are directly executable, most are utilities that you install into a database in order to add extra features to it.

As an example, to install the `pg_buffercache` module into a database named abc, the following command line would work (assuming the RedHat location for the file):

```
$ psql -d abc -f /usr/share/postgresql/contrib/pg_buffercache.sql
```

You could instead use the pgAdmin III GUI management utility, which is bundled with the Windows installer for PostgreSQL, instead of the command line:

1. Navigate down to the database you want to install the module into
2. Click on the SQL icon in the toolbar to bring up the command editor
3. Choose **File/Open**. Navigate to `C:\Program Files\PostgreSQL/version/share/contrib/pg_buffercache.sql` and open that file.
4. Execute using either the green arrow or **Query/Execute**.

You can do a quick test of whether the module installed on any type of system by running the following quick query:

```
SELECT * FROM pg_buffercache;
```

If any results come back, the module was installed. Note that `pg_buffercache` will only be installable and usable by database superusers.

Using a PostgreSQL's Extensions

PostgreSQL introduces another way to install contrib module, called extensions. If any contrib module build with the extension specification which includes:

- The Extension SQL file (`extension_name.sql`)
- The Extension control file (`extension_name.control`)
- The Extensions library file (`extension_name.so`)

Let's look at the following commands:

- **CREATE EXTENSION extension_name**: This command will install a new extension called dummy, so there will be no need to manually execute a sql file like we did while installing `pg_buffercache`.
- **ALTER EXTENSION extension name**: This command will alter the extension and is mostly used to upgrade an extension.
- **DROP EXTENIONS extension_name**: This command deletes a extension, after this clean-up is done.

pgFoundry

The official home of many PostgreSQL related projects is **pgFoundry** (`http://pgfoundry.org/`).

pgFoundry only hosts software for PostgreSQL, and it provides resources such as mailing lists and bug tracking in addition to file distribution. Many of the most popular PostgreSQL add-on programs are hosted there, including the following:

- Windows software allowing access to PostgreSQL through .Net and OLE
- Connection poolers pgpool and pgBouncer
- Database management utilities such as pgFouine, SkyTools, and pgtune

While sometimes maintained by the same people who work on the PostgreSQL core, pgFoundry code varies significantly in quality. One way to help spot the healthier projects is to note how regularly and recently new versions have been released.

Additional PostgreSQL related software

Beyond what comes with the PostgreSQL core, the contrib modules, and software available on pgFoundry, there are plenty of other programs which will make PostgreSQL easier and more powerful available from sources all over the Internet. There are actually so many available that choosing the right package for a requirement can itself be overwhelming.

Some of the best programs will be highlighted throughout the book, to help provide a shortlist of ones you should consider early on. This approach, where you get a basic system running and then add additional components as needed, is the standard way in which large open-source projects are built.

It can be difficult for some corporate cultures, ones where any software installation requires everything from approval to a QA cycle, to adapt to that style. To improve the odds your PostgreSQL installation will be successful in such environments, it's important to start early on introducing this concept early on. Additional programs to add components building on the intentionally slim database core will be needed later, and not all of what's needed will be obvious at the beginning.

PostgreSQL application scaling life cycle

While every application has unique growth aspects, there are many common techniques that you'll find necessary as the application using a PostgreSQL database becomes used more heavily. The chapters of this book each focus on one of the common aspects of this process. The general path database servers follow includes:

- Select hardware to run the server on. Ideally, you'll test that hardware to make sure it performs as expected too.
- Setup all the parts of database disk layout: RAID level, filesystem, and possibly table/index layout on disk.
- Optimize the server configuration.
- Monitor server performance and how well queries are executing.
- Improve queries to execute more efficiently, or add indexes to help accelerate them.
- As it gets more difficult to just tune the server to do more work, instead reduce the amount it has to worry about by introducing connection pooling and caching.
- Replicate the data onto multiple servers and distribute reads among them.
- Partition larger tables into sections. Eventually really large ones may need to be split so they're written to multiple servers simultaneously.

This process is by no means linear. You can expect to make multiple passes over optimizing the server parameters. It may be the case that you decide to buy newer hardware first, rather than launching into replication or partitioning work that requires application redesign work. Some designs might integrate caching into the design from the very beginning. The important thing is to be aware of the various options available, and to collect enough data about what limits the system is reaching to help you decide which of the potential changes are most likely to help.

Performance tuning as a practice

Work on improving database performance has its own terminology just like any other field. Here are some terms or phrases that will be used throughout the book:

- **Bottleneck** or **limiting factor**: Both these terms will be used to refer to the current limitation that is most hindering better performance..
- **Benchmarking**: This refers to running a test to determine how fast a particular operation can run. This is often done to figure out where the bottleneck of a program or system is.
- **Profiling**: This refers to monitoring what parts of a program are using the most resources when running a difficult operation such as a benchmark. This is typically used to help prove where the bottleneck is, and whether it's been removed as expected after a change. Profiling a database application usually starts with monitoring tools such as `vmstat` and `iostat`. Popular profiling tools at the code level include `gprof`, `oprofile`, and `dtrace`.

One of the interesting principles of performance tuning work is that, in general, you cannot figure out what the next bottleneck an application will run into is until you remove the current one. When presented with a system that's not as fast as someone would expect it to be, you'll often see people guessing what the current bottleneck is, or what the next one will be. That's generally a waste of time. You're always better off measuring performance, profiling the parts of the system that are slow, and using that to guess at causes and guide changes.

Let's say what you've looked at suggests that you should significantly increase `shared_buffers`, the primary tunable for memory used to cache database reads and writes. This normally has some positive impact, but there are also potential negative things you could encounter instead. The information needed to figure out which category a new application will fall into, whether this change will increase or decrease performance, cannot be predicted from watching the server running with the smaller setting. This falls into the category of chaos theory where even a tiny change in starting conditions can end up rippling out to a very different end condition, as the server makes millions of decisions impacted to a small degree by that change. Similarly, if `shared_buffers` is set too small, there are several other parameters that won't work as expected at all, such as those governing database checkpoints.

Since you can't predict what's going to happen most of the time, the mindset you need to adopt is one of heavy monitoring and change control. Monitor as much as possible, from application to database server to hardware. Introduce a small targeted change. Try to quantify what's different and be aware that some changes you have rejected as not positive won't always stay that way forever. Move the bottleneck to somewhere else, and you may discover that some parameter that didn't matter before is now suddenly the next limiting factor.

There's a popular expression on the mailing list devoted to PostgreSQL performance when people speculate about root causes without doing profiling to prove their theories: "less talk, more `gprof`". While `gprof` may not be the tool of choice for every performance issue, given it's more of a code profiling tool than a general monitoring one, the idea that you measure as much as possible before speculating as to root causes is always a sound one. And measure again to validate your change did what you expected, too.

Another principle that you'll find a recurring theme of this book is that you must be systematic about investigating performance issues. Do not assume your server is fast because you bought it from a reputable vendor; benchmark the individual components yourself. Don't start your database performance testing with application level tests; run synthetic database performance tests you can compare against other people's first. That way, when you run into the inevitable application slowdown, you'll already know your hardware is operating as expected and that the database itself is running well. Once your system goes into production, some of the basic things you might need to do in order to find a performance problem, such as testing hardware speed, become impossible to take the system down for.

You'll be in much better shape if every server you deploy is tested with a common methodology, which is exactly what later chapters in this book leads you through. Just because you're not a "hardware guy", that doesn't mean you should skip over the parts here that cover things like testing your disk performance. You need to do such work like that as often as possible when exposed to new systems. That's the only way to get a basic feel for whether something is operated within the standard range of behavior, or if, instead, there's something wrong.

Summary

PostgreSQL has come a long way in the last five years. After building solid database fundamentals, the many developers adding features across the globe have made significant strides in adding both new features and performance improvements in recent releases. The features added to the latest PostgreSQL 9.0, making replication and read scaling easier than ever before, are expected to further accelerate the types of applications the database is appropriate for. There are still some situations where PostgreSQL's feature set results in slower query processing than some of the commercial databases it might otherwise displace. If you're starting a new project using PostgreSQL, use the latest version possible, and definitely to deploy 9.6 or later. PostgreSQL works well in many common database applications, but there are certainly applications it's not the best choice for. Not everything you need to manage and optimize a PostgreSQL server will be included in a basic install. Expect to need some additional utilities that add features outside of what the core database aims to provide. Performance tuning is best approached as a systematic, carefully measured practice. In the following chapter, we will discuss the hardware best-suited for the PostgreSQL server.

2
Database Hardware

Database Hardware This chapter aims to help prioritize spending when planning out the purchase of a new server intended to run PostgreSQL. If you already have a running database server, following the instructions in the benchmarking chapters might be a more appropriate place to start than here. Ultimately, you may end up taking a few round trips, alternating through that material and what's covered here. For example, if you benchmark your server, and the disks seem slow, the background here might give you an idea what hardware change you could make to improve that. Once that's done, it's back to benchmarking again; repeat until performance matches expectations.

In this chapter, we will cover the following topics:

- Balancing hardware spending
- Reliable controller and disk setup

Balancing hardware spending

One of the reasons that working with open-source databases such as PostgreSQL can be so effective is that every dollar you save on software licensing can be put toward better hardware instead. The main three components you'll need to balance in your budget are CPUs, memory, and disks, with the disk controller as a related and critical part, too.

CPUs

Currently, available processors are bundling at least two, and possibly as many as eight, cores into each CPU, making the core count the figure of merit for most database applications. There are two basic decisions you need to make when deciding which CPU solution would best match your database application:

- Which processor family? Nowadays, this normally boils down to choosing among the various 64-bit product lines from Intel or AMD, although there are some other, less popular, choices still floating around (Itanium, SPARC, and so on).
- Do you get more cores or faster cores?

The complete detailed list of multicore processors can be found at: `https://en.wikipedia.org/wiki/Multi-core_processor`.

These choices are sometimes more tied together than you might think. Currently, Intel has a lead in delivering individual processor cores that are the fastest around, often due to faster transfers between the processor and system RAM. But the processors and related parts are more expensive, too. AMD is still competitive at providing more cores per dollar, and their server class designs normally do a good jobs making the best of the memory available to each core. But if what you want is many more affordable cores instead, that's where AMD is stronger. AMD also has a better history of making its fastest processors available in configurations with many sockets, when you want to put more than two physical CPUs into a single server.

The best way to figure out which class of database app you have--more cores or faster cores--is to monitor an existing server using tools such as top. If there are a small number of processes running using a single CPU each, that's the sort of workload where faster cores are better. This tends to happen if you have giant queries running in a batch fashion, for example, when large quantities of data need to be sorted to deliver any single report. But if all the CPUs are active with many more concurrent processes instead, you'd likely benefit more from having more cores. That's normally what you'll see in applications with a larger user count, such as databases backing web applications.

If you don't have the benefit of a working system to inspect, you can try to guess which type of situation you're in by noting the limitations of the database. Early versions of PostgreSQL do not allow splitting a single query across more than one core, what's called parallel query by some other databases that support it, but PostgreSQL 9.6 starts introducing parallelism. In PostgreSQL 9.6, *Parallel sequential scans*, *parallel joins* and *parallel aggregates* are introduced. So in pre 9.6 releases if you have any one query or a small number of queries that must run as fast as possible, the only way to do that is to prioritize getting faster cores, but in PostgreSQL 9.6 you can have some kind of parallelism.

Another situation where getting a faster core is a better choice, is if you need to prioritize data loading or export situations. PostgreSQL's best performing data import method, `COPY`, can easily become (but isn't always) limited by CPU performance, where that turns into the bottleneck for operations. While it's possible to split input files into pieces and load them in parallel, that's something you'll need to build or acquire yourself, rather than something the server knows how to do for you. Exporting a copy of the database using the `pg_dump` utility is another example of something that can become CPU-limited on some systems.

PostgreSQL 9.6 allows splitting a single query across more than one core, what's called parallel query, like some other databases.

Memory

How much to prioritize memory for your application really depends on the size of the working set of data needed to handle the most common operations. Generally, adding more RAM will provide a significant performance boost. There are a few situations where you'd be better served doing something else instead:

- If your dataset is small enough to fit into a smaller amount of RAM, adding more won't help you much. You probably want faster processors instead.
- When running applications that scan tables much larger than what you can feasibly purchase as RAM, such as in many data warehouse situations, you might be better served by getting faster disks rather than more memory.

The normal situation where more memory helps most is when the data you access frequently will fit with the larger amount but not with the smaller. This happens more often than you might think because of the way database B-tree indexes are stored. Even if you can't fit the entirety of a table, or even its index, in memory, being able to store a good-sized fraction of the index can mean that index-based data lookups will be significantly sped up. Having the most popular blocks from the top of the tree structure cached is helpful even if you can't fit all the leaves into memory too.

Once you have an application running, you can usually get a much better idea of how memory is being used by looking inside the PostgreSQL buffer cache (and potentially inside the operating system one as well) and seeing what data it prefers to keep around. The section on using `pg_buffercache` in `Chapter 5`, *Memory for Database Caching*, shows how to monitor that data.

The `pgfincore` (`http://pgfoundry.org/projects/pgfincore`) is a set of functions to handle low-level management of relations. The code can be downloaded from `git://git.postgresql.org/git/pgfincore.git`.

Disks

While it's always possible to run into situations where the CPU in your database server is its bottleneck, it's downright likely that you'll run into a disk bottleneck--particularly if you only have a drive or two in the system. The two basic choices in hard drives were the inexpensive ATA (also known as IDE) drives used in desktops versus the more serious SCSI drives aimed at servers.

Both technologies have marched forward, and the current choice you're most likely to run into when configuring a database server is whether to use **Serial ATA** (**SATA**) or **Serial Attached SCSI** (**SAS**). It's possible to find nearly identical drives available in both interfaces, and there are even drive controllers that allow attaching either kind of drive. Combined with a narrowing performance difference between the two, picking between them is harder than ever.

The broad parameters of each technology are straightforward to compare. Here's the state of things at the time of writing:

- **SAS disks**:
    - The maximum available RPM is higher: 10 K or 15 K
    - Not as much drive capacity: 73 GB-1 TB are popular sizes
    - Cost per MB is higher

- **SATA disks**:
    - Drives typically have a slower RPM--7,200 is standard; some 10K designs exist, such as the Western Digital VelociRaptor
    - Higher drive capacity--2 TB available
    - Cost per MB is lower

Generally, you'll find individual SAS disks to be even faster than SATA ones with similar specifications. In particular, you're likely to see better seek performance on random I/O due to faster drive mechanics in SAS, and sometimes a faster transfer rate from the disk, too. Also, because the SAS drives have supported advanced features, such as command queuing, for longer, it's more likely your operating system will have matching support to take advantage of them, too.

There are some other options in the area of hard disks, as follows:

- **Parallel ATA**: `https://en.wikipedia.org/wiki/Parallel_ATA`
- **Serial ATA**: `https://en.wikipedia.org/wiki/SATA`
- **SCSI**: `https://en.wikipedia.org/wiki/SCSI`
- **Solid-state drive**: `https://en.wikipedia.org/wiki/SSD`

RAID

The **Redundant Array of Inexpensive Disks** (**RAID**) approach is the standard way to handle both the performance and reliability limitations of individual disk drives. A RAID array puts many disks, typically of exactly the same configuration, into a set that acts like a single disk-but with either enhanced performance, reliability, or both. In some cases, the extra reliability comes from computing what's called parity information for writes to the array. Parity is a form of checksum on the data, which allows for reconstruction even if some of the information is lost. RAID levels that use parity are efficient, from a space perspective, at writing data in a way that will survive drive failures, but the parity computation overhead can be significant for database applications.

The most common basic forms of RAID arrays used are as follows:

- **RAID 0, also called striping**: Multiple disks are used at the same time, spreading reads and writes over each of them in parallel. This can be almost a linear improvement (two disks reading twice as fast as a single one), but a failure on any volume in the set will lose all the data.

- **RAID 1, also called mirroring**: Here, more copies of the same data are put onto multiple disks. This can sometimes improve performance--a good RAID 1 mirroring across two disks might handle two reads by sending one to each drive. Reads executed in parallel against both drives can effectively double average seeks per second. But generally, the reason for RAID 1 is redundancy: if a single drive fails, the system will continue operating using the other one.
- **RAID 10 or 1+0**: This first takes pairs of disks and mirrors them using RAID 1. Then, the resulting set is striped using RAID 0. The result provides both high performance and the ability to tolerate any single disk failure, without as many ways for speed to suffer in the average and worst case as RAID 5/6. RAID 10 is particularly appropriate for write-heavy environments, where the parity computation overhead of RAID 5/6 can cause disk performance to suffer. Accordingly, it's the preferred RAID level for high-performance database systems.
- **RAID 5, also called striped with parity**: This approach sits midway between 0 and 1. You stripe data across multiple drives similarly to RAID 0, which improves read performance. But some redundant data is added to a parity drive. If one of the disks in the array is lost, the missing data can be recomputed from the ones left using that parity information. While this is efficient in terms of how little space is wasted relative to the tolerance for disk failures provided, write performance in particular can suffer in RAID 5.
- **RAID 6**: Similar to RAID 5, but utilizes two independent parity functions distributed across the drives (`http://www.freeraidrecovery.com/library/what-is-raid.aspx`), enabling survival even with two disk failures. It has the same fundamental advantages and disadvantages. RAID 6 is an increasingly common way to cope with the fact that rebuilding a RAID 5 array after a disk loss can take a really long time on modern high--capacity drives. The array has no additional fault tolerance during that period, and seeing a second drive failure before that rebuild finishes is not that unlikely when it takes many hours of intense disk activity to rebuild. Disks manufactured in the same batch are surprisingly likely to fail in groups.

Differences between RAID 5 and RAID 6 can be found at `http://www.freeraidrecovery.com/library/raid-5-6.aspx`.

To be fair, in any disk performance comparison, you need to consider that most systems are going to have a net performance from several disks, such as in a RAID array. Since SATA disks are individually cheaper, you might be able to purchase considerably more of them for the same budget than had you picked SAS instead. In cases where your application accelerates usefully from being spread over more disks, being able to get more of them per dollar can result in an overall faster system. Note that the upper limit here will often be your server's physical hardware.

You only have so many storage bays or controller ports available, and larger enclosures can cost more, both up-front and over their lifetime. It's easy to find situations where smaller numbers of faster drives, which SAS provides, are the better way to go. This is why it's so important to constantly be benchmarking both hardware and your database application, to get a feel for how well it improves as the disk count increases.

Drive error handling

Just because it's possible to buy really inexpensive SATA drives and get good performance from them, it doesn't necessarily mean you want to put them in a database server. It doesn't matter how fast your system should be if it's down because of a bad drive.

The first step toward reliable hard drive operation is for the drive to accurately report the errors it does run into. This happens via two mechanisms: error codes reported during read and write operations, and drive status reporting via the **SMART** protocol. SMART provides all sorts of information about your drive, such as its temperature and the results of any self-tests run.

When they find bad data, consumer SATA drives are configured to be aggressive in retrying and attempting to correct the error automatically. This makes sense given that there's typically only one copy of that data around, and it's fine to spend a while to retry rather than report the data lost. But in a RAID configuration, as often found in a database environment, you don't want that at all. It can lead to a timeout, and generally makes things more difficult for the RAID controller. Instead, you want the drive to report the error quickly, so that an alternative copy or copies can be used instead. This form of error handling change is usually the main difference in SATA drives labeled enterprise or RAID edition nowadays: those will report errors quickly, where the non-RAID versions will not. Having to purchase the enterprise version of a SATA hard drive to get reliable server error handling operation does close some of the price gap between them and SAS models. This can be adjustable in drive firmware, however; see `http://en.wikipedia.org/wiki/TLER` for more information about drive features in this area.

Generally, all SAS disks will favor returning errors so data can be reconstructed rather than trying to self-repair. Additional information about this topic is available, as part of a commentary from network appliance about the drive reliability studies mentioned in the following section, at `http://storagemojo.com/2007/02/26/netapp-weighs-in-on-disks/`.

Using an external drive for a database:
External drives connected over USB or FireWire can be quite crippled in their abilities to report SMART and other error information, due to both the limitations of the common USB/Firewire bridge chipsets used to connect them and the associated driver software. They may not properly handle write caching for similar reasons. You should avoid putting a database on an external drive using one of those connection methods. Newer external drives using external SATA (eSATA) are much better in this regard, because they're no different from directly attaching the SATA device.

Drive firmware and RAID

In some disk-array configurations, it can be important to match the firmware version of all the drives used for reliable performance. SATA disks oriented at consumers regularly have large changes made to the drive firmware, sometimes without a model number change. In some cases, it's impossible to revert to an earlier version once you've upgraded a drive's firmware, and newer replacement drives may not support running an earlier firmware either. It's easy to end up in a situation where you can't purchase a replacement for a damaged drive even if the model is still on the market.

If you're buying a SAS drive, or one of the RAID-oriented enterprise SATA ones, these tend to have much better firmware stability. This is part of the reason these drives lag behind consumer ones in terms of maximum storage capacity. Anyone who regularly buys the latest, largest drives available on the market can tell you how perilous that is-new drive technology is rather unreliable. It's fair to say that the consumer market is testing out the new technology. Only once the hardware and associated firmware has stabilized does work on the more expensive, business-oriented versions, such as SAS versions, begin. This makes it easier for the manufacturer to keep firmware revisions stable as new drive revisions are released-the beta testing by consumers has already worked out many of the possible bugs.

SSDs

The latest drive technology on the market right now is **Solid State Drive** (**SSD**), also called flash disks. These provide permanent memory storage without any moving parts. They can vastly outperform disks with moving parts, particularly for applications where disk seeking is important-often the case for databases-and they should be more reliable, too.

There are three major reasons why more databases don't use SSD technology yet:

- Maximum storage capacity is low, and databases tend to be big
- Cost for the storage you do get is fairly high
- Most SSD designs do not have a well-designed write--cache on them

Due to how the erasing mechanism on a SSD works, you must have a write cache - typically a few kilobytes, to match the block size of the flash cells--for them to operate in such a way that they will last a long time. Writes are cached until a full block of data is queued up, then the flash is erased and the new data written.

While small, these are still effectively a write--back cache; with all the potential data corruption issues any such design has for database use (as discussed in detail later in this chapter). Some SSDs include a capacitor or similar battery backup mechanism to work around this issue; the ones that do not may have very rare, but still real, corruption concerns.

Until SSD manufacturers get better about describing exactly under what conditions the write-cache in their device can be lost, this technology remains an unknown risk for database use and should be approached with caution. The window for data loss, and therefore database corruption, is very small, but it's often there. And you usually can't resolve the issue by using a controller card with its own battery-backed cache. In many cases, current generation cards won't talk to SSDs at all, and even if they can, the SSD may not accept and honor the same commands to disable its write cache that a normal drive would--because said cache is so central to the longevity aspects of the drive.

A detailed description of SSD and comparison with HDD can be found at `http://www.storagereview.com/ssd_vs_hdd`.

Disk controllers

One of the most critical aspects of PostgreSQL performance is also one of the easiest to overlook. Several hardware trends have made the disk controller seem less important now:

- Increasing CPU speeds and improved **direct memory access** (**DMA**) approaches make offloading disk work to a controller card seemingly less valuable. There is little chance that your controller will be any faster at previously expensive tasks such as RAID parity computation than your main system processor(s).
- Recent operating-system filesystem technology such as Linux Software RAID and ZFS make hardware RAID, with its often-proprietary storage, seems less desirable.
- disk hardware almost completely disconnected from the interface that your operating system sees.

Virtual machines and cloud deployments make the underlying disk hardware almost completely disconnected from the interface that your operating system sees. Faster drives such as SSD seem to eliminate the need for an intermediate intelligent controller.

Do not be fooled, however. When it comes to committing information into a database, spinning disk media--and even flash media such as SSDs--has some limitations that no amount of software cleverness can overcome.

Hardware and software RAID

When implementing a RAID array, you can do so with special hardware intended for that purpose. Many operating systems nowadays, from Windows to Linux, include software RAID that doesn't require anything beyond the disk controller on your motherboard.

Hardware RAID has some advantages. In many software-RAID setups, such as the Linux implementation, you have to be careful to ensure the system will boot off either drive in case of a failure. The BIOS provided by hardware cards normally takes care of this for you. And in cases where a failed drive has been replaced, it's usually easier to set up an automatic rebuild with a hardware-RAID card.

When hard drives fail, they can take down the entire system in the process if they start sending bad data to the motherboard. Hardware RAID controllers tend to be tested for that scenario, whereas motherboard drive controllers aren't, necessarily.

The main disadvantage to hardware RAID, beyond cost, is that if the controller fails, you may not be able to access the drives anymore using a different controller. There is an emerging standard for storing the RAID metadata in this situation, the SNIA Raid **Disk Data Format** (**DDF**). It's not well supported by controller card vendors yet, however.

The biggest advantage of hardware RAID in many systems is the reliable write caching they provide. This topic is covered in detail later in this chapter.

Recommended disk controllers

There are plenty of disk controllers on the market that don't do well at database tasks. Here are a few products that are known to work well with the sort of hardware that PostgreSQL is deployed on:

- LSI's **MegaRAID** line has been a source for reliable, medium performance SCSI and now SAS/SATA controllers for many years. They tend to have smaller cache sizes and their older products in particular were not always the fastest choice available, but their
- Dell has offered a rebranded LSI MegaRAID card as their **PowerEdge RAID Controller** (**PERC**) for some time now. The PERC6 is based on the LSI SAS design mentioned in the previous point, as are its replacements, the PERC H700 and H800 (avoid the H200, which has no write cache at all). The PERC5 and earlier models tended to be slow, and the Dell customized firmware often didn't work as well as the ones in the genuine LSI models. These issues are all cleared up in the PERC6 and later models, which can easily clear 1 GB/s of reads from a properly configured 24-disk array.
- 3ware was one of the first companies to offer SATA RAID solutions, and they're particularly well known for providing excellent Linux drivers. Some of the earlier 3ware models had unimpressive performance; the current 9690SA is a solid mid-range performer if configured correctly. 3ware has gone through some company transitions; they were bought by AMCC, who were then bought by LSI. Eventually, you can expect that 3ware will be just another LSI line.
- HP provides a few RAID cards in their Smart Array series of products, including the P400, P600, and P800. The main difference between these cards is performance. The P800 is well respected as a card with high performance; the P400 and P600 are considered at best medium-speed performers.
- Emulex and QLogic provide the most popular high-performance cards for attaching fiber channel disk arrays to a server.

- Areca is a less well-known company than the rest on this list, but they've gained a following among fans of high-performance SATA cards; some models support SAS as well. Areca cards are featured in a few **white-box** vendor systems provided by resellers, for those who prefer not to deal with the big vendors mentioned here. One concern with Areca is getting a management utility that is compatible with your system. The more expensive models that include a built-in management network port, what they call their out-of-band manager, are easiest to deal with here--just access the card over the network via its web console.

Driver support for Areca cards depends heavily on the OS you're using, so be sure to check this carefully. Under Linux, for example, you may have to experiment a bit to get a kernel whose Areca driver is extremely reliable, because this driver isn't popular enough to get a large amount of testing. The 2.6.22 kernel works well for several heavy PostgreSQL users with these cards.

Typically, the cheapest of the cards you'll find on the preceding list currently sells for around $300 USD. If you find a card that's cheaper than that, it's not likely to work well. Most of these are what's referred to as **Fake RAID**. These are cards that don't actually include a dedicated storage processor on them, which is one part that bumps the price up substantially.

Instead, Fake RAID cards use your system's CPU to handle these tasks. That's not necessarily bad from a performance perspective, but you'd be better off using a simple operating system RAID (such as the ones provided with Linux, or even Windows) directly. Fake RAID tends to be buggy, have low-quality drivers, and you'll still have concerns about the volume not being portable to another type of RAID controller. They won't have a battery-backed cache, either, which is another major component worth paying for in many cases.

Prominent vendors of Fake RAID cards include **Promise** and **HighPoint**. The RAID support you'll find on most motherboards, such as Intel's RAID, also falls into the fake category. There are some real RAID cards available from Intel, though, and they manufacture the I/O processor chips used in several of the cards mentioned here.

Even just considering the real hardware RAID options here, it's impossible to recommend any one specific card, because business purchasing limitations tend to reduce the practical choices. If your company likes to buy hardware from HP, the fact that Areca might be a better choice is unlikely to matter; the best you can do is know that the P800 is a good card, while their E200 is absent from the preceding list for good reason: it's slow. Similarly, if you have a big Dell purchasing contract already, you're likely to end up with a PERC6 or H700/800 as the only practical choice. There are too many business-oriented requirements that filter down what hardware is practical to provide a much narrower list of suggestions than mentioned here.

Attached storage - SAN and NAS

If you are connecting hard drives directly to your server through the motherboard or add-in cards, without leaving the case itself, it's referred to as **Direct Attached Storage** (**DAS**). The other alternative is to use an external interface, usually fiber channel or Ethernet, and connect a **Storage Array Network** (**SAN**) or **Network Attached Storage** (**NAS**) to hold the database disks. SAN and NAS hardware is typically much more expensive than DAS, and easier to manage in complicated ways. Beyond that, comparing the two is somewhat controversial.

There are external drive arrays available that attach over SAS or eSATA cables, so they appear as direct-attached storage even though they are technically external to the server chassis. Dell's PowerVault is a popular and relatively inexpensive example known to scale to 192 DAS drives.

A SAN or NAS (when SAN is used later in this section, it's intended to refer to both) has a few clear advantages over direct storage:

- Easier to use many drives. It's hard to get direct storage to go much over 24 drives without moving into multiple external storage units, but that's only a medium-sized SAN that can be managed as a single component.
- Read/write caching is often much larger, where 16 GB of cache is not unusual in a SAN, while direct storage will normally top out at closer to 1 GB. Also, SAN designs can include a way to dump even that large cache to disk cleanly in the event of any power failure, making it very hard to end up with a corrupted database in the process, no matter how long the outage.

- SANs are easier to make redundant across multiple servers. Typically, you'll have at least two fiber channel ports in a SAN, making it easy to connect to two systems if you want. Then, either server can access the data, as long as you partition access between the two properly.
- The management interface for a SAN will usually include fancier rebuild possibilities, and features such as snapshots that can make backup and mirroring operations much easier.

There are a few potential drawbacks as well:

- Performance can suffer compared to direct storage. Everything else being equal, going through the SAN/NAS interface (fiber channel, Ethernet, and so on) adds latency no matter what, and can introduce a write bottleneck, too. This can really slow your system down if, for example, you're using an interface such as Gigabit Ethernet, with many disks behind it. You can easily fill the network capacity long before you're reached the speed of all available drives in that case.
- SANs are very complicated. It's common to need to hire a consultant just to get the system working as you expect for your workload.
- Costs are much higher in terms of performance per dollar on a SAN compared to direct storage. If you really need the redundancy and management features a SAN provides, they can make perfect sense. This is particularly true if the SAN is so large it's serving out work for many systems at once, which can make the whole system easier to cost-justify.

If you want performance at a reasonable price, direct storage is what you'll end up with. If you need a SAN or NAS, the reasons why are likely related to your business rather than its performance.

If you follow the disk benchmarking practices recommended in this book, you should be able to nail down results quickly enough to make good performance a required component of the sales cycle for the SAN. Considering their price tag, you certainly can make the vendor of the SAN do some work as part of selling it if you benchmark before the system is completely paid for, in a standard way whose results aren't likely to be disputed. This can sidestep the need to hire an external consultant, by leveraging the resources inside the vendor instead.

Reliable controller and disk setup

PostgreSQL uses a **write-ahead log** (**WAL**) to write data in a way that survives a database or hardware crash. This is similar to the log buffer or REDO log found in other databases. The database documentation covers the motivation and implementation of the WAL at `https://www.postgresql.org/docs/current/static/wal-intro.html`.

To quote from the introduction:

> *WAL's central concept is that changes to data files (where tables and indexes reside) must be written only after those changes have been logged, that is, after log records describing the changes have been flushed to permanent storage.*

This procedure ensures that if your application has received `COMMIT` for a transaction, that transaction is on permanent storage, and will not be lost even if there is a crash. This satisfies the durability portion of the **atomicity, consistency, isolation, durability** (**ACID**) expectations databases aim to satisfy.

The tricky part of the WAL implementation is the *flushed to permanent storage* part, which you might be surprised to learn will take several pages to cover in this chapter alone-and come back up again in a later ones, too.

Hard drive reliability studies

Generally, expected reliability is also an important thing to prioritize. There have been three excellent studies of large numbers of disk drives published in the last few years:

- **Google**: *Failure Trends in a Large Disk Drive Population* (`http://research.google.com/archive/disk_failures.pdf`)
- **Carnegie Mellon study**: *Disk failures in the real world* (`http://www.cs.cmu.edu/~bianca/fast07.pdf`)
- **University of Wisconsin-Madison and Network Appliance**: *An Analysis of Data Corruption in the Storage Stack*, (`http://www.usenix.org/event/fast08/tech/full_papers/bairavasundaram/bairavasundaram.pdf` (long version), or `http://www.usenix.org/publications/login/2008-06/openpdfs/bairavasundaram.pdf` (shorter version))

The data in the Google and Carnegie Mellon studies don't show any significant bias toward the SCSI/SAS family of disks being more reliable. But the U of W/Netapp study suggests that "SATA disks have an order of magnitude higher probability of developing checksum mismatches than Fibre Channel disks." That matches the idea suggested above, that error handling under SAS is usually more robust than on similar SATA drives. Since they're more expensive, too, whether this improved error handling is worth paying for depends on your business requirements for reliability. This may not even be the same for every database server you run. Systems where a single master database feeds multiple slaves will obviously favor better using components in the master as one example of that.

You can find both statistically reliable and unreliable hard drives with either type of connection. One good practice is to only deploy drives that have been on the market long enough that you can get good data from actual customers on the failure rate. Newer drives using newer technology usually fail more often than slightly older designs that have been proven in the field, so if you can't find any reliability surveys, that's a reason to be suspicious. The reliability is more important then the speed and performance, because data is a precious entity in database terminology, so selecting a reliable drive is very important in database deployment.

Write-back caches

The CPUs and memory in your server are quite fast compared to its disk drives. Accordingly, making the rest of the system wait for the disks, particularly when things need to be written out, can drag overall performance down heavily. Systems that wait for the disks to complete their writes before moving into their next task are referred to as having a **write-through cache**. While the data may be stored temporarily in a memory cache, until it's made it all the way through to the physical disk, any write an application requested isn't considered complete.

The normal solution to making that faster is to introduce a different type of write cache between the program doing the writing and the disks. A **write-back cache** is one where data is copied into memory, and then control returns to the application that requested the write. Those writes are then handled asynchronously, at some future time dictated by the design of the write-back cache. It can take minutes before the data actually makes it to disk.

When PostgreSQL writes information to the WAL, and sometimes when it writes to the regular database files, too, that information must be *flushed to permanent storage* in order for the database's crash corruption defense mechanism to work. So, what happens if you have a write-back cache that says the write is complete but it really isn't? People call these lying drives, and the result can be very bad.

If you have a system with a write-back cache and a system crash causes the contents of that write-back cache to be lost, this can corrupt a PostgreSQL database stored on that drive and make it unusable! You can discover it takes expert intervention to even get the database to start again, and determining what data is damaged will be difficult.

Consider the case where you have committed a transaction. Details of that new transaction might be spread across two data blocks on the drive. Now, imagine that one of those made it to disk before the system crashed, but the other didn't. You've now left the database in a corrupted state--one block refers to a transaction that doesn't exist where it's supposed to in the other block.

Had all of the data blocks related to the WAL been written properly, the database WAL can correct this error after the crash. But the WAL protection only works if it can get honest information about whether information has been written to the disks properly or not, and the lying write-back caches do not report that.

Sources of write-back caching

Servers are filled with write caches you need to be aware of:

- **Operating system write cache**: This cache can easily be gigabytes in size. Typically, you can flush data out of this cache by forcing a `sync` operation on the block that needs to be stored on disk. On POSIX systems (which includes all UNIX-like ones), this is done with the `fsync` or `fdatasync` calls. In some cases, it's possible to write directly in a `sync` mode, which is effectively a write followed by `fsync`. The `postgresql.conf` setting `wal_sync_method` controls which method is used, and it's possible to disable this altogether to optimize for speed instead of safety.
- **Disk controller write cache**: You'll find a write cache on most RAID controller cards, as well as inside external storage such as a SAN. Common sizes right now are 128 MB to 512 MB for cards, but gigabytes are common on a SAN. Typically, controllers can be changed to operate completely in write-through mode, albeit slowly. But by default, you'll normally find them in write-back mode. Writes that can fit in the controller's cache are stored there; the operating system is told the write is complete, and the card writes the data out at some future time. To keep this write from being lost if power is interrupted, the card must be configured with a battery. That combination is referred to as a **battery-backed write cache** (**BBWC**).

- **Disk drive write cache**: All SATA and SAS disks have a write cache on them that on current hardware is 8 MB to 32 MB in size. This cache is always volatile--if power is lost, any data stored in there will be lost. And they're always write-back caches if enabled.

How can you make sure you're safe given all these write-back caches that might lose your data? There are a few basic precautions to take:

- Make sure whatever filesystem you're using properly implements `fsync` calls, or whatever similar mechanism is used, fully. More details on this topic can be found in the `wal_sync_method` documentation and in information about filesystem tuning covered in later chapters.
- Monitor your driver controller battery--some controller cards will monitor their battery health, and automatically switch from write-back to write-though mode when there is no battery or it's not working properly. That's a helpful safety measure, but performance is going to drop hard when this happens.
- Disable any drive write caches--most hardware RAID controllers will do this for you, preferring their own battery-backed caches instead.

Disk controller monitoring

When you have a RAID controller card with a battery-backed cache, you probably expect you'll need to monitor the card to determine when disks fail. But monitoring controller battery health is an equally critical aspect of maintaining a reliable database system when you're using this technology. If the battery fails and you're using it in write-back mode, your writes are not safe. Similarly, if your power fails, preference should be given to shutting the database server down after a few minutes of power loss than trying to keep it going. Integrating in-power monitoring via a UPS or similar mechanism should be part of your database server configuration, so that a short outage results in an orderly shutdown. Consider the purpose of the controller battery to protect yourself from really unexpected outages, such as someone tripping over the power cord. Even if the manufacturer claims the controller battery will last through days of downtime, that's not a good reason to believe you'll actually be safe for an extended outage. You should consider the battery as something you'd prefer to use only for a few minutes of protection. That may be the reality, particularly in a case where the battery is older and has lost much of its capacity-and some controller batteries don't start out with very much capacity. Be sure to run your own tests rather than blindly believing the manufacturer specifications: your data depends on it.

Better RAID controllers will automatically disable write-back mode if their battery stops working normally. If performance suddenly drops on an older server, this is one potential cause.

Oh, and don't forget that every UPS has a battery that degrades over time as well--all the more reason to arrange an orderly shutdown of your server during a power outage, rather than optimistically presuming you can keep it running until power returns.

Disabling drive write caches

If your card doesn't disable all the drive write caches, or if you're using a software RAID approach, you'll need to turn the caches off yourself. The best way to do this is to see if it's possible to change the default write cache state using the utilities provided by the drive manufacturer.

You should be able to do this via software as well. Here is a sample session from a Linux system checking the write cache, toggling it off, confirming the change took, and then toggling it on again:

```
# hdparm -I /dev/sda | grep "Write cache"
 *  Write cache
# sudo hdparm -W 0 /dev/sda

/dev/sda:
setting drive write-caching to 0 (off)
write-caching =  0 (off)
# hdparm -I /dev/sda | grep "Write cache"
    Write cache
# hdparm -W 1 /dev/sda

/dev/sda:
setting drive write-caching to 1 (on)
write-caching =  1 (on)
```

Only the `-W 0` configuration is completely safe for database use. The PostgreSQL WAL documentation suggests similar commands to use for other operating systems.

Performance impact of write-through caching

If you don't have a battery-backed write cache, and therefore can't utilize some memory-based cache to accelerate `fsync` writes, commit performance on your database can be quite bad. The worst case here is where you have a single client that is issuing a commit after every statement it executes. The reality of how a hard drive works means that an individual write happens once each time the drive spins around.

Here are those measurements for the common drive speeds available right now, with computed maximum commit rate:

Rotation speed	Rotation time (ms)	Max commits/second
5,400	11.1	90
7,200	8.3	120
10,000	6.0	166
15,000	4.0	250

It's important to realize how limiting this can be.

If you have a common 7,200-RPM hard drive, no single client can commit more than 120 transactions/second in any situation where all that's available is a write-back cache.

It doesn't matter how many disks you have in a RAID array, or how you configure your software. You must have hardware with a battery, enabling a non-volatile write-back cache, in order to safely exceed this limit.

Some PostgreSQL installs use a RAID controller card just for this purpose, to provide a BBWC, in **Just a Bunch of Disks** (**JBOD**) mode--where no RAID is being done on the controller at all. Sometimes disks are used directly, others layer software RAID on top, which can have some advantages compared to hardware RAID.

If you have more than one client, you'll get more done per commit. It's normal to see >500 committed transactions per second if you have a larger number of clients all committing regularly, because each flushed disk write will include any queued-up commit requests from other clients, too. The other common technique here is to batch commits into larger pieces, perhaps going through 1,000 records at a time rather than a single one, in order to reduce the average impact of commit latency.

Another approach for accelerating systems that don't have a usable write cache is asynchronous commit, covered in `Chapter 6`, *Server Configuration Tuning.*

Summary

Building a database server with good performance is hard work. There are many individual components that are each available in multiple quality levels and corresponding costs. And there are plenty of small details you must get right, or you'll risk data corruption. Luckily, you don't have to start from scratch. Stick to common, well-understood components with known good performance, while keeping an eye on reliability, too, and you can build a well-balanced database server for a reasonable budget. Always make sure to run your own benchmarks on the result, though! It's very easy to sabotage even good equipment with the wrong configuration when running a database. Allocating your hardware budget among CPUs, memory, and disks is very application dependent, so you need to be very careful where to spend more. Also carefully selecting and configuring your controller and disk caches is critical for reliable database operation.

Make sure you are monitoring components in your server that are likely to fail or have a known lifetime, particularly hard drives and the batteries in disk controller cards. Any disk-write caches in your system must be confirmed to properly support the write flushing mechanism the database uses, or database corruption can result.

After discussing the details of the hardware needed by PostgreSQL, it's really important to discuss how to benchmark the hardware. Benchmarking the hardware is as important as benchmarking the database. Therefore before studying the PostgreSQL performance tuning and benchmarking, the next chapter will cover the basic concept of hardware benchmarking.

3
Database Hardware Benchmarking

After all the theory in the preceding chapter, about what makes some systems perform well or poorly, you might be wondering just how your own system measures up. There are several reasons to always do your own hardware benchmarks. The first is simply to be systematic about your performance process. If you always measure things yourself, you'll get a better feel for what good and bad performance looks like, one that can help tip you off to even subtle problems.

Second, in the case of disks in particular, problems here are a very common underlying cause of database performance issues. If your disks are slow, and there are many ways in which that can happen, your database will likely be slow too. It's important, when this happens, to have accurate data on whether the problem is likely at the hardware or software level.

The goal of your basic hardware testing should be to look for large configuration errors, not to analyze every possible parameter. The sorts of problems you need to be most concerned about are really obvious, and detailed optimization is better done at the database-application level.

In this chapter we will look into the following topics:

- CPU and memory benchmarking
- Physical disk performance
- Disk benchmarking tools

CPU and memory benchmarking

The first thing worth testing on a new system is the speed of its memory because, if this is slow, both the CPU and disks will suffer accordingly. You might wonder why this is so important. It's because database work is intensive in both these areas. PostgreSQL works with database data in 8K pages, and it's constantly shuffling those around to system memory as needed to satisfy queries. And looking through those pages for the specific rows needed is CPU intensive. Even on benchmarks that focus on disk-heavy work, the speed of the underlying memory can have a surprisingly high impact on results.

Memtest86+

One of the most valuable tools in the PC hardware technician's bag is **Memtest86+**, a program whose primary purpose is to find intermittent errors in PC memory. It's a great way to burn - in new hardware and confirms it works as expected.

You can download Memtest86+ directly from `http://www.memtest.org/` and create a bootable CD to run the program. Note that it's now included as one of the boot options for many Linux distributions, both on the installer media and when starting the OS normally. Any Ubuntu installer CD, for example, will include the Memtest86+ current when that version was released. Generally, if the program does not fully recognize your CPU hardware information, you probably need a newer version to make sure you're getting accurate results.

Once you've started it from a bootable CD, Memtest86+ reports the speed of the various types of memory at each of the cache levels it identifies. The most useful one is the MBps value reported on the *memory: line*. This will be how fast memory access to the main system RAM is happening, and is good for confirming that performance matches baselines. It can be a quick way to confirm performance and stability if you're adjusting clocking on your memory or CPU to try and speed things up.

One problem with this program is that it requires physical access to the system to run, which isn't always practical. Since Memtest86+ is only running a single process to test memory, its reported total speed number isn't really representative of the capabilities of modern processors. For that you'll need STREAM, which runs easily from a remote session, too.

STREAM memory testing

STREAM is a memory bandwidth-testing program that came from testing high-performance systems for scientific research. The program is hosted at `http://www.cs.virginia.edu/stream/`, and the website includes a variety of sample reports from the program, too.

The STREAM project provides some binaries for several platforms, even Windows. The available Linux binary hasn't worked out well in my own testing. In addition, STREAM aims only to track the maximum bandwidth the system is capable of. One of the things you should also be interested in for PostgreSQL use is how much memory bandwidth a single process can reach. As individual queries in PostgreSQL will only run on one CPU, if that number is low, you may be disappointed with how lone queries run, even on an otherwise idle server.

To try and work on both these problems, for Linux systems, my stream-scaling script, available at `http://github.com/gregs1104/stream-scaling`, tries to automate a more comprehensive view of your memory performance. It downloads the program, measures the total amount of cache in all the system processors, and automatically compiles STREAM to use a value much larger than that. It then loops from a single active thread (presumably running on a single core) upward, until it's using every processor on the server.

Even if you're sold on that idea, it will take you a while to get your own library of references performance-data assembled. A starting set from systems that I've confirmed results from myself is included on the site, and a few examples from there are covered in the following sections.

STREAM and Intel versus AMD

The following table shows the reported STREAM speeds by processor/RAM for a few servers, as the number of cores utilized increases. All except for the one Opteron model (featuring eight sockets of six core processors) are Intel systems. RAM included here is all double data rate synchronous dynamic random access memory, more commonly known as **DDR RAM**.

The main two standards in use in current hardware are DDR2 and DDR3, and the DDR RAM column shows which standard is used and what clock speed the memory is running at. The performance numbers shown for different total core counts are in units of MBps:

Processor	Cores	Frequency (GHz)	DDR RAM (MHz)	1 Core	2 Core	4 Core	All Core
T7200	2	2.00	2/667	2965	3084		3084
Q6600	4	2.40	2/800	4383	4537	4390	4390
Opteron 8431 (8 X 6)	48	2.40	2/800	4038	7996	13520	27214
Xeon E5506	4	2.13	3/800	7826	9016	9297	9297
i7 860	8	2.80	3/1600	9664	13096	14293	13231

This shows clear limiting of bandwidth available to a single core on all of the recent systems with many cores, relative to total system bandwidth. That particular problem has been more restrictive on AMD's systems than Intel's. Even though this large AMD system can achieve 27 GB/s when heavily tasked, a single process barely clears 4 GB/s. This is exactly why there aren't more AMD processors included in this list. From mid-2008 to mid-2010, AMD has lagged considerably behind Intel in terms of memory technology on systems with small numbers of processor sockets. They've fallen out of favor with the purchasing I've been involved in as a direct result. Note the large memory-speed jump for single core results starting with the Intel 5500 series processor in the preceding table. That represents the introduction of Intel's Nehalem architecture, which included a shift to faster DDR3 RAM, among other major memory improvements, and was what pushed Intel far ahead for a solid two years.

A detailed hardware benchmarking presentation can be found at `http://2ndquadrant.com/media/pdfs/talks/pg-hw-bench-2010.pdf`.

CPU benchmarking

It's rather hard to find any CPU benchmark that is more representative of database performance more useful than just using a database to do something processor-intensive. You can easily build some quick PostgreSQL-oriented CPU tests using the database `psql` client and its `\timing` feature, which shows you how long each statement takes to run. Here's an example that just exercises the CPU and memory, by adding the first million integers together with the always-handy `generate_series` set returning function:

```
postgres=# \timing
Timing is on.
postgres=# SELECT sum(generate_series) FROM generate_series(1,1000000);
      sum
--------------
 500000500000
(1 row)

Time: 216.980 ms
```

Here's another more complicated example that may use some disk accesses too, in addition to stressing CPU/memory. It depends on the amount of RAM in your server:

```
postgres=# \timing
Timing is on.
postgres=# CREATE TABLE test (id INTEGER PRIMARY KEY);
CREATE TABLE
Time: 30.455 ms
postgres=# INSERT INTO test VALUES (generate_series(1,100000));
INSERT 0 100000
Time: 185.109 ms
postgres=# EXPLAIN ANALYZE SELECT COUNT(*) FROM test;
                                                     QUERY PLAN
---------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=1693.00..1693.01 rows=1 width=8) (actual time=14.396..14.397 rows=1 loops=1)
   ->  Seq Scan on test  (cost=0.00..1443.00 rows=100000 width=0) (actual time=0.010..7.919 rows=100000 loops=1)
 Planning time: 1.780 ms
 Execution time: 14.428 ms
(4 rows)

Time: 32.862 ms
```

`EXPLAIN` is the psql command to explain the query, which shows the query timing and the query plan.

Both the insertion time and how long it takes to count each value are interesting numbers. The latter also includes some CPU/memory-intensive work related to updating the hint bit values PostgreSQL uses to track transaction visibility information. See `Chapter 7`, *Routine Maintenance,* for more information about that.

Remember, the point of your CPU testing is not to map out a comprehensive view of its performance in every regard. What you should focus on is making sure performance matches similar hardware and that, if this is an upgrade, it exceeds the expectations you have from older systems.

`Chapter 8`, *Database Benchmarking*, will provide a more useful look at CPU capacity from a PostgreSQL perspective. Running a `SELECT`-only test using `pgbench` and more clients than your system has cores, when all the data fits in RAM, is a very good way to see if your CPUs are delivering expected performance. That's my preferred way to see how fast a new CPU I'm evaluating will work for real CPU- and memory-limited workloads. Just note that the `pgbench` in PostgreSQL versions before 9.0 can easily act as a bottleneck on results; more on this topic in `Chapter 8`, *Database Benchmarking*.

A `pgbench` is a PostgreSQL contrib module, which is used to benchmark the PostgreSQL database.

Sources of slow memory and processors

If your memory doesn't look to be as fast as it should be, or your CPU results look suspicious, there are a few common things to look for to figure out why.

Most memory is now designed to work in a dual-channel configuration, with pairs of memory put only into specific slots. If that's not done correctly, you'll halve memory speed by running in a single-channel setup. Memtest86+ can note when this is happening, and your BIOS may realize it if you look for the information.

Poor-quality RAM can introduce a surprisingly large drop in system performance. And just because you have fast memory in your server, it doesn't mean the motherboard is taking advantage of it. The defaults on some systems are quite conservative. Nowadays, your system should be looking up **serial presence detect** (**SPD**) information, provided by your RAM, to determine how fast it should run. But that doesn't always default to optimal performance, and manual tweaking of the memory timing can be required.

Recent high-performance PC RAM aimed at desktop systems uses a newer standard, the **Extreme Memory Profile** (**XMP**) protocol, for the same purpose, to communicate the speed it's capable of running at to the system BIOS when you boot. But if your BIOS doesn't default to checking and using XMP, which some don't, your RAM will run at notably non-extreme speeds. You should be able to find out how fast your RAM is expected to run as a series of timing values. The Intel i7 860 system using DDR3-1600 mentioned above has timing values of 8-8-8-24, for example. The motherboard did not run the RAM at those speeds until I'd adjusted several settings in it. And just because your hardware vendor should be taking care of all this for you doesn't mean it's safe to ignore this whole issue. It's easy for anyone to miss a step and ship a system with degraded performance.

A link to describe the detail of RAM timings is:
`http://www.hardwaresecrets.com/understanding-ram-timings/`

Another problem that you can run into is using memory that doesn't work well with the clock speed of the processor you're using. Processors are often **locked** to certain multiplier possibilities that are relative to the speed at which the main system bus runs. The motherboard ends up doing a complicated negotiation game between the processor and the RAM to find a common denominator speed to run everything at. For example, one of my older systems supported either DDR2-667 or DDR-800 RAM, running at a memory clock of 333 or 400MHz. The system processor ran at 2.4 GHz, and only supported limited multiplier combinations. It turned out that if I used DDR2-667, the common frequency the motherboard settled on was running the memory bus at 300 MHz, with the CPU using an 8X multiplier. So, the RAM was essentially 10% under-clocked relative to its capabilities. Upgrading to DDR2-800 instead used a 400 MHz clock and a 6X CPU multiplier. That's a 33% jump in memory speed just from using a better grade of RAM, to better match the CPU clock possibilities, and overall system performance improved proportionately.

In addition to getting the memory and multiplier details right, processor power management is an increasing source of issues when benchmarking hardware. Many operating systems now default to having modest or aggressive processor power management active by default. This is a surprisingly common issue on Linux, for example. The normal warning sign is that the processor is only shown as running at 1 GHz in `/proc/cpuinfo`, with correspondingly slow results on some benchmarks. Normally, you will need to adjust the Linux CPU governor setup to optimize for performance, rather than lower power use, at least for the duration of the benchmark. Exact details for how to adjust the governor vary by Linux distribution. You may want to return to optimizing for lower energy use at the expense of some performance afterward, once you've confirmed performance can be good when needed.

Physical disk performance

While a lot of high-level information about disk performance has been mentioned already, if you want to get useful benchmarks from drives, you'll need to know a bit more about their physical characteristics. Our discussion will drop back to theory for a bit, followed by examples of real measured disks that demonstrate common things you can expect to see.

Random access and I/O per second

Enterprise storage vendors like to talk in terms of I/O per second, or **IOPS**. If you're buying a SAN for example, expect to be asked "how many IOPS do you expect total and per spindle?" and for measurements provided by the vendor proving good performance to be in this unit. This number represents typical disk performance on a seek-heavy workload and, unfortunately, it is a poor one to fixate on for database applications. Databases are often complicated mixes of I/O with caching involved-sequential reads, seeks, and commits all compete-rather than always being seek-bound.

Spindle is often used as a synonym for a single disk drive, and is used interchangeably here that way. It's more correctly used to only refer to a single section of a disk, the part that rotates. In this case, common use trumps correctness for most writing about this subject.

It's straightforward to compute IOPS for a single disk. You'll need to track down the manufacturer data sheet where they give the detailed timing specifications for the drive. The Seagate Momentus 7200.4 laptop drive used in the examples here has the following specifications:

- **Spindle speed**: 7,200 RPM
- **Average latency**: 4.17 ms
- **Random read seek time**: 11.0 ms

This models the fact that every disk access on a drive requires the following:

- Seeking to the right track on the disk. That's the **random read seek time**.
- Waiting for the sector we want to read to show up under the read head. That's the **average [rotational] latency** time.

The average latency figure here represents rotational latency. That will always be exactly 1/2 of the rotation time of the drive. In this case, 7,200 RPM means one rotation happens every 1/120 of a second, which means a rotation every 8.33 ms. Since, on average, you won't have to wait for a full rotation, that's halved to give an average, making for an expected rotation latency time of 4.17 ms. All 7,200 RPM drives will have an identical rotational latency figure, whereas seek times vary based on drive size, quality, and similar factors.

IOPS is simply a measurement of the average time for both those operations, the seek latency and the rotation latency, inverted to be a rate rather than an elapsed time. For our sample disk, it can be computed as follows:

```
Rotational latency RL = 1 / RPM / 60 / 2 = 4.17ms
Seek time S=11.0ms
IOPS = 1/(RL + S)
IOPS = 1/(4.17ms + 11ms) = 65.9 IOPS
```

Here are a few resources discussing IOPS you might find helpful:

- `http://www.techrepublic.com/blog/the-enterprise-cloud/calculate-iops-in-a-storage-array/`
- `http://www.ryanfrantz.com/posts/calculating-disk-iops/`

Remember that IOPS is always a worst-case. This is the performance the drive is guaranteed to deliver if it's being hammered by requests from all over the place. It will often do better, particularly on sequential reads and writes.

Sequential access and ZCAV

In many database situations, what you're also concerned about is the streaming sequential read or write rate of the drive, where it's just staying in one area rather than seeking around. Computing this value is complicated by the nature of how disks are built.

The first thing to realize about modern hard disks is that the speed you'll see from them highly depends on what part of the disk you're reading from. Disks spin at one speed all the time, referred to as **constant angular velocity** (**CAV**). A typical drive nowadays will spin at 7,200 RPM, and the actual disk platter is circular. When the disk read/write head is near the outside of the disk, the speed of the part passing underneath it is faster than on the inside. This is the same way that, on a car, the outside edge of a tire travels further than the inside one, even though the actual rotation count is the same.

Because of this speed difference, manufacturers are able to pack more data onto the outside edge of the drive than the inside. The drives are actually mapped into a series of zones with different densities on them. There is a longer discussion of this topic at `http://www.coker.com.au/bonnie++/zcav/`, and use of that ZCAV tool will be shown below in the section, *Disk benchmarking tools*.

The practical result is that the logical beginning part of the disk is going to be significantly faster than its end. Accordingly, whenever you benchmark a disk, you have to consider what part of that disk you're measuring. Many disk benchmark attempts give bad data because they're comparing a fast part of the disk, likely the first files put onto the disk, with ones created later that are likely on a slower part.

Short stroking

Since disks have this very clear faster portion to them, and capacities are very large, one observation you can easily make is that you should put the most important pieces of data on the early parts of the disk. One popular technique, named short stroking, limits the portion of the disk used to only include the fastest part, assuring you'll only be accessing its best area. Short stroking can be done just by adjusting the disk's partition table to only include the early part. You might partition the slower portion anyway, but just not use it regularly. Saving it for backups or migration use can be worthwhile. Occasionally, you can force short stroking via more physical means, such as a disk vendor or RAID controller tool that allows limiting the capacity exposed to the operating system.

Commit rate

As covered in the previous chapter, how quickly data can actually be committed permanently to disk is a critical performance aspect for database transaction processing. It's important to measure this area carefully. Speeds that are dramatically higher than expected are usually a sign that one of the write caches has been put into a volatile write-back mode, which, as already explained, can result in data loss and database corruption. Some examples of how that can happen will be covered in Chapter 4, *Disk Setup*.

If you don't have any non-volatile caching available, the basic commit rate for a drive will be similar to its IOPS rating. Luckily, PostgreSQL will put multiple transactions into a physical commit if they aren't happening quickly enough.

PostgreSQL test_fsync

In a source code build of PostgreSQL, the `src/tools/fsync` directory contains a program, named `test_fsync`, which might also be included in some packaged versions. This aims to test the commit rate for each of the ways a given PostgreSQL install might commit records to disk. Unfortunately, this program doesn't give results consistent with other tests, and before PostgreSQL 9.0, it's in the wrong units (elapsed times instead of operations per second). Until it's improved a bit further, its output can't be relied upon.

The INSERT rate

Each time you `INSERT` a record in a standard PostgreSQL install, it does a commit at the end. Therefore, any program that does a series of inserts in a loop, and times them, can measure the effective commit rate, presuming the records are small enough that true disk throughput doesn't become the limiting factor. It's possible to run exactly such a test using the `pgbench` tool shipped with PostgreSQL. You should be able to write your own similar test in any programming language you're familiar with that can issue PostgreSQL `INSERT` statements one at a time. Just make sure you don't batch them into a larger transaction block. That's the right approach if you actually want good performance, but not for specifically testing the commit rate using small transactions.

Windows commit rate

On the Windows platform, where `sysbench` and `test_fsync` will not be available, an `INSERT` test is really the only good option for testing commit rate. Note that the PostgreSQL `wal_sync_method`, needs to be set properly for this test to give valid results. Like most platforms, the Windows defaults will include unsafe write-back cache behavior.

Disk benchmarking tools

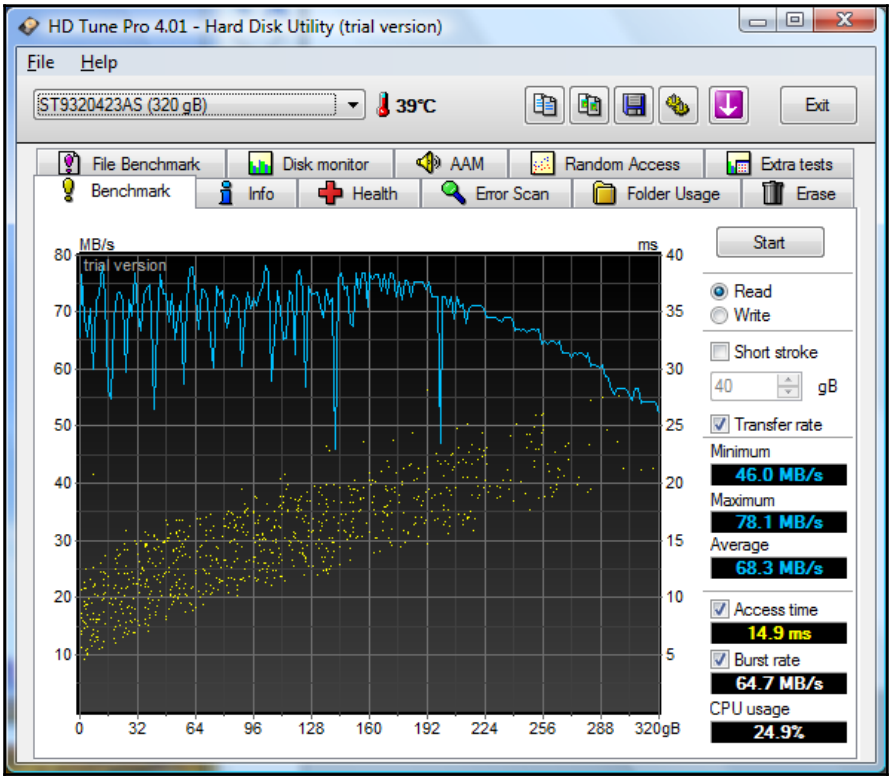
Now it's time to see some real disks measured. For the first few examples here, the drive being tested is a 320 GB Seagate Momentus 7200.4 3 Gbps SATA, model number ST9320423AS. This is one of the faster 2.5" laptop drives on the market, and its more detailed access-time specifications were given in the preceding IOPS section. The results shown are from an installation into a Lenovo ThinkPad T60 with an Intel Core 2 Duo T7200 running at 2.0 GHz.

We'll start with the HD Tune program running on Windows, because its graphs are extremely nice and it measures almost everything you'd hope for. Its graphs demonstrate several aspects of general disk performance more clearly than the command-line tools for UNIX systems covered in the following sections.

Basic disk benchmarking using HD Tune

A great tool for the sort of basic disk benchmarking needed for databases on Windows systems is HD Tune, available at `http://www.hdtune.com/`. The program is free for a trial period, with a slightly limited feature set, and is modestly priced to purchase. The free features are sufficient for database hardware validation.

Here is what the output from HD Tune looks like running the basic transfer rate benchmark on this disk:



These are respectable results from a single drive of any sort, particularly a laptop one. The left axis and the upper line graph is charting MBps at that point on the disk. The right axis label is in milliseconds, and the scattered dots in the middle and bottom are showing the access time along points in the drive. Note how transfer speed falls and access speed rises as you move further along the logical part of this disk into the slower ZCAV zones.

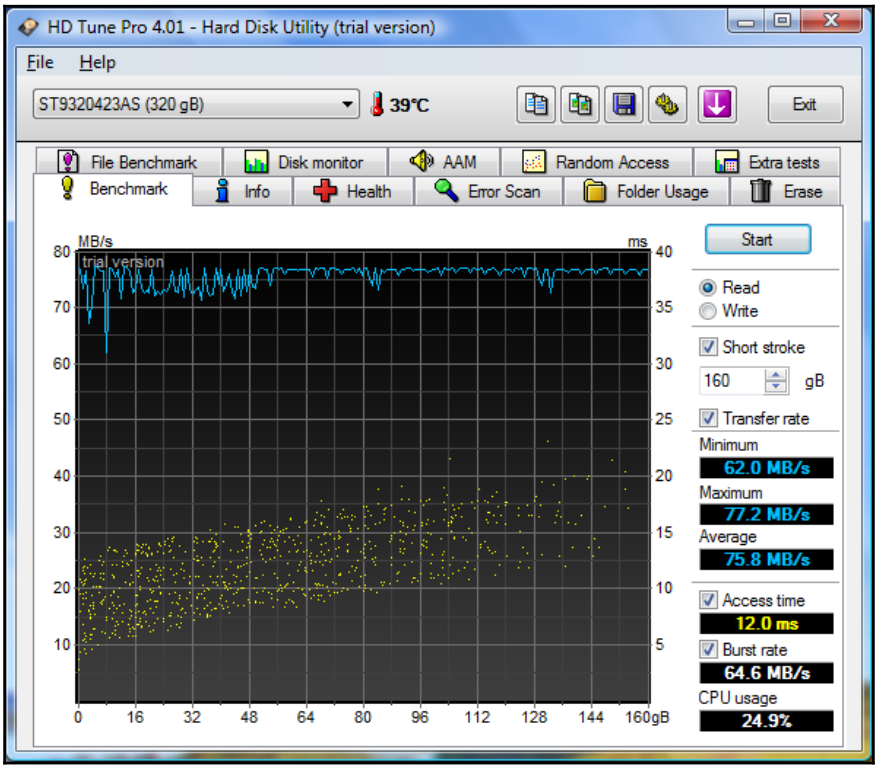
An average of 68.3 MBps is in the middle range of what you'll see on drives at the time of writing. High-performance desktop SATA or SAS drives will do better; some desktop and most laptop drives will do worse. The real-world seek access time of 14.9 ms is a similarly average value, and you can see it matches the drive specification.

There's one thing really odd about this graph, though. Note how the early part, from 0 to 192 GB, is both flat on top and somewhat jagged. Both of those are warning signs that something is wrong. If you look at the Bonnie++ ZCAV results below, you'll see what the drive's performance really looks like. The performance should be dropping evenly from the very beginning, and the graph should be smoother. When you see a flat top where a ZCAV decline should be, it normally means the rate is being limited by some bottleneck other than the drive speed itself. This sort of issue is exactly why it's worthwhile to get familiar with what drives look like when performing well, so you can spot the ones that aren't.

Short stroking tests

HD Tune includes a particularly easy-to-use short stroking test feature that shows you how performance would change if only using a portion of the disk.

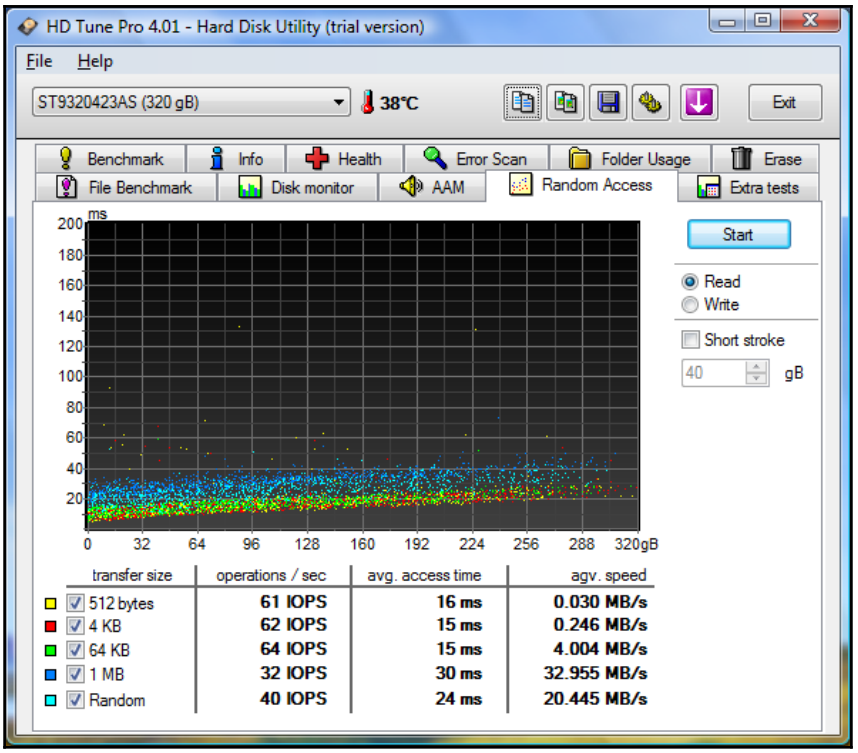
For our example, the first 160 GB were obviously the fastest parts, making that a reasonable short-stroked restriction:



You can see that not using the whole disk range via short stroking considerably decreases worst-case seek time, which makes it particularly appropriate for applications that are more concerned about worst-case latency than disk capacity. In some cases, you may discover that SATA disks with very high capacities end up being much closer in performance to more expensive SAS disks just by applying this technique. For example, the first 100 GB of a 1 TB SATA drive is extremely fast, due to how dense that information is on the drive. You might be surprised at how well it compares with, say, a 143 GB SAS drive, even with a higher rotation speed.

Measuring IOPS using HD Tune

HD Tune includes a **Random Access** test that gives its results in terms of the standard IOPS figure, at various block sizes:



This is not far from the computed value derived above for this drive: 69.5 IOPS.

The average speed computed number here (the poorly labelled `agv. speed` column) is always a good one to note when doing seek-based tests. It's easiest to explain what that means with a database-specific example. Consider that PostgreSQL reads in blocks of 8 KB each. Based on the above, we could expect about 64 IOPS out of this drive as its worst-case performance, doing nothing but random seeks, at that block size. That turns into the total transfer rate shown here:

```
64 IO/S * 8KB/IO * (1 MB / 1024 KB) = 0.5MB/s
```

That's what `agv. speed` is showing: the computed transfer speed for various block sizes.

This is the real world of disk performance. While you might think a disk is capable of 60 MBps or more of transfers, on a truly random workload, you might get 0.5 MB or less. This is a laptop disk, but only seeing 1 to 2 MBps on completely random tasks is typical for desktop- and server-class drives, too.

With enough practice, on a UNIX system, it's even possible to stare at the output from `vmstat` and `iostat`, see how busy the drives are and the actual read/write block counts, and make a rough estimate of the seek versus sequential workload breakdown from those numbers. If the drives are always busy but only getting 0.5 MBps, it has to be almost all seek. If they're busy half the time and getting 30 MBps, that's probably a sequential read hitting a bottleneck on the processing side of things.

Unpredictable performance and Windows

Serious database administrators have a strong historical bias toward using UNIX-like systems for their servers. The first examples here are using Windows instead because the graphs produced are easier to read, and are therefore better for introducing the concepts of this section. But doing so reminded me why Windows is not the preferred database-hosting operating system for so many people.

Getting useful benchmark results requires the system be quiescent: free of other programs running that would spoil the results of what is intended to be measured. When booting into Windows Vista to generate these results, I discovered the **TrustedInstaller** process was hogging a considerable amount of CPU and disk resources. It turned out Windows Update had decided it was time to install the next major Vista Service Pack, it was downloading pieces in the background, and pushing me toward the upgrade at every opportunity. It was two hours later before I had completed all the background activity it compelled me to take care of and had an idle system capable of running these tests.

Disk throughput in case of sequential read and write

`dd` is a standard UNIX utility that's capable of reading and writing blocks of data very efficiently. To use it properly for disk testing of sequential read and write throughput, you'll need to have it work with a file that's at least twice the size of your total server RAM. That will be large enough that your system cannot possibly cache all of the read and write operations in-memory, which would significantly inflate results. The preferable block size needed by `dd` is 8-KB blocks, to match how the database is going to do sequential read and write operations. At that size, a rough formula you can use to compute how many such blocks are needed to reach twice your RAM size is as follows:

```
blocks = 250,000 * (gigabytes of RAM)
```

Once you know that number, the following simple commands will time writing out a file large enough to not fit in the OS RAM cache, and then read the results back:

```
time sh -c "dd if=/dev/zero of=bigfile bs=8k count=blocks &&
 sync" time dd if=bigfile of=/dev/null bs=8k
```

Since this will run for a while without displaying anything interesting, you can watch `vmstat` or `iostat` during the test (in another Terminal) to see what's happening. The vmstat's `bi` and `bo` (block in and out) numbers will match the current read/write rate. You should also note the CPU percentage required to reach the peak rate. Displayed CPU percentages are often relative to the total CPU capacity of the system. So, if you have four CPUs, and you're seeing 25% overall CPU usage, it could be a single CPU running at 100%.

Once the times are reported, you can then divide the file size in MB (=1024 * GB) by the number of seconds of runtime, and get a MBps write and read score.

Recent `dd` versions, on Linux at least, will report a MBps value at the end of their run. Note that the value reported will be a bit inflated, because `dd` will report it is finished before the actual blocks are all flushed to disk. This is why the preceding recipe includes a `sync` at the end--this makes sure the time reported includes that overhead. The raw transfer rate reported by `dd` will usually be a bit higher than what you compute when taking this into account.

The `dd` numbers you'll see should closely match the `Bonnie++` block output/input numbers, as demonstrated in the following section. If you intend to eventually run `Bonnie++`, there's little sense in performing this test too. Using `dd` is mainly helpful for UNIX systems where you don't want to install anything just to test the disks out, including ones that don't have the development tools to build additional software installed.

bonnie++

The standard program used to run a simple disk I/O benchmark on a UNIX-like system is bonnie++. In its default configuration, it will create a file (or set of files) twice as large as the physical RAM in the server, to prevent the entire test from being cached in RAM, then read that whole file back in again. The main virtue of the program is that you can just run it from a directory on the volume you want to test and it usually does the right thing without further input.

Here's a full example of downloading `bonnie++`, building it, running it, and getting an HTML report from it:

```
$ wget http://www.coker.com.au/bonnie++/bonnie++-1.03e.tgz
$ tar xvfz bonnie++-1.03e.tgz
$ cd bonnie++-1.03e/
$ ./configure
$ make
$ ./bonnie++ -f -n 0 | tee `hostname`.bonnie
$ chmod +x ./bon_csv2html
$ cat `hostname`.bonnie | grep "," | ./bon_csv2html  >
`hostname`.htm
```

You'll want to do this on the disk you want to test, or to run the Bonnie++ program from where you compiled it while in a directory on the test disk. Note that the program won't run as root. You'll need to make the directory you're running it in owned by a regular user and then log in as that user before running it.

The preceding example runs the program with two command-line options you will likely always want to use:

- `-n 0`: Skip file creation test
- `-f`: Fast mode, skip per character I/O tests

Neither of those tests are very informative nowadays. What you really want are the sequential block read and write numbers, as well as the seeks result.

You'll have two files come out of this. If you use the suggested file-naming convention shown in the example, a full set of results will be in a file named after your hostname, with the extension `.bonnie`. Without the fast mode enabled, it looks like the following (one run exactly as above will have less information), again from our sample laptop disk:

```
Version 1.03e ------Sequential Output------ --Sequential Input- -
Random-
-Char-    -Block-   -Rewrite- -Char-     -Block-    -Seeks-
Machine  Size  K/s %CP K/s %CP K/s %CP K/s %CP K/s %CP  /s %CP
meddle     4G 44754 96 58386 24 30439 12 52637 97 71059 19 232.8 0
```

The create tests were deleted from the output shown here to save some space, and the above reformatted a bit to fit better into the width of the page here. The results will also be summarized in a list of comma-separated values that look like as follows:

```
meddle,4G,44754,96,58386,24,30439,12,52637,97,71059,19,232.8,0,16,
 11349,36,+++++,+++,+++++,+++,+++++,+++,+++++,+++,+++++,+++
```

The values showing up with + characters are tests that didn't produce useful output. This comma-delimited part can be sent through a program named `bon_csv2html` to produce an HTML-formatted version of the results, which is much easier to read. In the preceding full example, that gets redirected to a file named after your host, with the extension `.htm`.

If you did happen to compute just by running the program with its default parameters, you want to ignore the per-character and create results, and look at the block output/input ones instead.

The `Random Seeks` number reported by Bonnie++ is not a simple read-only seek test. Here's its actual description from the manual:

> *The test runs* `SeekProcCount` *processes (default 3) in parallel, doing a total of 8000 random seek reads to locations in the file. In 10% of cases, the block read is changed and written back.*

This actually makes it a mixed read/write test, which is really more useful for something like a database simulation anyway. Note that the duration of this test--8,000 seeks--is short enough that powerful systems nowadays, with large caches, can easily give inflated results here, at intermittent times. You might see 500 seeks per second on one run, followed by 2,000 per second on the next. It's important to confirm the seek figure using additional tools that run longer.

Bonnie++ 2.0

The preceding results were generated with Bonnie++ 1.03e, the most recent version from the stable release series at this point. If you have a terabyte or larger drive, you'll need 1.03e at a minimum to get useful results; the earlier 1.0 releases don't handle that right. Currently, development is nearly complete on an updated Bonnie++ 2.0. There are already plenty of systems where the V1 Bonnie++ results aren't very useful. Ideally, you'd default to trying the snapshots of the experimental V2 releases (currently at 1.96), and only fall back to the 1.0 series if that doesn't work for you, but as described in the following section, that may not always be practical. I normally end up needing to use both.

Here's a command line that works with V1.96 and provides a faster and more accurate test than the earlier versions:

```
bonnie++ -f -n 0 -c 4
```

In addition to the flags already discussed in older version `-c` is another flag introduced in bonnie++ 2. 0:

- `-c 4`: Concurrency, use four processes at once

Tests here suggest very high values for concurrency fail to deliver any incremental improvement, but going from one up to two, three, or four can improve results in a multi-core system.

Bonnie++ ZCAV

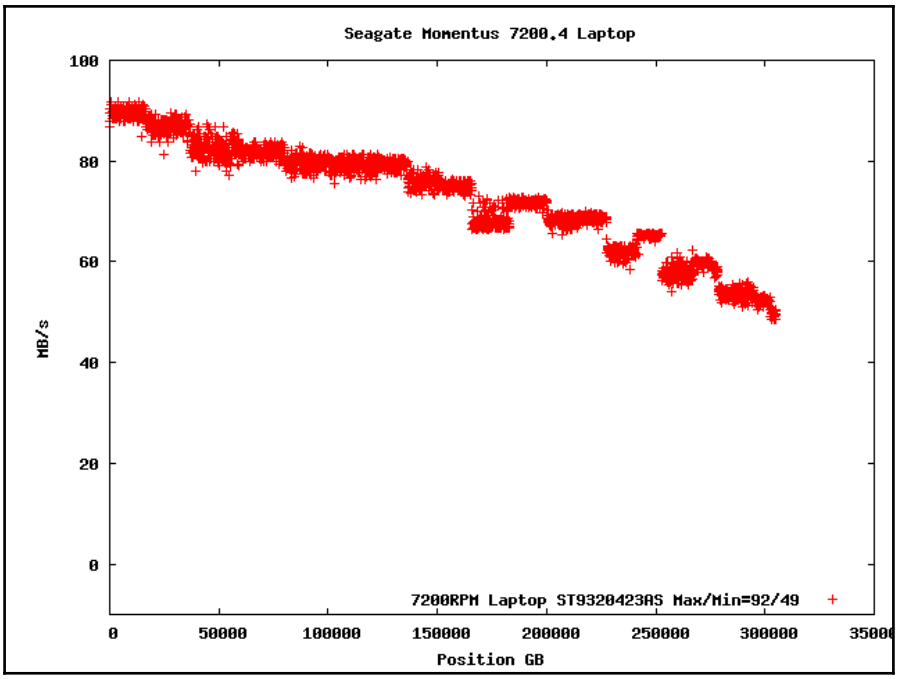
Just like HD Tune, Bonnie++ has a utility, a separate binary named `zcav`, which will track transfer rate across the disk. It produces a simple text file you can save to a file:

```
./zcav -f/dev/sda > sda.zcav
```

You will probably need to run this program as root, since it accesses the whole drive. You can convert the program's output to a graph using the **gnuplot** software.

Unfortunately, results from `zcav` on the experimental branch using version 1.96 haven't been usable in my own tests. I find myself needing to drop back to the 1.03e version for `zcav` to work properly. Hopefully, this will be sorted out before the official version 2.0 is released.

There are some examples of `zcav` results on the Bonnie++ site; here's the output showing performance on the sample laptop used for all the examples so far:



Following is the gnuplot script that produced it:

```
unset autoscale x
set autoscale xmax
unset autoscale y
set autoscale ymax
set xlabel "Position MB"
set ylabel "MB/s"
set key right bottom
set title "Seagate Momentus 7200.4 Laptop"
set terminal png
set output "laptop-zcav.png"
plot "sda.zcav" title "7200RPM Laptop ST9320423AS
 Max/Min=92/49"
```

The ZCAV software didn't produce any of the max/min data or titles shown on that graph; those were all manually edited to make a nicer graph after extracting the values from the raw text file. This is an example of how the Linux tools can give the same quality of basic results as something like HD Tune, but you have to put a bit more work into it.

There is one positive aspect to this extra work: the bottleneck on the early part of the disk seen on the Windows HD Tune results isn't there. Performance peaks at 92 MBps and falls slowly and steadily through all the transfer zones on the disk in the classic ZCAV pattern. Now that you see exactly what this sort of graph is supposed to look like, if you revisit the HD Tune once again, it should be obvious why I immediately suspected a problem when seeing it--the shape just wasn't right. This is exactly the same system, just booted into Ubuntu Linux instead of Windows Vista, which strongly suggests that whatever was limiting the Windows transfer rate to 78 MBps, and therefore flattening the left side of the HD Tune graph, was just some sort of Vista software problem.

This particular issue, noticing that something might be wrong with the disk configuration just because it didn't look right compared to similar hardware seen in the past, is exactly why it's worthwhile even for database administrators to become familiar with hardware benchmarking basics. A more serious problem of this type could easily throttle database performance, and without knowing the hardware is to blame, much time could be wasted attacking that problem from the database side.

sysbench

While initially targeted as a MySQL tool, the `sysbench` program available from `http://sysbench.sourceforge.net/` is also valuable for testing low-level aspects of performance that impact databases, ones that apply equally well to PostgreSQL. It even supports running read-only tests against PostgreSQL databases, too, a feature not demonstrated here. The use in this chapter is just for its low-level hardware tests.

These specific `sysbench` tests should be considered secondary to the ones already shown. If you have confirmed sequential read, write, and seek speeds are good with Bonnie++ and know commit rates are good from doing `INSERT` tests with PostgreSQL, you really don't need this data too. It's mainly useful as an alternative way to do more thorough and specific seek- and commit-rate tests, and is not recommended as your primary testing tool.

Here's an example of downloading and compiling `sysbench` without support for any database, the only way it will be used here:

```
$ git clone https://github.com/akopytov/sysbench.git
$ cd sysbench
$ ./autogen.sh
$ ./configure --without-mysql
$ make
```

This following example shows the running of the simplest test available, the one for CPU speed. The results the test gives are not very useful for comparing modern processors, but it does confirm the program is working quickly:

```
$ sysbench/sysbench --test=cpu run
sysbench 0.5:  multi-threaded system evaluation benchmark

Running the test with following options:
Number of threads: 1
Random number generator seed is 0 and will be ignored


Prime numbers limit: 10000

Threads started!


General statistics:
    total time:                          10.6209s
    total number of events:              10000
    total time taken by event execution: 10.6175s
    response time:
         min:                                  0.98ms
         avg:                                  1.06ms
         max:                                  3.58ms
         approx.  95 percentile:               1.33ms

Threads fairness:
    events (avg/stddev):           10000.0000/0.00
    execution time (avg/stddev):   10.6175/0.00
```

You may discover that it takes more work on the `configure` and `make` steps to build `sysbench` for your system than shown here.

Once you've compiled the program, you don't even need to install it onto your system. It will run fine from a local build in your home directory.

Seek rate

Unlike the rest of the tests `sysbench` is used for in this chapter, the seek-rate test requires a three-step process, where test files are created, the test is run, and then those files are cleaned up. You also have options for how many threads to keep active, how large the file should be, and what read/write mode to use. The following script shows how to run a seek test with the most commonly changed portions as environment variables:

```
$ cat bench.sh
#!/bin/sh
PREFIX="$HOME/sysbench"
THREADS=1
GBSIZE=4
MODE=rndrd
$PREFIX/sysbench/sysbench --test=fileio --num-threads=$THREADS
--file-num=$GBSIZE --file-total-size=${GBSIZE}G --file-block-size=8K
--file-test-mode=rndrd --file-fsync-freq=0 --file-fsync-end=no prepare
$PREFIX/sysbench/sysbench --test=fileio --num-threads=$THREADS
--file-num=$GBSIZE --file-total-size=${GBSIZE}G --file-block-size=8K
--file-test-mode=rndrd --file-fsync-freq=0 --file-fsync-end=no run --ma-time=60
```

To run the script, copy the contents into a file named `bench.sh`:

```
$ chmod +x bench.sh
```

The output is as follows:

```
$ ./bench.sh
sysbench 0.5:  multi-threaded system evaluation benchmark

4 files, 1048576Kb each, 4096Mb total
Creating files for the test...
Extra file open flags: 0
Reusing existing file test_file.0
Reusing existing file test_file.1
Reusing existing file test_file.2
Reusing existing file test_file.3
No bytes written.
sysbench 0.5:  multi-threaded system evaluation benchmark

Running the test with following options:
Number of threads: 1
Random number generator seed is 0 and will be ignored


Extra file open flags: 0
4 files, 1Gb each
4Gb total file size
Block size 8Kb
Number of IO requests: 10000
Read/Write ratio for combined random IO test: 1.50
Using synchronous I/O mode
Doing random read test
Threads started!

Operations performed:  10000 reads, 0 writes, 0 Other = 10000 Total
Read 78.125Mb  Written 0b  Total transferred 78.125Mb  (3.516Gb/sec)
460850.73 Requests/sec executed

General statistics:
    total time:                          0.0217s
    total number of events:              10000
    total time taken by event execution: 0.0194s
    response time:
         min:                                  0.00ms
         avg:                                  0.00ms
         max:                                  0.02ms
         approx.  95 percentile:               0.00ms

Threads fairness:
    events (avg/stddev):           10000.0000/0.00
    execution time (avg/stddev):   0.0194/0.00
```

Removing test files

Unlike the Bonnie++ seek test, which aimed at just twice your total RAM by default, you can easily make this test span a large portion of the disk instead. Any seek results should always include what portion of the disk the seeking took place over. To get useful results from larger disks, you might want to use hundreds of GB-worth of data on this test, instead of just the 2 * RAM that Bonnie++ uses for its seek testing.

fsync commit rate

It's possible to use `sysbench` to measure how fast commits can be flushed to disk using the standard `fsync` call, just like the database defaults to. Note that in this case, the file size being used is `16384` bytes, even though PostgreSQL block writes are `8192` bytes. The version tested here didn't work correctly with the block size reduced that far, and since the actual amount of bytes doesn't impact the commit rate until it becomes much larger anyway, it's not worth worrying about. The following is a Linux-specific script that includes disabling then re-enabling the drive's write cache. The basic `sysbench` call can be used on any platform that program runs on, however:

```
#!/bin/sh
DRIVE="/dev/sda"
PREFIX="$HOME/sysbench"
# Disable write cache
hdparm -W 0 $DRIVE

echo fsync with write cache disabled, look for "Requests/sec"
$PREFIX/sysbench/sysbench --test=fileio --file-fsync-freq=1 --file-num=1
--file-total-size=16384 --file-test-mode=rndwr run

# Enable write cache (returning it to the usual default)
hdparm -W 1 $DRIVE
echo fsync with write cache enabled, look for "Requests/sec"
$PREFIX/sysbench/sysbench --test=fileio --file-fsync-freq=1 --file-num=1
--file-total-size=16384 --file-test-mode=rndwr run
```

On a standard 7,200 RPM drive, spinning 120 times per second and therefore limited to that as its maximum commit rate, the version with the write cache disabled would look like as follows:

```
104.81 Requests/sec executed
```

The cached version will likely show thousands of commits per second.

Complicated disk benchmarks

There are many more complicated disk benchmark programs available:

- **iozone**: `http://www.iozone.org/` allows testing all sorts of disk scenarios.
- **fio**: `http://freshmeat.net/projects/fio/` lets you completely script exactly what benchmark scenario you want to run. Many samples are available at `http://wiki.postgresql.org/wiki/HP_ProLiant_DL380_G5_Tuning_Guide`.
- **pgiosim**: `http://pgfoundry.org/projects/pgiosim/` simulates very specific types of PostgreSQL workloads.

There are a few issues that make all of these less useful than the tools covered here. The first is that these are complicated tools to set up and interpret the results of. Correspondingly, when you do find a problem, if it involves a hardware vendor issue, there's no way they will trust or attempt to replicate things discovered via these tests. The dd, Bonnie++, and HD Tune are, on their respective platforms, known to be simple, reliable, easy-to-interpret tools. If you can show your vendor a problem using one of those tools, there's little they can do to wriggle out of that. Even `sysbench` is a bit more complicated than you'd want to rely upon in a vendor dispute. Simplicity and transparency with your vendor is much more important for doing basic database performance testing than being able to test more complicated scenarios.

And really, if your goal is eventually to tune a database application, tuning exercises should primarily be done at that level anyway. Once you're sure the basic hardware works as expected, move right onto database-level tests and see if tuning changes have any impact. That's much more likely to find out the real issues that do and don't matter than these complicated synthetic disk benchmarks.

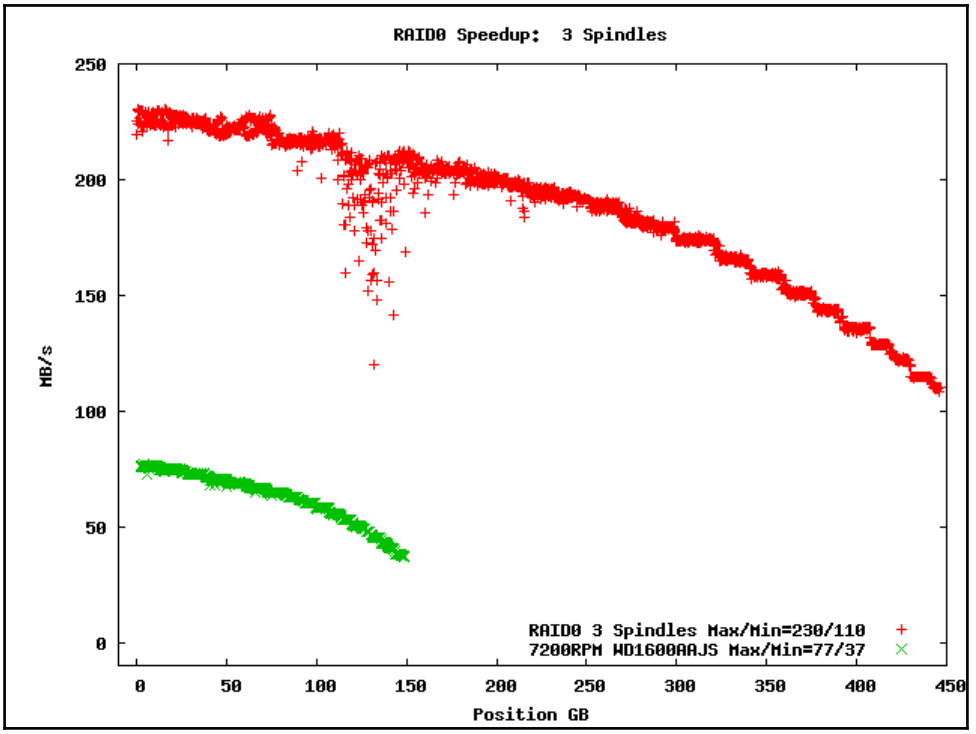
Sample disk results

Here's a summary of what was measured for the laptop drive tested in detail above, as well as a desktop drive, both alone and in a RAID array, as a second useful data point:

Disks	Seq read	Seq write	Bonnie++ seeks	sysbench seeks	Commits per second
Seagate 320 GB 7200.4 Laptop	71	58	232 @ 4 GB	194 @ 4 GB	105 or 1048
WD160 GB 7200 RPM	59	54	177 @ 16 GB	56 @ 100 GB	10212
3X WD160 GB RAID0	125	119	371 @ 16 GB	60 @ 100 GB	10855

Note how all the seek-related information is reported here relative to the size of the area being used to seek over. This is a good habit to adopt. Also, note that in the laptop rate, two commit rates are reported. The lower value is without the write cache enabled (just under the rotation rate of 120 rotations/second), while the higher one has it turned on--and therefore provides an unsafe, volatile write cache.

The other two samples use an Areca ARC-1210 controller with a 256 MB battery-backed write cache, which is why the commit rate is so high yet still safe. The hard drive shown is a 7,200-RPM 160-GB Western Digital SATA drive, model WD1600AAJS. The last configuration there includes three of that drive in a Linux Software RAID0 stripe. Ideally, this would provide 3X as much performance as a single drive. It might not look like this is quite the case from the Bonnie++ read/write results: those represent closer to a 2.1X speedup. But this is deceptive, and once again it results from ZCAV issues. Using the Bonnie++ ZCAV tool to plot speeds on both the single drive and RAID0 configurations, you get the following curves:



The max and min transfer speed numbers derived from the raw data are almost exactly tripled here:

- 3 X 37=111 MB theoretical min; actual is 110 MB
- 3 X 77=231 MB theoretical max; actual is 230 MB

That's perfect scaling, exactly what you'd hope to see when adding more disks to a RAID array. This clearly wasn't doing the right thing when only looking at average performance, probably because the files created were not on exactly the same portion of the disk to get a fair comparison. The reason why ZCAV issues have been highlighted so many times in this chapter is because they pop up so often when you attempt to do fair comparison benchmarks of disks.

Disk-performance expectations

So, what are reasonable expectations for how your disks should perform? The final example shown in the preceding section demonstrates how things should work. Any good drive nowadays should have sequential transfers of well over 50 MBps on its fastest area, with 100 MBps being easy to find. The slowest part of the drive will be closer to half that speed. It's good practice to try and test an individual drive before building more complicated arrays using them. If a single drive is slow, you can be sure an array of them will be bad, too.

The tricky part of estimating how fast your system should be is when you put multiple drives into an array.

For multiple disks into a RAID1 array, the sequential read and write speed will not increase. However, a good controller or software RAID implementation will use both drives at once for seeking purposes, which might as much as double measurements of that rate.

When multiple drives are added to a RAID0 array, you should get something close to linear scaling of the speeds, as shown in the preceding section. Two 50-MBps drives in RAID0 should be at least close to 100 MBps. It won't be perfect in most cases, but it should be considerably faster than a single drive.

Combinations like RAID10 should scale up sequential reads and writes based on the number of drive pairs in RAID0 form, while also getting some seek improvement from the RAID1 mirroring. This combination is one reason it's preferred for so many database disk layouts.

If you're using RAID5 instead, which isn't recommended for most databases, read speed will scale with the number of drives you use, while write speed won't increase.

Sources of slow disk and array performance

Most of the time, if you meet expectations for sequential read and write speeds, your disk subsystem is doing well. You can measure seek time, but there's little you can do to alter it besides add more disks. It's more a function of the underlying individual physical drives than something you can do anything about. Most problems you'll run into with slow disks will show up as slow read or write speeds.

Poor-quality drivers for your controller can be a major source of slow performance. Usually, you'll need to connect the same drives to another controller to figure out when this is the case. For example, if you have an SATA drive that's really slow when connected to a RAID controller, but the same drive is fast when connected directly to the motherboard, bad drivers are a prime suspect.

One problem that can significantly slow down read performance in particular is not using sufficient read-ahead for your drives. This normally manifests itself as writes being faster than reads, because the drive ends up idle too often while waiting for the next read request to come in. This subject is discussed more in `Chapter 4`, *Disk Setup*.

Conversely, if writes are very slow relative to reads, check the write caching policy and size on your controller if you have one. Some will prefer to allocate their cache for reading instead of writing, which is normally the wrong decision for a database system. Reads should be cached by your operating system and the database; it's rare they will ask the controller for the same block more than once, whereas it's quite likely your OS will overflow the write cache on a controller by writing heavily to it.

RAID controller hardware itself can also be a bottleneck. This is most often the case when you have a large number of drives connected, with the threshold for what large means dependent on the speed of the controller. Normally, to sort this out, you'll have to reduce the size of the array temporarily, and see if the speed drops. If it's the same even with a smaller number of drives, you may be running into a controller bottleneck.

The connection between your disks and the rest of the system can easily become a bottleneck. This is most common with external storage arrays. While it might sound good that you have a gigabit link to a networked array over Ethernet, a fairly common NAS configuration, if you do the math, it's at most 125 MBps--barely enough to keep up with two drives, and possible to exceed with just one. No way will that be enough for a large storage array. Even Fiber Channel arrays can run into their speed limits and become the bottleneck for high sequential read speeds if you put enough disks into them. Make sure you do a sanity check on how fast your drive interconnect is relative to the speed you're seeing.

It's also possible for a disk bottleneck to actually be somewhere on the CPU or memory side of things. Sometimes disk controllers can use quite a bit of the total system bus bandwidth. This isn't as much of a problem with modern PCI Express controller cards, which use the higher transfer rates available, but you do need to make sure the card is placed in a slot and configured so it's taking advantage of those. Monitoring overall system performance while running the test can help note when this sort of problem is happening; it will sometimes manifest as an excessive amount of CPU time being used during the disk testing.

Poor or excessive mapping of your physical disks to how the operating system sees them can also slow down results by more than you might expect. For example, passing through Linux's LVM layer can cause a 10-20% speed loss, compared to just using a simple partition instead. Other logical volume abstractions, either in hardware or software, can drop your performance, too.

One performance aspect that is overrated by storage vendors in particular is aligning filesystem blocks with those of blocks on physical storage, or with the **stripes** of some RAID array configurations. While theoretically such a misalignment can turn a single physical block write into two when blocks straddle a stripe boundary, you'll really need to have quite a system before this is going to turn into your biggest problem. For example, when doing random database writes, once the disk has done a seek to somewhere, it makes little difference whether it writes one or two blocks once it arrives. By far the biggest overhead was the travel to the write location, not the time spent writing once it got there. And since it's easy to have your bottlenecks actually show up at the block level in the OS or controller, where strip splits aren't even noticed, that makes this problem even less likely to run into. Alignment is something worth investigating if you're trying to get good performance out of RAID5, which is particularly sensitive to this problem. But for most systems using the better performing RAID levels, trying to tune here is more trouble than it's worth. Don't be surprised if a storage vendor, particular one defending an underperforming SAN, tries to blame the performance issues on this area, though. You'll likely have to humor them by doing the alignment just to rule that out, but don't expect that to change your database performance very much.

The length of this list should give you an idea why doing your own testing is so important. It should strike you that there are a whole lot of points where a disk configuration can go wrong in a way that slows performance down. Even the most competent vendor or system administrator can easily make a mistake in any one of these spots that cripples your system's disk speed and, correspondingly, how fast the database running it will get work done. Would you believe that even excessive vibration is enough to considerably slow down a drive nowadays? It's true!

Summary

Doing your own hardware benchmarking serves two complementary purposes. Knowing how fast your current systems are relative to one another, and being able to evaluate a candidate new server via the same measurements, is extremely valuable for helping nail down where the bottlenecks in your hardware are.

Second, the difference between reality and your hardware vendor's claims or reputation can be quite large. It is not safe to assume that your system is fast because you bought it from a reputable vendor. You should not assume a SAN is properly configured when delivered simply because it's a very expensive item and you were told it's already optimized for you. Systems are complicated, odd hardware interactions are inevitable, and not everyone involved in sales is going to be completely honest with you.

At the same time, you don't need to be a benchmarking expert to do useful hardware-validation tests. It's actually counterproductive to run really complicated tests. If they don't give expected results, it will be hard to get your vendor to acknowledge the result and replicate the issue where it can be resolved. It's better to stick with the simplest possible, industry-standard tests for benchmarking, rather than attempt to do really complicated ones. If it takes a complicated test requiring hours of custom application setup to show a performance problem, the odds that your vendor is going to help resolve the issue are low. Much more likely, your application will be blamed. If, on the other hand, you can easily replicate the problem using the UNIX standard `dd` command in a few minutes, it's difficult to refute that the lowest levels of hardware/software are to blame. Finally, doing some heavy benchmarking work when a new server arrives will do one additional thing: put some early stress on the hardware while it's still new. It's always better to deploy a system that's already gotten a workout to prove itself.

Always run your own basic hardware benchmarks on any system you intend to put a database on. Simpler tests your vendor can replicate if you run into a problem are better than complicated ones. memtest86+, STREAM, sysbench, hdtuune, dd, bonnie++, and sysbench are all useful tools for measuring various aspects of system performance. Disk drive testing needs to be very sensitive to how disk speed changes over the surface of the drive. IOPS is a common way to measure disk and disk array performance, but it's not very well matched to the requirements of database applications. Speeds on a seek-heavy workload can be much slower than you might expect based on a disk's sequential read/write performance. Commit rate needs to be measured to confirm the caching levels you believe are active really are, since that impacts database reliability. Complicated tests are better done using benchmarks of real database applications, rather than focusing on synthetic disk tests.

After selecting suitable hardware and benchmarking the hardware, the next task is to select the operating system and the file system for the database. The scope of the next chapter is to discuss the different kinds of file systems as well as the pros and cons of that file system. Disk setup and configuration is also part of the next chapter.

4
Disk Setup

Most operating systems include multiple options for the filesystem to store information onto disk. Choosing between these options can be difficult, because it normally involves some tricky speed versus reliability trade-offs. Similarly, how to set up your database and spread its components across many available disks also has trade-offs, with speed, reliability, and available disk space all linked. PostgreSQL has some facilities to split its database information over multiple disks, but the optimal way to do that is very much application dependent.

In this chapter we will cover the following topics:

- Maximum filesystem sizes
- Filesystem crash recovery
- Linux filesystems
- Solaris and FreeBSD filesystems
- Disk layout for PostgreSQL

Maximum filesystem sizes

One of the topics discussed for each filesystem is how large of a volume you can put on it. For most of them, that number is 16 TB, a shared artifact of using 32-bit numbers to represent filesystem information. Right now, it's quite easy to exceed 16 TB in a volume created with a moderately sized array of terabyte or larger hard drives. That makes this number an increasingly problematic limit.

There are three levels of issues you can run into here:

- The data structures of the filesystem itself don't support large volumes
- Tools used to create and manipulate the filesystem do not handle large sizes
- The disk-partitioning scheme needed to boot the operating system doesn't handle large volumes

The last of those is worth spending a moment on, since that problem is mostly independent of the filesystem specific details of the first two.

Most PC hardware on the market, with the notable exception of Apple Intel Mac OS X systems, partition drives using the **Master Boot Record** (**MBR**) partitioning scheme (`https://en.wikipedia.org/wiki/Apple_Partition_Map`).

This only supports partition sizes up to 2 TB in size (`https://en.wikipedia.org/wiki/Master_boot_record`). Creating a larger partition that you can boot from will require a different partitioning scheme. One possibility is the **GUID Partition Table** (**GPT**) scheme, promoted by Intel as part of their **Extensible Firmware Interface** (**EFI**) standard intended to replace the old PC BIOS booting methods. Apple is so far, the main early adopter of EFI and GPT. See `http://en.wikipedia.org/wiki/GUID_Partition_Table` for more information about GPT partitions, their support in various operating systems, and the backward compatibility possibilities available.

This area is not very well explored or supported yet. Proceed with caution if you expect to need greater than 2 TB volumes on PC hardware, even if your chosen filesystem obviously supports it. At a minimum, expect that you may need to have a smaller operating system disk that you boot from, only using GPT partitioning on the data disks mounted after the OS has started.

Filesystem crash recovery

Filesystem writes have two major components to them. At the bottom level, you are writing out blocks of data to the disk. In addition, there is an amount of filesystem metadata involved too. Examples of metadata include the directory tree, the list of blocks and attributes associated with each file, and the list of what blocks on disk are free.

Like many disk-oriented activities, filesystems have a very clear performance versus reliability trade-off they need to make. The usual reliability concern is what happens in situations where you're writing changes to a file and the power goes out in the middle of the process.

Consider the case where you're writing out a new block to a file, one that makes the file bigger (rather than overwriting an existing block). You might do that in the following order:

1. Write data block.
2. Write file metadata referencing use of that block.
3. Add data block to the list of used space metadata.

What happens if power goes out between Step 2 and Step 3 here? You now have a block that is used for something, but the filesystem believes it's still free. The next process that allocates a block for something is going to get that block, and now two files would refer to it. That's an example of a bad order of operations which no sensible filesystem design would use. Instead, a good filesystem design would:

1. Add data block to the list of used space metadata.
2. Write data block.
3. Write file metadata referencing use of that block.

If there was a crash between Step 1 and Step 2 here, it's possible to identify the blocks that were marked as used, but not actually written to usefully yet. Simple filesystem designs do that by iterating over the entire disk blocks allocated, reconciling the list of blocks that should be used or are free against what's actually used. Examples of this include the `fsck` program used to validate simple UNIX filesystems and the `chkdsk` program used on FAT32 and NTFS volumes under Windows.

Journaling filesystems

The more modern approach is to use what's called a journal to improve this situation (`https://en.wikipedia.org/wiki/Journaling_file_system`). A fully journaled write would look like this:

1. Write transaction start metadata to the journal.
2. Write used space metadata change to the journal.
3. Write data block change to the journal.
4. Write file metadata change to the journal.
5. Add data block to the list of used space metadata.
6. Write data block.
7. Write file metadata referencing use of that block.
8. Write transaction end metadata to the journal.

What this gets you is the ability to recover from any sort of crash the filesystem might encounter. If you didn't reach the final step here for a given write transaction, the filesystem could just either ignore (data block write) or undo (metadata write) any partially completed work that's part of that transaction. This lets you avoid long filesystem consistency checks after a crash, because you'll just need to replay any open journal entries to fix all the filesystem metadata. The time needed to do that is proportional to the size of the journal, rather than the old filesystem checking routines whose runtime is proportional to the size of the volume.

The first thing that should jump out at you here is that you're writing everything twice, plus the additional transaction metadata, and therefore more than doubling total writes per update in this situation.

The second thing to note is more subtle. Journaling in the filesystem is nearly identical to how write-ahead logging works to protect database writes in PostgreSQL. So if you're using journaling for the database, you're paying this overhead four times. Writes to the WAL, itself a journal, are journaled, and then writes to the disk are journaled, too.

Since the overhead of full journaling is so high, few filesystems use it. Instead, the common situation is that only metadata writes are journaled, not the data block changes. This meshes well with PostgreSQL, where the database protects against data block issues but not filesystem metadata issues.

Linux filesystems

Linux is a particularly good example to start with for discussing filesystem implementation trade-offs, because it provides all of the common options among its many available filesystems.

ext2

The **second extended filesystem** (**ext2**) is the filesystem for the Linux kernel. ext2 is the oldest Linux filesystem still viable for use now. ext2 does not have any journaling available (`https://en.wikipedia.org/wiki/Ext2`). Therefore, any system that uses it is vulnerable to long `fsck` recovery times after a crash, which makes it unsuitable for many purposes. You should not put a database volume on ext2. While that might theoretically work, there are many known situations, such as any user errors made during the quite complicated `fsck` process, that can break the write ordering guarantees expected by the database.

Because not journaling any writes is faster, ext2 volumes are used sometimes for PostgreSQL WAL volumes, which require very little in the way of data guarantees. Also, WAL volumes tend to be small filesystems without many files on them, and thus fairly quick to run `fsck` on. However, this does introduce the likely possibility of a failing automatic `fsck` check on reboot after a crash. That requires user intervention, and therefore will make the server fail to start fully.

Rather than presume you need to start with ext2, a sensible approach is to start with standard ext3. Switch to writeback ext3 if the WAL disk is not keeping up with its load, and only if that too continues to lag behind, consider dropping to ext2. Since the WAL writes are sequential, while ones to the database are often random, it's much harder than you might think to have the WAL be a bottleneck on your system, presuming you've first put it into its own disk(s). Therefore, using ext2 without proving it is necessary, falls into the category of premature optimization--there is a downside and you should only expose yourself to it when needed. Only if the WAL is a measured bottleneck should you consider the faster but riskier ext2.

ext3

The **third extended file system** (**ext3**) is a default filesystem for many Linux distributions (`https://en.wikipedia.org/wiki/Ext3`). The ext3 filesystem adds a journal kept in a regular file on top of the ext2 filesystem. If the journal is empty (which will be the case on a clean server shutdown), you can even open an ext3 filesystem in ext2 mode. It's backward compatible with ext2, and it's possible to convert a volume in either direction: ext2 to ext3 or ext3 to ext2.

There are three levels of journaling available in ext3, specified as options when mounting the filesystem:

- `data=writeback`: Data changes are not journaled at all. Metadata changes are journaled, but the order in which they are written relative to the data blocks is not guaranteed. After a crash, files can have extra junk at their end from partially completed writes, and you might have a mix of old and new file data.

- `data=ordered`: Metadata is journaled, but data changes are not. However, in all cases, the metadata writes only occur after the associated data has already been written--thus the `ordered` name. After a crash, it is possible to get a mix of old and new data in the file, as in the writeback mode, but you'll never end up with a file of incorrect length. The main differences in behavior here, compared to fully journaled writes, are when you're changing blocks already on disk, when there is a data change but no associated metadata change. In cases where the file is new or the block being written is at the end, expanding the size of the file, the behavior of ordered mode is functionally equivalent to journal.
- `data=journal`: Both file data and filesystem metadata are written to the journal before the main filesystem is touched.

Choosing between these three isn't as easy as it might appear at first. You might think that writeback mode is the fastest mode here in all cases. That's true, but on systems with a functional write cache, in particular, the difference between it and ordered, is likely to be quite tiny. There is little reason to prefer writeback from a performance perspective when your underlying disk hardware is good, which makes the exposure to risk even less acceptable. Amusingly, the usual situation in which writeback mode is said to be safe for PostgreSQL use, is one where the underlying writes involve a battery-backed write cache, but that's exactly the situation under which the penalty of ordered mode is the least expensive.

In short, there are very few reasons to ever use writeback for your main database. The main weakness of writeback mode--that files can sometimes be extended with garbage bytes-is not an issue for the WAL files. Those are both extended to their full size before use and written to with checksumming that rejects incorrect data. Accordingly, writeback is an appropriate mode to consider for a filesystem that only holds WAL data. When pressed for speed improvements on WAL writes, writeback is preferred to using ext2 from an integrity point of view, because the minimal metadata journaling you will get in this mode will prevent long `fsck` recovery times after a crash.

For the database disks, the choices are ordered or journal. On simple benchmarks such as bonnie++ (more on this in the *XFS* section), ordered will sometimes lose as much as half its throughput (relative to raw disk speed) because of the double writing that the journal introduces. However, that does not mean the journal will be half as fast for your database writes! Like the WAL, the journal is written out to a contiguous portion of disk, making it mostly sequential writes, and the subsequent updates to the data blocks are then written later. It turns out that this behavior makes journal model particular well suited to the situation where there are concurrent reads mixed with writes--which is exactly the case with many database applications.

In short, journal should not be immediately discarded as an option just because it delivers poor results in synthetic workloads. If you can assemble a reasonable simulation of your application running, including realistic multi-user behavior, it's worth trying the journal mode in addition to ordered. You shouldn't start there though, because in simpler workloads, ordered will be faster, and the additional integrity provided by journal the mode is completely optional for a PostgreSQL database.

Note that, switching to a different mode for your root filesystem isn't as easy as changing the mount options in the `/etc/fstab` file. Instead, you'll need to edit the bootloader (typically grub) configuration file and add the change as a kernel option like this `rootflags=data=journal` because the root drive is mounted before the `fstab` file is even consulted.

One of the limitations of ext3 that is increasingly relevant with today's hard drive sizes is that on common Intel and AMD processors, an ext3 filesystem can only be 16 TB in size, and individual files are limited to 2 TB.

ext4

The evolutionary replacement for ext3, ext4 was announced as production quality as of Linux kernel 2.6.28. A variety of fixes involving delayed allocation were applied between that and version 2.6.30, but more importantly for PostgreSQL, some bugs involving `fsync` handling were not fully corrected until kernel 2.6.31.8/2.6.32.1. Kernel 2.6.32 is the first version that includes an ext4 version that should be considered for a production PostgreSQL database. That is the version that both RHEL 6 and Ubuntu 10.04 are using, in the first long-term release from each that includes ext4 support.

The 16 TB filesystem limit of ext3 theoretically does not exist for ext4, but as this is being written, the associated `mkfs` utility is still stuck at that limit. The *Ext4 Howto* article at `https://ext4.wiki.kernel.org/index.php/Ext4_Howto` is the definitive source for updates about progress in removing that limitation.

From the perspective of PostgreSQL, the main improvement of ext4 over ext3 is its better handling of write barriers and `fsync` operations. See the following section about barriers for more information.

XFS

Unlike ext3, XFS was designed by SGI for efficiently journaling from the start, rather than having journaling added on to an existing filesystem. As you might predict, the result is a bit faster than ext3, some of which is just because of better efficiency in the journal implementation. However, some of this speed results from the fact that XFS only journals metadata, and it doesn't even have an option to try and order the data vs. metadata writes. Accordingly, XFS is most like ext3's writeback mode. One critical difference is that in situations where garbage blocks may have been written to a file, the main concern with ext3 writeback, the journal playback in XFS will instead zero out these entries. Then they are unlikely to be interpreted as real data by an application. This is sufficient to keep PostgreSQL from being confused if it tries to read them, as a zeroed block won't have the right header to look like either a WAL block or a database data block. Some consider this a major flaw of XFS, that it took what could have been a perfectly good copy of a block and erased it. The way PostgreSQL does its writes, such damage will be sorted out by the WAL. Just make sure you have the `full_page_writes` configuration parameter turned on.

How much faster is XFS? Running a simple test comparing ext3 in two of its modes vs. XFS, it does quite a bit better. The following table shows some simple bonnie++ 1.96 results, which are not indicative of the performance difference you'll see on general database applications. The results show the range appearing in three runs against a single SATA drive:

Filesystem	Sequential write (MB/s)	Sequential read (MB/s)
ext3 data=ordered	39-58	44-72
ext3 data=journal	25-30	49-67
XFS	68-72	72-77

It's clear that raw XFS performance has less variation than these two ext3 modes, and that its write performance is much faster. The writeback mode of ext3 wasn't included because it's not safe, due to its tendency to add non-zero garbage blocks to your files after a crash.

On systems with larger disk arrays, the delayed allocation feature of XFS, and its associated I/O scheduling, aim to provide larger writes that are likely better aligned with RAID stripe sizes. Combine delayed allocation with a generally more efficient design, and XFS's performance advantage over ext3 can really look compelling on huge volumes. To further improve performance on larger systems, there are several options available for adjusting XFS memory use for things such as the in-memory size of the metadata journal, as well as what its target pre-allocation size should be.

Historically, XFS has not been popular among Linux users or distributors. However, XFS easily handles files of over a million terabytes, making it the primary Linux solution currently available for files greater than 16 TB. This has resulted in XFS having a comeback of sorts in recent enterprise-oriented Linux distributions, where administrators are likely to run into that file limit. Recently, RedHat's RHEL 5.4 release added preliminary XFS support specifically to address that need, and their RHEL 6 release treats it as a fully supported filesystem on par with ext3 and ext4.

Since it defaults to using write barriers (described in more detail below), XFS is also paranoid about drives with volatile write caches losing writes. To prevent that, it is aggressive in sending drive cache flush commands to the underlying disks. This is what you want from a PostgreSQL database running on XFS using regular hard drives that have their write cache turned on. However, if you have a non-volatile write cache, such as a battery-backed write controller, this cache flushing is wasteful. In that case, it's recommended to use the `nobarrier` option when mounting the XFS filesystem in order to disable its cache flushing.

Other Linux filesystems

There are a few other filesystem options for Linux that are not well enough explored for use with PostgreSQL:

- **JFS**: Performs similarly to XFS but with less CPU usage. However, it is considered less stable than the recommended choices here. It's also not as well supported by mainstream Linux distributors. Is it because it is not as well supported by mainstream Linux distributors that it gets less testing as a result, or are there fewer users because of the stability issues? Regardless, JFS was never very popular, and it seems to be losing ground now.
- **ReiserFS**: Being the first journaling filesystem integrated into the Linux kernel, for some time ReiserFS was the preferred filesystem for major Linux distribution, SuSE. Since SuSE abandoned it in late 2006 for ext3, ReiserFS adoption has been shrinking steadily since. At this moment, the current ReiserFS v3 is considered stable, but its replacement, ReiserFS v4, has yet to be merged with the mainline Linux kernel. The uncertain future of the filesystem limits interest in it considerably. This is unfortunate given that the transparent compression feature of ReiserFS v4 would be particularly valuable for some database workloads, such as data warehouses where large sequential scans are common.

- **Btrfs**: This Oracle-sponsored filesystem is considered the future of Linux filesystems even by the primary author of ext4. At this point, code development hasn't reached a stable release. At a future time when that does happen, this filesystem has some unique features that will make it a compelling option to consider for database use, such as its support for easy snapshots and checksums.

While each of these filesystems has some recognized use cases where they perform well compared to the more mainstream choices, none of these are considered compelling for database use in general, nor in PostgreSQL, due to maturity issues as this is being written. Btrfs in particular, may change in that regard as unlike the other two, it has a healthy development community working on it still.

Write barriers

The `Chapter 2`, *Database Hardware*, already mentioned that most hard drives have volatile write caches in them, and that the WAL writing scheme used by the database isn't compatible with those. The important requirement here is that when a file sync operation (`fsync` on UNIX) occurs, the filesystem must make sure all related data is written out to a non-volatile cache or disk itself. Filesystem journal metadata writes have a similar requirement. The writes for the metadata updates require that the journal updates are first written out to preserve proper ordering.

To support both these situations, where something needs to be unconditionally written to disk and where an ordering is required, the Linux kernel implements what they call write barriers. To quote from the kernel documentation on barriers:

> *"All requests queued before a barrier request must be finished (made it to the physical medium) before the barrier request is started, and all requests queued after the barrier request must be started only after the barrier request is finished (again, made it to the physical medium)."*

This sounds comforting, because this requirement that data be on the physical medium matches what the database expects. If each database sync request turns into a write barrier, that's an acceptable way to implement what the database requires--presuming that write barriers work as advertised.

Drive support for barriers

In order to support barriers, the underlying disk device must support flushing its cache, and preferably a write-through mode as well.

SCSI/SAS drives allow writes (and reads) to specify **Force Unit Access** (**FUA**), which directly accesses the disk media without using the cache-what's commonly called write--through. They also support a `SYNCHRONIZE CACHE` call that flushes the entire write cache out to disk.

Early IDE drives implemented a `FLUSH CACHE` call and were limited to 137 GB in size. The ATA-6 specification added support for larger drives at the same as time it introduced the now mandatory `FLUSH CACHE EXT` call. That's the command you send to a drive that does what filesystems (and the database) currently need for write cache flushing. Any SATA drive on the market now will handle this call just fine; some IDEs and the occasional rare early SATA drives available many years ago, did not. Today, if you tell a drive to flush its cache out, you can expect it to do so reliably.

SATA drives that support Native Command Queuing can also handle FUA. Note that, support for NCQ in Linux was added as part of the switch to the libata driver in kernel 2.6.19, but some distributions (such as RedHat) have backported this change to their version of the earlier kernels they ship. You can tell if you're using libata, either by noting that your SATA drives are named starting with sda, or by running:

```
$ dmesg | grep libata
```

The exact system calls used will differ a bit, but the effective behavior is that any modern drive should support the cache flushing commands needed for barriers to work. Linux tests the drives out to confirm this is the case before letting you enable barriers, so if they're on, they are expected to work.

Filesystem support for barriers

ext3 theoretically supports barriers when used with simple volumes. In practice, and for database purposes, they are not functional enough to help. The problem is that `fsync` calls are not correctly translated into write barrier in a form that will always flush the drive's cache. It's just not something built into ext3 in the right form. And if you are using Linux software RAID or the **Logical Volume Manager** (**LVM**) with ext3, barriers will not be available at all anyway.

What actually happens on ext3 when you execute `fsync` is that all buffered data on that filesystem gets flushed out in order to satisfy that call. You read that right--all cached data goes out every time a `fsync` occurs. This is another reason why putting the WAL, which is constantly receiving `fsync` calls, onto its own volume is so valuable with ext3.

XFS does handle write barriers correctly, and they're turned on by default. When you execute a `fsync` call against a file on an XFS volume, it will just write out the data needed for that one file, rather than the excessive cache flushing that ext3 does. In fact, one of the reasons this filesystem has a bad reputation in some circles relates to how early it introduced this feature. It enabled barriers, before either the drives were available or the underlying device drivers did the right thing, to flush data out, which meant the barriers it relied upon for data integrity were not themselves reliable. This resulted in reports of XFS causing corruption, caused by bugs elsewhere in the software or hardware chain, but XFS was blamed for them.

ext4 also supports barriers, and they're turned on by default. The `fsync` calls are implemented properly as well. The end result is that you should be able to use ext4, leave drive caching on, and expect that database writes will be written out safely anyway. Amusingly, it was obvious that ext4 was finally doing the right thing in 2.6.32 because of the massive performance drop in PostgreSQL benchmarks. Suddenly, `pgbench` tests only committed a number of transactions per second that corresponded to the rotation speed of the underlying disks, rather than showing an inflated result from unsafe behavior.

General Linux filesystem tuning

Regardless of what filesystem you choose, there are some general Linux tuning operations that apply.

Read-ahead

The first parameter you should tune on any Linux install is the device read-ahead. When doing sequential reads that seem to be moving forward, this feature results in Linux asking for blocks from the disk ahead of the application requesting them. This is key to reaching full read performance from today's faster drives. The usual symptom of insufficient read-ahead is noting that write speed to a disk is faster than its read speed. The impact is not subtle; proper read-ahead can easily result in 100% or larger increase in sequential read performance. It is the most effective filesystem adjustment to make on Linux if you want to see benchmarks like the bonnie++ read speed jump upwards. That corresponds to a big increase in large sequential I/O operations in PostgreSQL, too, including sequential scans of tables and bulk operations like `COPY` imports.

You can check your current read-ahead using the `blockdev` command:

```
$ blockdev --getra /dev/sda
```

The default is 256 for regular drives, and may be larger for software RAID devices. The units here are normally 512 bytes, making the default value equal to 128 KB of read-ahead. The normal properly tuned range on current hardware usually works out to be 4096 to 16384, making this change:

```
$ blockdev --setra 4096 /dev/sda
```

A reasonable starting point. If you run bonnie++ with a few read-ahead values, you should see the sequential read numbers increase as you tune upwards, eventually leveling off. Unfortunately, read-ahead needs to be set for each drive on your system. It's usually handled by putting a `blockdev` adjustment for each device in the `rc.local` boot script.

The Linux read-ahead implementation was developed with PostgreSQL as an initial target, and it's unlikely you will discover increased read-ahead detuning smaller reads as you might fear. The implementation is a bit smarter than that.

File access times

Each time you access a file in Linux, a file attribute called the file's last access time (`atime`) is updated. This overhead turns into a steady stream of writes when you're reading data, which is an unwelcome overhead when working with a database. You can disable this behaviour by adding `noatime` to the volume mount options in `/etc/fstab`, as in this example:

```
/dev/sda1 / ext3 noatime,errors=remount-ro 0 1
```

There are two additional levels of access time updates available in some Linux kernels: `nodiratime` and `relatime`, both of which turn off a subset of the `atime` updates. Both of these are redundant if you use the preferred `noatime`, which disables them all.

Read caching and swapping

Linux will try to use any extra RAM for caching the filesystem. And that's what PostgreSQL would like it to do. When the system runs low on RAM, the kernel has a decision to make. Rather than reduce the size of its buffer cache, instead, the OS might swap inactive disk pages out. How often to consider this behavior is controlled by a tuneable named swappiness. You can check the current value on your system (probably 60) by looking at `/proc/sys/vm/swappiness` and the easiest way to make a permanent adjustment is to add a line to `/etc/sysctl.conf` like this:

```
vm.swappiness=0
```

A value of `0` prefers shrinking the filesystem cache rather than using swap, which is the recommended behavior for getting predictable database performance. You might notice a small initial decrease in performance at high memory usages though. The things that tend to be swapped out first are parts of the operating system that are never used, and therefore never missed. So evicting them for buffer cache space is the right move in that specific case. It's when you run so low on memory that more things start to get swapped out that the problems show up. As is often the case, optimizing for more predictable behavior (avoiding swap) might actually drop performance for a some cases (items swapped weren't necessary). Tuning for how to act when the system runs out of memory is not an easy process.

Related to this parameter is Linux's tendency to let processes allocate more RAM than the system has, in the hopes that not all of it will actually be used. This Linux overcommit behaviour should be disabled on a PostgreSQL server by making this change to the `sysctl` configuration:

```
vm.overcommit_memory=2
```

Both of these changes should be considered as part of setting up any Linux PostgreSQL server. A good practice here is to bundle them in with increasing the shared memory parameters to support larger values of `shared_buffers`, which requires editing the same `sysctl` file.

Write cache sizing

On the write side of things, Linux handles writes to disk using a daemon named `pdflush`. It will spawn a number of `pdflush` processes (between two and eight) to keep up with the amount of outstanding I/O. `pdflush` is not very aggressive about writes, holding to the theory that if you wait longer to write things out, you will optimize total throughput. When you're writing a large dataset, both write combining and being able to sort writes across more data will lower average seeking around while writing.

The main driver for when things in the write cache are aggressively written out to disk are two tuneable kernel parameters:

- `/proc/sys/vm/dirty_background_ratio`: Maximum percentage of active RAM that can be filled with dirty pages before `pdflush` begins to write them.
- `/proc/sys/vm/dirty_ratio`: Maximum percentage of total memory that can be filled with dirty pages before processes are forced to write dirty buffers themselves during their time slice, instead of being allowed to do more writes. Note that all processes are blocked for writes when this happens, not just the one that filled the write buffers. This can cause what is perceived as an unfair behavior where one `write-hog` process can block all I/O on the system.

The default here depends on your kernel version. In early 2.6 kernels, `dirty_background_ratio=10` and `dirty_ratio=40`. This means that a full 10% of RAM can be dirty before `pdflush` really considers it important to work on clearing that backlog. When combined with ext3, where any `fsync` write will force the entire write cache out to disk, this is the recipe for a latency disaster on systems with large amounts of RAM. You can monitor just exactly how much memory is queued in this fashion by looking at `/proc/meminfo` and noting how large the value listed for `dirty` gets.

Recognizing these defaults were particularly bad, in Linux kernel 2.6.22 both values were lowered considerably. You can tune an older kernel to use the new defaults like this:

```
echo 10 > /proc/sys/vm/dirty_ratio
echo 5 > /proc/sys/vm/dirty_background_ratio
```

And that's a common recipe to add to the `/etc/rc.d/rc.local` file on RHEL4/5 server installs in particular. Otherwise, the write stalls that happen when `dirty_ratio` is exceeded, can be brutal for system responsiveness. The effective lower limit here is to set `dirty_ratio` to 2 and `dirty_background_ratio` to 1, which is worth considering on systems with less than 8 GB of RAM. Note that changes here will detune average throughput for applications that expect large amounts of write caching. This trade-off, that maximum throughput only comes with an increase in worst-case latency, is very common.

I/O scheduler elevator

One of the more controversial tunable Linux performance features, in that there's no universally accepted guidelines available, is that of the I/O scheduler choice. The name elevator is used due to how they sort incoming requests. Consider a real elevator, currently at the first floor. Perhaps the first person in requests the ninth floor, then the next person the third. The elevator will not visit the floors in the order requested; it will visit the third floor first, then continue to the ninth. The scheduler elevator does read and write sorting to optimize in a similar fashion.

You can set the default scheduler elevator at kernel boot time. Here's an example from a RedHat Linux system, changing the default elevator to the deadline option:

```
kernel /vmlinuz-2.6.18-128.el5 ro root=/dev/sda1 elevator=deadline
```

The exact location of the file where you have a similar kernel boot line at depends on your Linux distribution and which boot loader you use. There are too many possibilities to list them all here; instructions for your Linux distribution on how to install a custom kernel should point you in the right direction.

As of Linux kernel 2.6.10, you can adjust the scheduler for individual devices without rebooting:

```
$ echo cfq > /sys/block/sda/queue/scheduler
$ cat /sys/block/sda/queue/scheduler
  noop anticipatory deadline [cfq]
```

The four elevator choices are as follows:

- `elevator=cfq`: Completely Fair Queuing tries to divide available I/O bandwidth equally among all requests. This is the default for most Linux installations.
- `elevator=deadline`: Deadline aims to provide something like real-time behavior, where requests are balanced so no single process `starves` due to having to wait too long for a read or write.

- `elevator=noop`: The no operation scheduler doesn't do any complicated scheduling, it handles basic block merging and sorting before passing the request along to the underlying device.
- `elevator=as`: Anticipatory scheduling intentionally delays I/O requests in the hope of bundling more of them together in a batch.

People seem drawn to this area, hoping that it will have a real impact on the performance of their system, based on the descriptions. The reality is that these are being covered last because this is the least-effective tunable mentioned in this section. Adjusting the I/O scheduler in most cases has a minimal impact on PostgreSQL performance. If you want to improve read performance, adjusting read-ahead is vastly more important. And if you want to tweak write performance, adjusting the dirty cache writeback behavior is the primary thing to consider (after tuning the database to reduce how much dirty data it generates in the first place).

There are a few cases where the I/O scheduler can be effective to tune. On systems with a lot of device read and write cache, such as some RAID controllers and many SANs, any kernel scheduling just gets in the way. The operating system is sorting the data it knows about, but not considering what data is already sitting in the controller or SAN cache. The noop scheduler, which just pushes data quickly toward the hardware, can improve performance if your hardware has its own large cache to worry about.

On desktop systems with little I/O capacity, the anticipatory schedule can be helpful to make the most of the underlying disk(s), by better sorting read and write requests into batches. It's unlikely to be suitable for a typical database server.

The other two options, CFQ and deadline, are impossible to suggest specific use cases for. The reason for this is that the exact behavior depends on both the Linux kernel you're using and the associated workload. There are kernel versions where CFQ has terrible performance, and deadline is clearly better, because of bugs in the CFQ implementation in that version. In other situations, deadline will add latency--exactly the opposite of what people expect--when the system has a high level of concurrency. You're not going to be able to usefully compare them with any simple benchmark. The main differences between CFQ and deadline only show up when there are many concurrent read and write requests fighting for disk time. Which is optimal is completely dependent on that mix.

Anyone who tells you that either CFQ or deadline is always the right choice doesn't know what they're talking about. It's worth trying both when you have a reasonable simulation of your application running, to see if there is a gross difference due to something like a buggy kernel. Try to measure transaction latency, not just average throughput, to maximize your odds of making the correct choice here. One way to measure query latency is to just enable logging query times in the database configuration, perhaps using `log_min_duration_statement`, then analyzing the resulting log files. But if the difference is difficult to measure, don't be surprised. Without a compelling reason to choose otherwise, you should prefer CFQ, as it's the kernel default and therefore much more widely tested.

Solaris and FreeBSD filesystems

While not identical, the common BSD heritage of Solaris and FreeBSD gives their respective filesystems much in common and both implement the same basic ZFS code as their current, most advanced filesystem. Choosing between the older UFS options and ZFS involves the usual performance and reliability trade-offs found in so many other disk related options. In general, ZFS is particularly good at handling very large databases, while UFS can perform better on smaller ones. The feature sets are different enough for this may not be the deciding factor for your installation though.

Solaris UFS

The original **Unix File System** (**UFS**) implementation, also called the Berkley Fast File System or UFS1, originated in BSD UNIX. It later appeared in several commercial UNIX variations, including Solaris. The current Solaris UFS adds two major features not found in the original UFS--support for larger files and filesystems (up to 16 TB) and logging.

The logging here is again similar to the database write-ahead logging and the journaling used in Linux. And like Linux's ext3 journal mode, there are known situations where having logging on turns out to be a performance improvement for UFS, because it turns what would otherwise be random writes into a series of sequential ones to the log. Logging is turned on by default on current Solaris releases; in ones before Solaris 9 04/04 U6, it had to be specifically enabled by adjusting filesystem options in `/etc/vfstab` to include it:

```
/dev/dsk/c0t0d0s1 /dev/rdsk/c0t0d0s1 / ufs 1 yes logging
```

In current versions where it's on by default, it can be disabled (which is not recommended) by using the `no_logging` mount option.

UFS is not tuned very well out-of-the-box for PostgreSQL use. One major issue is that it only tries to cache small files, which means it probably won't do what the database expects on the large database ones. The maximum amount of RAM used for caching is tiny--12% of SPARC systems and only 64 MB on Intel/AMD x64 systems. Reads and writes are not done in particularly large blocks either, which can be a problem for some workloads.

The last of those is the most straightforward to fix. When executing physical I/O, requests are broken up into blocks no larger than the `maxphys` parameter. This also serves as a limiter on read-ahead, and is therefore quite important to adjust upward. Normal practice is to adjust this to match the maximum allowed by the typical device drivers, 1 MB. The `klustsize` parameter works similarly for reads and writes to swap:

```
$ set maxphys=1048576
$ set klustsize=1048576
```

The code that stops UFS from caching larger files in memory is named `freebehind`. For PostgreSQL, you want this turned off altogether by setting it to 0, rather than trying to tune its threshold. The total size of the UFS filesystem cache, called the segment map, is set differently based on whether you're using a SPARC or x64 Solaris version.

On SPARC Solaris systems:

```
$ set freebehind=0
$ set segmap_percent=60
```

This size (60% of total RAM) presumes you're setting `shared_buffers` to 25% of total RAM and have a dedicated database server, so 85% of RAM is available for effective database cache. You might want a smaller value on some systems.

On Intel/AMD x64 Solaris systems:

```
$ set ufs:freebehind=0
$ set segmapsize=1073741824
```

This example sets the cache size to 1 GB. You'll need to adjust this fixed value on each server based on the amount of RAM in your server, rather than being able to use a percentage.

Solaris systems allow turning off access time tracking using `noatime` as a mounting option, similarly to Linux. This is a useful small optimization to enable.

A final Solaris specific option to consider with UFS relates to WAL writes. Since the WAL isn't read from, except after a crash, any RAM used to cache it is wasted. The normal way to bypass that is to use UNIX direct I/O, which doesn't go through the filesystem cache. But that isn't fully implemented in PostgreSQL on Solaris. If you separate out the `pg_xlog` directory onto its own filesystem, you can mount that using the `forcedirectio` option to get optimal behavior here. That will quietly convert all the WAL writes to direct I/O that bypasses the OS cache.

FreeBSD UFS2

The FreeBSD community improved basic UFS in its own way, into what it calls UFS2. This also expanded filesystem capacity to far beyond 2 TB, although there are still some user tools that may not support this yet; be sure to investigate that carefully before presuming you can host a large database on FreeBSD.

Rather than implement a journal (Linux) or logging (Solaris), FreeBSD's solution to filesystem integrity issues during a crash and resulting long filesystem recovery times is a technique called soft updates. This orders writes such that the only type of integrity issue after a crash are blocks marked as used but not actually "claimed" by any file. After an unclean shutdown, the filesystem can be brought up almost immediately. What's called a background `fsck` then runs against a static snapshot of the filesystem, searching for unclaimed blocks to clean them up. This removes the overhead of journaling, while avoiding the worst of the traditional non-journaled filesystem issues--long integrity check times that hold up booting after a crash. While not as common as journaling, this technique has been in use on FreeBSD for ten years already without major issues.

Given that PostgreSQL layers its own integrity checks on top of what the filesystem implements, UFS2 certainly meets the requirements of the database. You just need to be aware that the `fsck` activity related to crash recovery will be a background activity competing with database reads and writes, and that might happen during the worst time: just after the server has started, when all of its caches are empty.

Just as on Linux, proper read-ahead significantly improves sequential read speed on FreeBSD. The parameter is sized via the filesystem block size, which defaults to 8 KB. The common useful range for this parameter is 32 to 256. Adjust the value by adding a line to `/etc/sysctl.conf` like this:

```
vfs.read_max = 32
```

To make this active, execute the following command:

```
$ /etc/rc.d/sysctl start
```

The size of the write cache is set by the `vfs.hirunningspace` parameter. The guidelines in the FreeBSD Handbook suggest only increasing this to, at most, a small number of megabytes. You may want to increase this value to something larger if you are using a system with a battery-backed write controller, where the kernel can likely dump much larger amounts of writes onto disk in a batch without significant latency issues.

ZFS

Few filesystems have ever inspired the sort of zealous advocacy that fans of ZFS regularly display. While it has its weak points, in many respects, ZFS is a reinvention of the filesystem concept with significant advantages. One thing that's different about ZFS is that it combines filesystem and RAID capabilities into an integrated pair. The RAID-Z implementation in ZFS is a worthwhile alternative to standard RAID5 and RAID6 installations.

ZFS defaults to working in records of 128 KB in size. This is much larger than a PostgreSQL block, which can cause a variety of inefficiencies if your system is regularly reading or writing only small portions of the database at a time (like many OLTP systems do). It's only really appropriate if you prefer to optimize your system for large operations. The default might be fine if you're running a data warehouse that is constantly scanning large chunks of tables. But standard practice for ZFS database installations that do more scattered random I/O is to reduce the ZFS record size to match the database one, which means 8K for PostgreSQL:

```
$ zfs set recordsize=8K zp1/data
```

You need to do this before creating any of the database files on the drive, because the record size is actually set per file. Note that this size will not be optimal for WAL writes, which may benefit from a larger record size like the default.

One important thing to know about ZFS is that unlike Solaris's UFS, which caches almost nothing by default, ZFS is known to consume just about all the memory available for its **Adaptive Replacement Cache** (**ARC**). You'll need to reduce those amounts for use with PostgreSQL, where large blocks of RAM are expected to be allocated for the database buffer cache and things like working memory. The actual tuning details vary based on Solaris release, and are documented in the *Limiting the ARC Cache* section of the *ZFS Evil Tuning Guide* article at
`http://www.solaris-cookbook.eu/solaris/solaris-10-zfs-evil-tuning-guide/` or `http://www.serverfocus.org/zfs-evil-tuning-guide`.

For FreeBSD, refer to `http://wiki.freebsd.org/ZFSTuningGuide` for similar information. One of the scripts suggested there, `arc_summary.pl`, is a useful one in both its FreeBSD and Solaris incarnations, for determining just what's in the ARC cache and whether it's using its RAM effectively. This is potentially, valuable tuning feedback for PostgreSQL, where the OS cache is used quite heavily, but such use is not tracked for effectiveness by the database.

ZFS handles its journaling using a structure called the intent log. High performance systems with many disks commonly allocate a dedicated storage pool just to hold the ZFS intent log for the database disk, in the same way that the database WAL is commonly put on another drive. There's no need to have a dedicated intent log for the WAL disk too, though.

Similar to XFS, if you have a system with a non-volatile write cache such as a battery-backed write controller, the cache flushing done by ZFS will defeat some of the benefit of that cache. You can disable that behaviour by adjusting the `zfs_nocacheflush` parameter; the following line in `/etc/system` will do that:

```
set zfs:zfs_nocacheflush = 1
```

You can toggle the value to 1 (no cache flushing) and back to 0 (default, flushing enabled) with the following on a live filesystem:

```
echo zfs_nocacheflush/W0t1 | mdb -kw
echo zfs_nocacheflush/W0t0 | mdb -kw
```

ZFS has a few features that make it well suited to database use. All reads and writes include block checksums, which allow ZFS to detect the, sadly, common situation where data is quietly corrupted by RAM or disk errors. Some administrators consider such checksums vital for running a large database safely. Another useful feature is ZFS's robust snapshot support. This makes it far easier to make a copy of a database you can replicate to another location, backup, or even to create a temporary copy you can then rollback to an earlier version. This can be particularly valuable when doing risky migrations or changes you might want to back out.

Because of the robustness of its intent log and block checksum features, ZFS is one filesystem where disabling PostgreSQL's `full_page_writes` parameter is a candidate optimization with little risk. It's quite resistant to the torn pages issue that makes that parameter important for other filesystems. There is also transparent compression available on ZFS. While expensive in terms of CPU, applications that do lots of sequential scans of data too small to be compressed by the PostgreSQL TOAST method, might benefit from reading more logical data per physical read, which is what should happen if compression is enabled.

The choices for Windows are much easier, because there's only one of them that makes sense for a database disk (`https://en.wikipedia.org/wiki/File_Allocation_Table`).

FAT32

While generally deprecated at this point, it's still common for USB devices in particular to be formatted with the older FAT32 filesystem. Since FAT32 is not journaled, unclean shutdown requires running the `chkdsk` utility to recover from. Due to this and the lack of security on FAT32 files, the PostgreSQL installer does not support installing a database cluster directly onto a FAT32 volume. However, it's possible to install onto this filesystem if you manually run the `initdb` command after regular installation. This is not recommended for a reliable database configuration.

NTFS

Microsoft's flagship filesystem is NTFS. This filesystem uses non-ordered metadata journaling, similar to the writeback journal mode on Linux. This is generally reliable, but on rare occasions you might run into a volume that cannot be mounted in Windows without running the old `chkdsk` utility in a special mode as part of booting the system.

As described earlier, PostgreSQL installations on Windows using NTFS should prefer to set the following:

```
open_datasync=fsync_writethrough
```

Set it at the database configuration, which configures the filesystem to flush its WAL writes through any volatile disk write caches.

It's possible to create the functional equivalent of a symbolic link on NTFS called a filesystem junction. The Junction utility available at `https://technet.microsoft.com/en-us/sysinternals`, allows creating those. This lets NTFS volumes on Windows to relocate `pg_xlog` or other parts of the database tree to another filesystem, and to use tablespaces, too.

Theoretically, you can create an NFTS volume of 16 TB using the default cluster size of 4 KB, the same limit as most Linux deployments. It's even straightforward to create a volume with larger clusters though; at 64 KB, NTFS might support 256 TB volumes instead. As noted at the beginning of this chapter, you can partition larger sizes using something like GTP and have Windows recognize the result.

It's possible to turn on automatic compression of a NTFS volume, potentially valuable for the same reasons mentioned in the ZFS section. In practice, systems doing the sort of large queries that benefit from this are not commonly practical on Windows anyway due to its limitations in shared memory and connection capacity.

Adjusting mounting behaviour

Similar to the UNIX `noatime` parameter, it's possible to prevent NTFS from constantly updating when files were last accessed. Another useful tweak is to turn off the 8.3 filename generation Windows does for backward compatibility with older applications. These behaviors are disabled like this:

```
fsutil behavior set disablelastaccess 1
fsutil behavior set disable8dot3 1
```

Be careful when disabling the 8.3 filename generation in particular. One area this has been known to cause problems is if the `%TEMP%` or `%TMP%` environment variables are using the short filenames to access their files. You may need to update these variables to refer to the full pathname instead.

Disk layout for PostgreSQL

Since PostgreSQL uses standard filesystems for all its files, there are several parts of the database you can relocate to somewhere else just by moving the associated files and adding a symbolic link to the new location.

Symbolic links

Symbolic links (also called a **symlink**) are just entries in a filesystem directory that point toward another location. UNIX systems originally preferred to use what's called hard links, which link to the new location quite directly. The entry in the filesystem literally points to another spot on disk. To make this easier to manage, the normal approach now is to use soft symlinks, which are easily visible. The most common thing to relocate using a symlink in PostgreSQL is the WAL transaction log. You can do this after the database cluster is created (but with the server down!) like this:

```
$ cd $PGDATA
$ mv pg_xlog /disk
$ ln -s /disk/pg_xlog pg_xlog
$ ls -l pg_xlog
lrwxrwxrwx 1 postgres postgres 11 2010-04-27 17:35 pg_xlog ->
  /disk/pg_xlog
```

Starting in PostgreSQL 8.3, it's possible to use the `--xlogdir` parameter when running `initdb` to create the cluster. This doesn't work any differently--it will just create the soft symlink for you. The above technique is still quite common, and there's no reason to prefer one over the other besides what seems easier to you.

Tablespaces

The main unit of storage for a PostgreSQL database is the tablespace. Tablespaces are described accurately by their name: they're a space to put tables (and indexes). The idea is that, every logical disk you want to use for a distinct purpose gets assigned a tablespace name, and then when you create a table, you reference that tablespace to put it there:

```
$ mkdir /disk/pgdata
$ psql
postgres=# CREATE TABLESPACE disk LOCATION '/disk/pgdata';
postgres=# CREATE TABLE t(i int) TABLESPACE disk;
```

Tablespaces are also implemented inside the database using symbolic links, and your OS needs to support them (or an equivalent like the NTFS junction) for this to work. Databases and tables are by default, created in a virtual tablespace named `pg_default`. You can change that by setting the `default_tablespace` parameter in the server configuration. It's also possible to relocate an entire database by setting the `TABLESPACE` parameter when running `CREATE DATABASE`.

Database directory tree

With that background in mind, here are the major directories inside the PostgreSQL database structure that `initdb` creates:

- `base`: This is the location that holds `pg_default`, the default tablespace. The template databases and any additional ones created without an explicit tablespace assignment will all be placed here. The next chapter digs into the details of how the `base/` directory is organized into databases and tables.
- `global`: Holds the `pg_global` virtual tablespace. This is where system tables shared by all databases in a cluster, such as the database roles and other system catalog data, are stored at.
- `pg_clog`: The transaction commit log data is stored here. This data is actively read from, by `VACUUM` in particular, and files are removed once there's nothing interesting in them. This directory is one of the reasons PostgreSQL doesn't work very well if you mount the database in a way that bypasses the filesystem cache. If reads and writes to the commit logs are not cached by the operating system, it will heavily reduce performance in several common situations.
- `pg_stat_tmp`: This directory contains the files used to store database statistics. These files should never get very big, and if they do, you'll see the database statistics process start consuming more resources.
- `pg_tblspc`: When you create a new tablespace, this directory is where the symbolic link is saved to.
- `pg_xlog`: The database WAL used for crash recovery is stored here.
- `pg_subtrans`: Contains data related to subtransactions.
- `pg_multixact`: Contains multitransaction status data. In situations where your application heavily uses shared row locks, use of this direction could be heavy.
- `pg_twophase`: Holds data related to two-phase commits.

Generally, the first thing people relocate onto its own disk is `pg_xlog`. Then they add more tablespaces to split out heavily accessed tables. Next up is to move temporary files. The directories containing transaction details are much less likely candidates to be split to their own disks, but applications where this has improved performance have been observed by PostgreSQL users.

Temporary files

There are a few situations where PostgreSQL saves temporary files, ones that are not critical to database operation. Tables created using `CREATE TEMPORARY TABLE` and their respective indexes are one source. Probably more importantly, when the database is doing a query that involves a sort operation, and the data exceeds `work_mem`, temporary files are created for that purpose. So in situations where your users will be doing lots of sorting of large tables, like in a data warehouse, there can be quite a bit of activity going to disk for this purpose.

The `temp_tablespaces` database parameter allows relocating all the temporary tables to one chosen from the list of additional tablespaces provided. If you put more than one tablespace on that list, which is used is selected at random when a transaction first does something that needs one. However, when a second or later bit of temporary storage is needed by a transaction, the database actually iterates over the list sequentially, therefore spreading activity more evenly across all the provided tablespaces.

By default, temporary objects are created in the default tablespace. They'll all appear in `base/pgsql_tmp` unless relocated. If you're not using temporary tables heavily, monitoring activity and disk space used in that directory can be used to estimate how heavy the sorting activity is on your database. This technique works even on earlier PostgreSQL releases that don't have the `log_temp_files` parameters that can also be used for that purpose. Monitoring disk space used can be useful even if that's available, because it gives you an idea of how much concurrent sorting is happening, something that's hard to tell from log file entries.

One interesting property of temporary files is that they're prime candidates for storage even on less reliable drives, such as you might have your operating system on. It's possible to safely, and sometimes quite usefully, put the temporary files onto a directory on your OS disks if they are underutilized. Just be careful because if those disks are lost, you'll need to recreate that empty tablespace on the OS drives of the replacement, or remove it from the `temp_tablespaces` list.

Disk arrays, RAID, and disk layout

Many people start their PostgreSQL tuning work by asking about how to spread the various pieces of the database across a large disk array. Recommending how to do that is really application specific, and it's only after covering so much background in the last few chapters that the right context is available to even discuss it.

If you have a really large number of disks available, a dozen or more, presuming that disk space needed for the database wasn't a problem, a good configuration would look like this:

Location	Disks	RAID Level	Purpose
`/`	2	1	Operating system
`$PGDATA`	6+	10	Database
`$PGDATA/pg_xlog`	2	1	WAL
`Tablespace`	1+	None	Temporary files

Here we're presuming that you want every bit of important data to be redundant, and therefore are using the mirroring facilities of RAID1 to do that. This gives a 3:1 ratio between database read/write throughput and that of the WAL, which is usually enough that activity to each to be balanced well. If you only have two pairs of drives for those (the database is on one pair and the WAL on another), it's likely the WAL will get considerably less activity than the database.

The leftover disk here from the hypothetical set of 12 would be assigned as a hot spare, something it's always a good idea to allocate. Given that arrays tend to be an even number of disks, pairing some temporary space with a hot spare can work out better than having two hot spares. Another typical use for a single leftover disk is to create a place to store non-critical backups and other working files, such as a database dump that needs to be processed before being shipped elsewhere.

Here's where things get tricky...what if instead, the above was done like this:

Location	Disks	RAID Level	Purpose
`/`	12	10	OS, DB, WAL

Would that perform better or worse than the manually split setup for the same number of disks shown above? The answer to that is simple: there's no way to predict that. Having your data striped across so many disks is going to improve the average performance of every operation you do on this array. This setup is almost certainly better for a workload where random seeks are the norm. However, there are several database components that are not accessed randomly. The `pg_xlog` parameter is only written to sequentially. Temporary files are not strictly sequential, but they're not heavily random either. And writes to the operating system drive normally happen without the force of a sync operation forcing them to disk, which means that the OS is free to delay those writes for improved write combining and sorting--both of which reduce seeks.

Putting everything onto one big array will reduce the chances to take advantages of optimizations like that, because the cache flushing required to make the database happy is likely to force out plenty of other data too, particularly on ext3.

With that perspective, you can think about this a bit differently:

Function	Cache Flushes	Access Pattern
Operating system	Rare	Mix of sequential and random
Database	Regularly	Mix of sequential and random
WAL	Constant	Sequential
Temporary files	Never	More random as client count increases

The exact mix of sequential and random behavior for your database is completely application dependent. How many concurrent users are involved, impacts both database and temporary file patterns--and again, what your application is doing impacts whether temp files are even important. Any attempt to optimize disk layout that doesn't take into account the access pattern of your app, including concurrency, is unlikely to predict performance correctly.

Even if you do have enough information to believe that you can predict an optimal layout, odds are still against you. To quote noted computer scientist Donald Knuth, *...the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail*. You would be unwise to presume that you will be the lucky person who guesses correctly.

If you do split something out, make sure to measure the improvement to confirm whether you were right. Ideally, you'd be able to simulate that in a test environment before doing so on a production system. In the real world, most development servers have far fewer disks than the production ones, making that tough to simulate. One good place to try it is during acceptance testing of new production hardware. You should be doing general system burn-in and testing of any candidate replacement for a production database server. That's a great time to experiment with (and measure!) changes in application performance, including different disk configurations.

Disk layout guidelines

There are a few guidelines that can help you prune down the possible configurations:

- Avoid putting the WAL on the operating system drive, because they have completely different access patterns and both will suffer when combined. Normally this might work out fine initially, only to discover a major issue when the OS is doing things like a system update or daily maintenance activity. Rebuilding the filesystem database used by the `locate` utility each night is one common source on Linux for heavy OS disk activity.
- If you have evidence you don't do any large sorting, the temporary files can be kept at their default location, as part of the database disk.
- On Linux systems running ext3, where `fsync` cache flushes require dumping the entire OS cache out to disk, split the WAL onto another disk as soon as you have a pair to spare for that purpose.

Beyond those rough rules, it's hard to settle arguments between the *put everything in one big array and let the OS sort it out* versus *break out everything into individually assigned disk so the DBA can tune* crowds. As is the case with many types of database tuning, in order to really know for sure what works best here, you'll need to simulate both configurations, using a benchmark workload as similar as possible to your production workload.

As for which side I personally fall onto, I prefer to split things up. The reason for that is not because I expect it to perform better, but because it's unquestionably easier to quantify what an application is doing that way. Being able to measure the exact balance of data going to database, WAL, and temporary disks is valuable both for general optimization, as well as for finding bottlenecks in application design. I consider the value of being able to monitor to that level to be the tiebreaker between these schools of thought on database layout.

Remember that the hardware your application is currently running on is not necessarily going to be what it runs on forever. If I'm planning out a new server, and I can predict the WAL versus DB activity ratio based on measurements of the old system, that's not just guessing anymore, and the odds are much better that it will be successful. But if the old system only has a large pile of disks, information gathered from them isn't nearly as useful in capacity planning for the future.

Summary

There aren't a lot of easy answers for what filesystem to use or how to arrange your disks. You need to error on the side of paranoia, not performance, if you expect to keep your database intact under the sort of odd things that will happen to it in production. And unfortunately, maximum reliability is usually slower, too. The most valuable thing you can do is be systematic in how you test. Instrument what you're doing, benchmark whenever possible, and always try to collect reproducible examples of realistic workloads whenever they appear. Filesystem crash protection is most commonly done by journaling writes, which adds overhead but makes recovery time after unclean shutdown predictable. On Linux, the ext3 filesystem allows a wide variety of tuning possibilities for its journal. While not the best performer, its occasional problem spots are at least well understood. Both XFS and ext4 are at early production quality right now, and expected to become increasingly viable alternatives to ext3 in the near future. Solaris and FreeBSD both have older UFS filesystems and the newer ZFS available. ZFS has several features that make it uniquely suited for large database installations. The NTFS filesystem on Windows shares many fundamentals with the UNIX-based ones popular on other operating systems. Increasing read-ahead, stopping updates to file access timestamps, and adjusting the amount of memory used for caching are common tuning techniques needed to get good performance on most operating systems. PostgreSQL allows relocating parts of the database through both symbolic links and the creation of tablespaces. In the next chapter, we will discuss how PostgreSQL manages memory and cache. We will also study how to optimally manage the available memory for better performance of the database. After discussing the hardware and filesystem, the next topic is the memory. The focus of the next chapter is memory configuration for better performance of the PostgreSQL server.

5
Memory for Database Caching

When you start a PostgreSQL server, it allocates a fixed size block of shared memory that all access to the information in the database passes through. In addition, each client that connects to memory uses up its own bit of memory, expanding as the client uses resources such as sorting space and storing data about pending transactions to commit.

Some settings in the database can be adjusted by the client after they connect. For example, the `work_mem` setting, a limiter on how much memory can be used for sorting, can be increased by a client after it connects. These allocations use non-shared memory, and tuning them is covered in the next chapter.

The major component to the shared memory used by the server is a large block allocated for caching blocks read from and written to the database. This is set by a parameter named `shared_buffers`. Monitoring and optimizing how this memory is used is the major topic of this chapter. It is one of the most important parameters to get good performance, and one of the hardest to predict an optimum value for.

In this chapter, we will look into the following topics:

- Memory units in the postgresql.conf file
- Increasing UNIX shared memory parameters for larger buffer sizes
- Crash recovery and the buffer cache
- Database buffer cache versus operating system cache
- Analyzing buffer cache contents

Memory units in the postgresql.conf file

All of the shared memory settings and the starting client settings for the database are stored in the `postgresql.conf` file. In earlier versions, you had to know the unit for each memory related setting: some were in units of 1K, some 8K, it was difficult to keep track of. Nowadays, you can still set values like that, but the preferred practice is to use a memory size instead. For example, if you wanted to set the `wal_buffers` value that controls how much memory to use for buffering write-ahead log data, you can do that now with a line like this in the `postgresql.conf` file:

```
shared_buffers = 128MB    # min 128kB
```

If you use the `SHOW` command to display the value for this setting, it will write it similarly (although it's possible the value will get rescaled to display better). But the database still converts this value into its own internal units, which for this parameter happens to be 8K blocks. It's helpful to be aware of this because you'll need to know the actual values to understand some of the internal information covered in this chapter. Like `shared_buffer`, there are other memory configuration settings, for example, `work_mem`, `maintenance_work_mem`, `autovacuum_work_mem`, and `dynamic_shared_memory_type`. Details of each and every configuration will be covered later in this book.

The `pg_settings` view in the database can be used to see how that conversion happens. It's also helpful to know that the `current_setting()` function can be used to get the same basic information as `SHOW`, but in a way you can use in queries. You can combine these two to help see the relationship between the internal way the server stores parameters and what they actually represent:

```
$ psql postgres
psql (9.6.1)
Type "help" for help.

postgres=# show shared_buffers;
 shared_buffers
----------------
 8MB
(1 row)

postgres=# SELECT name,
postgres-#    setting,
postgres-#    unit,
postgres-#   current_setting(name)
postgres-#       FROM pg_settings
postgres-#       WHERE name='wal_buffers';
    name     | setting | unit | current_setting
-------------+---------+------+-----------------
 wal_buffers | 2048    | 8kB  | 16MB
(1 row)
```

Increasing UNIX shared memory parameters for larger buffer sizes

When you use the `initdb` command to create a new PostgreSQL cluster, the server detects how large of a shared memory block it can allocate by starting at a moderate value and decreasing it until the allocation is successful. This is necessary because on many platforms, including some very popular UNIX ones, the default values for how much shared memory you can allocate is very low. Even on recent software such as the constantly updated Linux kernels and really small values are possible on older systems.

The default memory sizes in the `postgresql.conf` file are not optimized for performance or for any idea of a typical configuration. They are optimized solely so that the server can start on a system with low settings for how much shared memory it can allocate, because that situation is so common.

There are a variety of common errors you can get when a PostgreSQL server fails to start, they are documented at `http://www.postgresql.org/docs/current/static/server-start.html` and the one related to shared memory parameters being too low looks like this:

```
FATAL: could not create shared memory segment: Invalid argument
DETAIL: Failed system call was shmget(key=5440001, size=4011376640,
 03600).
```

The kernel resources documentation page at `http://www.postgresql.org/docs/current/static/kernel-resources.html` goes over how to increase this parameter in detail for most platforms. If you have a system that supports the `getconf` command, which many UNIX-like ones do, the following program will produce reasonable starting values by asking `getconf` for information about how much memory is in your system. Create a `shmsetup.sh` file with the following contents:

```
#!/bin/bash
# simple shmsetup script
page_size=`getconf PAGE_SIZE`
phys_pages=`getconf _PHYS_PAGES`
shmall=`expr $phys_pages / 2`
shmmax=`expr $shmall \* $page_size`
echo kernel.shmmax = $shmmax
echo kernel.shmall = $shmall
```

After changing the permission of the `shmsetup.sh` file, execute the script as follows:

```
$ chmod +x shmsetup.sh
$ ./shmsetup.sh
kernel.shmmax = 2123816960
kernel.shmall = 518510
```

This sets the maximum shared block size to one half of total memory and outputs values suitable for a Linux `/etc/sysctl.conf` file. Here's an example from a system with 2 GB of physical RAM, run as root:

```
# ./shmsetup >> /etc/sysctl.conf
# sysctl -p
kernel.shmmax = 2123816960
kernel.shmall = 518510
```

Here `shmmax` is the maximum size (in bytes) for a single shared memory segment, set to 1 GB. And `shmall` is the total amount of shared memory (in pages) that all processes on the server can use. The number of bytes in a page depends on the operating system; 4096 is a common value. A more robust version of this program is included as one of this book's code samples, which handles this page math for you.

Kernel semaphores

While not a memory allocation figure, another occasional `sysctl` tweaking requirement for PostgreSQL is to increase the number of system semaphores, an object used for process communication, available. The defaults on a recent Linux system look like this:

```
$ ipcs -l

------ Messages Limits --------
max queues system wide = 1684
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384

------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 4177919
max total shared memory (kbytes) = 17112760316
min seg size (bytes) = 1

------ Semaphore Limits --------
max number of arrays = 128
max semaphores per array = 250
max semaphores system wide = 32000
```

```
max ops per semop call = 100
semaphore max value = 32767
```

One of the parts trimmed from the `ipcs` output also shows the shared memory limits, so you can also double-check them with it. Setting higher semaphore limits is done with a kernel parameter that combines the main four limits here into one line. This maps into the format like this:

```
$ sysctl kernel.sem
kernel.sem = 250 32000 100 128
```

All four of the values here might need to be increased on systems with a large number of processes, setting the same way as the increased shared memory sizes.

Estimating shared memory allocation

It's possible to predict how much memory the PostgreSQL server is expected to allocate given the server parameters. Very little of this is likely to matter to you because all the other sizes are dwarfed by `shared_buffers` unless your client count is extremely high. Assuming a PostgreSQL 9.6 server, these are the default values for the settings involved here:

Parameter	Default value
`max_locks_per_transaction`	64
`max_connections`	100
`autovacuum_max_workers`	3
`max_prepared_transactions`	0
`block_size`	8192
`wal_block_size`	8192
`wal_buffers`	-1 (min 32kB, -1 sets based on `shared_buffers`)

There's no default value that you can assume for `shared_buffers`, because that value is detected at server cluster creation time. The default setting of 0 for `max_prepared_transactions` makes the prepared transaction space allocation go away altogether.

Prepared transactions involve the two-phase commit features of the database, and have nothing to do with the much more common prepared statements that are used for things such as preventing SQL injections.

To further simplify the memory estimate computation, you can just add in the autovacuum workers as clients (103 instead of 100). The main elements of the allocation estimate table then becomes as follows:

Usage	**Approximate shared memory bytes**
Connections+AV workers	1.9 MB
Shared disk buffers	8400 * `shared_buffers`
WAL buffers	64 kB
Fixed space requirements	770 kB

That's about 2.8 MB plus whatever the `shared_buffers` amount comes out as. Since a typical value for `shared_buffers` starts at 24 MB or more, it is obviously the dominant thing to be concerned about when estimating shared memory consumption.

Inspecting the database cache

You can look inside the current contents of the PostgreSQL `shared_buffers` database cache using the `pg_buffercache` module. This is one of the optional `contrib` modules available that ships with PostgreSQL; see the first chapter for more information about making sure that these are installed. An introduction to `pg_buffercache` is available as part of the PostgreSQL documentation at `http://www.postgresql.org/docs/current/static/pgbuffercache.html`.

The information on how the database stores information in this chapter assumes you have this module installed, so that you can look at how blocks in shared memory change as you perform various activities. This is the best way to understand how the relevant database internals work. On a production server, using `pg_buffercache` may not be as vital, but it's extremely valuable for learning how the database works with its shared memory well enough to tune memory use.

When hearing about a database block that's in memory, you'll sometimes see that referred to as a page instead. It's normal to hear people speak of the database buffer cache being organized into 8K pages of memory. The only real distinction between these two terms, which are normally used interchangeably, is that a page in memory actually holds a block plus a small amount of overhead to identify what block this is--what's referred to as the buffer header.

Installing pg_buffercache into a database

The various parts of `pg_buffercache` include a library written in C and some SQL that handles viewing the information. While the buffer cache itself is shared among every database in the cluster, `pg_buffercache` only can show you really useful information related to the database you're currently connected to. And its SQL component needs to be installed in each database you want to monitor (but not the templates one) as well.

In order to install this utility or use it, you will need to be connected to the database as its superuser. Here's a sample that installs `pg_buffercache` into a new database, presuming the RedHat Linux standard directory installation tree; substitute your own PostgreSQL installation directory into here:

```
$ createdb pgbench
$ psql pgbench
psql (9.6)
Type "help" for help.

pgbench=# CREATE EXTENSION pg_buffercache;
CREATE EXTENSION
```

You can confirm that the utility is working as expected by looking at how large your system `shared_buffers` is, and noting that the count of entries returned by `pg_buffercache` matches it:

```
pgbench=# SELECT name,
                  setting,unit,
                  current_setting(name)
           FROM pg_settings
            WHERE name='shared_buffers';
      name       | setting | unit | current_setting
-----------------+---------+------+-----------------
 shared_buffers | 1024    | 8kB  | 8MB
(1 row)

pgbench=# select count(*) from pg_buffercache;
 count
-------
  1024
(1 row)
```

Note the small value, here, `128MB`. This is from a Linux system running kernel 3.16.0 that has gigabytes of RAM.

There are several examples of useful reports you can produce with `pg_buffercache` near the end of this chapter.

Database disk layout

But in order to fully interpret the information returned by utilities such as `pg_buffercache`, and therefore use that information to adjust memory sizing, it's helpful to know a bit about how databases are stored on disk. On UNIX-like systems, typically the `$PGDATA` environment variable on your system will point to your database cluster, and underneath there the `base/` directory actually contains the database tables.

If you're not sure where your database is located on disk, but if you can connect to the database, the location of this and other important files on the server is available from the `pg_settings` view. This particular detail is in the `data_directory` setting; here's an example of locating it, and a query showing the locations of other interesting files related to the server:

```
postgres=# show data_directory;
      data_directory
---------------------------
 /usr/local/pgsql/bin/data
(1 row)

postgres=# select name,setting from pg_settings where category='File Locations';
       name        |                  setting
-------------------+-------------------------------------------
 config_file       | /usr/local/pgsql/bin/data/postgresql.conf
 data_directory    | /usr/local/pgsql/bin/data
 external_pid_file |
 hba_file          | /usr/local/pgsql/bin/data/pg_hba.conf
 ident_file        | /usr/local/pgsql/bin/data/pg_ident.conf
(5 rows)
```

Assuming `$PGDATA` is set to point to `data_directory`, if you create a table and want to have a look for the associated data file, that would start like this:

```
postgres=# SELECT t.relname,
         d.datname,
         'base/' || d.oid || '/' || t.oid AS PATH
         FROM pg_class t, pg_database d
         WHERE d.datname LIKE 'postgres'
         AND t.relname LIKE 'foo';
 relname | datname  |       path
---------+----------+------------------
 foo     | postgres | base/12669/68509
(1 row)
```

Let's look at the following code:

```
$ls data/base/
1 12170 12175 16384
```

These three directories each contain one database: `template0`, `template1`, and the default `postgres` database. Both template databases are used to create new databases; their use is documented at `http://www.postgresql.org/docs/current/static/manage-ag-templatedbs.html`.

The cryptic numbers are what are called an **object identifier** or **OID**. In early PostgreSQL versions, every row in every table received an OID, which was in some cases used as a way to uniquely identify that row. To reduce overhead, these no longer are included in individual rows by default. You can still do so if you want, and many of the system catalog tables are created including them. There is more about the OID and other hidden system columns at `http://www.postgresql.org/docs/current/static/ddl-system-columns.html`.

Another handy system column to know about is `ctid`, which can still be used as a way to uniquely identify a row, even in situations where you have multiple rows with the same data in them. This provides a quick way to find a row more than once, and it can be useful for cleaning up duplicate rows from a database too.

To decipher the structure of what's in the `base/` directory, we need the OID of the database and the relation OID (a table is a type of relation) for the `foo` table just created. You can determine those using the `pg_database` and `pg_class` system views; your exact numbers here will likely be different if you try this yourself:

```
postgres=# SELECT datname,oid FROM pg_database WHERE datname='postgres';
 datname  |  oid
----------+-------
 postgres | 12669
(1 row)

postgres=# SELECT relname, oid, relfilenode
      FROM pg_class
      WHERE relname='foo';
 relname |  oid  | relfilenode
---------+-------+-------------
 foo     | 68509 |       68509
(1 row)
```

Here the OID and the file it's stored in are the same. There are a few things you can do to a table that will change the name of the file used to store it, without changing the OID. Examples include `TRUNCATE`, `REINDEX`, and `CLUSTER`. To find out where a relation is really at, you do need to check the `relfilenode` field in its `pg_class` catalog entry.

If you want to do this sort of investigation from the command prompt, the `oid2name` command (another optional `contrib` module similar to `pg_buffercache`) can provide the same information in cases where the OID still matches the filename:

```
laptop56:bin ibrar$ ./oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  12669       postgres  pg_default
  12668      template0  pg_default
      1      template1  pg_default
```

Let's look at the following code:

```
$ oid2name -t foo
From database "postgres":
Filenode Table Name
----------------------
   68509     foo
```

In this example, having not inserted anything into this table yet, you'd expect the file for it to be empty, and for there to be no blocks containing its data in the buffer cache, which is the case:

```
laptop56:bin ibrar$ ls -l $PGDATA/base/12669/68509
-rw-------  1 ibrar  wheel  0 May  8 23:28 /usr/local/pgsql/bin/data//base/12669/68509
```

```
postgres=# SELECT reldatabase,
         relfilenode,
         relblocknumber
      FROM pg_buffercache
      WHERE relfilenode = 68509;
 reldatabase | relfilenode | relblocknumber
-------------+-------------+----------------
(0 rows)
```

Creating a new block in a database

Once you insert something into this table, a page is allocated in the buffer cache to hold that new information, and a standard 8K block is allocated on disk:

```
postgres=# INSERT INTO foo(a, b) VALUES (0,0);
INSERT 0 1
postgres=# SELECT reldatabase,
           relfilenode,
           relblocknumber
           FROM pg_buffercache
           WHERE relfilenode = 68509;
 reldatabase | relfilenode | relblocknumber
-------------+-------------+----------------
       12669 |       68509 |              0
(1 row)
```

You can always see that file on disk:

```
laptop56:bin ibrar$ ls -l $PGDATA/base/12669/68509
-rw-------  1 ibrar  wheel  8192 May  8 23:51 /usr/local/pgsql/bin/data//base/12669/68509
laptop56:bin ibrar$ ./psql postgres
psql (9.6.1)
Type "help" for help.

postgres=# SELECT reldatabase, relfilenode,
         relblocknumber, isdirty, usagecount
      FROM pg_buffercache
      WHERE relfilenode=68509;
 reldatabase | relfilenode | relblocknumber | isdirty | usagecount
-------------+-------------+----------------+---------+------------
       12669 |       68509 |              0 | f       |          4
(1 row)
```

Note that this block is dirty. That means that the version in memory has not been written out to disk yet. Also note that the `usagecount` value for it is 1. This means that one database process has accessed this block since its section of the buffer cache was last considered as a candidate for reuse.

Showing the `usagecount` value in `pg_buffercache` is only available in PostgreSQL 8.3 and later. You can leave that part out of the various examples shown in this chapter. You'll just not be able to get quite as much information about how popular the blocks you're viewing statistics about are.

Writing dirty blocks to disk

There are a couple of ways blocks that are dirty can be written to disk. The easiest one to trigger at will is a checkpoint:

```
postgres=# CHECKPOINT;
CHECKPOINT
postgres=# SELECT reldatabase, relfilenode,
          relblocknumber, isdirty, usagecount
      FROM pg_buffercache
      WHERE relfilenode=68509;
 reldatabase | relfilenode | relblocknumber | isdirty | usagecount
-------------+-------------+----------------+---------+------------
       12669 |       68509 |              0 | f       |          4
(1 row)
```

You can check the file on disk:

```
laptop56:bin ibrar$ ls -l $PGDATA/base/12669/68509
-rw-------  1 ibrar  wheel  8192 May  8 23:51 /usr/local/pgsql/bin/data//base/12669/68509
```

You can see the block is no longer dirty after the checkpoint, and the last updated time for the file saving its data on disk has moved forward to match this write.

From the perspective of the database, each table is a series of numbered blocks, each 8K in size, starting at 0 when you create the first row and moving forward from there. As you further extend the size of the table, the upper limit for individual files is 1 GB. Above that size, you'll see multiple files.

There are two other ways that a dirty block can be written out, which are described in the next few sections. To cover the full lifecycle of how blocks get in and out of the buffer cache, a diversion into crash recovery and checkpoints is needed to explain some of the activity you'll see going on.

Crash recovery and the buffer cache

If you had to write out every database change to disk immediately after it's made, the database performance would really suffer. This is particularly true on blocks that you change all the time, which would then be written very often. But you have to be able to recover if the server crashes before things are written out completely too. Periodic database checkpoints take care of that.

Checkpoint processing basics

A checkpoint iterates over every dirty block in the system as of a point in time, writing them out to disk. Then that information is flushed to permanent storage, via the same mechanisms WAL writes are. Once that's done, if your system crashes, recovery from the crash will start from this new last point where the database was sure everything on disk was consistent. In this way, the database constantly moves forward the recovery starting position to the point in time the previous checkpoint started at. Everything that happened to dirty blocks before then is guaranteed to have been written to the database blocks on disk, and therefore the amount of potential information that might be lost in a crash is kept under control.

Write-ahead log and recovery processing

In the second chapter, the PostgreSQL write-ahead log was introduced as the mechanism used to recover from data lost in a crash. Its documentation starts at `http://www.postgresql.org/docs/current/static/wal.html`.

The WAL is mainly a stream of information about database blocks. When you commit a data block change, if this is the first time since the last checkpoint the block has been changed, the entirety of the original block is written out. That means an 8K WAL write even if you're changing only a single byte. This behavior is optional, but necessary if you want safe crash recovery. You need to set `full_page_writes` to on or off in the `postgresql.conf` file, a complete description of `full_page_writes` can be found in the PostgreSQL documentation at `http://www.postgresql.org/docs/current/static/runtime-config-wal.html`. Once a page has been written out at least once after a checkpoint, just the information about the row that's changed needs to be written.

PostgreSQL has a command called `CHECKPOINT` to enforce the transaction log checkpoint; complete details about checkpoint can be found at the PostgreSQL documentation at `http://www.postgresql.org/docs/current/static/sql-checkpoint.html`.

One side effect of this structure is that just after a checkpoint, there is a potential for write I/O to the WAL to spike, because every page that is dirtied is going to get a full page write. Once the more popular blocks such as high-level index structures have been fully written once, additional WAL updates to those pages will only include the row deltas, and therefore generate less write volume than the first change.

If you ever wondered how the database was able to recover from crashes even if they involved partial writes of blocks to disk, this is the mechanism that makes it happen. After a crash, recovery starts at wherever the last checkpoint ended at. If you know exactly what was originally in each block you modify, and what changed in it each time it's modified, replaying those changes will include everything needed to put the disk block back into the same state it would have ended up in had the server not crashed.

Checkpoint timing

Checkpoints are valuable because they reduce the amount of time required to recover from a crash, so you want to perform them as often as you can stand. But the overhead of both the checkpoint writes and the subsequent full page writes to the WAL is expensive, which means you don't want checkpoints to happen too often.

You can adjust how often checkpoints occur by modifying two database tuneables that control the two primary ways by which a checkpoint starts:

- `min_wal_size` and `max_wal_size`: In the latest release of PostgreSQL the old parameter, `checkpoint_segments`, is replaced with two new parameters, `min_wal_size` and `max_wal_size`. The `max_wal_size` parameter is the maximum WAL size between automatic checkpoints. The old value `checkpoint_segments` can be replaced with this formula:

    *max_wal_size = (3 * checkpoint_segments) * 16MB*

- `checkpoint_timeout`: When `checkpoint_timeout` of time has passed since the last checkpoint, another one will occur regardless of how many segments have been used. This is accounted for in the `pgstat_bgwriter.checkpoints_timed` statistic. By default the checkpoint timeout is 5 minutes.

- `checkpoint_completion_target`: The parameter enforces PostgreSQL to write data slower, which means that PostgreSQL should take *checkpoint_timeout * checkpoint_comlpetion_target* amount of time to write the data. By default the value is 0.5.
- `checkpoint_warning`: A message to the server log written if checkpoints occurrence (because of filing of segments) is closer to this amount of time. There will be no message if `checkpoint_timeout` is less than `checkpoint_timeout`. By default it is disabled.

- `checkpoint_flush_after`: Enforces the operating system to issue write on underlying storage when this number is written in case of checkpoint.
- `checkpoints_req`: If you do something that manually triggers a checkpoint, such as executing the `CHECKPOINT` command, these are counted in the `checkpoints_req` category. In addition to monitoring the statistics, you can see exactly what happens during each checkpoint by turning on the `log_checkpoints` parameter.

If you have `checkpoint_segments` set very low--the default is a tiny 3 resulting in a checkpoint after only 48 MB of WAL writes--and you reach a segment driven checkpoint required fast enough, a warning about this tuning error is shown in the database logs.

The `log_checkpoints` and `pgstat_bgwriter` parameters are only available in PostgreSQL 8.3 and later. In earlier versions, the only useful way to determine when a checkpoint happened was to set the `checkpoint_warning` parameter to a very high value--such as its maximum of 3,600 seconds--which then forces a harmless warning entry into the database log file every time a checkpoint happens. Unless it coincides with a message about the segments being too low, you won't know for sure whether this checkpoint was triggered for time or segment reasons. Generally you can make a good guess just by noting when the previous checkpoint was. If it was `checkpoint_timeout` before this one, it's likely a time-driven checkpoint.

Checkpoint tuning based on what's inside the buffer cache and the statistics shown in `pg_stat_bgwriter` is a major subject for a later chapter. The basics of their impact on disk I/O can be explained simply--but it does depend heavily on the version of the database you're running.

The `checkpoint_segments` is no longer available in the latest version of PostgreSQL, use this formula instead:
*max_wal_size = (3 * checkpoint_segments) * 16MB*

Checkpoint spikes

As you might guess, if there's a lot of dirty data in a large shared buffer cache, writing it all out once will really cause I/O on your disks to increase. This is particularly true in PostgreSQL versions 8.2 and earlier, which just write the entire dirty portion of the cache out as fast as possible--regardless of its impact on the rest of the system. In versions older than PostgreSQL 9.6 the checkpoints wrote out dirty pages in random order, or you can say when this happened to appear in shared buffers. This is a serious performance hit on rotating media. PostgreSQL 9.6 introduces a new technique in which checkpoint driven writes are done as block numbers and files.

Spread checkpoints

Starting in PostgreSQL 8.3, the spread checkpoint feature, tuned by `checkpoint_completion_target`, aims to reduce the amount of writes in any unit of time by spreading them out over a much longer period of time. For recovery to work, it's only necessary that a checkpoint finishes before the next one can start. There's no reason you can't have one checkpoint start almost immediately after the previous one, executing slowly writing everything that happened while the previous checkpoint was active. This is exactly what the spread checkpoint feature does. It estimates when the next checkpoint is coming based on whether checkpoints seem to be happening because of the timeout being reached or because the checkpoint segment limit was reached, and aims to finish at the completion target percentage of that value.

It is still possible to run into checkpoint spikes even given the spread checkpoint feature. They are particularly common on filesystems that don't know how to force to disk the changes to a single file at a time. On Linux, ext3 has a particularly bad limitation. Any attempt to sync a single file will write the entire write cache for that device out to accomplish that. This can easily lead to a large checkpoint I/O spike as many files worth of data are all forced out at once. Better filesystems such as XFS and ext4 don't suffer from that particular problem, but checkpoint spikes are still possible even on them.

Database block lifecycle

The normal sequence by which a block makes its way through shared memory and either onto (or back onto) disk works like the following.

A page to hold the database block needed is requested for allocation in the shared buffer cache. If that block already exists in there, no disk read is needed. Otherwise, ask the operating system to read the page into memory. In either case, at this point the page in memory will have its usage count increased, because presumably it's being requested so that it can be used for something. The `pgstat_bgwriter.buffers_alloc` statistics will be incremented to count this allocation.

Changes are made to the block. In the preceding example, an `INSERT` created a new row in a new block. You might also `UPDATE` or `DELETE` something that changes a block. The block is now dirty.

The transaction commits to the WAL. At this point, whether or not the dirty block is written out to the database file on disk in the near future, the information needed to make sure that happens eventually is safe--presuming your hardware doesn't have the sort of problematic write-back cache warned against in the disk hardware section.

The dirty block is written to disk, the buffer page is marked clean.

Those last two steps are asynchronous: the statistics update and the write of the dirty block don't happen instantly, they happen when it's the appropriate time for them to.

Dirty block write paths

Once a block has been made dirty in shared memory, there are three possible ways it can be written back to the database, each tracked by a counter in `pg_stat_bgwriter`:

- `buffers_checkpoint`: Clearing all of the dirty blocks included as part of a checkpoint write are accounted for here. This value jumps all at once when the checkpoint finishes, which makes monitoring this an alternate way to figure out when checkpoints finished. This is additional monitoring along with the log files.
- `buffers_backend`: A backend (any process besides the background writer that also handles checkpoints) tried to allocate a buffer, and the one it was given to use was dirty. In that cache the backend must write the dirty block out itself before it can use the buffer page.

- `buffers_clean`: The type of backend write described previously makes some other process stall for a moment while it writes out dirty data. To keep that from happening too often, the background writer process scans forwarding looking for blocks that might be allocated in the near future that are dirty and that have a low usage count (alternatively called the **Least Recently Used** or **LRU** blocks). When it finds them, it writes some of them out pre-emptively, based on historical allocation rates. Should they get dirty again before they're reallocated to a backend, that effort was wasted. The usage count and dynamically tuned approach used by the background writer made that unlikely.

Generally, the best type of write here is one done by a checkpoint. In each of the other cases, it's possible that this block will get dirty again before the next checkpoint, which makes the earlier write a waste of resources. More on using this `pg_stat_bgwriter` data to tune write performance will appear in later chapters:

```
postgres=# SELECT * FROM pg_stat_bgwriter;
-[ RECORD 1 ]---------+------------------------------
checkpoints_timed     | 8
checkpoints_req       | 6
checkpoint_write_time | 7378
checkpoint_sync_time  | 8
buffers_checkpoint    | 79
buffers_clean         | 0
maxwritten_clean      | 0
buffers_backend       | 6
buffers_backend_fsync | 0
buffers_alloc         | 710
stats_reset           | 2017-05-08 23:17:03.411527+05
```

Database buffer cache versus operating system cache

Unlike many traditional database products, PostgreSQL does not assume or even prefer that the majority of the memory on the system be allocated for its use. Most reads and writes from the database are done using standard operating system calls that allow the operating system's cache to work in its usual fashion. In some configurations WAL writes will bypass the OS cache, that's the main exception.

If you're used to a database where most system RAM is given to the database and the OS cache is bypassed using approaches such as synchronous and direct writes, you don't want to set up PostgreSQL the same way. It will be downright counterproductive in some areas. For example, PostgreSQL's stores commit log information in the `pg_clog` directory. This data is both written to and read from regularly, and it's assumed that the operating system will take care of optimizing that access. If you intentionally bypass PostgreSQL's intentions by using mounting tricks to convert writes to synchronous or direct, you may discover some unexpected performance regressions due to this issue, among others.

So why not just give all the RAM to the OS to manage? The main reason that the PostgreSQL shared buffer cache can do better than the operating system is the way it keeps a usage count of buffers. This allows buffers to get a "popularity" score from 0 to 5, and the higher the score the less likely it is that those buffers will leave the cache. The way this works, whenever the database is looking for something to evict to make more space for data it needs, it decreases that usage count. Every increase in usage count makes that block harder to get rid of. The implementation used is called a clock-sweep algorithm.

Typical operating system caches will only give any buffer one or two chances before that data is evicted, typically with some form of LRU algorithm. If your database has data in it that accumulates a high usage count, it's likely that data is being served better staying in the database's shared RAM than in the operating systems.

There are also some optimizations to the PostgreSQL buffer cache to deal with large queries, such as sequential scans of the entire database and VACUUM, which lets it hold onto important data that's likely to be blown out of the operating system cache as part of doing that job.

Doubly cached data

One reason not to make the shared buffer cache too large is that the OS cache is also being used for reads and writes, and it's extremely likely that there's going to be some wasted overlap there. For example, if you read a database block from disk that's not been requested before by the server, it's first going to make its way into the OS cache, and then it will be copied into the database buffer cache, what's referred to as double buffering. The OS copy and the database cache copy will both be evicted from their respective caches eventually, but for some period of time there's going to be duplication there. Keeping `shared_buffers` at only a modest percentage of RAM reduces the amount of such data likely to be duplicated.

Inspecting the OS cache

On some operating systems, it's possible to extract information from the operating system's cache and combine it with details of the PostgreSQL one. If available, this provides a more comprehensive view of the data than looking just at the database side of things. Projects viewing this data are still in the early stages; however, some example implementations include the following:

- `http://www.kennygorman.com/wordpress/?p=250`: Python script showing PostgreSQL objects in Linux memory
- `http://pgfoundry.org/projects/pgfincore/`: `pgfincore` provides this data directly inside the database itself, where you might combine it with data from `pg_buffercache`

Checkpoint overhead

If your database buffer cache is large, it's also possible that a large quantity of data could be dirty when a checkpoint starts, causing a checkpoint I/O spike. When using a later PostgreSQL version that supports spread checkpoints, you can tune checkpoint frequency to reduce the average size of these spikes, and you'll also have the `pg_stat_bgwriter` view to help you with that optimization.

But ultimately you should realize that every bit of data that's inside of the database's shared buffer cache needs to be accounted for at checkpoint time, while data in the OS cache does not. The flip side to this is that keeping data in `shared_buffers` can reduce total write volume, when you modify the same data block more than once per checkpoint--which is common for index blocks in particular. Sizing to optimize has to consider both sides of this trade-off. You should aim for the most frequently used blocks to end up in database shared memory, while the less popular blocks spend more time in the OS cache.

Starting size guidelines

The "lore" for sizing `shared_buffers` generally specifies a suggestion in terms of a percentage of system RAM. Some advocate only 10-15%. The thorough academic exploration in the paper *Tuning Database Configuration Parameters with iTuned* at `http://www.cs.duke.edu/~shivnath/papers/ituned.pdf` found 40% optimal on a 1 GB system being tested on a wide range of queries. And occasionally reports appear where 60% or more of total RAM turns out to be optimal.

Generally, if you have a server where OS overhead is small relative to total memory (any modern system with 1 GB or more of RAM), giving the database 25% of total RAM is a reasonable starting setting for `shared_buffers` in the middle of the effective range. It may not be optimal, but it's unlikely to be so high that double buffering becomes an issue. And it's likely to be far better than the tiny default forced on the database by the low shared memory parameters of typical operating system kernel defaults.

Platform, version, and workload limitations

In addition to systems with small amounts of RAM, there are several situations where even 25% of RAM is too much:

- **Early versions**: Using a large buffer cache was made possible by improvements in PostgreSQL 8.1. Earlier versions were hard pressed to even use 512 MB of RAM effectively. Unless you intend to benchmark to prove larger increases are valuable, a reasonable upper limit for 8.0 or earlier is 32768 pages (256 MB- versions this early don't allow using memory units for their parameters either).
- **Windows**: For reasons not yet fully identified by the PostgreSQL community, the Windows platform does not perform well with large shared memory allocations. Peak performance on Windows installs is typically with `shared_buffers` in the 64 MB to 512 MB range. Using 256 MB would be an aggressive, but likely still safe, maximum value to use, again unless you can benchmark to prove further increases are useful.
- **Checkpoint spikes**: In versions 8.2 and earlier, PostgreSQL performed its checkpoint writes in what's essentially one big dump. This has an extremely high potential to cause other I/O on the system to pause, often for several seconds, until the checkpoint write finishes. If you have a version of PostgreSQL before 8.3, and you have a write-heavy workload, you are more likely to suffer from checkpoint spike pauses than you are to benefit from having more shared memory for queries--given that the OS cache will supplement for you there. 128 MB of RAM is a reasonable starting size for write-intensive workloads in earlier versions that do not support spread checkpoints.
- **Shared server**: If this is not a dedicated database server, you need to carefully consider the requirements of other applications before allocating a large portion of memory that is dedicated to the database. Since PostgreSQL will always fall back to sharing the OS cache even if it doesn't have dedicated RAM, smaller values for `shared_buffers` can make it much easier to build a high performance host that's doing both database and application duties.

- **Very large systems**: Experiments on systems with large amounts of RAM suggest that probably due to internal partitioning issues in the buffer cache (which haven't been adjusted since PostgreSQL 8.4), setting it larger than approximately 8 GB can be counterproductive. If you have a system with a very large amount of RAM, there is not much information available on where cache sizing breaks down, but it's likely in the multiple gigabyte range. On servers with 8 GB or more of RAM, you might start with only 2 GB dedicated to the database, and only resize upward if cache inspection suggests it's likely to be productive.

The typical PostgreSQL default install, where `shared_buffers` is 128 MB or less, is unlikely to perform very well except in the least demanding situations. But there are plenty of PostgreSQL systems that achieve good performance with `shared_buffers` increased no further than the 128 MB to 256 MB range. If you're going to follow standard guidelines and use a large amount of RAM, you owe it to yourself to confirm it is being used well.

Analyzing buffer cache contents

You've seen and been told how using a block will increase its usage count. You've also seen and been told how a dirty block might make its way out to disk. This wasn't intended just as an academic exercise. Believe it or not, all of this information is useful for determining how large your shared buffer cache should be!

If you want to do better than following a rule of thumb for how big to set `shared_buffers` relative to the OS cache, you have two options. You can run your own benchmarks with your application and see how the results vary based on the amount of shared memory dedicated to the database. Just be careful to account for the influence of the larger OS cache when running multiple such tests. Or you can use knowledge of how the buffer cache works from inside of it to help make that decision.

Inspection of the buffer cache queries

Note that due to multiple changes between PostgreSQL in version 8.3, a few of these queries will work without modification on earlier versions. You will have to remove all references to `usagecount` and in some cases you may need to adjust how values are cast between types.

For the following example outputs (which are scripted into the code sample named `bufcache.sh` included with this book), `shared_buffers` is set to 256 MB, and `pgbench` is initialized with a scale of 50 (full coverage of `pgbench`, another database `contrib`. utility, appears in a later chapter):

```
$ pgbench -i -s 50 pgbench
```

```
pgbench=# SELECT
          pg_size_pretty(pg_database_size('pgbench'))
          AS db_size;
 db_size
---------
 755 MB
(1 row)
```

Now run the benchmark:

```
$ pgbench -S -c 8 -t 25000 pgbench
starting vacuum...end.
transaction type: <builtin: select only>
scaling factor: 50
query mode: simple
number of clients: 8
number of threads: 1
number of transactions per client: 25000
number of transactions actually processed: 200000/200000
latency average = 0.130 ms
tps = 61359.047683 (including connections establishing)
tps = 61396.498665 (excluding connections establishing)
```

This makes for a total database size greater than the shared buffer cache can hold, forcing some prioritization of what is and isn't kept in there via the `usagecount` mechanism. After creation, the queries run against the accounts table by `pgbench` make sure the cache is being used by some sort of workload worth analyzing: in this case, many clients are only doing reads from the accounts table, the largest in the database.

The `pg_buffercache` module needs to be installed into the `pgbench` database before any of these queries are run as well.

`pg_buffercache` requires broad locks on parts of the buffer cache when it runs. As such, it's extremely intensive on the server when you run any of these queries. They are not recommended for regular monitoring use. A snapshot on a daily basis or every few hours is usually enough to get a good idea of how the server is using its cache, without having the monitoring itself introduce much of a load. These are not queries you want to run every minute.

Top relations in the cache

The following example appears in the documentation as an example of how to use `pg_buffercache`, and it's quite a good way to start your analysis:

```
pgbench=# SELECT
 c.relname,
 count(*) AS buffers
FROM pg_class c
 INNER JOIN pg_buffercache b
  ON b.relfilenode=c.relfilenode
 INNER JOIN pg_database d
  ON (b.reldatabase=d.oid AND d.datname=current_database())
GROUP BY c.relname
ORDER BY 2 DESC
LIMIT 10;
          relname          | buffers
---------------------------+---------
 pgbench_accounts_pkey     |     358
 pgbench_accounts          |     315
 pg_depend_reference_index |      12
 pg_depend                 |      10
 pg_rewrite                |       6
 pg_depend_depender_index  |       5
 pg_description            |       5
 pg_amop                   |       5
 pg_extension              |       5
 pg_init_privs             |       5
(10 rows)
```

Removing the system tables (which has been done on all the examples shown here) shows that almost all of the cache is being used by the `pgbench_accounts` table and the index enforcing its primary key, as expected given those accounts are all we were running the `SELECT` statements against:

```
       relname         | buffers
-----------------------+---------
pgbench_accounts       |    9334
pgbench_accounts_pkey  |    6936
```

Summary by usage count

That doesn't give any interesting information about the usage counts of either table though; this query will show them:

```
pgbench=# SELECT
          usagecount,count(*),isdirty
          FROM pg_buffercache
          GROUP BY isdirty,usagecount
          ORDER BY isdirty,usagecount;
 usagecount | count | isdirty
------------+-------+---------
          0 |   511 | f
          1 |   241 | f
          2 |    49 | f
          3 |    20 | f
          4 |    67 | f
          5 |   134 | f
          5 |     2 | t
(7 rows)
```

The results don't prove anything interesting so far, they just suggest there's not very much accumulating a high usage count on a straight percentage basis.

Buffer contents summary with percentages

This query is where you probably want to start when analyzing a new database. It lets you quickly see not just how much data is being cached for each table but how much that represents relative to its total size:

One troublesome change in PostgreSQL 8.4 that commonly impacts code like this is that `pg_relation_size()` and `pg_total_relation_size()` were changed to run against a `regclass` type instead of a text name. These examples sidestep that by using a join with `pg_class` that provides an OID, but your own queries similar to these may not be able to use that trick.

```
pgbench=# SELECT
 c.relname,
 pg_size_pretty(count(*) * 8192) as buffered,
 round(100.0 * count(*) /
  (SELECT setting FROM pg_settings
   WHERE name='shared_buffers')::integer,1)
  AS buffers_percent,
 round(100.0 * count(*) * 8192 /
  pg_relation_size(c.oid),1)
  AS percent_of_relation
FROM pg_class c
 INNER JOIN pg_buffercache b
  ON b.relfilenode = c.relfilenode
 INNER JOIN pg_database d
  ON (b.reldatabase = d.oid AND d.datname = current_database())
GROUP BY c.oid,c.relname
ORDER BY 3 DESC
LIMIT 10;
          relname           | buffered | buffers_percent | percent_of_relation
----------------------------+----------+-----------------+---------------------
 pgbench_accounts_pkey      | 2752 kB  |            33.6 |                 2.5
 pgbench_accounts           | 2408 kB  |            29.4 |                 0.4
 pg_depend_reference_index  | 96 kB    |             1.2 |                28.6
 pg_operator                | 88 kB    |             1.1 |                73.3
 pg_depend                  | 80 kB    |             1.0 |                18.2
 pg_rewrite                 | 56 kB    |             0.7 |                58.3
 pg_amop                    | 48 kB    |             0.6 |               100.0
 pg_depend_depender_index   | 40 kB    |             0.5 |                12.5
 pg_description             | 40 kB    |             0.5 |                14.7
 pg_extension               | 40 kB    |             0.5 |               500.0
(10 rows)
```

The `pg_relation_size()` function does not include data that has been stored in a TOAST table associated with this one, and therefore it may not be an accurate assessment of your table's size. In PostgreSQL 9.0, you can replace this with the `pg_table_size()` function added in that version, which does include TOAST information in its sum. See `http://www.postgresql.org/docs/current/interactive/storage-toast.html` for an introduction to TOAST, and `http://www.postgresql.org/docs/current/interactive/functions-admin.html` for documentation of the system size functions.

Now we're seeing something useful. While `pgbench_accounts_key` is only using up to 33.6.% of the cache, it's holding onto nearly 100% of the data in that index. This tells us that the database believes that keeping that data in memory is really important. A breakdown of the `usagecount` data by relation will give more insight into how that is happening.

Buffer usage count distribution

This query breaks down usage counts by relation and usage count, so you can see exactly how usage count distribution differs between tables:

```
pgbench=# SELECT
        c.relname, count(*) AS buffers,usagecount
        FROM pg_class c
        INNER JOIN pg_buffercache b
        ON b.relfilenode = c.relfilenode
        INNER JOIN pg_database d
        ON (b.reldatabase = d.oid
        AND d.datname = current_database())
        AND relname LIKE 'pgbench_%'
        GROUP BY c.relname,usagecount
        ORDER BY c.relname,usagecount;
        relname         | buffers | usagecount
------------------------+---------+------------
 pgbench_accounts       |     254 |          0
 pgbench_accounts       |      45 |          1
 pgbench_accounts_pkey  |     248 |          0
 pgbench_accounts_pkey  |      46 |          1
 pgbench_accounts_pkey  |       1 |          3
 pgbench_accounts_pkey  |      48 |          4
(6 rows)
```

The primary key index here isn't using that many buffers, but the ones it does have in the cache average a very high usage count. Meanwhile, the regular accounts data takes up much more room, but extremely little of it gets a `usagecount` greater than 1. This explains why the database considers `pgbench_accounts_pkey` as a popular relation, it should keep a large portion in memory, while `pgbench_accounts` is being treated as more transient data. This is really what you want. The ability to look up data in the accounts table requires having the primary key index, so you'd hope and expect as much of that index to be cached as possible.

Using buffer cache inspection for sizing feedback

The preceding examples give you the basic two things to look for when deciding if your `shared_buffers` cache is large enough. First, compare what percentage of the relations you believe are important to your application's performance appear to be cached. If this number is low, you may benefit from a larger buffer cache. A helpful secondary look at this information is available if you combine this with hit rates from views such as `pg_stat_user_tables`, which will be covered in a later chapter.

The balance of popular (high usage count) versus transient (low usage count) pages tells you a lot about whether your cache is sized appropriately. If most of your pages have low usage counts (0,1), but you're still getting good hit rates from the table statistics, you can probably decrease the size of the buffer cache without a performance loss. Even the simplest operating system LRU algorithm is capable of making good caching decisions where there aren't particularly popular pages to prioritize over transient ones.

But if you are accumulating a large number of blocks with a high usage count (4,5), this is strong evidence that your data is being served well and stored in the database shared buffer cache, and that your application might benefit from it being even larger.

Summary

Because database servers have so many different types of workloads they might encounter, it's difficult to give any hard rules for optimal configuration just based on server hardware. Some applications will benefit from having really large amounts of dedicated database memory in the form of `shared_buffers`; others will suffer large checkpoint spike problems if you do that. PostgreSQL versions starting with 8.3 do provide you with tools to help monitor your system in this area though. If you combine that with some investigation of just how the server is using the memory you've allocated for it, and preferably add in some of the monitoring techniques covered in later chapters, you'll be much better informed. A quick look inside of the actual contents of the database buffer cache will answer all sorts of questions about how the server is using memory, and be much more accurate for planning purposes than guessing.

PostgreSQL allocates one large shared memory block on server startup to cache reads and writes to database disk blocks. This cache works in cooperation with, rather than replacing the operating system cache, and it should be sized as only a moderate percentage of total RAM. The default shared memory allocation limits on many systems are extremely low, and will need to be increased to allow proper shared memory sizing for the database. Crash recovery is done by writing full copies of the blocks being modified to the WAL before changing the associated file on the database disk. How far back recovery must go in order to repair damage after a crash moves forward as each checkpoint finishes. Checkpoints need to be tuned carefully to limit crash recovery time, while not impacting the rest of the system's performance. In the next chapter, we will study how to tune the PostgreSQL server for optimal performance.

6
Server Configuration Tuning

The main tunable settings for PostgreSQL are in a plain text file named `postgresql.conf` that's located at the base of the database directory structure. This will often be where `$PGDATA` is set to on UNIX-like systems, making the file `$PGDATA/postgresql.conf` on those platforms.

This chapter mirrors the official documentation's structure regarding these parameters at `http://www.postgresql.org/docs/current/static/runtime-config.html`, but rather than describing the meaning of every parameter, this chapter is more focused on the guidelines for setting the most important values from the perspective of someone interested in performance tuning. This should be considered a supplement to rather than a complete replacement for the extensive material in the manual.

Another live resource related to this subject is the article *Tuning Your PostgreSQL Server* at `http://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server`, which has some overlap with the information covered here. The wiki structure of this article makes it conducive to being kept current, so it may eventually include details about future PostgreSQL versions that aren't yet released.

In this chapter, we will cover the following topic:

- Interacting with the live configuration

Interacting with the live configuration

There are many ways to change the database's parameters beyond just editing its configuration file and restarting. Understanding these can be critical to reducing server downtime just for routine configuration changes, as well as making sure you're adjusting the parameter you want at the time when you want the change to take effect.

Defaults and reset values

The database has two things that you might refer to as defaults depending on your context. The first type of default is what the server will set the value to if you don't ever change it, the setting the server starts with before it's even read the `postgresql.conf` file. Starting in PostgreSQL 8.4, you can check this value using the `pg_settings` view by looking at the `boot_val` column:

```
postgres=# \d pg_settings;
     View "pg_catalog.pg_settings"
     Column      |  Type   | Modifiers
-----------------+---------+-----------
 name            | text    |
 setting         | text    |
 unit            | text    |
 category        | text    |
 short_desc      | text    |
 extra_desc      | text    |
 context         | text    |
 vartype         | text    |
 source          | text    |
 min_val         | text    |
 max_val         | text    |
 enumvals        | text[]  |
 boot_val        | text    |
 reset_val       | text    |
 sourcefile      | text    |
 sourceline      | integer |
 pending_restart | boolean |
```

Detail about this view can be found on Official documentation of PostgreSQL:
`http://www.postgresql.org/docs/current/static/view-pg-settings.html`

Once the server has started, and you make changes to the parameters, there's also a default value that those parameters will return to if you use the `RESET` command documented at `http://www.postgresql.org/docs/current/static/sql-reset.html` to return them to their starting value; this is labeled as `reset_val` in the `pg_settings` view.

Allowed change context

Every configuration setting has an associated context in which it's allowed to be changed. The best way to determine the allowed change context for a setting is to ask the database directly. This example shows one entry with each context type (the actual result if you run this query will include every server parameter):

```
postgres=# select name,context from pg_settings limit 6;
          name           |  context
-------------------------+------------
 allow_system_table_mods | postmaster
 application_name        | user
 archive_command         | sighup
 archive_mode            | postmaster
 archive_timeout         | sighup
 array_nulls             | user
(6 rows)
```

The context field isn't documented very well in the official manual. Here are the meanings of the various settings you'll find there, sorted from the hardest to the easiest to change:

- `internal`: These settings are mainly the database internals set at compile time. They're included in the view for your information, but cannot be changed without recompiling the server.
- `postmaster`: These are only updated when a full server restart is performed. All shared memory settings fall into this category.
- `sighup`: Sending the server a HUP signal will cause it to reload the `postgresql.conf` file, and any changes made to this parameter will be immediately active. See the next section for information about how to do this.
- `backend`: These settings are similar to the `sighup` ones, except that the changes made will not impact any already running database backend sessions. Only new sessions started after this will respect the change. There are very few parameters with this property, and most impact behavior that only happens when backend sessions start or end. The example shown previously, `log_connections`, can't be made retroactive in order to log a connection that's already been made. Only new connections made after `log_connections` is enabled will be logged.
- `superuser`: This can be modified by any database superuser (usually the person who created the database and commonly postgres) at any time, and made active without even requiring a full configuration reload. Most of the settings in this category relate to logging various aspects of statements executed by the server.
- `user`: Individual user sessions can adjust these parameters at any time. Their changes will only impact that session. Most parameters here alter how queries execute, which allows you to tweak the query execution on a per-session basis.
- `superuser-backend`: These settings can be changed in the `postgresql.conf` file without restarting the PostgreSQL server. These values can be changed for a particular session using libpq's `PGOPTIONS`. This value cannot be changed after starting the session, and only the superuser can change these setting.

As you might imagine from this list, the answer to a seemingly simple question such as "what's the current value of `work_mem`?" can have a very different answer depending on when you ask and the context involved. It might be initially set to one value in the `postgresql.conf` file, have the default changed to another by a configuration reload, and then be changed again by one backend adjusting its individual setting before running a query.

Reloading the configuration file

There are three ways you can get the database to reload its configuration in order to update values in the `sighup` category. If you're connected to the database as a superuser, `pg_reload_conf` will do that:

```
postgres=# SELECT pg_reload_conf();
pg_reload_conf
----------------
t
```

You can send a HUP signal manually using the UNIX `kill` command:

```
$ ps -eaf | grep "postgres -D"
postgres 11185   1 0 22:21 pts/0  00:00:00
/home/postgres/inst/bin/postgres -D /home/postgres/data/
$ kill -HUP 11185
```

Finally, you can trigger a `SIGHUP` signal for the server by using `pg_ctl`:

```
$ pg_ctl reload
LOG: received SIGHUP, reloading configuration files
server signaled
```

No matter which approach you use, you'll see the following in the database log files afterwards to confirm that the server received the message:

```
LOG: received SIGHUP, reloading configuration files
```

You can then confirm that your changes have taken place as expected using commands such as `SHOW`, or by looking at `pg_settings`.

Commented-out settings

What happens when you had something set manually but then disable it on a running server? It depends on the version you're running. Let's say your `postgresql.conf` file started with the following setting active:

```
checkpoint_timeout = 5min
```

You now edit the file to comment that setting out:

```
#checkpoint_timeout = 5min
```

Then you tell the server to re-read the configuration file:

```
$ pg_ctl reload
```

`#checkpoint_timeout` is a `sighup` context setting. The `reload` command throws a note in the log file.

```
LOG: received SIGHUP, reloading configuration files
LOG: parameter checkpoint_timeout removed from configuration file,
 reset to default
```

You can confirm the setting using the `SHOW` command.

```
postgres=# SHOW checkpoint_timeout;
checkpoint_timeout
--------------------
5min
(1 row)
```

This behavior is both complicated and version-dependent. Experienced PostgreSQL administrators will usually double-check parameters they intended to change afterwards using the `SHOW` statement, or by looking at `pg_settings`, to make sure the settings match what was expected.

Another complicated area here is that it's possible to include additional configuration files from inside the master `postgresql.conf`. These effectively work as if you'd inserted the text of the file into the spot you included it in. You can see the file's settings originated from using `pg_settings` in this case, along with what line the active version came from. Also note that if you set a parameter more than once, only the last setting will be registered.

Server-wide settings

While these parameters might be adjustable in other contexts, generally the ones in this section are only adjusted in the `postgresql.conf` file before the server is started.

Database connections

There are many configuration parameters that control how people can connect remotely and locally to the database. The full list is documented at `http://www.postgresql.org/docs/current/static/runtime-config-connection.html`.

listen_addresses

Any installation that requires a connection from a remote system will need to change `listen_addresses` to allow that. The default only allows local connections from users logged into the same system as the database server. A common approach is to accept incoming connections from anywhere-as far as the primary configuration file is concerned-like this:

```
listen_addresses = '*'
```

Then you should set up the `pg_hba.conf` file described at `http://www.postgresql.org/docs/current/static/auth-pg-hba-conf.html` to control who can actually connect. There is a potential performance concern to this approach in that filtering out connections using a more focused setting for `listen_addresses` can be more efficient than letting clients connect. If you let a client connect and then validate them against `pg_hba.conf`, this will use additional service resources, and introduces the possibility of a malicious user; launching a denial of service attack via this route.

In practice, few PostgreSQL servers are directly exposed to Internet-wide queries anyway. Normally, you'd filter out the default port for PostgreSQL (5432) at the firewall level to handle this level of security, which is the most efficient approach and a common implementation shared for securing other applications too. If you have a system that's exposed to the world, which is increasingly common for situations such as cloud-hosted databases, make sure to use all three layers of defense. Restrict who can connect at the firewall level, reduce what addresses you listen on, if practical, and lock down who can get access to the database using `pg_hba.conf`.

max_connections

One of the settings that you'll always find set to a value, typically 100, in the `postgresql.conf` generated by `initdb` is `max_connections`. Because each connection uses a small amount of shared memory, as described in the previous chapter, it's possible for systems with particularly limited shared memory defaults to not even allow this many connections. Accordingly, like `shared_buffers`, some investigation is done when the database cluster is created, and then the largest value supported-up to 100-is saved into the default configuration. In practice, the amount of nonshared memory each client uses for things such as sorting will dwarf this, but the shared component can't be completely ignored.

It is important not to set this parameter to a value much higher than you need it to be. There are several downsides to larger settings. The first is wasted shared memory, typically the last of the things to be concerned about as the amount per connection is small.

There are other resources a client can use, however, with memory allocations for sorting (controlled via `work_mem`, which we'll discuss shortly) normally the largest. If a large number of connections are allowed, to be safely conservative, these settings must be made smaller too so that the potential for excess resource use is limited.

Windows PostgreSQL servers can be extremely limited in the number of connections they can support due to resource allocation issues. It's common for them to only support about 125 connections before running out of memory in the desktop heap area. See `https://wiki.postgresql.org/wiki/Running_%26_Installing_PostgreSQL_On_Native_Windows` for more details about this issue and the potential workarounds for it.

Finally, establishing connections in PostgreSQL should be considered a resource-intensive operation. It is not the goal of the database server to make the process of establishing a database connection, authenticating, and then executing a query a lightweight operation. Generally, while connection overhead starts to become a bottleneck on a general server operation after several hundred connections, the exact threshold varies depending on hardware and configuration. Certainly if you intend to support thousands of queries at once, you cannot do that by allowing each client to connect to the database directly. Putting connection pooling software between the application and the database is the common way to handle this scalability limitation, and that topic is covered in a later chapter.

Shared memory

The shared memory settings are important to get right because they will always require a full database restart to change--the server cannot re-allocate shared memory dynamically.

shared_buffers

Setting `shared_buffers` usefully was the topic of most of the last chapter.

Free space map settings

Space left behind from deletions or updates of data is placed into a free space map by `VACUUM`, and then new allocations are made from that free space first rather than by allocating new disk for them.

Starting in PostgreSQL 8.4, the **free space map** (**FSM**) is stored on disk, and therefore scales in size automatically. In PostgreSQL versions up to 8.3, the FSM was stored in shared memory--which required monitoring how much of it was being used--potentially increasing it in size. The `max_fsm_pages` and `max_fsm_relations` parameters have been removed to simplify the free space map administration. More details about the free space map can be found in the official documentation at `https://www.postgresql.org/docs/current/static/storage-fsm.html`.

Logging

The general logging setup is important but somewhat outside the scope of this book. You may need to set parameters such as `log_destination`, `log_directory`, and `log_filename` to save your log files in a way that is compatible with the system administration's requirements of your environment. On most systems these will all be set to reasonable defaults to get started. A later chapter will cover adjustments to these for CSV logging, which can be helpful for analyzing query timing.

On UNIX-like systems, it's common for some of the database logging to be set in the script that starts and stops the server, rather than directly in the `postgresql.conf` file. If you instead use the `pg_ctl` command to manually start the server, you may discover that the logging ends up on your screen instead. If you want to duplicate that behavior, you'll need to look at the script that starts the server normally (commonly `/etc/init.d/postgresql`) to determine what it does. In most cases, you just need to add `-l logfilename` to the `pg_ctl` command line to redirect its output to the standard location.

log_line_prefix

The default `log_line_prefix` parameter is empty, which is not what you want. A good starting value here is the following:

```
log_line_prefix='%t:%r:%u@%d:[%p]: '
```

This will put the following into every log line:

- `%t`: Timestamp
- `%u`: Database username
- `%r`: Remote host connection
- `%d`: Database
- `%p`: Process ID of connection

It may not be immediately obvious what you'd want all of these values for, particularly the process ID. Once you've tried to chase down a few performance issues, the need for saving these values will be more obvious, and you'll be glad to already have this data logged.

Another approach worth considering is setting `log_line_prefix` such that the resulting logs will be compatible with the `pgFouine` program, described in the monitoring chapter. That is a reasonable, general purpose logging prefix, and many sites end up needing to do some sort of query analysis eventually.

log_statement

The options for this setting are as follows:

- `none`: Do not log any statement-level information.
- `ddl`: Log only **data definition language** (**DDL**) statements such as `CREATE` and `DROP`. This can normally be left in, even in production, and can be used to catch major changes introduced accidentally or intentionally by administrators.
- `mod`: Log any statement that modifies a value, which is essentially everything except for simple `SELECT` statements. If your workload is mostly `SELECT`-based with relatively few data changes, it may be practical to leave this enabled all the time.
- `all`: Log every statement. This is generally impractical to leave on in production due to the overhead of the logging. However, if your server is powerful enough relative to its workload, it may be practical to keep it on all the time.

Statement logging is a powerful technique for finding performance issues. Analyzing the information saved by `log_statement` and related sources for statement-level detail can reveal the true source of many types of performance issues. You will need to combine this with the appropriate analysis tools, which are covered in `Chapter 7`, *Routine Maintenance*.

log_min_duration_statement

Once you have some idea of how long a typical query statement should take to execute, this setting allows you to log only the ones that exceed the threshold you set. The value is in milliseconds, so you might set the following:

```
log_min_duration_statement=1000
```

With this, you'll only see statements that take longer than 1 second to run. This can be extremely handy for finding out the source of outlier statements that take much longer than most to execute.

If you are running version 8.4 or later, you might instead prefer to use the `auto_explain` module (`https://www.postgresql.org/docs/current/static/auto-explain.html`) instead of this feature. This will allow you to actually see why the queries that are running slowly are doing so by viewing their associated `EXPLAIN` plans.

Vacuuming and statistics

PostgreSQL databases require two primary forms of regular maintenance as data is added, updated, and deleted.

`VACUUM` cleans up after old transactions, including removing information that is no longer visible and returning freed space to where it can be reused. The more often you `UPDATE` and `DELETE` information from the database, the more likely you'll need a regular vacuum cleaning regime. But even static tables with data that never changes once inserted still need occasional care here.

`ANALYZE` looks at tables in the database and collects statistics about them. This could be information such as estimates of how many rows they have and how many distinct values are in them. Many aspects of query planning depend on this statistics data being accurate.

There's more about VACUUM in Chapter 7, *Routine Maintenance*, and the use of statistics is covered as part of Query Optimization.

autovacuum

Because both these tasks are critical to database performance over the long term, starting in PostgreSQL 8.1 there is an autovacuum daemon available that will run in the background to handle these tasks for you. Its action is triggered by the number of changes to the database exceeding a threshold it calculates based on the existing table size.

The parameter for autovacuum is turned on by default in PostgreSQL 8.3, and the default settings are generally aggressive enough to work out of the box for smaller databases with little manual tuning.

Enabling autovacuum

If you have autovacuum available and it's not enabled, then you need to set it as follows:

```
autovacuum = on     # Enable autovacuum subprocess? 'on'
                    # requires track_counts to also be on.
```

As you can see in the preceding comment, track_count can also be set to true. In older versions, you needed to set stats_start_collector and stats_row_level to true. In the latest version (9.3) and above, both parameters are merged into track_count.

Note that, as warned in the documentation, it's also wise to consider adjusting superuser_reserved_connections to allow for autovacuum's processes in these earlier versions.

The autovacuum you'll get in 8.1 and 8.2 is not going to be as efficient as what comes in 8.3 and later. You can expect it to take some fine tuning to get the right balance of enough maintenance without too much overhead, and because there's only a single worker, it's easier for it to fall behind on a busy server. This topic isn't covered at length here. It's generally a better idea to put time into planning an upgrade to a PostgreSQL version with a newer autovacuum than to try and tweak an old one extensively, particularly given that there are so many other performance issues that cannot be resolved easily in the older versions.

maintainance_work_mem

A few operations in the database server need working memory for larger operations than just regular sorting. `VACUUM`, `CREATE INDEX`, and `ALTER TABLE ADD FOREIGN KEY` can both allocate up to `maintainance_work_mem`-worth of memory instead. Because it's unlikely that many sessions will be performing one of these operations at once, it's possible to set this value much higher than the standard per-client `work_mem` setting. Note that at least `autovacuum_max_workers` (defaulting to three starting in version 8.3) will allocate this much memory, so consider those sessions (perhaps along with a session or two performing a `CREATE INDEX`) when setting this value.

Assuming you haven't increased the number of autovacuum workers, a typical high setting for this value on a modern server would be at 5% of the total RAM, so that even five such processes wouldn't exceed a quarter of the available memory. This works out to approximately 50 MB of `maintainance_work_mem` per every GB of server RAM.

default_statistics_target

PostgreSQL makes its decisions about how queries execute based on statistics collected about each table in your database. This information is collected by analyzing the tables, either with the `ANALYZE` statement or via autovacuum. In either case, the amount of information collected during the analyze step is set by `default_statistics_target`. Increasing this value makes `ANALYZE` take longer, and since autovacuum's `ANALYZE` happens regularly, this turns into increased background overhead for database maintenance. But if there aren't enough statistics about a table, you can get bad plans for queries against it.

The default value for this setting used to be the very low figure of 10, but was increased to 100 in PostgreSQL 8.4. Using that larger value was popular in earlier versions too, for general improved query behavior. Indexes using the `LIKE` operator also tend to work much better with values greater than100 than below it, due to a hard-coded change at that threshold.

Note that increasing this value results in a net slowdown on your system if you're not ever running queries where the additional statistics result in a change to a better query plan. This is one reason why some simple benchmarks show PostgreSQL 8.4 as slightly slower than 8.3 at default parameters for each, and in some cases you might return an 8.4 install to a smaller setting. Extremely large settings for `default_statistics_target` are discouraged because of how much overhead they incur.

If there is just a particular column in a table that you know needs better statistics, you can use `ALTER TABLE SET STATISTICS` on that column to adjust that particular setting. This works better than increasing the system-wide default and making every table pay for that requirement. Typically the columns that really require a lot more statistics to work properly will require a setting near the maximum of 1,000 (increased to 10,000 in later versions) to get a serious behavior change, which is far higher than you'd want to collect data for on every table in the database.

Checkpoints

The mechanics of how checkpoints work were covered in the previous chapter, along with the principle tunables involved. The discussion here will focus mainly on the common practice for initially setting these values.

In the latest release of PostgreSQL, the old parameter `checkpoint_segments` is replaced with two new parameters, `min_wal_size` and `max_wal_size`. The `max_wal_size` parameter is the maximum WAL size between automatic checkpoints. The old value `checkpoint_segments` can be replaced with this formula:

*max_wal_size = (3 * checkpoint_segments) * 16MB*

If WAL disk usage is less than the `min_wal_size` parameter, the old WAL are recycled.

checkpoint_timeout

The default for this setting of 5 minutes is fine for most installations. If your system isn't able to keep up with writes to the point where the timeout is the main thing driving when the checkpoints happen, it's reasonable to consider increasing this value. Aiming for 10 minutes or more between checkpoints isn't dangerous; again, it just increases how long database recovery after a crash will take. Since this is just one component to the database server downtime after a crash, that's something you need a healthy respect for.

checkpoint_completion_target

If you have increased `max_wal_size`, it's reasonable to also increase `checkpoint_competion_target` to its practical maximum of 0.9. This gives maximum checkpoint spreading, which theoretically means the smoothest I/O too. In some cases, keeping the default of 0.5 will still be better, however, as it makes it less likely that one checkpoint's writes will spill into the next one.

It's unlikely a value below 0.5 will be very effective at spreading checkpoints at all, and unless you have an extremely large value for the number of segments, the practical difference between small changes in its value are unlikely to matter. One approach for the really thorough is to try both 0.5 and 0.9 with your application and see which gives the smoother disk I/O curve over time, as judged by OS-level monitoring.

wal_buffers

While the documentation on `wal_buffers` suggests the default of 64 KB is sufficient so long as no single transaction exceeds that value, in practice write-heavy benchmarks see optimal performance at higher values than you might expect from that--at least 1 MB or more. With the only downside being increased use of shared memory, and since there's no case where more than a single WAL segment could need to be buffered, given modern server memory sizes the normal thing to do nowadays is to just set the following:

```
wal_buffers=16MB
```

Then you can forget about it as a potential bottleneck or item to tune further. Only if you're tight on memory should you consider a smaller setting.

wal_sync_method

`Chapter 2`, *Database Hardware*, touched on the importance of setting up your server to avoid volatile write-back caches. One purpose of `wal_sync_method` is to tune such caching behavior.

The default behavior here is somewhat different from most options. When the server source code is compiled, several possible ways to write are considered. The one that is believed to be most efficient then becomes the compiled-in default. This value is not written to the `postgresql.conf` file at `initdb` time, though, making it different from other auto-detected, platform-specific values, such as `shared_buffers`.

Before adjusting anything, you should check what your platform detected as the fastest safe method using `SHOW`; here is a Linux example:

```
postgres=# show wal_sync_method;
 wal_sync_method
-----------------
 fdatasync
```

On both Windows and Mac OS X platforms, there is a special setting to make sure the OS clears any write-back caches. The safe value to use on these platforms that turns on this behavior is:

```
wal_sync_method=fsync_writethrough
```

If you have this setting available to you, you really want to use it! It does exactly the right thing to make database writes safe while not slowing down other applications the way disabling an entire hard drive write cache will do.

This setting will not work on all platforms, however. Note that you will see a performance drop going from the default to this value, as is always the case when going from unsafe to reliable caching behavior.

On other platforms, tuning `wal_sync_method` can be much more complicated. It's theoretically possible to improve write throughput on any UNIX-like system by switching from any write method that uses a `write`/`fsync` or `write`/`fdatasync` pair to one using a true synchronous write. On platforms that support safe `DSYNC` write behavior, you may already see this as your default when checking it with `SHOW`:

```
wal_sync_method=open_datasync
```

You won't see it explicitly listed in the configuration file as such. If this is the case on your platform, there's little optimization beyond this that you can likely do. `open_datasync` is generally the optimal approach, and when available it can even use direct I/O as well to bypass the operating system cache.

The Linux situation is perhaps the most complicated. As shown, this platform will default to `fdatasync` as the method used. It is possible to switch this to use synchronous writes with:

```
wal_sync_method=open_sync
```

In many cases, you can discover this is faster--sometimes much faster--than the default behavior. However, whether this is safe or not depends on your filesystem. The default filesystem on most Linux systems, ext3, does not handle `O_SYNC` writes safely in many cases, which can result in corruption. See *PANIC caused by open_sync on Linux* at `http://archives.postgresql.org/pgsql-hackers/2007-10/msg01310.php` for an example of how dangerous this setting can be on that platform. There is evidence this particular area has finally been cleaned up on recent (2.6.32) kernels when using the ext4 filesystem instead, but this has not been tested extensively at the database level yet.

In any case, your own tests of `wal_sync_method` should include the **pull the cord** test, where you power the server off unexpectedly, to make sure you don't lose any data with the method you've used. It might also be a good idea to test at a very high load for a long period of time to find intermittent bugs that might cause a crash.

PITR and WAL replication

The `archive_mode`, `archive_command`, and `archive_timeout` settings are discussed in `Chapter 14`, *Scaling with Replication*.

Per-client settings

While all of the settings in this section can be adjusted per client, you'll still want good starting settings for these parameters in the main configuration file. Individual clients that need values outside of the standard can always do so using the `SET` command within their session.

effective_cache_size

As mentioned in the last chapter, PostgreSQL is expected to have both its own dedicated memory (`shared_buffers`) in addition to utilizing the filesystem cache. In some cases, when making decisions such as whether it is efficient to use an index or not, the database compares the sizes it computes against the effective sum of all these caches; that's what it expects to find in `effective_cache_size`.

The same rough rule of thumb that would put `shared_buffers` at 25% of system memory would set `effective_cache_size` to between 50% and 75% of RAM. To get a more accurate estimate, first observe the size of the filesystem cache:

- **UNIX-like systems**: Add the `free` and `cached` numbers shown by the `free` or `top` commands to estimate the filesystem cache size
- **Windows**: Use the Windows Task Manager's **Performance** tab and look the `System Cache` size

Assuming you have already started the database, you need to then add the `shared_buffers` figure to this value to arrive at a figure for `effective_cache_size`. If the database hasn't been started yet, usually the OS cache will be an accurate enough estimate when it's not running. Once it is started, most of the database's dedicated memory will usually be allocated to its buffer cache anyway.

The `effective_cache_size` parameter does not allocate any memory. It's strictly used as input on how queries are executed, and a rough estimate is sufficient for most purposes. However, if you set this value much too high, actually executing the resulting queries may result in both the database and OS cache being disrupted by reading in the large number of blocks required to satisfy the query believed to fit easily in RAM.

It's rare you'll ever see this parameter tuned on a per-client basis, even though it is possible.

synchronous_commit

In the hardware chapter, the overhead of waiting for physical disk commits was stressed as a likely bottleneck for committing transactions. If you don't have a battery-backed write cache to accelerate that, but you need better commit speed, what can you do? The standard approach is to disable `synchronous_commit`, which is sometimes alternatively referred to as enabling asynchronous commits. This groups commits into chunks at a frequency determined by the related `wal_writer_delay` parameter. The default settings guarantee a real commit to disk at most 600 milliseconds after the client commit. During that window--which you can reduce in size with a corresponding decrease in speed-up--if your server crashes, that data will not be recovered afterwards.

Note that it's possible to turn this parameter off for a single client during its session:

```
SET LOCAL synchronous_commit TO OFF;
```

You don't have to make it a server-wide choice. This provides you with the option of having different physical commit guarantees for the different types of data you put into the database. An activity monitoring table that is frequently inserted into, where a fraction of a second of loss is acceptable, would be a good candidate for an asynchronous commit. An infrequently written table holding real-world monetary transactions should prefer the standard synchronous commit.

work_mem

When a query is running that needs to sort data, the database estimates how much data is involved and then compares it to the `work_mem` parameter. If it's larger (and the default is only 1 MB), rather than do that sorting in memory, it will write all the data out and use a disk-based sort instead. This is much, much slower than a memory-based one. Accordingly, if you regularly sort data, and have memory to spare, a large increase in `work_mem` can be one of the most effective ways to speed up your server. A data warehousing report on a giant server might run with a gigabyte of `work_mem` for its larger reports.

The catch is that you can't necessarily predict the number of sorts any one client will be doing, and `work_mem` is a per-sort parameter rather than a per-client one. This means that memory use via `work_mem` is theoretically unbounded, were a number of clients sorting enough large things to happen concurrently.

In practice, there aren't that many sorts going on in a typical query, usually only one or two. And not every client that's active will be sorting at the same time. The normal guidance for `work_mem` is to consider how much free RAM is around after `shared_buffers` is allocated (the same OS caching size figure needed to compute `effective_cache_size`), divide this figure by `max_connections`, and then take a fraction of that figure; 1/2 of that would be an aggressive `work_mem` value. In that case, only if every client had two sorts active all at the same time would the server be likely to run out of memory, which is an unlikely scenario.

The `work_mem` computation is increasingly used in later PostgreSQL versions for estimating whether hash structures can be built in memory. Its use as a client memory size threshold is not limited just to sorts; that's simply the easiest way to talk about the type of memory allocation decision it helps to guide.

Like `synchronous_commit`, `work_mem` can also be set per client. This allows an approach where you keep the default to a moderate value, and only increase sort memory for the clients that you know are running large reports.

random_page_cost

This parameter is common to tune, but explaining what it does requires a lot of background about how queries are planned. That will be covered in the chapter devoted to query optimization. Particularly in earlier PostgreSQL versions, lowering this value from its default--for example a reduction from 4.0 to 2.0--was a common technique for making it more likely that the planner would use indexed queries instead of the alternative of a sequential scan.

With the smarter planner in current versions, this is certainly not where you want to start tuning at. You should prefer getting better statistics and setting the memory parameters as primary ways to influence the query planner.

constraint_exclusion

If you are using PostgreSQL 8.3 or earlier versions, and you are using the database's table inheritance feature to partition your data, you'll need to turn this parameter on. The reasons for that are covered in the partitioning chapter of this book.

Starting in 8.4, `constraint_exclusion` defaults to a new smarter setting named `partition` that will do the right thing in most situations without it ever needing to be adjusted.

Tunables to avoid

There are a few parameters in the `postgesql.conf` file that have gathered up poor guidance in other guides you might come across, and they might already be set badly in a server with a configuration you're now responsible for. Others have names suggesting a use for a parameter that doesn't actually exist. This section warns you about the most common of those to avoid adjusting.

fsync

If you just want to ignore crash recovery altogether, you can do that by turning off the `fsync` parameter. This makes the value for `wal_sync_method` irrelevant, because the server won't be doing any WAL sync calls anymore.

It is important to recognize that if you have any sort of server crash when `fsync` is disabled, it is likely that your database will be corrupted and will no longer start afterwards. Despite this being a terrible situation to be running a database under, the performance speedup of turning crash recovery off is so large that you might come across suggestions you disable `fsync` anyway. You should be equally hesitant to trust any other advice you receive from sources suggesting this, as it is an unambiguously dangerous setting to disable.

One reason this idea gained traction is that in earlier PostgreSQL versions there was no way to reduce the number of `fsync` calls to a lower number, to trade-off some amount of reliability for performance. Starting in 8.3, in most cases where people used to disable `fsync` it's a better idea to turn off `synchronous_commit` instead.

There is one case where `fsync=off` may still make sense--Initial bulk loading. If you're inserting a very large amount of data into the database, and do not have hardware with a battery-backed write cache, you might discover this takes far too long to ever be practical. In this case, turning the parameter off during the load--where all data can easily be recreated if there is a crash causing corruption--may be the only way to get loading time below your target. Once your server is back up again, you should turn it right back on again.

Some systems will also turn off `fsync` on servers with redundant copies of the database--for example, slaves used for reporting purposes. These can always resynchronize against the master if their data gets corrupted.

full_page_writes

Much like `fsync`, turning this parameter off increases the odds of database corruption in return for an increase in performance. You should only consider adjusting this parameter if you're doing extensive research into your filesystem and hardware in order to assure partial page writes cannot happen.

commit_delay and commit_siblings

Before `synchronous_commit` was implemented, there was an earlier attempt to add that sort of feature enabled by the `commit_delay` and `commit_siblings` parameters. These are not effective parameters to tune in most cases. It is extremely difficult to show any speedup by adjusting them, and quite easy to slow every transaction down by tweaking them. The only case where they have shown some value is for extremely high I/O rate systems. Increasing the delay a very small amount can make writes happen in bigger blocks, which sometimes turn out better aligned when combined with larger RAID stripe sizes in particular.

max_prepared_transactions

Many people see this name and assume that since they use prepared statements, a common technique to avoid SQL injection, they need to increase this value. This is not the case; the two are not related. A prepared transaction is one that uses `PREPARE TRANSACTION` for **two-phase commit** (**2PC**). If you're not specifically using that command and 2PC, you can leave this value at its default. If you are using those features, only then will you likely need to increase it to match the number of connections.

Query enable parameters

It's possible to disable many of the query planner's techniques in the hopes of avoiding a known bad type of query. This is sometimes used as a work-around for the fact that PostgreSQL doesn't support direct optimizer hints for how to execute a query. For example, you might be shown this as a suggestion:

```
enable_seqscan = off
```

Such a method is often suggested as a way to force the use of indexes instead of sequential scans.

Generally, this is a bad idea, and you should improve the information the query optimizer is working with so it makes the right decisions instead. This topic is covered in a later chapter in this book dedicated to query optimization.

New server tuning

There are a few ways to combine all of this information into a process for tuning a new server. Which is best is based on what else you expect the server to be doing, along with what you're looking to adjust yourself versus taking rule of thumb estimates for.

Dedicated server guidelines

Initial server tuning can be turned into a fairly mechanical process:

1. Adjust the logging default to be more verbose.
2. Determine the maximum figure to set `shared_buffers` to. Start at 25% of system memory. Consider adjusting upward if you're on a recent PostgreSQL version with spread checkpoints and know your workload benefits from giving memory directory to the buffer cache. If you're on a platform where this parameter is not so useful, limit its value or adjust downward accordingly.

If your OS is 32 bit, then the `shared_buffer` value must be less than 2.5 GB, and in case of Windows OS, a larger value won't provide benefits.

3. Estimate your maximum connections generously, since this is a hard limit: clients will be refused connection once it's reached.

4. Start the server with these initial parameters. Note how much memory is still available for the OS filesystem cache:

```
$ free -m
  total    used  free  shared  buffers   cached
Mem: 3934     1695  2239    11    89     811
-/+ buffers/cache: 793   3141
Swap:       4091   0    4091

$ pg_ctl start

$ free -m
  total     used  free shared  buffers   cached
Mem:  3934      1705  2229   22    89    822
-/+ buffers/cache:  793  3141
Swap:         4091  0    4091
```

5. Adjust `effective_cache_size` based on `shared_buffers` plus the OS cache. A value between 50% to 75% of total memory is useful.
6. Divide the OS cache size by `max_connections`, then by 2. This gives you an idea of a maximum reasonable setting for `work_mem`. If your application is not dependent on sort performance, a much lower value than that would be more appropriate.
7. Set `maintenance_work_mem` to around 50 MB per GB of RAM:

```
Set max_wal_size to 1GB.
```

8. In older versions, set the value of `checkpoint_segments`. Here is the formula to calculate `max_wal_size`:

$$max_wal_size = (3 * checkpoint_segments) * 16MB$$

9. If you have server-class hardware with a battery-backed write cache, a setting of 32 would be a better default.
10. If you're using a platform where the default `wal_sync_method` is not safe, change it to one that is.
11. Increase `wal_buffers` to 16 MB.
12. For PostgreSQL versions before 8.4, consider increases to both `default_statistics_target` (to 100, the modern default) and `max_fsm_pages` based on what you know about the database workload.

Once you've set up a number of servers running your type of applications, you should have a better idea what sort of starting values make sense to begin with. The values for `max_wal_size` and `work_mem` in particular can end up being very different from what's suggested here.

Shared server guidelines

If your database server is sharing hardware with another use, particularly the common situation where a database-driven application is installed on the same system, you cannot be nearly as aggressive in your tuning as the method we just looked at. An exact procedure is harder to outline. What you should try to do is use--tuning values for the memory-related values on the lower side of recommended practice:

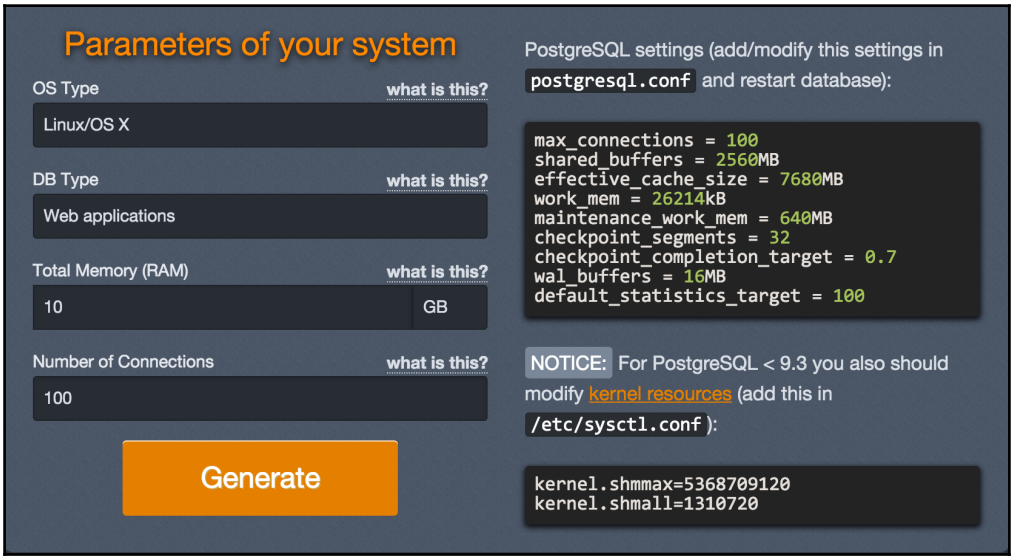
- Only dedicate 10% of RAM to `shared_buffers` at first, even on platforms where more would normally be advised
- Set `effective_cache_size` to 50% or less of system RAM, perhaps less if you know your application is going to be using a lot of it
- Be very stingy about increases to `work_mem`

The other suggestions in the previous section should still hold--using larger values for `max_wal_size` and considering the appropriate choice of `wal_sync_method`, for example, are no different on a shared system than on a dedicated one.

Then, simulate your application running with a full-sized workload, and then measure the available RAM to see if more might be suitable to allocate toward the database. This may be an iterative process, and it certainly should be matched with application-level benchmarking if possible. There's no sense in giving memory to the database on a shared system if the application, or another layer of caching, such as at the connection pooler level, would use it more effectively. That same idea gets reasonable starting settings and tunes iteratively based on monitoring--this works well for a dedicated server too.

pgtune

Starting with PostgreSQL 8.4, the pgtune program--available from `http://pgfoundry.org/projects/pgtune/` can be used to create an initial `postgresql.conf` file for a new server. It allows you to suggest what the intended type of workload is, ranging from a dedicated data warehouse server down to a developer workstation. Based on that input and certain system parameters, such as the amount of RAM in the server, it produces a tuned configuration for the major system parameters following similar methods to those described in this chapter. This is not going to be as accurate as following the guidelines for a dedicated server and measuring everything yourself, but it will get you up and running with a configuration that's generally the right size very quickly. Any reasonably sized `postgresql.conf` should easily outperform the default one, given that it's optimized only for low-shared memory use. There is a web interface (found at `http://pgtune.leopard.in.ua/`) where `postgresql.conf` settings can be generating by prodding some of the parameters:



Summary

There are almost 200 values you might adjust in a PostgreSQL database's configuration, and getting them all right for your application can be quite a project. The guidelines here should get you into the general area that you should start from. They should also help you avoid the most common pitfalls, and give you an idea as to what settings are more likely to be valuable when you do run into trouble. The default values in the server configuration file are very short on logging information and have extremely small memory settings. Every server should get at least a basic round of tuning to work around the worst of the known issues. The memory-based tunables, primarily `shared_buffers` and `work_mem`, need to be adjusted carefully and in unison to make sure your system doesn't run out of memory altogether.

The autovacuum process is also critical in making sure the query planner has the right information to work with, as well as for keeping tables maintained properly. In many cases, the server does not need to be restarted to make a configuration change, and many parameters can even be adjusted on a per-client basis for really fine tuning.

7
Routine Maintenance

PostgreSQL aims to be easy to maintain. But like any database, heavy activity can lead to performance dropping due to overhead. The approach taken to handle concurrent read and write scalability in the database can leave behind significant amounts of data that needs to be cleaned up properly. Understanding why that happens is valuable for modeling just how often related maintenance needs to occur. Another aspect of maintaining any database server is monitoring how well the queries it runs execute.

In this chapter, we will look into the following topics:

- Transaction visibility with multiversion concurrency control
- Vacuum
- Autoanalyze
- Index bloat
- Dump and restore
- Vacuuming the database/table
- CLUSTER
- Reindexing
- Detailed data and index page monitoring
- Monitoring query logs

Transaction visibility with multiversion concurrency control

One decision any database designer needs to make is how to handle the situation where multiple clients are interacting with the same data. PostgreSQL uses a popular approach called **multiversion concurrency control** (**MVCC**) to handle this job. MVCC is also used in BerkeleyDB, SQL Anywhere, Oracle, and many other database products; it's a general technique rather than something specific to PostgreSQL. The introduction to MVCC in the documentation at `http://www.postgresql.org/docs/current/static/mvcc-intro.html` makes the concept sound more complicated than it is. It's easier to understand with some simple examples, which we'll get to shortly.

Visibility computation internals

Understanding how this is implemented in the server is helpful both to help predict how statements will act and so that you can interpret the diagnostic information available.

As transactions are created in the database, PostgreSQL advances a transaction ID counter, usually just called XID, to keep track of them. When you insert a row into the database, or update an existing row, the new row created by that operation saves the session's transaction ID into a field named the insertion XID, also referred to as `xmin`. This is the minimum XID capable of seeing this bit of information, once it's been committed:

```
postgres=# CREATE TABLE foo(bar int);
CREATE TABLE
postgres=# INSERT INTO foo VALUES(1);
INSERT 0 1
postgres=# SELECT xmin,xmax,bar from foo;
   xmin   | xmax | bar
----------+------+-----
 10708401 |    0 |   1
(1 row)
```

When a query or statement starts (tracking at either granularity can happen, depending on what mode you're in--more on this later), it notes the current transaction ID as a hard line for what it should consider visible. As rows are considered for inclusion in the query results, if they are committed and their insertion `xmin` is less than that number, they show up in the query.

A similar mechanism handles deletion. Each row has a delete XID, also referred to as `xmax`, that starts blank to indicate that the row is live. When you delete a row, the current transaction ID becomes its `xmax` to say it should no longer be visible after that point in time (or, technically, that point in transaction history). As queries that are running consider rows for visibility, they note any non-blank `xmax` and only include the row if the commit of that deletion happened before the query's start XID.

The `xmin` and `xmax` data is essentially the visibility lifetime of the row in transaction ID terms. The row is only visible from the perspective of a query whose transaction number is between those two values. What you might not expect is that you can actually look at most of these internal details to see how they work as you run experiments, including the current transaction ID (`txid_current`) and the `xmin`/`xmax` data for every row you can see.

The computations are actually a bit more complicated than this because they also consider the transactions that were in progress when the query was started. You can see exactly what information is saved by a query in order to perform its visibility computations by using the `txid_current_snapshot()` function.

Updates

Let's create a simple table with one row to get started:

```
postgres=# CREATE TABLE t (s SERIAL, i INTEGER);
CREATE TABLE
postgres=# INSERT INTO t(i) VALUES(0);
INSERT 0 1
postgres=# SELECT *, xmin, xmax FROM t;
 s | i |   xmin   | xmax
---+---+----------+------
 1 | 0 | 10420499 |    0
(1 row)

postgres=# SELECT txid_current();
 txid_current
--------------
     10420500
(1 row)
```

So there's the new row inserted, with the transaction ID `xmin` active for the statement that inserted it. There's no maximum yet because nothing has ever updated or deleted this row.

Now run `psql`, start a transaction with `BEGIN`, and update the row--but don't commit it just yet:

```
postgres=# BEGIN;
BEGIN
postgres=# SELECT txid_current();
 txid_current
--------------
     10420501
(1 row)

postgres=# UPDATE t SET i=100 WHERE s=1;
UPDATE 1
postgres=# SELECT *, xmin, xmax FROM t;
 s |  i  |   xmin   | xmax
---+-----+----------+------
 1 | 100 | 10420501 |    0
(1 row)
```

From the perspective of this session, the row has been changed. But since it has not been committed yet, other sessions will not see it. You can prove that by connecting to the database from another client session and looking at `t`; it will show the original value still:

```
postgres=# SELECT *, xmin, xmax FROM t;
 s | i |   xmin   |   xmax
---+---+----------+----------
 1 | 0 | 10420499 | 10420501
(1 row)
```

But it already knows that once transaction 10420501 commits, this version of the row is obsolete.

This behavior is the essence of MVCC: each database client session is allowed to make changes to a table, but it doesn't become visible to other sessions until the transaction commits. And even after that point, sessions that are already open will not normally see that change until they are finished. Similarly, once a session starts a transaction, it's blind to most changes made to the database after that point, and its own changes are kept private until it commits. And even those aren't considered final, even in the session that made the change--if you `ROLLBACK` its transaction, the original values had better still be around to return to. In this case, even though the last row here is shown with a lifetime ending with an `xmax` value of 10420501, if that transaction rolls back that doesn't matter.

As you might guess from studying the example, the server actually makes a second row on disk when you issue an `UPDATE` statement. Obviously the original one can't be overwritten, given this behavior. It has no choice but to duplicate the original row, apply the update, and then save this new version. The change from the old to the new version is marked with a transaction identification number, which is what's used to determine which sessions that the updated row should be visible to.

The first thing to recognize, then, is that an `UPDATE` statement consists of the following steps:

1. Read the original row in.
2. Change the fields adjusted by the `UPDATE` statement.
3. Save the new row into a newly allocated location with a new transaction ID.
4. When no one else is using it anymore, `VACUUM` will delete the original row.

Anything that you `UPDATE` will temporarily take up twice as much disk space. It's not just overwriting the original row with the new values; both old and new versions will be available for some time.

Row lock conflicts

There is one tricky part left here. At the point the preceding process stopped, the first session has an open transaction trying to update the row using `WHERE s=1`. What happens if we try to do something similar in the second session before it's either been committed or rolled back?

```
$ psql -c "UPDATE t SET i=i+1 WHERE s=1;"
```

Guess what? This statement will hang! The reason is that we've reached the limits of how many sessions can be isolated from one another safely. Once two sessions are both trying to update the same row, some more complicated rules come into play.

Whenever you try to grab a lock on a specific row--which includes `UPDATE`, `DELETE`, and the locks `SELECT FOR UPDATE`/`SELECT FOR DELETE` obtain--it has to wait for anyone that already has a lock on that row.

Details about `SELECT for UPDATE/DELETE` can be found at `http://www.postgresql.org/docs/current/static/sql-select.html#SQL-FOR-UPDATE-SHARE`.

Once the original locker completes its work, then some decisions have to be made.

If the transaction holding the lock rolled back, then no harm done; the newer one continues the originally planned work. But if the original row the second session was trying to update was changed, then what happens next depends on the configuration of your session.

PostgreSQL operates in two modes for resolving the order of operations in these situations. In `Read Committed` mode, the default, having another row get changed underneath of a session isn't a disaster. What the server will now do is start the original work it was planning to do again with the new copy of the row. If the old row was selected via a `WHERE` clause, the new one will be checked to see if that condition still applies. If not, the new copy is now ignored--that `WHERE` clause doesn't find it anymore. But if all of the reasons for why that row was being updated still seem to apply, the planned update will then just execute against the modified copy.

Returning to the example again for a second, if the original transaction commits, it will be the same as if this order of events happened:

```
UPDATE t SET i=100 WHERE s=1;
UPDATE t SET i=i+1 WHERE s=1;
```

This means that `i=101` is at the end. This is despite the fact that at the point where the second statement started, the other update wasn't complete or visible yet. It will wait for the update that is in the middle of completing and then add its own work on top of that. This is because `Read Committed` gets a new snapshot of database activity that it might want to pay attention to at the start of every statement. In cases where only reads are being done, this will be a consistent view throughout the whole transaction. In the event of changes by other sessions, some of that new data can enter into your open session this way.

Serialization

Now, if you are horrified by the idea that `UPDATE` and `DELETE` statements made by other sessions will leak into your view of the database state, even within a transaction, what you probably want is the other transaction isolation method, `serializable`.

There are actually two slightly different MVCC modes available in PostgreSQL to help address different requirements here. These reflect two behaviors from the SQL standard, introduced at `http://www.postgresql.org/docs/current/static/transaction-iso.html`. The problems described there, such as non-repeatable and phantom reads, require using the database's more complicated serializable mode to avoid, rather than the simpler and default read committed transaction mode. It's worth reading about those problems, because they tend to haunt complicated applications that never consider the sort of race conditions among database operations they occur during.

Following the same basic example, where an open session has an uncommitted `UPDATE` when a second tries to touch the same row, serialization initially works the same way as `Read Committed`. If `UPDATE`, `DELETE`, or `lock` for a row that's already locked is needed, the session waits for that to clear. If the transaction rolled back or didn't touch the data in the row, everything proceeds as in the cases already explained. But if the first transaction commits and it actually modified the row with `UPDATE` or `DELETE` rather than just having a lock on it, you're done. You will get this error:

```
ERROR: could not serialize access due to concurrent updates
```

Your application has no choice here but to rollback the entire transaction it was in the middle of and start over. This level of isolation is normally required if you have a transaction that executes multiple statements that absolutely must operate on an identical view of the database. `Read Committed` lets the visibility snapshot slip forward for things such as updates to commonly touched rows as each statement is processed. A serialized session aims to keep a consistent view for the whole transaction, and if something happens to make that possible, it throws an error rather than risk a problem.

Deletions

There's little different about what happens during `DELETE`. Close out any open sessions you might have with the unfinished transaction. Now let's start another transaction to delete a row:

```
postgres=# BEGIN;
BEGIN
postgres=# DELETE FROM t WHERE s=1;
DELETE 1
postgres=# SELECT * FROM t;
 s | i
---+---
(0 rows)
```

Once again, if you look at this table from another session, then, because it's not been committed yet, the row deleted here will still be there, just with an `xmax` value set:

```
postgres=# SELECT *, xmin, xmax FROM t;
 s |  i  |   xmin   |   xmax
---+-----+----------+----------
 1 | 100 | 10420504 | 10420507
(1 row)
```

Only in the first session has the row been deleted. This brings us to a second form of visibility that needs to be considered: when a row is deleted, it can't actually go away until any session that might need to see the row has ended. So, when you commit a delete, it doesn't actually change the record itself. Instead, it updates the visibility information around that row to say "this is deleted and may not be visible everywhere", and then each client who checks it will need to make that decision for itself. These are called dead rows in the database; regular rows are described as live.

Another interesting thing to note here is that the way this row looks is exactly like the one left behind by an `UPDATE` statement; it's just another row with an `xmax` set waiting for a commit, after which it can be removed. Because of MVCC, whether you update or delete a row, you get the same form of dead row left over at the end. The only exception is if you are updating in a situation where the HOT technique in the database can be used to speed your update, something covered in the following section. That may be a bit less expensive.

Advantages of MVCC

Why go through all this trouble? The transaction isolation of sessions from one another has an extremely valuable result: it avoids locking many resources that can block other clients from doing their work. The locks required to run a query and read data do not conflict with the ones that are needed to write to the database. Reading never blocks writing, and writing never blocks reading. This model scales quite well into large client counts without running into lock contentions for either tables or rows. You can still explicitly lock a row or table if needed, by using methods including the `LOCK` statement and the lock upgrades implemented in `SELECT FOR SHARE` and `SELECT FOR UPDATE` statements.

Disadvantages of MVCC

The main disadvantage of MVCC is the background work necessary to clean up its visibility data and keep disk space under control, the topic of the next section.

You should also be aware that MVCC cannot make all possible concerns about interactions between sessions and ordering magically go away. In fact, not seeing data that another session has been altering until later can introduce its own new class of problem for applications that are used to traditional locking approaches. You've seen how `UPDATE` and `DELETE` order might happen in an unexpected way with a regular `Read Committed` MVCC session, and you can't just make those go away with serializable without also preparing for some transactions to fail badly.

Transaction ID wraparound

The implementation of MVCC in PostgreSQL uses a transaction ID that is 32 bits in size. It's impractical to make it any longer because, as you've just seen, visibility information is stored in each row as `xmin` and `xmax` values. Having a larger ID would therefore increase the size of each row by a significant amount. A signed 32-bit number can only handle a range of about 2 billion transactions before rolling over to zero. When it exceeds its range, transactions that used to appear in the past will now appear to be from the future, which, as you might imagine, will wreak havoc. If this transaction wraparound ever happens to your database, it will fail to operate sanely, and the database will go far out of its way to keep that from happening.

The way that the 32-bit XID is mapped to handle many billions of transactions is that each table and database has a reference XID, and every other XID is relative to it. This gives an effective range of 2 billion transactions before and after that value. You can see how old these reference XID numbers are relative to current activity, starting with the oldest active entries, like this:

```
SELECT relname,age(relfrozenxid) FROM pg_class WHERE relkind='r'
  ORDER BY age(relfrozenxid) DESC;
SELECT datname,age(datfrozenxid) FROM pg_database ORDER BY
  age(datfrozenxid) DESC;
```

One of the things `VACUUM` does is push forward the frozen value once a threshold of transactions have passed, set by the `vacuum_freeze_max_age` parameter, and autovacuum has its own setting as `autovacuum_freeze_max_age`. This maintenance is also critical to cleaning up the commit log information stored in the `pg_clog` directory. Some transactions will *fall off the back* here, if they have a transaction ID so old that it can't be represented relative to the new reference values. These will have their XID replaced by a special magic value called the FrozenXID. Once that happens, those transactions will appear in the past relative to all active transactions.

The values for these parameters are set very conservatively by default--things start to be frozen after only 200 million transactions, even though wraparound isn't a concern until 2 billion. One reason for this is to keep the commit log disk space from growing excessively. At the default value, it should never take up more than 50 MB, while increasing the free age to its maximum (2 billion) will instead use up to 500 MB of space. If you have large tables where that disk usage is trivial and you don't need to run vacuum regularly in order to reclaim space, increasing the maximum free age parameters can be helpful to keep autovacuum from doing more work than it has to in freezing your tables.

Adjusting the minimum values for when freezing happens is only really a good idea in one situation: just after a form of bulk loading where there's no way you will be modifying the transactions you just added. The problem with using a low `vacuum_freeze_min_age` during normal use is that the freezing process will replace transaction information with the special FrozenXID, which loses potentially useful data about when those transactions committed. If you ever end up in the unfortunate position where you have to dig into transaction ordering forensics to track down how a problem happened, discovering that the XID data that would have helped sort that out has been frozen is not good news.

Therefore, the basic trade-off that you need to decide on is how much of this detailed transaction number diagnostic information you want to keep around, knowing that if you keep too much of it around, it will take up more disk space and make `pg_clog` less efficient. At the same time, deciding to delay vacuum for as long as possible also means that when it does happen, you will need to do a lot of work. More frequent `VACUUM` work means more regular small disruptions, though, so some prefer to just schedule that cleanup rather than risk it popping up at a bad time. It's tricky to provide general advice that works for everyone.

Starting in PostgreSQL 8.3, you can use the system functions described at `http://www.postgresql.org/docs/current/static/functions-info.html` to determine the latest transaction ID in use on the server. `txid_current()` is the simplest value to inspect, as shown in some of the early examples in this chapter. Watching that number grow over time lets you estimate how many transactions are occurring on your system. In order to compensate for transaction wraparound, the value returned is actually 64 bits wide, with the top 32 bits holding a counter of how many times the XID has crossed back to zero. This makes the value returned by `txid_current()` always move forward, even as the actual XID rolls over its upper limit.

Vacuum

If you surveyed a set of experienced PostgreSQL database administrators and asked what part of database maintenance requires the most work, the word vacuum would pop up quite often in those conversations. A combination of complexity and some unfortunate terminology choices makes this particular area of database management quite prone to problems and misunderstandings, relative to how little trouble most parts of PostgreSQL administration are.

The need for vacuum stems from the visibility approach that we just looked at. The root problem is that clients executing `UPDATE` or `DELETE` operations don't know everything happening on the server. They can't make the decision about whether the original, now dead, row can truly be deleted, which is only possible when there are in fact no clients left who need to see it. And sometimes the new rows are never actually committed, leaving behind a different sort of dead row: a rolled back one that never becomes visible.

Cleaning up after all these situations that produce dead rows (`UPDATE`, `DELETE`, `ROLLBACK`) is the job for an operation named vacuuming. It inspects what transactions have finished and cleans up rows that cannot be visible to any current or future query. It also handles the transaction ID wraparound quirks of the database.

Vacuum implementation

There are two primary sets of internal structures that serve as input and output from data vacuuming.

On the input side, each database row includes status flags called hint bits that track whether the transaction that updated the `xmin` or `xmax` values is known to be committed or aborted yet. The actual commit logs (`pg_clog` and sometimes `pg_subtrans`) are consulted to confirm the hint bits' transaction states if they are not set yet. Vacuum also writes part of the row itself to these hint bits, as it confirms the known state has changed.

Besides updating the data pages themselves, the additional output from vacuum is updated information in the free space map.

Regular vacuum

Vacuum does a scan of each table and index, looking for rows that can no longer be visible. The row hint bits are updated to reflect any information discovered, and newly freed space is inserted into the free space map.

Once the free space map for a table has entries on it, new allocations for this table will reuse that existing space when possible, rather than allocating new space from the operating system.

Returning free disk space

Usually, administrators are surprised to find that there is no reduction in disk space from using vacuum. There is only one situation where vacuum can actually reduce the size of a table. If the last data page of a table is empty, completely free of rows, and it's possible to obtain an exclusive lock on the table, vacuum executes a special disk release process. It will scan backward from the end, returning all pages it finds to the operating system as newly free space, until it finds a data page that isn't empty.

Accordingly, it's only possible for a regular vacuum to reduce the size of a table if it has a contiguous chunk of free space at the end of the table. One common situation that can produce this pattern of disk usage is when a table includes a fixed time period of data, with new records constantly inserted and older ones periodically deleted. If those deletions are cleaned up properly with vacuum, eventually you can expect that the table size will reach a steady-state.

In most other situations, vacuum will never release disk space. In regular operations, your goal should be for vacuum to run often enough that there is never a large amount of free space in any table to release. However, this is sometimes unavoidable; a large deletion of historical data is one way to end up with a table with lots of free space at its beginning.

Full vacuum

In versions of PostgreSQL before 9.0, the `VACUUM FULL` command, which is never executed by autovacuum, takes a more aggressive approach to space reuse. It compacts tables by moving rows to the earliest page they can be placed into. This makes the one situation where vacuum can release disk space, when it's all at the end of the table, very likely. If you have any dead rows in your table, `VACUUM FULL` will relocate them one at a time to an earlier part of the table, and then shrink it accordingly once the end is all empty space.

There are two major downsides to doing that. The first is that it's very time and resource intensive. `VACUUM FULL` is likely to take a very long time if your table is larger, and during that time it will have an exclusive lock on the table. That's a very bad combination. The second issue is index bloat, covered in more detail in the following section, and equally troublesome.

PostgreSQL 9.5 introduced a new parameter, `jobs`, to run `VACUUM` in parallel. It speeds up the process, but the downside is that it increases the load on the server.

PostgreSQL 9.0 has introduced a rewritten `VACUUM FULL` command that is modeled on the `CLUSTER` implementation of earlier versions. It does not have the same issues described in this section. Instead, it has the limitation that you need enough disk space to hold a fresh copy of the table in order for it to work.

Removing tuple timeout

As discussed, `VACUUM` physically deletes old tuples until the last transaction can see them. But sometimes the transaction takes too long and holds the tuples for a very long time. A new configuration parameter was introduced in PostgreSQL 9.6 called `old_snapshot_threshold`. This parameter can be configured to specify how long this snapshot is valid for. After that time, the dead tuple is a candidate for deletion, and if the transaction uses that tuple, it gets an error.

HOT

One of the major performance features added to PostgreSQL 8.3 is **HOT** (**Heap Only Tuples**). HOT allows the reuse of space left behind by dead rows resulting from the `DELETE` or `UPDATE` operations under some common conditions. The specific case that HOT helps with is when you are making changes to a row that does not update any of its indexed columns. When this happens, if the new second copy of the row can be fitted onto the same page as the existing one, it's put there without a new index entry being allocated for it. Instead, it's added to a list on that existing data block, if there's space, in a structure called an update chain.

In order to make this case more likely to occur, HOT also does a single-block mini-vacuum as part of its work, whenever it's practical to do so. This combination--doing a single block `VACUUM` and avoiding changes to the index structure--allows an `UPDATE` operation that used to touch both the data (heap) block and an index block to instead only touch the heap, hence the name (Heap Only).

Note that this doesn't completely eliminate the need for other forms of `VACUUM` because HOT may not be able to do the single-block vacuum. It also may not clean up after all possible cases involving aborted transactions.

The normal way to check if you are getting the benefit of HOT updates is to monitor `pg_stat_user_tables` and compare the counts for `n_tup_upd` (regular updates) versus `n_tup_hot_upd`.

One of the ways to make HOT more effective on your tables is to use a larger fill factor setting when creating them. Having extra empty space available in blocks gives HOT more room to do the shuffling around that it is good at without having to move data to new disk pages.

Cost-based vacuuming

Running a regular `VACUUM` command is a pretty intensive operation. It's likely to become the most resource-intensive process running on your server, and if autovacuum ran in that fashion, it would be unusable in many environments.

Fortunately, there's another approach available. Cost-based vacuuming limits the amount of disk I/O any given vacuum or autovacuum process is expected to do per unit of time. It works by assigning an estimated cost to every I/O operation, accumulating a total for each operation it performs, then pausing once an upper limit on cost per iteration is exceeded.

The cost estimation presumes three basic operations that vacuum performs, each with its own presumed amount of work to handle:

- **Page hit**: A data page was needed, but found in the shared buffer cache. It merely needs to be scanned for dead rows, with no physical I/O involved. This has a reference cost of 1 unit via `vacuum_cost_page_hit`, and you shouldn't ever adjust it.
- **Page miss**: The data page needed isn't already buffered, so it must be read from disk before it can be scanned. This default set via `vacuum_cost_page_miss` makes this cost 10.
- **Page dirty**: After being scanned, the page needed to be modified; it's now dirty and must be written out to disk. The default `vacuum_cost_page_dirty` is 20.

The preceding parameters are set globally for all tables in the database and impact both manual vacuum and autovacuum (which is described in more detail in the next section). The way these are described, such as saying things are read from disk, isn't quite right; they're read from the operating system cache, which may even have them in regular memory already. Partly because of that class of problem, it's difficult to adjust these values, and only recommended for very experienced PostgreSQL administrators to attempt. Measuring the real-world costs of these operations is hard, and there's not much evidence yet that doing so will improve significantly over the theoretical model used here. It's much better to start vacuum adjustment for workload by tweaking the higher-level settings--described next--instead of these.

A manual vacuum worker will execute until it has exceeded `vacuum_cost_limit` of the estimated I/O, defaulting to 200 units of work. At that point, it will then sleep for `vacuum_cost_delay` milliseconds, defaulting to 0--which disables the cost delay feature altogether with manually executed `VACUUM` statements.

However, autovacuum workers have their own parameters that work the same way. `autovacuum_vacuum_cost_limit` defaults to `-1`, which is shorthand for saying that they use the same cost limit structure (the ratios between individual costs) as manual vacuum. The main way that autovacuum diverges from a manual one is it defaults to the following cost delay:

```
autovacuum_vacuum_cost_delay = 20ms
```

So where a regular `VACUUM` will just keep going each time it accumulates 200 or more cost units of operations, autovacuum will instead sleep for 20 ms each time it reaches that point.

Note that if you want to adjust a manual `VACUUM` to run with the cost logic, you don't need to adjust the server `postgresql.conf` file--this is a user setting. You can tweak it before issuing any manual vacuum and it will effectively limit its impact for just that session:

```
postgres=# SET vacuum_cost_delay='20';
postgres=# show vacuum_cost_delay;
vacuum_cost_delay
-------------------
20ms
postgres=# VACUUM;
```

It's extremely difficult to turn all these estimated cost figures into a predicted real-world I/O figure. But if you combine adjusting this value with monitoring both the database and the I/O load at the operating system, it does allow some iterative tuning methods.

Autovacuum

With vacuum being so critical to performance, automating it as much as possible has been one of the most regularly improved aspects of the database during the last few years of development. Major advancements in each version include the following:

- **3, 7.4, 8.0**: autovacuum is available as a `contrib` module (`contrib/pg_autovacuum`). It requires custom scripting to keep it running all the time.
- **1**: The autovacuum daemon is introduced as a distinct process managed by the server, always running whenever the server is when enabled.

- **2**: Last vacuum and last analyze times are saved to `pg_stat_all_tables` so that you can monitor activity better.
- **3**: Autovacuum is turned on by default, and it can run multiple workers at once. Additional log detail is possible via `log_autovacuum_min_duration`.
- **4**: The disk-based free space map means no `max_fsm_pages` overflow. Control of per-table behavior is switched to use table storage parameters.
- **1**: There is no waiting time if it cannot acquire the lock on the table. Instead, it will be vacuumed later.
- **4**: `autovacuum_work_mem` controls the amount of memory used by the autovacuum worker.
- **9.6**: The system avoids re-vacuuming the frozen tuple-only pages.

Also note that in versions before 8.3, when it became enabled by default, running autovacuum also required some optional table statistics to be collected. The common recipe to make this feature work in 8.2 and earlier is:

```
stats_start_collector = on
stats_block_level = on
stats_row_level = on
autovacuum = on
```

Autovacuum logging

In versions before 8.3, autovacuum mentioned when it was visiting each database in the logs, but did not give much information beyond that. It's possible to watch it more directly by setting:

```
log_min_messages =debug2
```

But be warned that this level of debugging could potentially log a line or more for every statement your server executes, which is both a major performance drag and a drain on disk space. It's generally practical to use this much logging to track down an autovacuum problem only when you know the server is nearly idle.

In current versions, you can easily monitor the daemon's activity by setting `log_autovacuum_min_duration` to some number of milliseconds. It defaults to -1, turning logging off. When set to a value `>=0`, any autovacuum action taking longer than that amount of time will be logged. Since autovacuum can run quite often doing trivial operations you don't necessarily care about, setting this to a moderate number of milliseconds (for example 1000=1 second) is a good practice to follow.

Autovacuum monitoring

Since log files can easily get lost, the best way to approach making sure autovacuum is doing what it should is to monitor what tables it's worked on instead:

```
SELECT schemaname,relname,last_autovacuum,last_autoanalyze
FROM pg_stat_all_tables;
```

You can trim down the information here by changing all tables in the preceding command to name one of the alternate views; `pg_stat_sys_tables` shows only system tables, while `pg_stat_user_tables` shows only your user tables.

Autovacuum triggering

If any database is in danger of overflowing the maximum allowed transaction ID, autovacuum will first work on that situation. Autovacuum will kick in to work on XID wraparound even if you've disabled it. It looks at the transaction age, a measure of how many transactions have happened since a database was last reset to use a new XID basis. You can monitor the highest age on any database and see them for individual databases or tables using queries like these:

```
SELECT max(age(datfrozenxid)) FROM pg_database;
SELECT datname,age(datfrozenxid) from pg_database
ORDER BY age(datfrozenxid) DESC;
SELECT relname, age(relfrozenxid) FROM pg_class
WHERE relkind = 'r'
ORDER BY age(relfrozenxid) DESC;
```

Once that check is passed without requiring any work, autovacuum next considers the statistics collected for each table during routine operations. Each table in the database has an estimated live and dead row count, as well as an estimated total number of rows (its tuple count). These are combined to form a threshold for how many rows must have changed before autovacuum processes the table, like this:

```
autovacuum_vacuum_scale_factor * tuples +
  autovacuum_vacuum_threshold
```

Since `autovacuum_vacuum_threshold` is small by default (50), once your table is of modest size it will no longer need the main component here. Instead, you'll get autovacuum triggering via the scale factor, which defaults to triggering after changes impacting 20% of the table. Particularly on large tables, that value can be much too high--20% of a billion row table is quite a bit of dead space to be hanging around without vacuum cleanup.

This query will show you whether each table in your database qualifies for autovacuum processing, and exactly where that line is relative to the total amount of changes in there right now:

```
postgres=# SELECT *,
n_dead_tup > av_threshold AS "av_needed",
CASE WHEN reltuples > 0
THEN round(100.0 * n_dead_tup / (reltuples))
ELSE 0
END
AS pct_dead
FROM
(SELECT
N.nspname,
C.relname,
pg_stat_get_tuples_inserted(C.oid) AS n_tup_ins,
pg_stat_get_tuples_updated(C.oid) AS n_tup_upd,
pg_stat_get_tuples_deleted(C.oid) AS n_tup_del,
pg_stat_get_live_tuples(C.oid) AS n_live_tup,
pg_stat_get_dead_tuples(C.oid) AS n_dead_tup,
C.reltuples AS reltuples,
round(current_setting('autovacuum_vacuum_threshold')::integer
+ current_setting('autovacuum_vacuum_scale_factor')::numeric * C.reltuples)
AS av_threshold, date_trunc('minute',greatest(pg_stat_get_last_vacuum_time(C.oid),
pg_stat_get_last_autovacuum_time(C.oid))) AS last_vacuum, date_trunc('minute',greatest(pg_stat_get_last_analyze_time(C.oid),
pg_stat_get_last_analyze_time(C.oid))) AS last_analyze
FROM pg_class C
LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
WHERE C.relkind IN ('r', 't')
AND N.nspname NOT IN ('pg_catalog', 'information_schema') AND
N.nspname !~ '^pg_toast'
) AS av
ORDER BY av_needed DESC,n_dead_tup DESC;
 nspname | relname | n_tup_ins | n_tup_upd | n_tup_del | n_live_tup | n_dead_tup | reltuples | av_threshold | last_vacuum | last_analyze | av_needed | pct_dead
---------+---------+-----------+-----------+-----------+------------+------------+-----------+--------------+-------------+--------------+-----------+----------
 public  | t       |         2 |         2 |         5 |          1 |          3 |         0 |           50 |             |              | f         |        0
 public  | foo     |         3 |         0 |         2 |          1 |          2 |         0 |           50 |             |              | f         |        0
 public  | vehicle |         0 |         0 |         0 |          0 |          0 |         0 |           50 |             |              | f         |        0
 public  | car     |         0 |         0 |         0 |          0 |          0 |         0 |           50 |             |              | f         |        0
 public  | test    |         0 |         0 |         0 |          0 |          0 |    100000 |        20050 |             |              | f         |        0
(5 rows)
```

This simple query doesn't pay any attention to the fact that you can actually customize autovacuum parameters for each table.

Per-table adjustments

Sometimes individual tables have vacuum requirements that don't match the rest of the database, and it is possible to tweak the autovacuum behavior for each table.

The pgAdmin III GUI interface has a simple interface for this sort of adjustment, using its **Vacuum settings** tab for an individual table you're viewing. This is one area where adjustments with the command line can be hard to get right.

As of 8.4, per-table autovacuum information is stored as standard table storage parameters, similar to things such as the table's fill factor. These are normally set at table creation time (`http://www.postgresql.org/docs/current/static/sql-createtable.html`).

The `psql` utility displays table storage parameters when using its `d+` mode, as shown by this example, which demonstrates how to disable autovacuum on a table:

```
postgres=# ALTER TABLE t SET (autovacuum_enabled=false);
ALTER TABLE
postgres=# \d+ t
                                        Table "public.t"
 Column |  Type   |                   Modifiers                   | Storage | Stats target | Description
--------+---------+-----------------------------------------------+---------+--------------+-------------
 s      | integer | not null default nextval('t_s_seq'::regclass) | plain   |              |
 i      | integer |                                               | plain   |              |
Options: autovacuum_enabled=false
```

Common vacuum and autovacuum problems

In many of the situations where `VACUUM` and related cleanup, such as `VACUUM FULL` or `CLUSTER`,need to happen, such as index bloat and excessive dead tuples taking up disk space, the work involved can be so intensive that administrators decide that `VACUUM` is something to be avoided. This is actually the opposite of what should be concluded. In reality, the answer to most vacuum-related problems is to vacuum more often. That reduces the amount of work done by each individual vacuum, and it promotes earlier re-use of space that keeps table sizes from getting so big in the first place. Do not reach the wrong conclusion from a painful vacuum situation--that almost always means you didn't vacuum often enough, not that you should avoid running vacuum as much as possible in the future.

There are, however, a few known issues that you can run into with vacuum and autovacuum that will inhibit attempts to follow recommended vacuum practice, or otherwise seem unexpected.

Autovacuum is running even though it was turned off

If your table is approaching transaction ID wraparound, autovacuum will process it regardless of whether it's turned on or off globally, or for that table. This is one of the many reasons why running with autovacuum off, rather than tuning it to run smoothly all the time, is dangerous. If you haven't done that tuning, and just turned it off instead, when autovacuum does start anyway to deal with wraparound it will not be adjusted appropriately to avoid interference with any activity going on.

Autovacuum is constantly running

While autovacuum stays running all the time, it shouldn't be doing active work all the time. If it is, there are two common issues to investigate.

Each time vacuum runs out of its allocated space for `maintenance_work_mem`, it needs to start over and make an additional pass over the table. Accordingly, if you have set this tunable too low relative to the size needed to vacuum your tables, vacuum can run much less efficiently than it should, and therefore autovacuum will be running much more often just to keep up.

Another autovacuum problem related to having a large number of databases is that autovacuum tries to start one worker on each database every `autovacuum_naptime` second (defaulting to one minute, up the maximum set by `autovacuum_max_workers`. So, if you have 60 databases, a new worker will be started every second with the default parameters. You probably need to increase the nap time value if you have more than dozens of databases in your cluster.

Out of memory errors

If you are too aggressive with setting `maintenance_work_mem`, you may discover that autovacuum periodically causes out of memory errors on your system. You can typically identify that situation because the amount of memory listed in the logs for the allocation failure attempt will match that database setting.

Don't forget that up to `autovacuum_max_workers` can be active, each using that much memory. When estimating the available RAM on the system, it's better to err on the small side initially with this setting. Ramp it up after you've done enough monitoring to know what the peak memory usage on your server is.

In PostgreSQL 9.4, a new configuration parameter was introduced called `autovacuum_work_mem`. This is used to control the amount of memory used to control the memory usage of autovacuum.

Not keeping up on a busy server

The default value for `autovacuum_vacuum_cost_delay` of 20 ms is a medium one appropriate for a smaller system. If you have a busy server, you can easily discover that autovacuum never quite keeps up with incoming traffic. You can drop this setting significantly in order to give autovacuum more opportunities to keep pace with your server, particularly if you have a system that's capable of good I/O performance.

Large servers will commonly run with a cost delay in the 1 ms to 5 ms range. Your first response to not having autovacuum run often enough to keep up should be to reduce this parameter until you've reached that range. Since operating system timers sometimes won't delay for less than 10 ms at a time, at some point you may need to switch to increasing `autovacuum_cost_limit` instead to get additional autovacuum work accomplished. That's the general order in which to tune these two settings: the delay first, down to either 10 ms if that's the best your OS will do or down to 1 ms, and then the cost limit for additional increases in vacuum activity. Adjusting the rest of the parameters should be considered a last resort, as they are more complicated, more likely to be functional at the default settings, and easier to set badly.

Autovacuum is too disruptive

The opposite case is also possible: autovacuum can easily consume too much I/O capacity on a server without a lot of it to spare. In that case, increases of `autovacuum_vacuum_cost_delay` to as high as 100 ms are common. This is the preferred way to deal with the situation where vacuum seems to be consuming too many resources. Increasing the cost delay to make autovacuum's actions less disruptive is the way you should be dealing with the situation where it seems you can't run vacuum during your busy times.

Long-running transactions

One problem that can block all vacuum-related activity on a server is a long-running transaction. If you have a query running that needs to see very old information, the server can't clean up past that point until it completes. The easiest way to monitor long-running transactions is to watch for them in `pg_stat_activity`, using a query such as follows:

```
postgres=# SELECT  pid,
        current_timestamp - xact_start AS xact_runtime,
        query
        FROM pg_stat_activity
        ORDER BY xact_start;
 pid  | xact_runtime |                          query
------+--------------+-------------------------------------------------------
 6426 | 00:00:00     | SELECT  pid,                                          +
      |              |         current_timestamp - xact_start AS xact_runtime,+
      |              |         query                                         +
      |              |         FROM pg_stat_activity                         +
      |              |         ORDER BY xact_start;
(1 row)
```

This will show just how long any open transaction has been running.

Free space map exhaustion

One of the major problems in PostgreSQL versions up to 8.3 is that the free space map is stored in shared memory, and is of a fixed size. If your database has a large number of deletions, updates, and/or rollbacks in between when a manual or automatic vacuum runs, it's possible for this **free space map** (**FSM**) to run out of room and therefore not be able to track additional free space. This can quickly lead to catastrophic amounts of table and index bloat.

You can track how much of the FSM is being used only through watching the output of manual vacuums. Here is an example showing the information provided by an 8.3 server:

```
postgres=# VACUUM VERBOSE;
INFO: free space map contains 49 pages in 62 relations
DETAIL: A total of 992 page slots are in use (including overhead).
992 page slots are required to track all free space.
Current limits are: 204800 page slots, 1000 relations,
using 1265 kB.
```

In earlier versions, you'd see the totals but not the limits for reference. You can always check them using the `SHOW` command:

```
postgres=# SHOW max_fsm_pages;
max_fsm_pages
---------------
204800
```

Now in PostgreSQL free space is recorded in `_fsm` relation forks so `max_fsm_pages` and `max_fsm_relations` settings have been removed.

In either case, you want to confirm the total page slots used (992 in this example) with the upper limit (204800 here). If those numbers are even close, you should be thinking about increasing `max_fsm_pages` by a significant amount when you can next arrange to restart the server. One common approach is to double it each time you run into this situation, at least until you've reached the millions of slots, then continue increases at a more moderate pace. It's not unusual for a large system to set this parameter to 10 M or more. And if your number of relations is close to that limit, set by `max_fsm_relations`, you similarly need to increase it, albeit not as aggressively in most cases.

If you are analyzing your log files using `pgFouine`, it has a vacuum logging reader that knows how to look for the output from `VACUUM VERBOSE` in the logs. If you're using that software and a PostgreSQL version before 8.4, you should consider generating its FSM report regularly to monitor this area.

Recovering from major problems

If you ever find yourself with so much free space that you need to reclaim that `VACUUM FULL` seems the only way out, you should consider a few things.

First, can you use `CLUSTER` to rebuild the table? It essentially makes a clean second copy of the table and then substitutes it for the original once finished. While this still requires a long lock and some processing time, this is far more efficient in most cases than `VACUUM FULL`. Note that `CLUSTER` is only completely safe for this use as of PostgreSQL 8.3. In earlier versions, a common trick was to use `ALTER TABLE` in a way that would rewrite the entire table with a new version.

Second, how can you adjust your vacuum and/or autovacuum strategy to keep such a large problem from showing up again? Generally, the answer to this is "vacuum more often" rather than less.

If index bloat is the main issue, perhaps aggravated by a past `VACUUM FULL` run, `REINDEX` can be useful instead of `CLUSTER` for recovering from that. `REINDEX` doesn't require an exclusive lock, but it will block writers to the table while it is executing. There is, unfortunately, no simple way yet to run a `REINDEX` in a way similar to `CREATE INDEX CONCURRENTLY` such that it doesn't block other processes. It's sometimes possible to create a new index and use a pair of index renaming operations to swap it out for the one you want to rebuild, thus allowing you to drop the original. This doesn't work for any index that enforces a constraint, which keeps this technique from being useful on `UNIQUE` and `PRIMARY KEY` indexes. You also need to be careful that no transactions still using the old index are executing before the final setup here.

Autoanalyze

While technically a part of the same autovacuum daemon, the autoanalyze logic works a bit differently. The server keeps an internal count of how many tuples (rows) were in each table the last time it was analyzed. It then adds the estimated counts it keeps for live and dead tuples in the database (the ones you can see in `pg_stat_user_tables`) and triggers when these numbers drift too far. While regular autovacuum only cares about rows that are dead because of `UPDATE`, `DELETE`, and `ROLLBACK`, autoanalyze also counts tables that have expanded through simple `INSERT` additions too.

By the time you've made it through the much more difficult to set up and monitor autovacuum work on your server, the similar but simpler autoanalyze work should be straightforward. While you can't see the internal tuple count to predict exactly when it will kick in, the two main statistics that drive it, the live and dead tuple counts, are shown, along with the last time the two types of analyze were done in `pg_stat_user_tables`. They all appear in the more complicated autovacuum triggering example given in the preceding section too. Because running analyze on a table is relatively fast, and there's no concerns about XID wraparound, it's easier to just lower the parameters that control how often autoanalyze runs to make it happen more frequently, or to run it manually on a regular schedule.

Note that there are some known problems with autoanalyze triggering, including that it can be fooled into inactivity by HOT activity not adjusting its counters properly. This logic was improved in PostgreSQL 9.0 to make this class of problem less likely to run into. It's still a good idea to manually search out tables that haven't been analyzed for a long time and give them some maintenance attention if they might not have accurate statistics. It's not a hard bit of work to keep up with compared to keeping them free of dead rows via regular vacuum work.

Index bloat

PostgreSQL's default index type is the binary tree (B-tree). While a B-tree gives good index performance under most insertion orders, there are some deletion patterns that can cause large chunks of the index to be filled with empty entries. Indexes in that state are referred to as bloated. Scanning a bloated index takes significantly more memory, disk space, and potentially disk I/O than one that only includes live entries.

There are a few main sources for index bloat to be concerned about. The first involves deletion, and is concisely described by the documentation about routine reindexing:

> *"Index pages that have become completely empty are reclaimed for re-use. There is still a possibility for inefficient use of space: if all but a few index keys on a page have been deleted, the page remains allocated. So a usage pattern in which all but a few keys in each range are eventually deleted will see poor use of space."*

The second source has been removed as of PostgreSQL 9.0. In earlier versions, `VACUUM FULL` compacts tables by moving their rows to earlier positions in the table. The documentation on `VACUUM FULL` describes how that creates index bloat:

> *"Moving a row requires transiently making duplicate index entries for it (the entry pointing to its new location must be made before the old entry can be removed); so moving a lot of rows this way causes severe index bloat."*

A third source is that any long-running transactions can cause table bloat and index bloat just because they block the efforts of any vacuum procedure to properly clean up tables.

As we have already discussed, PostgreSQL is based on MVCC architecture. The same row can be visible to one transaction and not visible to another transaction. Rows that are not visible to any architecture are considered dead rows. In a table, rows can be updated and deleted, and this causes dead rows that in turn cause dead space. Bloating occurs because of this dead space. This bloating can cause performance degradation.

Measuring index bloat

When you first create a new table with an index on it and insert data into it, the size of the index will grow almost linearly with the size of the table. When you have bloated indexes, that proportion will be very different. It's not unusual for a bloated index to be significantly larger than the actual data in the table. Accordingly, the first way you can monitor how bloated an index is by watching the index size relative to the table size, which is easy to check with this query:

```
pgbench=# SELECT
  nspname,relname,
  round(100 * pg_relation_size(indexrelid) / pg_relation_size(indrelid)) / 100
   AS index_ratio,
  pg_size_pretty(pg_relation_size(indexrelid)) AS index_size,
  pg_size_pretty(pg_relation_size(indrelid)) AS table_size
FROM pg_index I
LEFT JOIN pg_class C ON (C.oid = I.indexrelid)
LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
WHERE
 nspname NOT IN ('pg_catalog', 'information_schema', 'pg_toast') AND
 C.relkind='i' AND
 pg_relation_size(indrelid) > 0;
 nspname |        relname        | index_ratio | index_size | table_size
---------+-----------------------+-------------+------------+------------
 public  | pgbench_branches_pkey |           2 | 16 kB      | 8192 bytes
 public  | pgbench_tellers_pkey  |           1 | 16 kB      | 16 kB
 public  | pgbench_accounts_pkey |        0.16 | 54 MB      | 320 MB
(3 rows)
```

As an example of what this looks like when run, if you create a `pgbench` database with a scale of 25, the index will be 16% of the size of the table.

If you don't have that data available, it's possible to get a rough estimate of how much dead row bloat is in a table or an index by running some computations based on the size of various structures in the table and index. The PostgreSQL monitoring plug-in for Nagios, `check_postgres`, includes such an estimate, and some other sources have their own checks that are often derived from that original source. A full list of bloat checking utilities is available at `http://wiki.postgresql.org/wiki/Index_Maintenance`.

Note that for the case we have been looking at, the `check_postgres` index bloat test (as of version 2.14.3) doesn't estimate the bloat change as significant. You can see exactly what query it estimates that with by running the program in verbose mode, copying the code it executes, translating the `\n` characters in there to either spaces or carriage returns, and then pasting the result into `psql`:

```
$ ./check_postgres.pl --action bloat -db pgbench -v -v -v
$ psql -d pgbench -x
pgbench=# SELECT
current_database(), schemaname, tablename,
reltuples::bigint, relpages::bigint, otta
...
iname      | accounts_pkey
ituples     | 1250000
ipages      | 5489
iotta      | 16673
ibloat      | 0.3
wastedipages   | 0
wastedibytes   | 0
wastedisize   | 0 bytes
```

In general, so long as you have an unbloated reference point, monitoring the index to table size ratio will give you a more accurate picture of bloat over time than something working with the table statistics like `check_postgres` can provide.

Fixing the index bloat

One of the performances hit is when an index becomes blotted. one major reason of index bloating is sparse deletions. Here are some articles about the index bloating `http://blog.ioguix.net/tag/bloat/`, `https://wiki.postgresql.org/wiki/Show_database_bloat`. The question is how to fix an index bloating when detected? There are several ways to fix that issue, which are:

Dump and restore

Dump/restore is the simplest way to overcome index bloating. The `pg_dump` utility can be used to take a dump of the database, then restore the operation and create the whole schema and reload the data again. This process overcomes the whole bloating issue. This is an expensive operation and sometimes seems too restrictive.

Vacuuming the database/table

Vacuuming is another solution to overcome index bloating. PostgreSQL has a utility called `VACUUM`, which is used for vacuuming the database. The `VACUUM` command reshuffles the rows to ensure that the page is as full as possible, but database file shrinking only happens when there are some 100% empty pages at the end of the file. This is the only case where `VACUUM` is useful to reduce the bloat. Its syntax is as follows:

```
postgres=# VACUUM FULL VERBOSE foo;
INFO:  vacuuming "public.foo"
INFO:  "foo": found 0 removable, 910001 nonremovable row versions in 4919 pages
DETAIL:  0 dead row versions cannot be removed yet.
CPU 0.03s/0.17u sec elapsed 0.21 sec.
VACUUM
```

CLUSTER

With the dump/restore solution, we came to the conclusion that rewriting the rows can fix index bloating. But dumping and restoring the whole database is a very expensive operation. There is another way to reorder the rows using the `CLUSTER` command. The `CLUSTER` command physically reorders rows based on the index. The `CLUSTER` command is used to create a whole initial copy of the table, and thee old copy of the data is dropped.

The `CLUSTER` command requires enough space, virtually twice the disk space, to hold the initial organized copy of the data.

The code is as follows:

```
CLUSTER table_name USING index_name
```

Reindexing

The reindexing of the index can be used to overcome index bloating. The reindex command is required when data is randomly scattered and needs reindexing:

```
REINDEX TABLE item;
```

The following is the output of this command:

Using pgstattuple to get the size of table / index.

```
postgres=# CREATE TABLE foo(a INTEGER, b VARCHAR);
CREATE TABLE
postgres=# CREATE INDEX bar ON foo (a);
CREATE INDEX
postgres=# INSERT INTO foo VALUES(generate_series(1,1000000), 'TEXT');
INSERT 0 1000000
postgres=# SELECT table_len/(1024*1024) table_size, tuple_count total_rows FROM pgstattuple('foo');
 table_size | total_rows
------------+------------
         42 |    1000000
(1 row)

postgres=# SELECT table_len/(1024*1024) table_size, tuple_count total_rows FROM pgstattuple('bar');
 table_size | total_rows
------------+------------
         21 |    1000000
(1 row)
```

Now Reindex the table and get sizes:

```
postgres=# DELETE FROM foo WHERE a  >10000 AND a <100000;
DELETE 89999
postgres=# SELECT table_len/(1024*1024) table_size, tuple_count total_rows FROM pgstattuple('foo');
 table_size | total_rows
------------+------------
         42 |     910001
(1 row)

postgres=# SELECT table_len/(1024*1024) table_size, tuple_count total_rows FROM pgstattuple('bar');
 table_size | total_rows
------------+------------
         21 |    1000000
(1 row)

postgres=# REINDEX TABLE foo;
ERROR:  canceling autovacuum task
CONTEXT:  automatic vacuum of table "postgres.public.foo"
REINDEX
postgres=# SELECT table_len/(1024*1024) table_size, tuple_count total_rows FROM pgstattuple('foo');
 table_size | total_rows
------------+------------
         42 |     910001
(1 row)

postgres=# SELECT table_len/(1024*1024) table_size, tuple_count total_rows FROM pgstattuple('bar');
 table_size | total_rows
------------+------------
         19 |     910001
(1 row)
```

pgstattuple is a PostgreSQL's extension. Install the extension using `CREATE EXTENSION pgstattuple;`.

Detailed data and index page monitoring

If you really want to get deep into just what's happening with the disk space use on your server, there are a few more PostgreSQL contrib modules that provide additional information available:

- `pgstattuple`: Maybe you don't trust the running estimates for dead rows that the database is showing. Or perhaps you just want to see how they are distributed. This information and lots of other tuple-level data is available via the `pgstattuple` module. The module includes functions to give detailed analysis of both regular row tuple data and index pages, which lets you dig into trivia, such as exactly how the B-tree indexes on your server were built.
- `pg_freespacemap`: This lets you look at each page of a relation (table or index) and see what's in the FSM for them. The data provided is a bit different in 8.4, where the FSM was rewritten, than it is in earlier versions. There is an example of using this utility inside of the Nagios `check_postgres` utility, a program discussed more in `Chapter 11`, *Database Activity and Statistics*.

Monitoring query logs

If you want to profile what your server has done, one of the most effective ways is to analyze the logs of the queries it executed. There are many ways you can approach that problem, and several tools available to then analyze the resulting log files.

Basic PostgreSQL log setup

This is what the default settings in the `postgresql.conf` setting look like for the main logging setup parameters:

```
log_destination = 'stderr'
logging_collector = off
log_line_prefix = ''
log_directory = 'pg_log'
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
```

It's important to know what all these lines mean before changing them:

- `log_destination`: Write server log messages to the standard error output of the process starting the server. If you started the server manually, these might appear right on your console. If you sent the server output to another spot, either using the `pg_ctl -l` option or by using command-line redirection, they'll go to the file you specified instead.
- `logging_collector`: Set on the logging collector, to capture all the log messages sent to stderr.
- `log_line_prefix`: An empty string means don't add anything to the beginning.
- `log_directory`: When the logging collector is on, create files in the `pg_log` directory underneath your database directory (typically `$PGDATA`).
- `log_filename`: Name any files the logging collector creates using date and time information.

The `log_directory` and `log_filename` parameters can be set to whatever makes sense for your environment. The other parameters have some specific situations where you will want to change them from the defaults, covered in the next few sections.

The server doesn't clean up log files if you turn collection on. You have to manage that yourself, perhaps using standard UNIX log rotation utilities.

Log collection

While most server startup scripts will save the log file output that the server writes for you, that's typically going to be into a single file, such as how RedHat systems create a `pg_startup.log` parameter for you.

The parameter shown in all these examples as `logging_collector` was renamed in PostgreSQL 8.3. Before then, it was known as `redirect_stderr`. The change in name reflects the fact that it's also used to redirect for the `csvlog` format, in addition to `stderr` as of that version. The new parameter has the same basic behavior.

If you want to save log files for query analysis, or you just want the log files to be created in batches (per day by default), you'll want to turn the `logging_collector` parameter on. Afterwards, rather than a single file, you'll get one named after whatever `log_filename` pattern has been specified.

log_line_prefix

There are three basic formats of line prefix that will match the standard of those introduced by the requirements of `pgFouine`, a popular log parsing utility covered in the following paragraph. To ensure compatibility with that tool, you should use one of these three in your `postgresql.conf`, if you don't have any specific reason not to:

- `log_line_prefix = '%t [%p]: [%l-1] '`
- `log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d '`
- `log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d,remote=%r '`

Note the trailing space in all these examples; it is important that you always include it in your prefix setting. The first line just logs basic information. The second upgrades it to also include the user and database information for the connection, which allows filtering on those two fields when running `pgFouine` or similar utilities. The final line also adds the remote host the connection came from, which for some applications is an extremely useful bit of information for analyzing the logs (even though `pgFouine` won't do anything with that).

Note that if you're using syslog or the CSV format logs, the timestamp and process information will be inserted for you automatically. You don't need to include it into the prefix of the PostgreSQL log output too.

Multi-line queries

Sometimes client tools and psql sessions can execute queries that span more than one line. Here's an example of such generated using psql:

```
postgres=# show log_line_prefix
postgres-# ;
```

With statement logging enabled, this will show up in the standard text logs like this:

```
2010-03-28 21:23:49 EDT [10669]: [1-1]
  user=postgres,db=postgres,remote=[local] LOG: duration: 0.923 ms
  statement: show log_line_prefix;
```

That's two lines of output. The dangling semicolon is the only thing on the second line, with no prefix appearing. Because of that, there's no way to associate that line with the rest of the query, given that log lines are not guaranteed to be grouped together--in a more complicated environment, another line might get in between these two in the text format logs.

Log file analysis tools that want to follow every line in a query have no way to sort out what to do here, and the better ones will warn you about this problem when they run, if it shows up. There are alternate logging formats besides simple plain text that avoid this issue.

Using syslog for log messages

On UNIX-like systems, one approach for database logging that leverages existing operating infrastructure is to redirect `log_destination` to use the syslog facility.

On Windows systems, you can use log messages to the Windows Event Log similarly to how syslog is used here.

The main thing to be concerned about when using syslog is a warning you'll find in the documentation at
`http://www.postgresql.org/docs/current/static/runtime-config-logging.html`.

The logging collector is designed to never lose messages. This means that in case of extremely high load, server processes could be blocked due to trying to send additional log messages when the collector has fallen behind. In contrast, syslog prefers to drop messages if it cannot write them, which means it's less reliable in those cases, but it will not block the rest of the system.

This means that the exact times when you most want data--when the server is overloaded--are the most likely to lose your logs if using syslog. That's one reason why it's not a more popular approach. The complexity of setting up syslog (or the improved `syslog-ng`) is another reason database administrators tend to avoid this setup.

In addition to allowing the use of common syslog analysis tools, another advantage of the syslog format is that it knows how to handle multi-line messages in a way that tools such as `pgFouine` can use.

CSV logging

Another way to avoid multi-line query issues is to use CSV logging, added to PostgreSQL in version 8.3. This logs every statement in a delimited fashion that clearly marks line breaks such that multi-line queries can be imported without a problem into tools. You can even import these logs files into the database for analysis; the documentation at `http://www.postgresql.org/docs/current/static/runtime-config-logging.html` shows a sample table for that purpose in the *Using CSV-format log output* section.

Note the format of the CSV output fields and therefore the sample table changed in 9.0 (`application_name` was added) compared with 8.3/8.4. Make sure you're using the right table definition for your version, or `COPY` will complain that the column count doesn't match when you try to import.

To turn on this feature, you need to adjust the log destination and make sure the collector is running:

```
log_destination = 'csvlog'
logging_collector = on
```

The server must be completely restarted after this change for it to take effect. After this change, the log files saved into the usual directory structure that the collector uses will now end with `.csv` rather than `.log`. If you followed the right documentation for your version to create the `postgres_log` file, you would import it like this:

```
postgres=# COPY postgres_log FROM '/home/postgres/data
  /pg_log/postgresql-2010-03-28_215404.csv' WITH CSV;
```

Having all of the log data in the database allows you to write all sorts of queries to analyze your logs. Here's a simple example that shows the first and last commit among the logs imported:

```
SELECT min(log_time),max(log_time) FROM postgres_log WHERE
  command_tag='COMMIT';
```

You might instead ask at what elapsed time since the start of the session each command happened at:

```
SELECT log_time,(log_time - session_start_time) AS elapsed FROM
  postgres_log WHERE command_tag='COMMIT';
```

Using this particular example against an imported `pgbench` run will show you that `pgbench` keeps one session open for all the work each client does.

You can use this log to figure out what statement executes before a given commit too, by looking for entries with the same `session_id` but with lower `session_line_num` values. Often when people run into situations where a `COMMIT` statement is the one that takes a long time, for example when a large queue of `CHECK` constraints have built up, knowing what happened before the commit is critical information, which is also difficult to discover. The CSV format logs make this easy to determine because they contain every bit of session information.

The main disadvantage of the CSV log format is that you're writing out a lot of logging information, and that can be too intensive on the exact type of system that needs better logging the most: ones that are overloaded by bad queries.

Logging difficult queries

A straightforward way to generate log files only for the complex, long running queries on your server--cutting out routine logging of small queries and therefore reducing logging overhead--is to set a minimum statement duration to log. A typical configuration would be:

```
log_min_duration_statement = 1000
log_duration = off
log_statement = 'none'
```

The first line there will log every statement that takes over 1000 ms (1 second) without logging anything else. The other two are actually the defaults.

An alternate approach you'll sometimes see instead aims to just capture every query:

```
log_min_duration_statement = -1
log_duration = on
log_statement = 'all'
```

Since setting `log_min_duration_statement` to 0 will also log every statement with a duration, that's an easier way to get this behavior. It only requires tweaking one value, and it's easy to adjust upward so that it's only triggered when appropriate. Because that's easier to manage, you should prefer that approach over increasing the overall statement verbosity.

This area of PostgreSQL continues to expand as advances on the server and on additional external utilities are finished.
`http://wiki.postgresql.org/wiki/Logging_Difficult_Queries` tracks the latest work in this area.

auto_explain

One of the great features added in PostgreSQL 8.4 is `auto_explain`, documented at `http://www.postgresql.org/docs/current/static/auto-explain.html`.

This is a `contrib` module, not necessarily installed by default. See the discussion of the `contrib` modules in `Chapter 1`, *PostgreSQL Versions*.

To enable the feature, edit your `postgresql.conf` to add parameters such as this:

```
shared_preload_libraries = 'auto_explain'
custom_variable_classes = 'auto_explain'
auto_explain.log_min_duration = '1s'
```

That will trigger `auto_explain` on any query longer than a second. You'll need to completely restart the server after making this change before it goes into effect.

Once that's done, any query that takes longer than this will be logged with a full `EXPLAIN` plan for the output. This is what one of the common slow `pgbench UPDATE` statements look like:

```
duration: 1955.811 ms plan:
Query Text: UPDATE pgbench_accounts SET abalance = abalance + -3410
  WHERE aid = 9011517;
Update (cost=0.00..10.03 rows=1 width=103)
  -> Index Scan using pgbench_accounts_pkey on pgbench_accounts
  (cost=0.00..10.03 rows=1 width=103)
  Index Cond: (aid = 9011517)",,,,,,,,,""
```

The plans will be spread across multiple lines if in the regular logs, but will be in a single column if loaded into a CSV format log. The `syslog` logs will also tag the multiple line output appropriately.

As for what you can use this for, a look at the logs shows the delay is happening even when updating the tiny branches table:

```
duration: 1490.680 ms plan:
Query Text: UPDATE pgbench_branches SET bbalance = bbalance + -2792
  WHERE bid = 4;
Update (cost=0.00..4.50 rows=1 width=370)
 -> Seq Scan on pgbench_branches (
     cost=0.00..4.50 rows=1 width=370)
Filter: (bid = 4)",,,,,,,,,""
```

Knowing information like this--that the `UPDATE` hang is happening even when updating a table extremely likely to be in the buffer cache--is extremely valuable for tracking down the true cause of slow statements.

Log file analysis

Once you have some log files saved, next you'll want to analyze the results. The common information you'll get is a profile of what queries were run, how long each took individually, and sums showing the totals broken down by type of query. There are several tools available for this purpose, with different trade-offs in terms of things such as how active their development community is, how fast the tool runs, and how extensive the installation requirements are.

Normalized query fingerprints

To a person, it would be obvious that the following two queries are fundamentally the same:

```
UPDATE pgbench_accounts SET abalance = abalance + 1631
 WHERE aid = 5829858;
UPDATE pgbench_accounts SET abalance = abalance + 4172
 WHERE aid = 567923;
```

This isn't necessarily obvious to a log file parsing program, though. Good database programming practice will often convert these to executing with prepared statements, which offers both a security improvement (resistance to SQL injection) as well as a potential performance gain. The `pgbench` program used to generate many of the examples in this book can be converted to them internally for executing statements:

```
$ pgbench -c4 -t 60 -M prepared pgbench
```

Using prepared statements makes it obvious that these statements have the same query fingerprint--that the main structure is the same:

```
UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = 2;
```

Some log file analysis programs are capable of performing a process named query normalizing in order to extract a query fingerprint from even unprepared statements. If you don't use prepared statements, it will be a feature you need if you want to group similar queries together for analysis purposes.

pg_stat_statements

While in many cases you might still want to save full logs, the `pg_stat_statements` feature added in PostgreSQL 8.4 can substitute as a way to analyze queries without needing an intermediate logging facility. The feature is documented at `http://www.postgresql.org/docs/current/static/pgstatstatements.html`.

This is a `contrib` module, not necessarily installed by default. See the discussion of the `contrib` modules in `Chapter 1`, *PostgreSQL Versions*. Like `pg_buffercache`, it needs to be installed in each database you use it against. Here's an example of installing it on a system with the RedHat Linux directory layout for PostgreSQL:

```
$ psql -d pgbench -f
/usr/share/postgresql/contrib/contrib/pg_stat_statements.sql
```

You enable this using `postgresql.conf` additions like these:

```
shared_preload_libraries = 'pg_stat_statements'
custom_variable_classes = 'pg_stat_statements'
pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

This is then followed by fully restarting the server. Once you've run some queries and therefore populated the information in `pg_stat_statements`, you can look at them the same as any other view:

```
pgbench=# SELECT round(total_time*1000)/1000 AS total_time,query
  FROM pg_stat_statements ORDER BY total_time DESC;
total_time | query
  78.104 | UPDATE pgbench_accounts SET abalance = abalance + $1
WHERE aid = $2;
  1.826 | UPDATE pgbench_branches SET bbalance = bbalance + $1
WHERE bid = $2;
  0.619 | UPDATE pgbench_tellers SET tbalance = tbalance + $1
WHERE tid = $2;
 0.188 | INSERT INTO pgbench_history (tid, bid, aid, delta,
 mtime) VALUES ($1, $2, $3, $4, CURRENT_TIMESTAMP);
 0.174 | SELECT abalance FROM pgbench_accounts WHERE aid = $1;
 0.026 | vacuum pgbench_branches
 0.015 | vacuum pgbench_tellers
 0.011 | BEGIN;
 0.011 | END;
 0.001 | truncate pgbench_history
```

It's extremely handy to have this facility built into the database, without needing external tools. There are a few limitations. No query fingerprint normalization can be done if you don't use query parameters--the preceding example required running `pgbench` in its prepared statement mode. The number of saved messages has a hard limit on it. It's easy to lose an interesting statement log as new activity occurs on the server.

And not necessarily every operation you can see in the logs will appear here. For example, note that in the preceding log from a standard `pgbench` test, timing for the `UPDATE` statements is listed, but the commits from the `END` statements are not accumulating a lot of time. A very different view of this information shows up if you look at the log file data, as shown in the next section.

PostgreSQL 9.0 adds an additional ability to `pg_stat_statements`. It can now also track the amount of database cache buffers used while processing a statement.

pgFouine

The most popular query log file analyzer for PostgreSQL is **pgFouine**, available at `http://pgfouine.projects.postgresql.org/`.

The program is available in packaged form for some operating systems that bundle large amounts of open source software with their distribution. pgFouine is available for RHEL and CentOS Linux systems via the EPEL repository, and is available as one of the standard packages on Debian/Ubuntu systems.

The program is a PHP script, and it has a moderate list of libraries it relies upon that are not bundled with PHP itself. As such, getting pgFouine working properly on a server that doesn't have a rich software library available can be difficult. This includes Windows and most older UNIX systems. Note that you don't necessarily have to run the program on the server itself. Copying the query log files daily from the database server onto another system, such as a web reporting host, is common practice from a performance point of view. You might, for example, have a Solaris database server (where installing the program could be difficult), then upload them to a Linux host, both to offload log file processing from the database hardware and to publish them out to a web application.

A basic test of pgFouine can use `postgresql.conf` with the default `stderr` log format, redirected into a file, with an appropriate prefix and logging all statements:

```
logging_collector = on
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d,remote=%r '
log_min_duration_statement = 0
```

Once you've generated some query traffic, you run the utility against the resulting log file:

```
$ pgbench -T 60 -c 4 pgbench
$ pgfouine -logtype stderr -file $PGDATA/pg_log/postgresql-2010-03-
  28_210540.log -top 10 -format text
```

```
#####  Overall statistics  #####

Number of unique normalized queries: 6
Number of queries:     8,001
Total query duration:  0.8s


#####  Queries by type  #####

SELECT:    1333     16.7%
INSERT:    1334     16.7%
UPDATE:    3999     50.0%


#####  Queries that took up the most time (N)  #####

1) 0.2s - 1,335 - END;
--
2) 0.1s - 1,332 - UPDATE pgbench_accounts SET abalance = abalance + 0 WHERE aid = 0;
--
3) 0.1s - 1,334 - UPDATE pgbench_branches SET bbalance = bbalance + 0 WHERE bid = 0;
--
4) 0.1s - 1,333 - UPDATE pgbench_tellers SET tbalance = tbalance + 0 WHERE tid = 0;
--
5) 0.1s - 1,333 - SELECT abalance FROM pgbench_accounts WHERE aid = 0;
--
6) 0.1s - 1,334 - INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (0, 0, 0, 0, CURRENT_TIMESTAMP);
--
```

```
#####  Most frequent queries (N)  #####

1) 1,335 - 0.2s - END;
--
2) 1,334 - 0.1s - UPDATE pgbench_branches SET bbalance = bbalance + 0 WHERE bid = 0;
--
3) 1,334 - 0.1s - INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (0, 0, 0, 0, CURRENT_TIMESTAMP);
--
4) 1,333 - 0.1s - UPDATE pgbench_tellers SET tbalance = tbalance + 0 WHERE tid = 0;
--
5) 1,333 - 0.1s - SELECT abalance FROM pgbench_accounts WHERE aid = 0;
--
6) 1,332 - 0.1s - UPDATE pgbench_accounts SET abalance = abalance + 0 WHERE aid = 0;
--


#####  Slowest queries (N)  #####

1) 0.00 - 1,335 - END;
--
2) 0.00 - 1,332 - UPDATE pgbench_accounts SET abalance = abalance + 0 WHERE aid = 0;
--
3) 0.00 - 1,334 - UPDATE pgbench_branches SET bbalance = bbalance + 0 WHERE bid = 0;
--
4) 0.00 - 1,333 - UPDATE pgbench_tellers SET tbalance = tbalance + 0 WHERE tid = 0;
--
5) 0.00 - 1,333 - SELECT abalance FROM pgbench_accounts WHERE aid = 0;
--
6) 0.00 - 1,334 - INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (0, 0, 0, 0, CURRENT_TIMESTAMP);
--
```

Two of the sections left out--most frequent queries and slowest queries--are not very useful for this pgbench test, but might be useful for your data set. Note that even though individual queries are logged here, and we didn't use prepared statements, the program figured out the common parameters anyway. They show up in the last section there, replaced with the value of 0, which is what you want to see a normalizing query analysis tool do.

You can easily create HTML output instead:

```
$ pgfouine -file $PGDATA/pg_log/postgresql-2010-03-28_210540.log -
  logtype stderr -format html > fouine.htm
```

Many other types of reports are also available. You can just run pgFouine or read its documentation for an idea of the options. pgFouine also includes a `VACUUM` logging analyzer useful for monitoring that area of your database, as documented at `http://pgfouine.projects.postgresql.org/vacuum.html`.

Because pgFouine needs to handle multi-line queries intelligently, you will end up with missing data with a simple configuration when such a query appears. The program's authors have traditionally recommended using the `syslog` format for all logs to avoid this. Recent pgFouine releases (starting with 1.1) can instead import CSV format logs by passing the `-logtype` CSV parameter into the program.

pgFouine is powerful, but it has some limitations. The heavy installation dependencies can be overwhelming for some environments. As a PHP script, it's not particularly efficient when dealing with really large query log files. And it's not very flexible about things such as setting `log_line_prefix`; you're compelled to adopt its standard there. But if you can work within those limitations, pgFouine is a very capable tool for finding what the slowest queries in your application are.

pqa

An ancestor of pgFouine, **Practical Query Analysis** or **pqa** is a project located at `http://pqa.projects.postgresql.org/` that provides some basic query analysis. The program is fairly small and has minimal dependencies due to its small feature set and use of Ruby as its programming language. However, it hasn't been updated in several years, and doesn't support recent features such as the CSV logging format.

eqpa

Another program similar to pqa and pgFouine is **Enterprise Postgres Query Analyze** or **epqa**, located at `http://epqa.sourceforge.net/`.

Compared to those other tools, eqpa aims to consume less memory and process logs faster, mainly due to its use of the more efficient string operations available in Perl. It will also read compressed log files without having to decompress them separately before running the tool. This tool also hasn't been updated in a few years and has a minimal development community compared to pgFouine.

pgsi

A slightly different spin on the usual query log impact analysis is done if you analyze logs with the **PostgreSQL System Impact** (**PGSI**) log analyzer (`http://bucardo.org/wiki/Pgsi`).

While this includes many of the same average and count statistics the other log analyzers here include, it also considers factors such as the average interval between when the query runs. This will make queries that are essentially always running easier to spot. pgsi is a relatively new project, but it already supports producing nice reports in both HTML and Mediawiki formats, and it's better at interpreting what you've put into your logs using the `log_line_prefix` mechanism as written (rather than requiring a custom format) than most of the tools like it.

mk-query-digest

The most recent tool to be introduced for analyzing PostgreSQL log files is `mk-query-digest`, available from `http://www.maatkit.org/doc/mk-query-digest.html`.

This program originated as a tool for MySQL databases that PostgreSQL log input was added to. One major advantage of its approach is that it includes some advanced query fingerprint techniques. It's also designed to work with the popular `memcached` query caching software; logging through a query cache can be tricky. It is also designed to be fast at processing large log files, and it has minimal installation requirements--just Perl and the libraries that ship with that language by default.

The main thing you might find missing from `mk-query-digest` are fancier output modes. It produces a basic text-only output. You will need additional layers of tools to get an HTML display, for example, and said tools are not available yet at the time of writing. However, given that this tool has both a strong basic feature set and its output is the same for each of the database formats it supports, there's a potential for those tools be written by a larger group than just the PostgreSQL community--or even yourself, given that it's a straightforward tool to interact with. Wrapping the output from this program into more web-suitable formats would not very difficult.

Summary

Properly maintaining every aspect of a database is time-consuming work, but the downside of not doing it can be even worse. This is particularly true when it comes to good vacuuming practice, where routine mistakes can build up over time into extremely expensive operations requiring downtime to fully recovery from. Similarly, monitoring your query logs to be proactive about finding ones that execute slowly takes regular review, but the downside there can also be downtime if your server falls apart under a heavy load.

MVCC will keep routine read and write conflicts from happening, but you still need to code applications to lock resources when only one session should have access to a row at once. Any time you `UPDATE`, `DELETE`, or `ROLLBACK` a transaction, it will leave a dead row behind (and potentially a dead index entry) that needs to be cleaned up later by some form of vacuum. Avoid ever using the `VACUUM FULL` command. Instead, use `CLUSTER` as the main way to reorganize a badly damaged table, or `REINDEX` to fix bad indexes. Saving basic query logs to disk with proper prefix lines is vital to catching execution slowdowns before they grow too bad.

Next, we will discuss PostgreSQL benchmarking using different benchmarking tools, such as pgbench.

8
Database Benchmarking

PostgreSQL ships with a benchmarking program named `pgbench` that can be used for a variety of tests. The default tests included are useful, but it also includes a database benchmarking scripting language that allows you to customize these scripts. You can even use the `pgbench` core as an efficient, scalable multi-client database program to write new tests. There are also some industry-standard tests available that let you compare PostgreSQL with other database products, albeit without officially audited results in most cases.

Topics we will cover in this chapter are:

- PostgreSQL benchmarking tool pgbench
- Built-in pgbench benchmarks
- Custom benchmark using pgbench.

pgbench default tests

The original inspiration for the pgbench test is the **Transaction Processing Performance Council** (**TPC**) benchmark named TPC-B: `http://www.tpc.org/tpcb/`.

Originally developed in 1990 (and now considered obsolete by TPC), this benchmark models a simple bank application that includes a set of bank branches, each of which has a number of tellers and accounts.

Table definition

The main table definition SQL adds these tables:

- `pgbench_branches`:

```
pgbench=# \d+ pgbench_branches
                        Table "public.pgbench_branches"
  Column  |     Type      | Modifiers | Storage  | Stats target | Description
----------+---------------+-----------+----------+--------------+-------------
 bid      | integer       | not null  | plain    |              |
 bbalance | integer       |           | plain    |              |
 filler   | character(88) |           | extended |              |
Indexes:
    "pgbench_branches_pkey" PRIMARY KEY, btree (bid)
Options: fillfactor=100
```

- `pgbench_tellers`:

```
pgbench=# \d+ pgbench_tellers;
                        Table "public.pgbench_tellers"
  Column  |     Type      | Modifiers | Storage  | Stats target | Description
----------+---------------+-----------+----------+--------------+-------------
 tid      | integer       | not null  | plain    |              |
 bid      | integer       |           | plain    |              |
 tbalance | integer       |           | plain    |              |
 filler   | character(84) |           | extended |              |
Indexes:
    "pgbench_tellers_pkey" PRIMARY KEY, btree (tid)
Options: fillfactor=100
```

- `pgbench_accounts`:

```
pgbench=# \d+ pgbench_accounts;
                        Table "public.pgbench_accounts"
  Column  |     Type      | Modifiers | Storage  | Stats target | Description
----------+---------------+-----------+----------+--------------+-------------
 aid      | integer       | not null  | plain    |              |
 bid      | integer       |           | plain    |              |
 abalance | integer       |           | plain    |              |
 filler   | character(84) |           | extended |              |
Indexes:
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
Options: fillfactor=100
```

- `pgbench_history`:

```
pgbench=# \d+ pgbench_history;
                              Table "public.pgbench_history"
 Column |            Type             | Modifiers | Storage  | Stats target | Description
--------+-----------------------------+-----------+----------+--------------+-------------
 tid    | integer                     |           | plain    |              |
 bid    | integer                     |           | plain    |              |
 aid    | integer                     |           | plain    |              |
 delta  | integer                     |           | plain    |              |
 mtime  | timestamp without time zone |           | plain    |              |
 filler | character(22)               |           | extended |              |
```

In PostgreSQL 9.6, these tables have the `pgbench_` prefix on their names. Previously this was dangerous without that `pgbench_`, because a badly written `pgbench` run could wipe out a real table named `accounts` far too easily. Having done that once, I can tell you it can be quite embarrassing to have to explain why all the accounts are now missing from a live database server.

The intention of the various filler fields is to make sure that each row inserted into the database is a full `100` bytes wide. This doesn't actually work as intended; the way data is inserted, the filler doesn't take up any space beyond a small amount of column overhead. This problem has never been corrected because doing so would make it no longer possible to compare historical `pgbench` results with more recent ones.

What you should realize is that every record type inserted by the standard `pgbench` test is extremely narrow, only a small number of bytes are inserted for each of them. In practice, even the smallest `INSERT` or `UPDATE` will dirty a standard 8 KB database page and require that large a write anyway. The fact that only a fraction is actually dirtied after each change does help reduce total WAL write volume after you've written the usual `full_page_writes` copy of each page, which will end up as the bulk of actual WAL throughput anyway.

Scale detection

The number of branches is referred to as the database scale: each branch adds another 10 tellers and 100,000 accounts to the database. This value is input when you initialize the database `pgbench` to create tables into it. Here's an example that creates a database for `pgbench` and then populates it with the `pgbench` tables:

```
$ createdb pgbench
$ pgbench -i -s 10 pgbench
```

This will initialize (`-i`) the pgbench tables using a scale of 10 (`-s 10`) into the database named `pgbench`. You don't have to use that name, but it's a common convention.

When `pgbench` runs in a regular mode (not initialization), it first checks if you have manually given it the database scale using `-s`; if so, it uses that value. If not and you're running one of the default tests, it will try to detect the scale using the following query:

```
pgbench=# SELECT count(*) FROM pgbench_branches;
 count
-------
    10
(1 row)
```

If you run a custom query, the scale will never be detected for you--it will always be assumed to be 1. You must be careful when running custom tests to set it yourself in any case where the test script you're running assumes the scale will be set properly. That includes customized variations of the built-in tests you might create, like those distributed with `pgbench-tools`.

Query script definition

As the preceding table code, the actual queries run by the built-in standard `pgbench` tests are compiled into its source code, which means the only way you can see them is to refer to documentation such as `http://www.postgresql.org/docs/current/static/pgbench.html`.

The default transaction script, what's called the "TPC-B (sort of)" transaction when you run the test (and referred to here as the "TPC-B-like" transaction), issues seven commands per transaction. It looks like this inside the program:

```
\set nbranches :scale
\set ntellers 10 * :scale
\set naccounts 100000 * :scale
\setrandom aid 1 :naccounts
\setrandom bid 1 :nbranches
\setrandom tid 1 :ntellers
\setrandom delta -5000 5000
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta
 WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts
 WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta
```

```
 WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta
 WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
 VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
```

The first three lines compute how many branches, tellers, and accounts this database has based on `:scale`, which is a special variable name set to the database scale, determined using the logic described in the last section.

The next four lines create random values to use, simulating a bank transaction where someone went to a specific teller at a specific branch and deposited/withdrew an amount of money from their account.

The presumption that one is using a regular teller for all transactions rather than a bank's **Automated Teller Machine** (**ATM**) is another clue as to how old the benchmarking being simulated here really is. `pgbench` is not simulating any real-world situation that still exists, and the TPC retired TPC-B as useful some time ago. `pgbench` should be considered a completely synthetic workload at this point, useful for testing some things but not indicative of true real-world database performance at all.

The main core here is five statements wrapped in a transaction block, so they execute either all together or nothing at all. That way the sum of money accounted for in the bank should match from the branch down to account level. Note that the statement will still take up some time and resources on the server even if it is rolled back.

Each of the statements has a very different practical impact:

- `UPDATE pgbench_accounts`: As the biggest table, this is the statement most likely to trigger disk I/O.
- `SELECT abalance`: Since the previous `UPDATE` will have left the information needed to answer this query in its cache, it adds very little work to the standard TPC-B-like workload.
- `UPDATE pgbench_tellers`: With so many fewer tellers than accounts, this table is small and it's likely to be cached in RAM; but it's large enough that it tends to be a source of lock.

- `UPDATE pgbench_branches`: As an extremely small table, the entire contents of this are likely to be cached already and essentially free to access. However, because it's small, locking might turn into a performance bottleneck here, if you use a small database scale and many clients. Accordingly, the general recommendation is always to make sure you scale clients for any test you run. In practice, this problem is not really that important, although you can see it in one of the examples below. Disk I/O on the accounts table and commit rate will be more of a limiter than locking concerns in most cases.
- `INSERT INTO pgbench_history`: The history table acts as an append-only one that is never updated or queried later, and accordingly it doesn't even have any indexes on it. As such, this insert turns out to be a low volume write, relative to the much more expensive indexed UPDATE statements.

This is the default test. There are two others:

- `-N`: The above, but skipping the two small `UPDATE` statements impacting tellers and branches. This reduces the main potential for lock contention, while keeping the primary source of heavy writes.
- `-S`: Just the `SELECT` statement on the accounts table, which then doesn't need to be wrapped in a transaction block either.

It's unlikely you'll have any use for the test that skips some of the `UPDATE` statements, but running just `SELECT` statements is very useful for examining the cache sizes on your system and for measuring maximum CPU speed.

Configuring the database server for pgbench

All the queries used for the built-in tests are simple: they are at most using a primary key index to look a row up, but no joins or more complicated query types. However, the statements executed will stress both the writes and the buffer cache's ability to keep up.

Accordingly, parameters you might adjust break down into three major categories:

- **Important to note and possibly tune**: `shared_buffers`, `checkpoint_segments`, `autovacuum`, `wal_buffers`, `Checkpoint_completion_target`.
- **Impacts test results significantly**: `wal_sync_method`, `synchronous_commit`, `wal_writer_delay`. Adjust on systems where a change is appropriate to get good and/or safe results.

- **Does not matter**: `effective_cache_size`, `default_statistics_target`, `work_mem`, `random_page_cost`, and most other parameters. The various `bgwriter` settings do have an impact, just usually not a positive one. Unlike some real workloads where it might help, making the background writer more aggressive, usually results in a net loss in raw throughput on `pgbench` tests, without improving latency either.

Sample server configuration

Some `pgbench` samples are provided below, with the following server specifications:

- **Processor name**: Intel Core i7
- **Processor speed**:2.3 GHz
- **Number of processors**:1
- **Total number of cores**:4
- **L2 cache (per core)**:256 KB
- **L3 cache**: 6 MB
- **Memory**: 16 GB

The operating system was on a different disk from any of the database files.

Here is a basic minimally tuned `postgresql.conf` capable of running `pgbench` usefully on such a server, with 8 GB of RAM, following the usual guidelines for `shared_buffers` sizing:

- `shared_buffers` = 6 GB
- `checkpoint_completion_target` = 0.9
- `wal_buffers` = 16 MB
- `max_connections` = 300
- `checkpoint_timeout` = 10 min

Running pgbench manually

As a simple example of running a test, you could execute the SELECT only test for a small number of transactions. Here's a quick test from the sample server:

```
$ pgbench -S -c 4 -t 20000 pgbench
starting vacuum...end.
transaction type: <builtin: select only>
scaling factor: 10
query mode: simple
number of clients: 4
number of threads: 1
number of transactions per client: 20000
number of transactions actually processed: 80000/80000
latency average = 0.106 ms
tps = 37886.806535 (including connections establishing)
tps = 37919.135311 (excluding connections establishing)
```

This is simulating the situation where four database clients are all active at the same time and continuously asking for data. Each client runs until it executes the requested number of transactions.

Since this is a Linux system, the `pgbench` driver program itself is known to limit performance here when run as a single thread, described in more detail below. Using more threads (only available in PostgreSQL 9.0 and later) shows a dramatic improvement in speed:

```
$ pgbench -S -c 4 -j 4 -t 20000 pgbench
starting vacuum...end.
transaction type: <builtin: select only>
scaling factor: 10
query mode: simple
number of clients: 4
number of threads: 4
number of transactions per client: 20000
number of transactions actually processed: 80000/80000
latency average = 0.101 ms
tps = 39730.271189 (including connections establishing)
tps = 39780.738514 (excluding connections establishing)
```

Rather than worrying about the transaction count, a simpler approach, available from PostgreSQL version 8.4, is to specify a runtime instead. The preceding code took 2.6 seconds; you could run for a full five seconds instead like this:

```
$ pgbench -S -c 4 -j 4 -T 5 pgbench
```

It's possible to get useful test results for a SELECT only test in a few seconds. You will need a much longer test, taking minutes, to get a TPC-B-like write test result that is meaningful. A 10-minute long TPC-B like test, which is usually long enough, could be run like this:

```
$ pgbench -c 4 -j 4 -T 600 pgbench
```

In earlier versions, before the timed run was available, estimating how many transactions were needed to get a useful runtime could take a few tries to get right.

Graphing results with pgbench-tools

Running a full and thorough `pgbench` evaluation of a server takes several days of system runtime. Starting with PostgreSQL 8.3, the `pgbench-tools` program, available from `http://git.postgresql.org/gitweb?p=pgbench-tools.git`, allows the automation of multiple runs of `pgbench`, including the production of graphs showing the results. Earlier PostgreSQL versions did not save all the necessary information to allow the graphing results over time.

On a system that supports running the `git` version control software, you can retrieve the source code like this:

```
$ git clone git://git.postgresql.org/git/pgbench-tools.git
$ cd pgbench-tools/
```

If you're on a database server that doesn't support `git`, you may need to run the preceding code on a newer system that does, then use that system to create an archive: here are two examples that produce a `.tar` and `.zip` file respectively:

```
$ git archive --prefix=pgbench-tools/ HEAD > pgbench-tools.tar
$ git archive --prefix=pgbench-tools/ --format=zip HEAD >
 pgbench-tools.zip
```

The current version of `pgbench-tools` does not work with PostgreSQL 9.6.

Configuring pgbench-tools

The `README` file that comes with `pgbench-tools` is the definitive documentation about how to use the program. The current recommended practice, which the examples here will follow, is to create two databases for it to use. The primary one should be named `pgbench` and it will contain the main database being tested. The second, named `results` by default, will contain a summary of information about each test after it finishes.

Here is a sample session that creates both databases and initializes the results database:

```
$ cd pgbench-tools/
$ createdb pgbench
$ createdb results
```

```
$ psql -d results -f init/resultdb.sql
BEGIN
psql:init/resultdb.sql:3: NOTICE:  table "testset" does not exist, skipping
DROP TABLE
CREATE TABLE
psql:init/resultdb.sql:9: NOTICE:  table "tests" does not exist, skipping
DROP TABLE
CREATE TABLE
psql:init/resultdb.sql:30: NOTICE:  table "timing" does not exist, skipping
DROP TABLE
CREATE TABLE
CREATE INDEX
psql:init/resultdb.sql:41: NOTICE:  table "test_bgwriter" does not exist, skipping
DROP TABLE
CREATE TABLE
CREATE FUNCTION
CREATE FUNCTION
COMMIT
```

Notice the messages about tables (which it tries to delete) not existing: these are normal, because `DROP TABLE IF EXISTS` isn't available until PostgreSQL 8.2, and `pgbench-tools` can, in special cases, be configured to run against 8.1 databases.

The results database doesn't even need to be on the same server as the one being tested. Making it remote stops `pgbench-tools` from disturbing the database being tested and the associated operating system cache, so this can be a better way to set things up. The default configuration and examples here don't assume you've done that simply because it complicates the configuration and use of the program, for limited benefit--the cache disruption is not that large relative to system RAM nowadays.

Once you know where each database is located, edit the `config` file at the root of the `pgbench-tools` directory, and update the settings for `TEST` and `RESULT` variables set there. You may want to customize the number of worker threads as well: see notes on that in the following sections and in the `pgbench-tools` documentation.

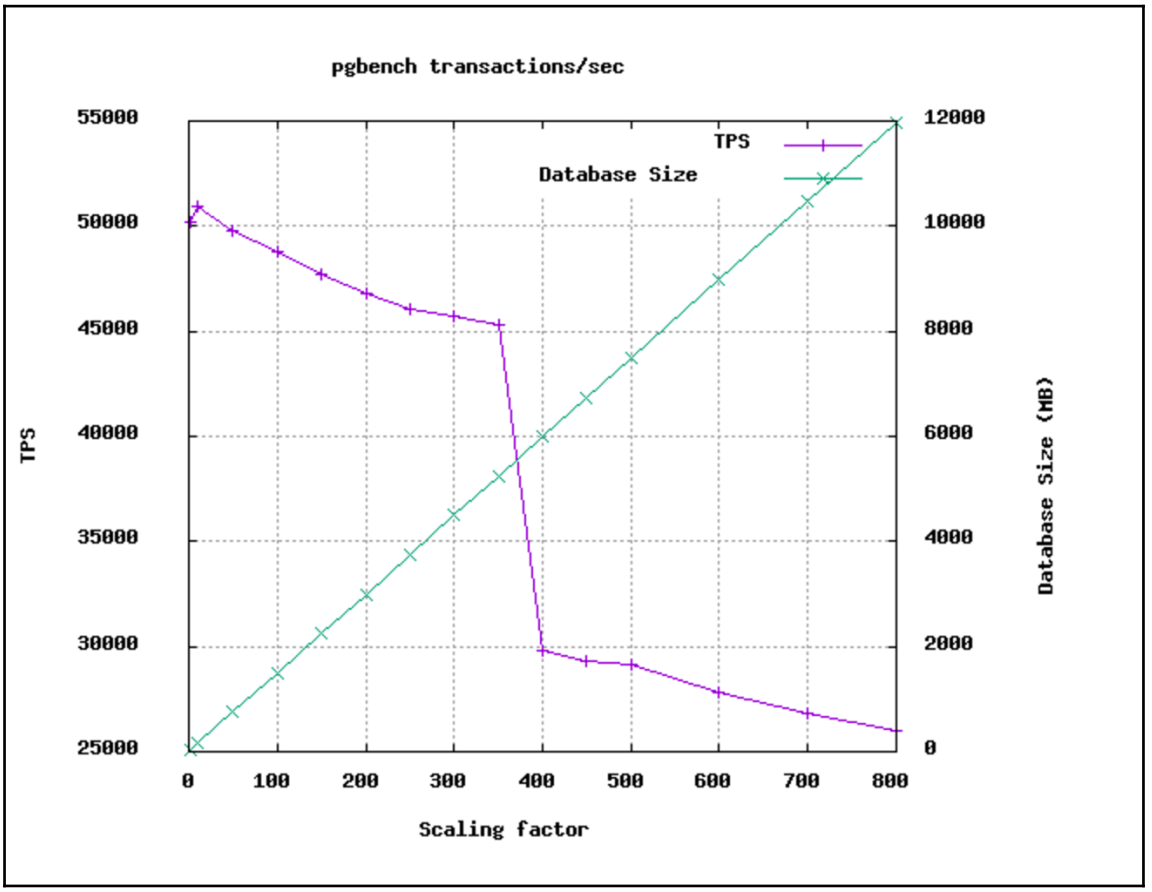
If you're using PostgreSQL 8.3, you will also need to follow the notes in the `config` file to swap which set of table names you are using. Some lines in the default one must be commented out, while others must be enabled in their place.

Sample pgbench test results

These all come from the sample server and configuration described previously, and are the sort of graph that you get from `pgbench-tools` each time it is run. These results were all created using PostgreSQL 9.6 and allow up to four `pgbench` worker threads (one per core).

SELECT-only test

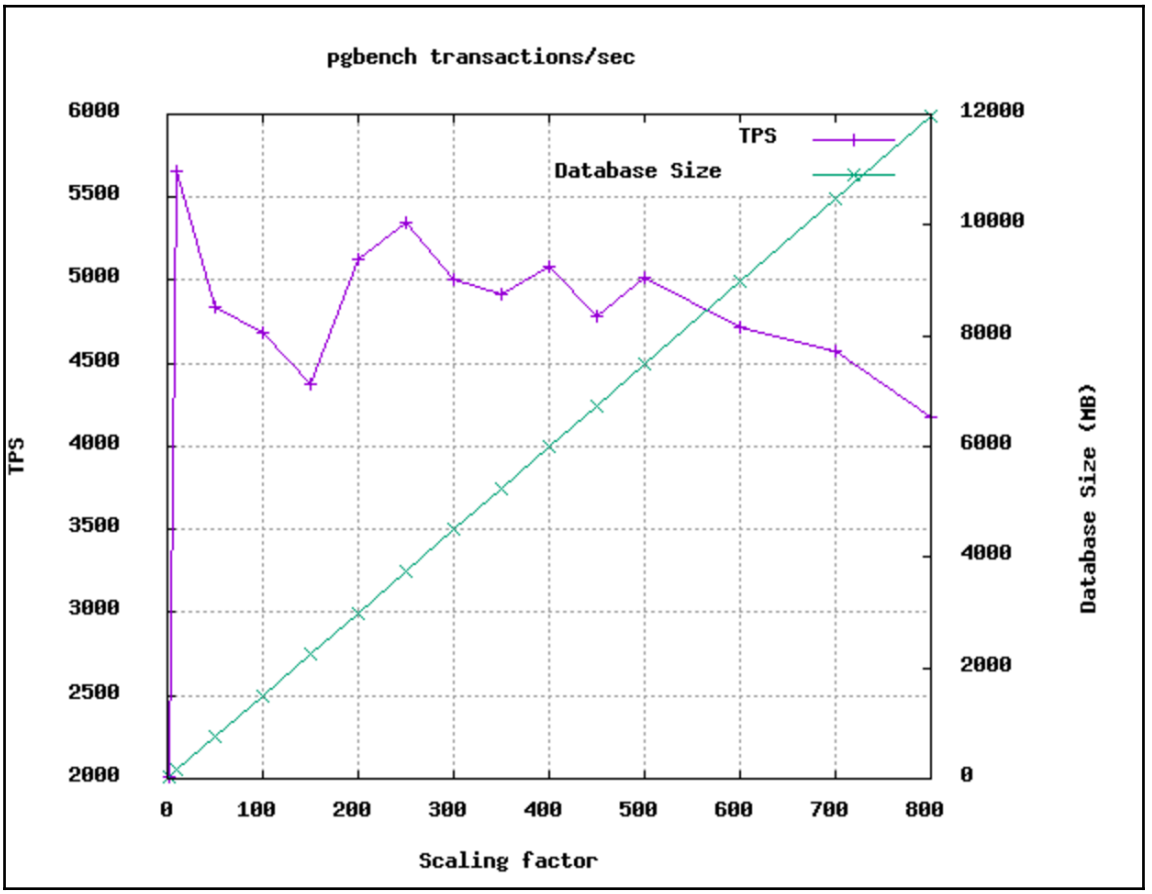
When running the simple read-only test, the results will be very good while the database fits in RAM, then they'll start to drop off fast after that. Note that the straight line you'll see on all these graphs is always the database size value, which is in units, shown by the right axis scale:



This curve has a similar shape no matter what hardware you use. The only thing that changes is where the two big break points are at, depending on how much RAM is in the server. You're limited by the speed of well-cached system RAM on the left side, by disk seek rate on the right, and a mix of the two in between.

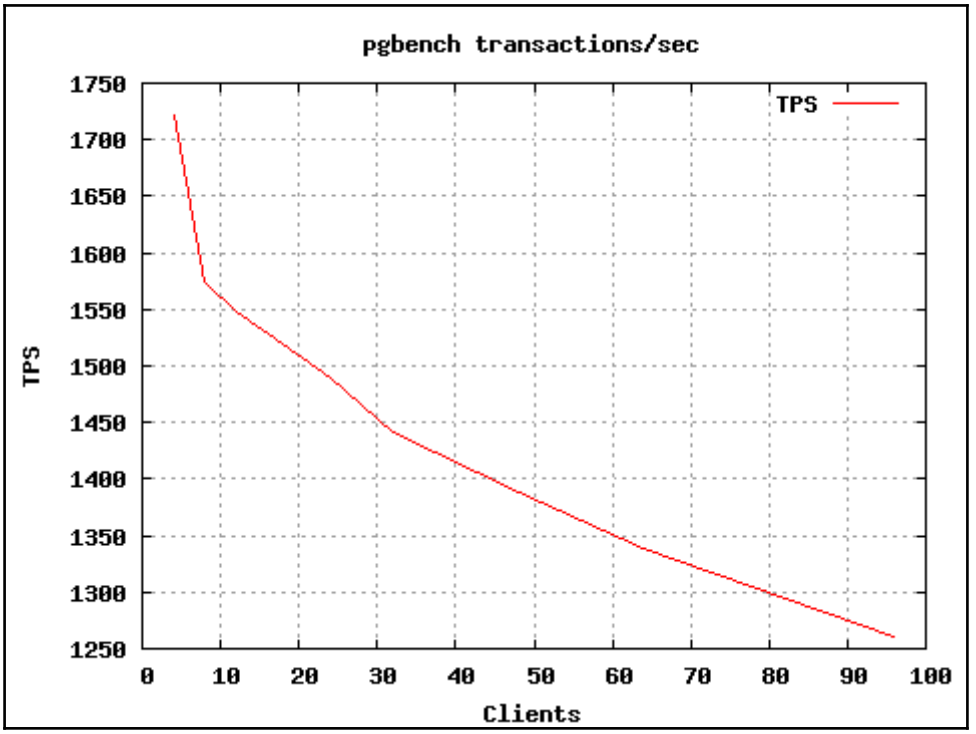
TPC-B-like test (Read/Write)

The scaling diagram has the same basic curve on the write-heavy test, but with fewer transactions occurring:



If you look carefully at the left-hand side of this, you can see one of the issues warned about earlier and mentioned in the `pgbench` documentation: results at a very low database scale are lower than when the database is a bit larger. This is due, in particular, to lock contention for the small branches table when running at higher client counts with a tiny scale.

The preceding results are averaged across all the client counts used. The number of clients also impacts results quite a bit:



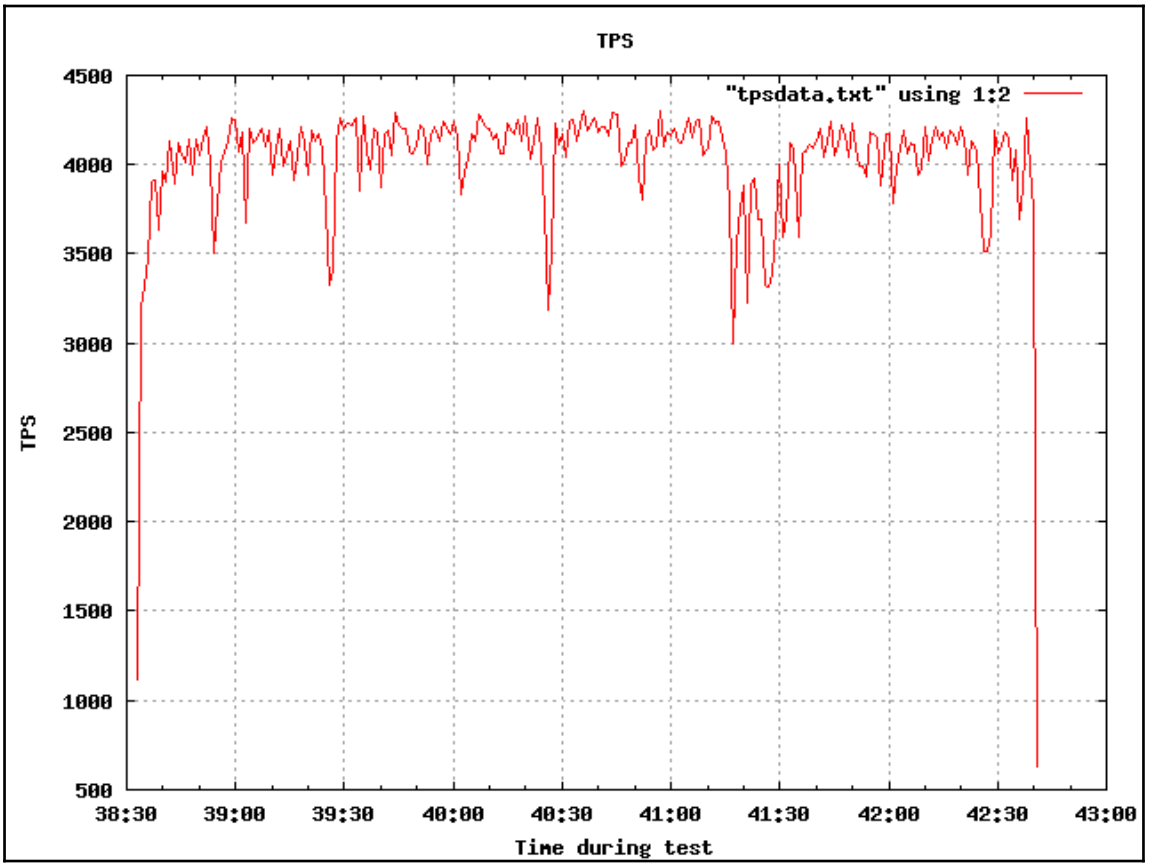
Correspondingly, the client results are also averaged out over all the scaling factors. The client scaling for the read-only test looks similar, just with smaller total values.

To get a really clear picture of the system, you should look at the web report produced by `pgbench-tools`. The scaling/client combination that produced the greatest single TPS value is usually an interesting number to note--the included utility script `fastest.sql` will show you the best results from a run.

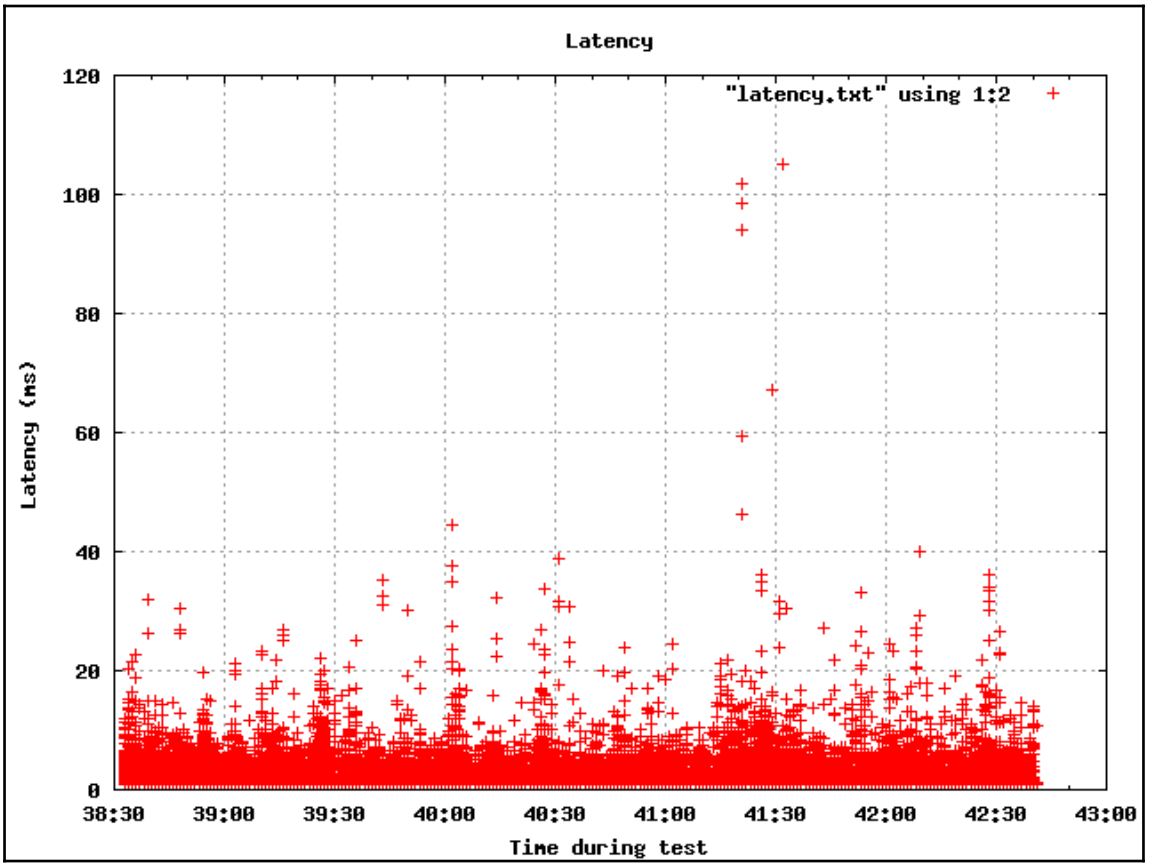
Latency analysis

Most of the time, you can think of latency--the time taken to process a single transaction--as the inverse of transactions/second. However, worst-case latency is an extremely important number to note for many applications, and you cannot determine it from any average measurement of TPS.

Accordingly, `pgbench-tools` always saves the latency data `pgbench` can produce, and it computes average, worst-case, and the 90th percentile of latency results in the HTML results it generates. This data, and the graphs that go along with it, are a major reason to use this toolset instead of running `pgbench` manually. If you're lucky, your results will look like this, a TPS over time graph from one of the fastest TPC-B-like individual runs in the set graphed above:

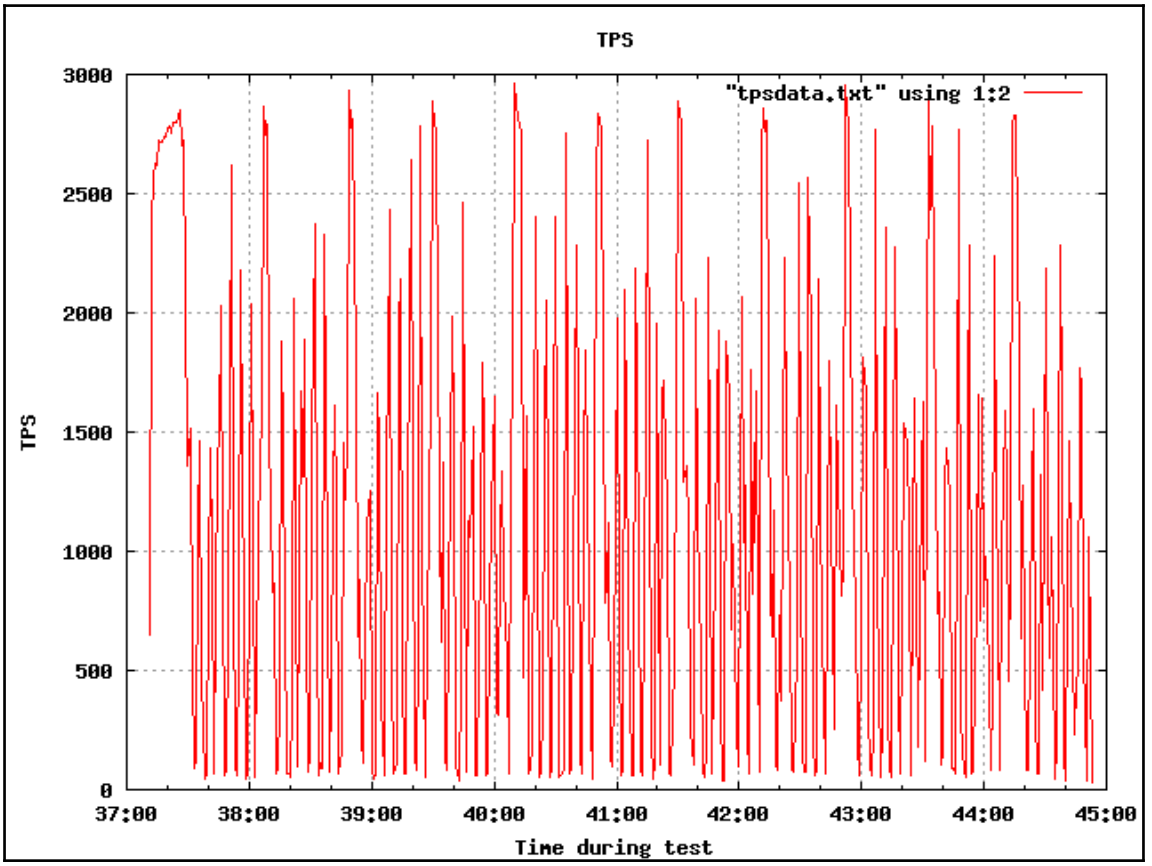


This run averaged 4,036 transactions/second, and aside from early ramp-up and shutdown, the transactions processed in any second was still quite high. The program also shows latency, how long each statement took to run, and graphs that for you. Graphing the latency of individual statements can show you both exactly how bad the worst-case is and what the distribution of those slow statements looks like:



Worst-case latency peaked at just over 100 milliseconds. The way the slow cases are clustered around the same period here is quite common in these results--when the server slows down, it will usually impact all the active clients at that time.

What does a bad result look like? The following example still had a reasonable average throughput of 1082 TPS, given it was running with a much larger database and client size (scale=250, clients=96). That's still an average latency of just under 1 millisecond. However, in this case, the worst-case latency is a frightening 6,348 milliseconds--some clients waited for over six seconds to get a response to their request! You can easily see how intermittent the server's responses were by looking at the TPS graph too:



The only reason the average TPS was so high is that when the server was working, it processed results quite quickly. But there were just as many periods it dropped to processing very few transactions in any given second, and sometimes it processed no transactions for several seconds.

When using pgbench to tweak write performance on a server, you need to be careful not to optimize for throughput at the expense of worst-case latency. Few applications would prefer an extra few percent throughput if it meant a several-second delay. The smoothness of the TPS graph, and correspondingly the worst-case latency results, are just as important (and possibly more so) than the average value.

Unfortunately, long delays are quite common on write-heavy pgbench tests, simply because of how it always writes at the maximum speed possible from the hardware. The typical cause is simple filling of the write cache on whatever controller is involved. In this example, the 256 MB write cache on the Areca controller used is simply taking a few seconds to clear when it's filled with random I/O that's blocking access to some resource a client needs. Behavior in this situation is commonly bad on Linux, where the kernel developers rarely optimize for latency at the expense of throughput, and it's easy to get backed up like this. More modern benchmarks typically introduce pauses, called think times, where clients aren't actively pushing work into the system, which allow their size to be scaled upwards and for you to see useful latency results without being hammered all the time by disk writes queuing. pgbench should not be considered representative of any real-world workload. Most programs don't constantly write like it does.

Sources for bad results and variation

There are many ways to get bad results from pgbench that don't mean anything valid. There are even more ways to get results that vary so much from one pgbench run to another that they don't mean what you would imagine them to. Normally, you should run any pgbench test at least three times and observe the average and variation before presuming that the test is valid. Taking the middle of three results is a common median technique for filtering those results into a single result for comparison.

You must let the test run for a while to get useful results. As a rule of thumb, if you haven't executed a minute's worth of runtime, you're unlikely to have gotten useful results at all. The main exception is the select-only test, which can give useful results in a few seconds. Generally, if you've seen >100,000 transactions, that's enough for you to start to believe the results, and fast hardware can execute that may SELECT statements in a few seconds nowadays. pgbench-tools includes some buffer report scripts aimed to help investigate this area, and the output from one of them is included in the web report results directory.

For write-heavy tests, such as the TPC-B-like default one, you need to run for long enough for a full checkpoint cycle to have occurred--at least a few minutes, sometimes as many as ten minutes--for the results to be meaningful. If you don't, the run-to-run variation you'll see will be very large, depending on whether that particular one included an active checkpoint or not in it.

Pay attention to the database size relative to the amount of RAM and other caches in your system, so that you understand what you're testing. Don't forget to set the database scale manually if executing a custom test yourself. `pgbench-tools` tries to take care of this detail for you, at least when you're using the standard `pgbench` tables.

Longer runs of `pgbench` will eventually end up triggering autovacuum and eventually can even bloat the database. This improved significantly in PostgreSQL 8.3 due to the HOT feature, and the overhead of autovacuum was lowered due to the FSM changes in 8.4. Because all the updates done in the standard `pgbench` test qualify for the mini-vacuum work that HOT can perform, and the updates to the `pgbench_accounts` table are the most difficult single statement to execute, it benefits quite a bit from that feature.

Whether autovacuum is running or not really alters how the test performs. You can get useful results with it turned on or off, just make sure you note which you're testing; be consistent about it, and understand that your test results will be impacted by that choice.

Developer PostgreSQL builds

If you have built PostgreSQL with the full set of development options turned on, your `pgbench` results will be considerably slowed by that. This is something to watch out for, particularly in the alpha and beta builds of the software, where these features may even be turned on in packaged versions such as the beta RPMs.

The worst performance hit comes from having assertions, checks in the code for unexpected conditions, enabled. The assertion overhead scales upward as you increase `shared_buffers`, which can result in the unexpected situation where large buffer cache sizes actually decrease performance on the benchmark. Whenever you're starting to use `pgbench`, it's a good idea to check that you don't have any assertions enabled by:

```
postgres=# show debug_assertions;
debug_assertions
------------------
off
```

That will keep you from wasting time running benchmarks that have confusing results.

Worker threads and pgbench program limitations

In earlier PostgreSQL versions, and by default in the current version, the `pgbench` program itself runs as a single process on the system. It queues up statements for each client to execute, then feeds them new ones as they report that they have finished responding to the previous one. In all situations, it is possible that the process will be slow if you have a large number of clients active. A common approach to work around that has been to run the `pgbench` program itself on another system, which unfortunately adds its own overhead too.

The situation is particularly bad on Linux systems running kernel 2.6.23 or later. The **Completely Fair Scheduler** (**CFS**) introduced in that version is not very fair to `pgbench`, when it runs in its standard mode using a socket connection. The result is that throughput only scales into the low tens of thousands of transactions per second, acting as if only one database core or processor is working correctly. Note how much the use of multiple threads speeded up on the initial read-only sample shown in the preceding text--an increase from 17 K to 31 K TPS--clearly shows that the single `pgbench` worker was the bottleneck. The worst manifestation of this problem is usually avoided when using `pgbench-tools`, as that always uses the TCP/IP host-based connection scheme instead.

Starting from PostgreSQL 9.0, you can work around these problems by increasing the number of worker threads that `pgbench` runs in parallel to talk to clients with. Each worker is assigned an equal share of clients that it opens and manages, splitting the load of clients up so that no single process has to service them all. The number of workers must be a multiple of the client count, so you need to choose that carefully. The program will abort complaining of bad input rather than do anything if it can't divide the client count requested equally among the number of workers specified; the way the program is written, each worker must have the same number of clients assigned to it.

Generally, you'll want one worker thread per core on your system if you have a version that this capability exists in. `pgbench-tools` allows you to set a maximum worker count while aiming to keep the clients properly divisible among workers. See the pgbench documentation for details. Using multiple workers is usually good for at least a 15% or greater speedup over using only a single worker, and as suggested earlier, increases of over 100% are possible. If you create too many workers--for example, making one worker per client when the number of clients far exceeds system cores--you can end up right back at where excessive swapping between processes on the system limits performance again.

pgbench custom tests

While the built-in tests are interesting, you can use `pgbench` instead as a testing harness to run your own database tests. This gets you multiple clients running at once, executing concurrent queries with a minimum of scripting.

Insert speed test

To give an idea of how this might be useful, imagine you want to test how fast your system can write data to disk using `INSERT` . You might copy the filler concept from the built-in tables (this is included in `pgbench-tools` as `init/insertsize.sql`):

```
create table data(filler text);
```

Then create a custom script that actually fills that much space up (also included as `insert-size.sql`):

```
insert into data (filler) values (repeat('X',:scale));
```

This borrows the concept of "scale" from the regular tests and uses it to specify the size of the data inserted. You can create your own variables with other names, pass values for them in on the command line, and then refer to them in your scripts.

Here's a sample session showing installation of this test using the sample version included with `pgbench-tools` , followed by manually running this custom test once with `pgbench` :

```
$ createdb insert
$ psql -d insert -f init/insertsize.sql
$ pgbench -s 1000 -f tests/insert-size.sql -n -c 4 -t 25000 insert
```

As with most custom tests, you have to manually specify the scale if you want that to be set correctly, and you have to skip the `VACUUM` step that is normally done on the standard tables with `-n`.

This data point isn't all that interesting on its own, but you can update the end of the `pgbench-toolsconfig` to run this test at a variety of sizes and client loads. Once you run that set, you'll see a more interesting pattern emerge:

```
SCALES="10 100 1000 10000"
SCRIPT="insert-size.sql"
TOTTRANS=100000
SETTIMES=1
SETCLIENTS="1 2 4 8 16"
SKIPINIT=1
```

On my test system, the amount of `INSERT` statements per second doesn't change very much whether you're writing 1 or 10,000 bytes into each one. It's primarily bounded by disk commit speed, which is always far less than the rated sequential write of the drive.

However, once you've reached a large enough number of clients combined with more bytes written each time, eventually the total completed `INSERT`/second rate starts to drop off. Find that point, measure actual disk writes using OS monitoring tools, and you'll have determined your database system's burst ability for writing out new data from inside the database itself, under a concurrent workload--something it's not very easy to measure directly.

This is a moderately complicated synthetic database benchmark, with variable client and workload sizes, and writing a client application this complicated from scratch would take a while. By leveraging `pgbench` and `pgbench-tools`, you can easily script something this complicated in only a few lines of code. You'll end up with a results database you can analyze for patterns using standard SQL queries and tools.

Transaction processing performance council benchmarks

The TPC-B benchmark is just one of many created by the Transaction Processing Performance Council. It's the second benchmark from them for what's called an **Online Transaction Processing** (**OLTP**) workload, which is one heavy on database writes. A more current benchmark in that same style is their TPC-C: `http://www.tpc.org/tpcc/`, which includes what can be a fairly complicated mix of transaction types that all revolve around order entry and related inventory in a warehouse.

Using genuine TPC benchmarks isn't practical to do on your own unless you're a major vendor prepared to license them and follow stringent reporting guidelines. However, there are free and open source clones of some of their benchmarks available, which follow the spirit of the benchmark without giving results that you can necessarily compare as directly across different types of systems. The **Database Test** (**dbt**) project has produced clones of some of these tests, outlined at `http://osdldbt.sourceforge.net/` but not necessarily available in the best form from there.

The benchmarking wiki page at `http://wiki.postgresql.org/wiki/Category:Benchmarking` lists a variety of benchmarking programs and resources related to using them with the database. The open-source alternative to TPC-C, named `dbt-2`, has its best documentation linked from there. It targeted PostgreSQL as its original development platform, although there are more databases supported now. If you'd like to simulate something that behaves more like a real-world workload than `pgbench`, but would like to keep things very heavy on writes, `dbt-2` is the standard program used in the PostgreSQL community for that purpose. Past tuning of the database itself has relied heavy on validation through improvements in `dbt-2` results.

The latest OLTP benchmark from the TPC is TPC-E, which models a large stock trading brokerage firm. An open version is available as `dbt-5`, but that code is still in active development. It may take a bit of work to get it running on your system.

Another popular test is TPC-H, which aims to model a decision support system running many moderately complicated queries at the same time. The TPC has released all the code and data necessary to run this particular benchmark, and one of the links on the benchmarking wiki page leads to information about that. However, not all the types of queries used by TPC-H are handled well by PostgreSQL, mainly because they haven't shown up in enough real-world workloads for anyone to optimize for them. The various commercial database vendors have to make sure they do well on each of these major benchmarks, so there's considerably more work put into making sure those products do well on TPC-C, TPC-E, and TPC-H in particular.

Summary

Benchmarking databases is a very extensive topic, and this chapter just introduces the major concepts. Having the `pgbench` tool bundled with the database is handy for doing smaller tests, but you need to be aware of its limitations before you rely too much on its results for your tuning efforts. The basis for the built-in `pgbench` tests is outdated, and it's unlikely to match real-world performance. It can still be useful as a synthetic performance test. You can write your own custom tests, either against the standard pgbench tables or new ones, and use the `pgbench` scripting capability to run those tests against multiple clients at once. Only a small number of the general database settings will impact `pgbench` results, since its queries are so simple. Using a benchmarking tool chain that graphs transaction rate and latency, such as `pgbench-tools`, is vital to monitor latency in addition to the throughput rate. Getting some of the modern, serious TPC benchmarking running with PostgreSQL will provide much more useful test results, but they are also much harder to get started with. The Next chapter will discuss the database index and how to improve performance using database indexes.

9
Database Indexing

An index is simply an organized list of values that appear in one or more columns in a table. The idea is that if you only want a subset of the rows in that table, a query can use the index to determine which rows match, rather than examining every row. Because an index has an order to it, it can also be used to speed up situations where a section of a table has to be sorted in order to return its values.

Indexes help the database cut down on the amount of data it needs to look at in order to execute a query. It's hard to write about indexes without knowing how queries are executed, and it's hard to discuss query execution without knowing what indexes do. This chapter tries to break that explanation deadlock by using simple index examples, where the associated query plans should make some sense even without the query tuning background covered in the next one.

Indexes should not be relied upon to force ordering of a query. If you want a query to be ordered a particular way, use `ORDER BY` to request it. The existence of an index with that order is more likely to be selected to execute that query, as it will skip a sorting step to do so. But just because a query normally uses an index that happens to have an order in it to return results, that does not mean you'll always receive them ordered that way. Query plans will change over time, to best match the underlying data as it too changes.

The main new thing to know to get started here is that if you put `EXPLAIN ANALYZE` in front of a `SELECT` statement, the database will show you how much work it expected that query to take, as well as how much it actually did. Reading `EXPLAIN` output is covered in more detail in the next chapter.

You might discover you get more out of this chapter if you return to it again after reading the next.

Topics we will cover in this chapter are:

- Index creation and maintenance
- Combining indexes
- Discussing different types of indexes and their use

Indexing example walkthrough

Discussion of indexes and query plans can quickly reach an overwhelming level of theory. Instead, this section will lead you through running various queries, with and without useful indexes, and showing how the query execution changes.

Measuring query disk and index block statistics

The best way to really understand how indexes work to save on the number of disk reads is to show how many blocks were actually used to satisfy that query. The following view merges together the two main sources for relevant table statistics, `pg_stat_user_tables` and `pg_statio_user_tables`:

```
postgres=# CREATE OR REPLACE VIEW table_stats AS
SELECT
  stat.relname AS relname,
  seq_scan, seq_tup_read, idx_scan, idx_tup_fetch,
  heap_blks_read, heap_blks_hit, idx_blks_read, idx_blks_hit
FROM
  pg_stat_user_tables stat
  RIGHT JOIN pg_statio_user_tables statio
    ON stat.relid=statio.relid;
CREATE VIEW
```

For the following examples, the following snippet of code is used after each statement (with `t` being the only table they use) to show the buffer usage count statistics that follow the query:

```
postgres=# SELECT pg_sleep(1);
 pg_sleep
----------

(1 row)

postgres=# \pset x on
Expanded display is on.
postgres=# SELECT * FROM table_stats WHERE relname='t';
-[ RECORD 1 ]--+--
relname        | t
seq_scan       | 0
seq_tup_read   | 0
idx_scan       |
idx_tup_fetch  |
heap_blks_read | 0
heap_blks_hit  | 0
idx_blks_read  |
idx_blks_hit   |

postgres=# SELECT pg_stat_reset();
-[ RECORD 1 ]-+-
pg_stat_reset |

postgres=# \pset x off
Expanded display is off.
```

The idea is that if you reset all the database statistics after the previous query, run a new query, then show the statistics, presuming no other activity is happening, the stats you see will only reflect what that last query did. The reason for the `pg_sleep` there is that statistics are at best only updated a few times per second. If you display them immediately after a query, they may not reflect recent activity. The `pset` commands are toggling on and off the optional query output mode in `psql` that shows each row returned on a separate line, which is much easier to read in the text for this data. This is equivalent to the `\x` you'll see sometimes used in other sections of this book, except rather than toggling the value it's explicitly setting it on or off; it works the same as specifying `-x` when running `psql` too.

Starting in PostgreSQL 9.0, it's possible to get a subset of this information--the part you're most likely to want--by using the `EXPLAIN (ANALYZE ON, BUFFERS ON)` syntax. An example is shown at the end of the walkthrough.

Running the example

The complete demo used to generate these examples is included with this book's sample files, and it was transformed into text by running the example like this:

```
$ psql -e -f indextest.sql > indextest.out
```

You might want to tinker with the source code or setup and try running it yourself in this way. The general technique used--create sample data, execute a query, gather usage statistics around it--is a very common one used for iteratively improving a given query as you tweak either its indexing or database parameters.

To keep complete control over what's going on while running this demo, the `autovacuum` parameter was turned off in the `postgresql.conf` file first:

```
$ psql -c "show autovacuum"
autovacuum
------------
off
```

If you have `autovacuum` turned on, you'll see slightly different results. It will do the appropriate `ANALYZE` work needed to make things work right automatically, rather than wait for the manual execution shown here to take care of that.

Sample data setup

To create some sample data to demonstrate how indexes work, let's create a table with a simple key/value structure. Each key can be an incrementing number, while the values vary from `0` to `10`. It's simple to use `generate_series` to create data like this:

```
postgres=# DROP TABLE IF EXISTS t;
DROP TABLE
postgres=# CREATE TABLE t(k serial PRIMARY KEY,v integer);
CREATE TABLE
postgres=# INSERT INTO t(v)
postgres-#   SELECT trunc(random() * 10)
postgres-#     FROM generate_series(1,100000);
INSERT 0 100000
```

Creating a serial field as a `PRIMARY KEY` will create a sequence and one index for you, as noted when you run the preceding code:

```
NOTICE:  CREATE TABLE will create implicit
 sequence "t_k_seq" for serial column "t.k"
NOTICE:  CREATE TABLE / PRIMARY KEY will create
 implicit index "t_pkey" for table "t"
```

In PostgreSQL 9.1, a new syntax introduced "UNIQUE/PRIMARY KEY USING" in which a unique and primary key can use an existing unique index.

The first valuable thing to know is how many pages (blocks of 8 KB each) the table has, and how many tuples (rows) the query planner believes the table to have. Since autovacuum is turned off, that information won't have appeared yet:

```
postgres=# SELECT relname,relpages,reltuples FROM pg_class WHERE relname='t';
 relname | relpages | reltuples
---------+----------+-----------
 t       |        0 |         0
(1 row)
```

This shows why turning autovacuum on by default in 8.3 was so valuable: in earlier versions, it was easy to miss this step, and end up running queries using completely wrong statistics information. A manual `ANALYZE` step will make sure all the statistics are current, and you can even compute the average number of rows stored per page of data. Here `VACUUM ANALYZE` is used because it also updates the hint bits related to the table, reducing variation in query runtimes later:

```
postgres=# VACUUM ANALYZE t;
VACUUM
postgres=# VACUUM;
VACUUM
postgres=# SELECT relname,relpages,reltuples FROM pg_class WHERE relname='t';
 relname | relpages | reltuples
---------+----------+-----------
 t       |      443 |     99882
(1 row)

postgres=# SELECT relname,round(reltuples / relpages) AS rows_per_page FROM pg_class WHERE relname='t';
 relname | rows_per_page
---------+---------------
 t       |           225
(1 row)
```

At a high level, then, these 100,000 rows of data are taking up 443 8 K blocks on disk, each one of which holds approximately 225 rows.

Simple index lookups

If you use the primary key on the table, which was built using an index, it's possible to retrieve a row using that index quite quickly. This uses what's unsurprisingly called an index scan:

```
postgres=# EXPLAIN ANALYZE SELECT count(*) FROM t WHERE k=1000;
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=4.31..4.32 rows=1 width=8) (actual time=0.029..0.029 rows=1 loops=1)
   ->  Index Only Scan using t_pkey on t  (cost=0.29..4.31 rows=1 width=0) (actual time=0.024..0.024 rows=1 loops=1)
         Index Cond: (k = 1000)
         Heap Fetches: 0
 Planning time: 0.263 ms
 Execution time: 0.061 ms
(6 rows)
```

Here is scan value after the previous query:

```
postgres=# SELECT * FROM table_stats WHERE relname LIKE 't';
-[ RECORD 1 ]--+--
relname        | t
seq_scan       | 0
seq_tup_read   | 0
idx_scan       | 1
idx_tup_fetch  | 1
heap_blks_read | 1
heap_blks_hit  | 0
idx_blks_read  | 1
idx_blks_hit   | 1
```

The second part here demonstrates what comes out of the `pg_stat_reset`/`table_stats` combination used around each of the statements in the example, and used in the provided source code example.

This demonstrates the first thing to recognize about indexes. Note the counts here for `heap_blks_hit` and `idx_blks_hit`. Even when you can use an index to look up something by its primary key, you cannot return the result without also looking at the associated row. Here there are three index blocks used to navigate down to the block where the row needed was located at, then that single heap (row data) block was returned. The reason for this is because PostgreSQL doesn't include visibility information in the indexes yet. The actual row must be consulted to determine if it's visible, it's not enough to confirm the key value "1000" exists in the index and then include it in the count here.

Accordingly, it's quite easy for an indexed scan to use more I/O than one that doesn't use an index. The index must be selective--filtering out a significant portion of the rows--before it is worth using.

Full table scans

If you ask for data that does not have a selective index available, such as anything related to the unindexed v column, the query can only occur by running a full sequential scan over all the rows in the table:

```
postgres=# EXPLAIN ANALYZE SELECT count(*) FROM t WHERE v=1;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=1716.27..1716.28 rows=1 width=8) (actual time=10.783..10.784 rows=1 loops=1)
   ->  Seq Scan on t  (cost=0.00..1691.53 rows=9898 width=0) (actual time=0.030..9.903 rows=9969 loops=1)
         Filter: (v = 1)
         Rows Removed by Filter: 90031
 Planning time: 0.083 ms
 Execution time: 10.809 ms
(6 rows)
```

Here is the result for the `SELECT` query:

```
postgres=# SELECT * FROM table_stats WHERE relname LIKE 't';
-[ RECORD 1 ]--+-------
relname        | t
seq_scan       | 1
seq_tup_read   | 100000
idx_scan       | 0
idx_tup_fetch  | 0
heap_blks_read | 0
heap_blks_hit  | 443
idx_blks_read  | 0
idx_blks_hit   | 0
```

You can see that every single one of the 443 blocks in this table were required to satisfy this query.

Another thing to recognize is that just because a query skips a portion of a table, that doesn't automatically mean that an index will be useful to execute it. You have to skip enough of a table for it to be worthwhile to do so. The following compound query, for example, still scans the whole table, even though it's only looking for part of the key values:

```
postgres=# EXPLAIN ANALYZE SELECT count(*) FROM t WHERE k>9000 AND v=5;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=1964.12..1964.13 rows=1 width=8) (actual time=12.559..12.559 rows=1 loops=1)
   ->  Seq Scan on t  (cost=0.00..1941.23 rows=9158 width=0) (actual time=1.006..11.874 rows=9080 loops=1)
         Filter: ((k > 9000) AND (v = 5))
         Rows Removed by Filter: 90920
 Planning time: 0.101 ms
 Execution time: 12.588 ms
(6 rows)
```

The `SELECT` query result:

```
postgres=# SELECT * FROM table_stats WHERE relname LIKE 't';
-[ RECORD 1 ]--+-------
relname        | t
seq_scan       | 1
seq_tup_read   | 100000
idx_scan       | 0
idx_tup_fetch  | 0
heap_blks_read | 0
heap_blks_hit  | 443
idx_blks_read  | 0
idx_blks_hit   | 0
```

Generally, any time a query is expected to hit a significant percentage of a table, you can expect a full sequential table scan to happen even if an index is usable there.

Index creation

To try and improve things, let's create an index on the value column, and look at how large that is:

```
postgres=# CREATE INDEX i ON t(v);
CREATE INDEX
postgres=# SELECT relname,reltuples,relpages FROM pg_class WHERE relname='i';
 relname | reltuples | relpages
---------+-----------+----------
 i       |    100000 |      276
(1 row)
```

Now, let's look at the table:

```
postgres=# SELECT relname,round(reltuples / relpages) AS rows_per_page FROM pg_class WHERE relname='i';
 relname | rows_per_page
---------+---------------
 i       |           362
(1 row)
```

One thing you should notice here is that the index is certainly not tiny compared to the table, which is surprisingly often the case. It's about half the size of the table, at 276 pages. You should always note how large any index created is relative to the table when determining if it's a worthwhile addition. The disk and maintenance overhead of an index can be considerable, and it needs to be justified by a performance improvement when running queries.

Lookup with an inefficient index

Was it worth creating this new index? Consider a simple query that uses it to filter out the table based on value:

```
postgres=# EXPLAIN ANALYZE SELECT COUNT(*) from t where v = 1;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=483.75..483.76 rows=1 width=8) (actual time=3.298..3.298 rows=1 loops=1)
   ->  Bitmap Heap Scan on t  (cost=12.17..482.50 rows=500 width=0) (actual time=0.695..2.499 rows=10001 loops=1)
         Recheck Cond: (v = 1)
         Heap Blocks: exact=443
         ->  Bitmap Index Scan on i  (cost=0.00..12.04 rows=500 width=0) (actual time=0.637..0.637 rows=10001 loops=1)
               Index Cond: (v = 1)
 Planning time: 0.070 ms
 Execution time: 3.324 ms
(8 rows)
```

Check the `SELECT` query for the same:

```
postgres=# SELECT * FROM table_stats WHERE relname LIKE 't';
-[ RECORD 1 ]--+------
relname        | t
seq_scan       | 0
seq_tup_read   | 0
idx_scan       | 1
idx_tup_fetch  | 10001
heap_blks_read | 0
heap_blks_hit  | 443
idx_blks_read  | 0
idx_blks_hit   | 29
```

We expect 1/10 of the table, or around 10,000 rows, to be returned by the index scan. That's what happens (9,903 in this case). It is effectively doing a scan on the whole table to do that though (443 blocks). This is clearly a step back from just looking at the whole table. This shouldn't come as any surprise really. Given how this table was created, there's likely to be a value where `v=1` on every data block, so we can't help but scan them all here. The planner doesn't know this (yet!) though, and it's actually made executing this query worse.

You might wonder why it executes faster than the above despite this. No special work was done to isolate out caching effects: most of the data here is already sitting in the database's buffer cache this time. Generally, you need to run any query you're measuring three times if you want to get a feel for caching. If the first is much slower, and the second and third are about the same runtime, it's likely that just the case data had to be read from disk or the OS cache for the first, while it was found in the database cache for the later ones.

Combining indexes

That doesn't mean the v index is useless though. If you are running something that is selective based on the k index, the one on v can be combined with that. With the index in place, this earlier example now uses a complicated query plan:

```
postgres=# EXPLAIN ANALYZE SELECT COUNT(*) from t where k > 9000 and v = 5;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=484.09..484.10 rows=1 width=8) (actual time=5.321..5.321 rows=1 loops=1)
   ->  Bitmap Heap Scan on t  (cost=12.08..483.67 rows=167 width=0) (actual time=1.764..4.429 rows=8988 loops=1)
         Recheck Cond: (v = 5)
         Filter: (k > 9000)
         Rows Removed by Filter: 887
         Heap Blocks: exact=443
         ->  Bitmap Index Scan on i  (cost=0.00..12.04 rows=500 width=0) (actual time=1.390..1.390 rows=9875 loops=1)
               Index Cond: (v = 5)
 Planning time: 0.111 ms
 Execution time: 5.357 ms
(10 rows)
```

Check the following `SELECT` query:

```
postgres=# SELECT * FROM table_stats WHERE relname LIKE 't';
-[ RECORD 1 ]--+-----
relname        | t
seq_scan       | 0
seq_tup_read   | 0
idx_scan       | 1
idx_tup_fetch  | 9875
heap_blks_read | 0
heap_blks_hit  | 443
idx_blks_read  | 28
idx_blks_hit   | 1
```

It doesn't turn out to be any more efficient (because all the data blocks were pulled anyway), but the amount of duplication isn't very bad--just a small amount of extra index overhead. It does demonstrate the next useful thing to know about indexes. We provided single column indexes covering `k` and on `v`. The query looked for specific ranges or values for both parameters. The query planner can combine those using what it calls a bitmap index scan. There was no need to create an index on the two fields together-(`k,v`) or (`v,k`)-to get this behavior. Two single column indexes were sufficient. This is normally the case with PostgreSQL. You can put an index on each of the columns you want to filter on, and they can be very efficiently combined in most cases. You really need to prove that a compound index covering more than one field is a worthwhile performance improvement before you should add one, particularly given they will only provide a useful speedup in queries that use that exact combination. Combining single column indexes is often as fast, and you can use them in any query that references a column that you indexed.

Planning for plan changes

This behavior--that index scans will flip into sequential ones once they access enough of a table--is one of the more surprising aspects to many people. Random disk access is so much slower than sequential scans that it makes sense to do so. But if all your data is cached in RAM (as is the case with the examples shown here), the advantages aren't as obvious. And that may very well always be the case with your data. In that case, you could even consider lowering database query parameters such as `random_page_cost` to make the optimizer understand your true situation.

This whole area is one of the things to be wary of when executing your own queries that expect to use indexes. You can test them out using some subset of your data, or perhaps using the initial production set. If you then deploy onto a system with more rows, or the production system expands its data size, these plans will change. What was once an indexed query might turn into an unindexed full sequential scan. What can be even more frustrating for developers is the opposite: a query coded against a trivial subset of the data might always use a sequential scan, just because the table is small and the selectivity of the index low. Run the same query against the production dataset, where neither is true anymore, and it could instead run using an index.

Whether an index will be used or not is a combination of both the query and the underlying table it's executing against, determined based on the statistics about the table. There is a mechanism that exists in PostgreSQL to manipulate the planning. In that mechanism, any scan can be disabled or enabled; and disabling and enabling of the scan affects the planning.

The following is the list of configurations that affect the query planning:

- enable_bitmapscan
- enable_hashagg
- enable_hashjoin
- enable_indexscan
- enable_indexonlyscan
- enable_material
- enable_mergejoin
- enable_nestloop
- enable_seqscan
- enable_sort
- enable_tidscan

Complete details of each can be found at PostgreSQL's official documentation:

`http://www.postgresql.org/docs/current/static/runtime-config-query.html`

In the case of the index scan, it can be forced by disabling sequential scans (`enable_seq_sacns=false`) and disabling nest-loops (`enable_nestloop=false`). If the index is still not used, the only reason behind this is the cost or the condition, which is not fulfilled for that index

Clustering against an index

If you want to look up things in this dataset based on the values in the v column, the fundamental issue here has already been described. Each page of data has about 226 rows in it, which you can expect to have around 22 rows that contain each value present. So there's no way to pick a value that only shows up in a subset of the table data blocks.

If there is a particular value you do want your data to be organized around, it's possible to do so using the `CLUSTER` command, described in more detail here:

```
CLUSTER t USING i;
ANALYZE t;
```

Since `autovacuum` is off, this is followed by a manual `ANALYZE` to make sure the change in structure of the table is noted. In particular, the statistics for the data will now reflect that the structure of the table is highly correlated with the `v` field.

Now, if you execute a query that only looks at a portion of the table based on the value field, that can be executed efficiently using an index scan:

```
postgres=# EXPLAIN ANALYZE SELECT COUNT(*) from t where v < 5;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=1788.09..1788.10 rows=1 width=8) (actual time=14.564..14.565 rows=1 loops=1)
   ->  Index Only Scan using i on t  (cost=0.29..1662.37 rows=50290 width=0) (actual time=0.026..10.254 rows=50274 loops=1)
         Index Cond: (v < 5)
         Heap Fetches: 50274
 Planning time: 0.084 ms
 Execution time: 14.595 ms
(6 rows)
```

Check the following output:

```
postgres=# SELECT * FROM table_stats WHERE relname LIKE 't';
-[ RECORD 1 ]--+------
relname        | t
seq_scan       | 0
seq_tup_read   | 0
idx_scan       | 1
idx_tup_fetch  | 50274
heap_blks_read | 0
heap_blks_hit  | 223
idx_blks_read  | 0
idx_blks_hit   | 139
```

Note that the combined I/O here (223 heap blocks, 139 index blocks) is indeed less than scanning the whole table (443) blocks looking for 50% of it. This is only true on an index scan that's expected to do sequential transfers; using a scattered 50% of an index would not be so efficient.

There's still a threshold here. At some point, getting even more of the table will switch to a sequential scan again:

```
postgres=# EXPLAIN ANALYZE SELECT COUNT(*) FROM t WHERE v<6;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=1843.47..1843.48 rows=1 width=8) (actual time=15.105..15.105 rows=1 loops=1)
   ->  Seq Scan on t  (cost=0.00..1693.00 rows=60190 width=0) (actual time=0.010..10.267 rows=60103 loops=1)
         Filter: (v < 6)
         Rows Removed by Filter: 39897
 Planning time: 0.063 ms
 Execution time: 15.127 ms
(6 rows)
```

The `SELECT` query:

```
postgres=# SELECT * FROM table_stats WHERE relname LIKE 't';
-[ RECORD 1 ]--+-------
relname        | t
seq_scan       | 1
seq_tup_read   | 100000
idx_scan       | 0
idx_tup_fetch  | 0
heap_blks_read | 132
heap_blks_hit  | 311
idx_blks_read  | 0
idx_blks_hit   | 0
```

This is because the index will not be considered selective enough to justify the overhead of using it anymore.

Explain with buffer counts

These examples should have proven to you that looking at the counts of blocks hit and read is valuable for determining whether a query plan is really being executed correctly or not. PostgreSQL 9.0 adds a feature to make this easier than before. Rather than looking at the `pg_stat*` data as done in the preceding example, you can request a count of buffers accessed directly when running `EXPLAIN`:

```
postgres=# EXPLAIN (ANALYZE ON, BUFFERS ON) SELECT count(*) FROM t WHERE v=5;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=313.43..313.44 rows=1 width=8) (actual time=2.148..2.148 rows=1 loops=1)
   Buffers: shared hit=31
   ->  Index Only Scan using i on t  (cost=0.29..288.29 rows=10057 width=0) (actual time=0.012..1.299 rows=10013 loops=1)
         Index Cond: (v = 5)
         Heap Fetches: 0
         Buffers: shared hit=31
 Planning time: 0.070 ms
 Execution time: 2.172 ms
(8 rows)
```

To see how well this works, try running that query again after dropping the index that's selective on `v`:

```
postgres=# DROP INDEX i;
DROP INDEX
postgres=# EXPLAIN (ANALYZE ON, BUFFERS ON) SELECT count(*) FROM t WHERE v=5;

                                                    QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=1718.14..1718.15 rows=1 width=8) (actual time=10.130..10.130 rows=1 loops=1)
   Buffers: shared hit=443
   ->  Seq Scan on t  (cost=0.00..1693.00 rows=10057 width=0) (actual time=0.008..9.271 rows=10013 loops=1)
         Filter: (v = 5)
         Rows Removed by Filter: 89987
         Buffers: shared hit=443
 Planning time: 0.071 ms
 Execution time: 10.152 ms
(8 rows)
```

That shows the sequential scan running against the full 443 pages of data this table is known to occupy, instead of the much smaller count the earlier query executed in.

Index creation and maintenance

Creating an index is one of the most intensive operations you can do on a database. When populating a database using tools such as `pg_restore`, the time spent building indexes can be the longest part of the data loading. And you can't necessarily just ignore them afterwards.

Index rebuilding can be an expected part of regular database maintenance, particularly in the case where many rows (but not all of them) are being deleted from a section of an index. There's more information about that topic back in `Chapter 7`, *Routine Maintenance*.

Unique indexes

A unique index ensures that you won't have more than one row containing a particular key value. Unique indexes are quite common in proper database design, both for improving performance--a unique index lookup is normally fast--as well as data integrity, preventing erroneous duplications of data. Only B-tree indexes can currently be used as unique ones.

There are three ways you can create a unique index, only two of which are recommended. The first you saw at the beginning of the chapter walkthrough: when you mark a field as `PRIMARY KEY`, a unique index is created to make sure there are no key duplicates.

But a primary key is just a specially marked case of having a unique constraint on a table. The following two statement sets give almost identical results:

```
CREATE TABLE t(k serial PRIMARY KEY,v integer);

CREATE TABLE t(k serial,v integer);
ALTER TABLE t ADD CONSTRAINT k_key UNIQUE (k);
```

Except that the primary key case is labeled better in the system catalog (and is therefore preferable to an internal database documentation standpoint). Note that the index created for the purpose of enforcing uniqueness is a perfectly functional index usable for speeding up queries. There's no need in this case to create a regular index on `k` so that you can look up values by the key faster, as the unique index provides that already.

It's also possible to directly create an index using `CREATE UNIQUE INDEX`. This is considered bad form and you should avoid doing so. It effectively creates a constraint, without labeling it as such in the list of constraints. If you want values to be unique, add a constraint; don't add the index manually.

One caveat when using unique indexes is that null values are not considered equal to one another. This means that you could have several entries inserted that each have a null value for their key. To avoid the problems this can introduce, it's recommended to always add `NOT NULL` to the fields that are going to participate in the index when the table is created. Rejecting null values when they don't make sense in your data is a good habit to adopt beyond this particular issue--most fields in a well-designed database will reject null values.

Concurrent index creation

When you build an index normally, it locks the table against writes. This means that reads using `SELECT` against the table will work fine, but any attempt to insert, update, or delete a row in it will block until the index creation is finished. The statement will pause and wait for the index lock to be released, rather than throwing an error immediately. Since index rebuilding on large tables can easily take hours, that's a problem. There's some potential for a deadlock producing error here as well, if the client backend already had some locks on that table before the index build began.

Accordingly, on any production database with significant table sizes, where a user being blocked for a long time is an issue, you usually want to build indexes using `CREATE INDEX CONCURRENTLY`. This is not the default because the concurrent index build is much less efficient than the standard one that takes a lock. It scans the table once to initially build the index, then makes a second pass to look for things added after the first pass.

There is also a risk to any concurrent index build on indexes that enforce some constraint, such as any `UNIQUE` index. If an unacceptable row is found during that second pass, such as a uniqueness violation, the concurrent index build fails, having wasted resources without accomplishing anything. The resulting index will be marked `INVALID` when viewed with the `psql` utility and will not be usable for queries. But changes to the table will still update values in it--with wasted overhead. You can recover from this situation by dropping the index and trying to build it again, presuming you know the same issue won't pop up again (for example, all duplicate values have been erased). You can also use `REINDEX` to fix it; but that can't be done concurrently, it will take the usual write lock the whole time.

Clustering an index

`CLUSTER` works by making a whole new copy of the table, sorted by the index you asked to traverse the original in. It's as if you had inserted them in that order in the first place. Once built, the original copy of the data is dropped. `CLUSTER` requires an exclusive lock on the table and enough disk space to hold the second, reorganized copy--essentially, twice the amount of disk space, temporarily. You should run `ANALYZE` afterwards in order to update the statistics associated with this table, to reflect its new ordering.

Clustering is useful for getting faster results from range-based queries, when selecting a set of values that is between two end points (with the beginning or end being considered an end point in that context). It can generally speed up access to fragmented data that might have been introduced from any number of sources.

Clustering is a one-time act. Future insertion does not respect the clustered order, even though the database does remember what index you last used if you `CLUSTER` again.

Starting in PostgreSQL 9.0, the way VACUUM FULL is executed uses the same basic logic as `CLUSTER`: rewrite the whole thing, rather than try to clean up problem spots inefficiently.

Fillfactor

When you create a new index, not every entry in every index block is used. A small amount of free space, specified by the `fillfactor` parameter, is left empty. The idea is that the first set of changes to that index, either updates or insertions, can happen on the same index blocks, therefore reducing index fragmentation.

The default `fillfactor` for B-tree indexes is 90%, leaving 10% free space. One situation where you might want to change this is a table with static data, where the data won't change after the index creation. In this case, creating the index to be 100% full is more efficient:

```
CREATE INDEX i ON t(v) WITH (FILLFACTOR=100);
```

On tables that are being populated randomly, or ones that are heavily updated, reducing the `fillfactor` from its default can reduce index fragmentation, and therefore the amount of random disk access required to use that index once it's more heavily populated.

Reindexing

When indexes become less efficient due to being spread over the disk inefficiently, either scattered around too randomly or too bloated with now unused values, the index will benefit from being rebuilt. This is most easily done with the `REINDEX` command, and that can't be done concurrently. More information about detecting when indexes need to be rebuilt and how to do so can be found in `Chapter 7`, *Routine Maintenance*.

Index types

Indexing in PostgreSQL is completely programmable. It's straightforward (albeit not too easy) to create a totally custom index type, even with customized operators for how to compare values. A few unusual index types are included with the core database.

B-tree

The standard index type is the B-tree, where the B stands for balanced. A balanced tree is one where the amount of data on the left and right side of each split is kept even, so that the amount of levels you have to descend to reach any individual row is approximately equal.

The B-tree can be used to find a single value or to scan a range, searching for key values that are greater than, less than, and/or equal to some value. They also work fine on both numeric and text data. Recent versions of PostgreSQL (8.3 and later) can also use an index to find (or avoid) `NULL` values in a table.

Text operator classes

It's possible to use an index on a text field to speed up finding rows that start with a substring of interest. A query like this that uses `LIKE` to match the start of a string:

```
SELECT * FROM t WHERE t.s LIKE 'start%';
```

Can use an index on that string field to return answers more quickly. This is not useful if your comparison tries to match the middle or end of the column though. In addition, if your database uses something other than the simple `C` locale, the default way values are compared in indexes can't be used for this purpose. You'll need to create the index using a special mode for locale sensitive character by character comparisons, like this:

```
CREATE INDEX i ON t (s text_pattern_ops);
```

In this example, s would be a text field; there are different operators for varchar and char types. In addition to `LIKE`, this form of index can also be used for some types of text regular expression matches, too.

See the *Operator Classes and Operator Families* section of the PostgreSQL manual for more information on this topic. You can confirm your locale, and therefore whether you need to use this approach, by looking at the `lc_collate` parameter:

```
postgres=# show lc_collate;
lc_collate
-------------
en_US.UTF-8
```

This sample database is not using the `C` locale, and therefore would need to use the `text_pattern_ops` operator class in order for `LIKE` queries to use an index on a text field.

Hash

The hash index type can be useful in cases where you are only doing equality (not range) searching on an index, and you don't allow NULL values in it. However, it is easy for hash indexes to become corrupt after a database crash, and therefore ineffective for queries until manually rebuilt. The advantages to using a hash index instead of a B-tree are small compared to this risk. You normally shouldn't ever use the hash index type. But if you are willing to spend a considerable amount of time analyzing its usefulness and ensuring the index will be rebuilt if it becomes corrupt, it is possible you can find a purpose for it.

GIN

Regular indexes are optimized for the case where a row has a single key value associated with it, so that the mapping between rows and keys is generally simple. The **Generalized Inverted Index** (**GIN**) is useful for a different sort of organization. GIN stores a list of keys with what's called a posting list of rows, each of which contain that key. A row can appear on the posting list of multiple keys, too.

With the right design, GIN allows efficient construction and search of some advanced key/value data structures that wouldn't normally map well to a database structure. It leverages the ability to customize the comparison operator class used for an index, while relying on a regular B-tree structure to actually store the underlying index data.

GIN is useful for indexing array values, which allows operators such as searching for all rows where the array column contains a particular value, or has some elements in common with an array you're matching against. It's also one of the ways to implement full-text search. There are several examples that use GIN that suggest how it can be utilized to build custom data storage among the example PostgreSQL `contrib` modules.

Anytime you're faced with storing data that doesn't match the usual single key to single value sort of structure regular database indexes expect, GIN might be considered as an alternative approach. It's been applied to a wide range of interesting indexing problems.

GiST

A **Generalized Search Tree** (**GiST**) provides a way to build a balanced tree structure for storing data, such as the built-in B-tree, just by defining how keys are to be treated. This allows using the index to quickly answer questions that go beyond the usual equality and range comparisons handled by a B-tree. For example, the geometric data types that are included in PostgreSQL include operators to allow an index to sort by the distance between items and whether they intersect.

GiST can also be used for full-text search, and it too has a very rich library of `contrib` modules that use its features. When you need a really customized type of index, with operators that go beyond the normal comparison ones, but the structure is still like a tree, consider using GiST when a standard database table doesn't perform well.

SP-GiST

SP-GiST is an abbreviation for **space-partitioned GiST**. SP-GiST supports partitioned search trees, which facilitate development of a wide range of different non-balanced data structures, such as quad-trees, k-d trees, and suffix trees (tries). The common feature of these structures is that they repeatedly divide the search space into partitions that need not be of equal size. Searches that are well matched to the partitioning rule can be very fast: `https://www.postgresql.org/docs/current/static/spgist-intro.html`.

Some of the information of the SP-GIST is taken from Purdue University's SP-GiST Indexing Project (`https://www.cs.purdue.edu/spgist/`).

BRIN

BRIN is a new index. **BRIN** stands for **Block Range Indexes**. The block ranges are ranges of pages that are adjacent in the table. For each block range, at least the minimum information of that range is stored by the index. Currently minimum and maximum values per block are stored by the index. A detailed description of the BRIN index can be found at PostgreSQL's official documentation: `http://www.postgresql.org/docs/current/static/brin-intro.html`.

This index is used for very large tables:

```
postgres=# CREATE TABLE foo ( a int, b varchar, c text);
CREATE TABLE
postgres=# INSERT INTO foo VALUES(generate_series(1, 1000000), 'Test Data', 'Test Column');
INSERT 0 1000000
postgres=# EXPLAIN SELECT count(*) FROM foo WHERE a > 100 AND a < 400;
                           QUERY PLAN
-----------------------------------------------------------------
 Aggregate  (cost=16735.89..16735.90 rows=1 width=8)
   ->  Seq Scan on foo  (cost=0.00..16728.07 rows=3125 width=0)
         Filter: ((a > 100) AND (a < 400))
(3 rows)
```

Create an index of column `a` and column `b` on the table `foo`:

```
postgres=# CREATE INDEX foo_idx ON foo USING BRIN(a, b);
CREATE INDEX
postgres=# EXPLAIN SELECT count(*) FROM foo WHERE a > 100 AND a < 400;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=1029.34..1029.35 rows=1 width=8)
   ->  Bitmap Heap Scan on foo  (cost=15.10..1028.58 rows=303 width=0)
         Recheck Cond: ((a > 100) AND (a < 400))
         ->  Bitmap Index Scan on foo_idx  (cost=0.00..15.03 rows=303 width=0)
               Index Cond: ((a > 100) AND (a < 400))
(5 rows)
```

"BRIN is a new index access method intended to accelerate scans of very large tables, without the maintenance overhead of B-trees or other traditional indexes. They work by maintaining "summary" data about block ranges. Bitmap index scans work by reading each summary tuple and comparing them with the query quals; all pages in the range are returned in a lossy TID bitmap if the quals are consistent with the values in the summary tuple, otherwise not. Normal index scans are not supported because these indexes do not store TIDs."
Ref:
`http://www.postgresql.org/message-id/E1XmpRL-0001Zh-Sd@gemulon.postgresql.org`
`http://michael.otacoo.com/postgresql-2/postgres-9-5-feature-highlight-brin-indexes/`

Index only scans

In PostgreSQL 9.2, a new feature has been added called Index Only Scans. This is a performance feature in which data for only certain types of queries is retrieved from the index. It avoids accessing the table if it can find the data in the index. The PostgreSQL planner selects "Index Only Scans" if the entire column in the query is part of the index. It is very obvious that if a tuple can be found in the index, then there is no need to get the tuple from heap because it can cause I/O and I/O is bad for performance. Further details about Index Only Scans can be found at `https://wiki.postgresql.org/wiki/Index-only_scans`.

Create a new table using the `CREATE TABLE` statement:

```
postgres=# CREATE TABLE foo ( a int, b varchar, c text);
```

Create an index of columns `a` and `b` on the table foo:

```
postgres=# CREATE INDEX foo_idx ON foo ( a, b);
```

Insert the dummy data into table `foo`:

```
postgres=# INSERT INTO foo VALUES(generate_series(1,10000),
 'Test Data', 'Test Column');
```

After that, explain the query where all the columns of the query are part of the index `foo_idx`. The PostgreSQL planner selects the `Index Only Scan` for that:

```
postgres=# EXPLAIN SELECT a FROM foo WHERE a = 100;
                              QUERY PLAN
------------------------------------------------------------------------
 Index Only Scan using foo_idx on foo  (cost=0.29..8.30 rows=1 width=4)
   Index Cond: (a = 100)
(2 rows)

postgres=# EXPLAIN SELECT a,b FROM foo WHERE a = 100;
                              QUERY PLAN
------------------------------------------------------------------------
 Index Only Scan using foo_idx on foo  (cost=0.29..8.30 rows=1 width=14)
   Index Cond: (a = 100)
(2 rows)
```

If the query has columns that are not part of the index, the planner does not use the `Index Only Scan`:

```
postgres=# EXPLAIN SELECT * FROM foo WHERE a = 100;
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using foo_idx on foo  (cost=0.29..8.30 rows=1 width=26)
   Index Cond: (a = 100)
(2 rows)
```

count(*)

The `count(*)` is an expensive operation in PostgreSQL because of its MVCC architecture. Due to PostgreSQL's MVCC architecture, there is no straightforward way to count the rows without traversing all the rows. Here is an example where we are counting the number of rows in a table and a sequential scan is used to count the rows in an expansive way:

```
postgres=# EXPLAIN SELECT Count(*) FROM foo;
                          QUERY PLAN
---------------------------------------------------------------
 Aggregate  (cost=199.00..199.01 rows=1 width=8)
   ->  Seq Scan on foo  (cost=0.00..174.00 rows=10000 width=0)
(2 rows)
```

But in a case where the query has a restriction clause (`WHERE` clause), the `Index Only Scan` is used instead of the sequential scan. It is much faster than the sequential scan and reduces the time of `count(*)`:

```
postgres=# EXPLAIN SELECT Count(*) FROM foo WHERE a = 100;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Aggregate  (cost=8.30..8.31 rows=1 width=8)
   ->  Index Only Scan using foo_idx on foo  (cost=0.29..8.30 rows=1 width=0)
         Index Cond: (a = 100)
(3 rows)
```

Visibility map

A feature called the **visibility map** was introduced in PostgreSQL version 8.4. The visibility map keeps track of pages that contain the tuple visible to all transactions. The `Index Only Scan` is used when all columns are part of the index and the visibility map shows that all the tuples on the page are visible to all. A detailed description of the visibility map can be found at `http://www.postgresql.org/docs/current/static/storage-vm.html`.

Advanced index use

Many PostgreSQL users will only ever use simple B-tree indexes on a single column. There are of course a few ways to build more complicated ones too, though.

Multicolumn indexes

Consider a database table that is storing a category and a subcategory for a series of data. In this case, you don't expect to ever specify a subcategory without also providing a category. This is the sort of situation where a multicolumn index can be useful. B-tree indexes can have to up 32 columns they index, and any time there's a natural parent/child relationship in your data this form of index might be appropriate.

Create an index for that sort of situation:

```
CREATE INDEX i_category ON t (category,subcategory);
```

The index could be used for queries that look like this:

```
SELECT * FROM t WHERE category='x' and subcategory='y';
SELECT * FROM t WHERE category='x';
```

But it is unlikely to be useful for this form:

```
SELECT * FROM t WHERE subcategory='y';
```

Had you instead created two indexes for both category and subcategory, those could satisfy either individual type of search, as well as the combined one (via combining with a bitmap index scan). However, the result would likely be larger and involve more overhead to update than the multicolumn version, which is the downside to that flexibility, as well as being a bit less efficient when a query selecting on both columns is run.

Indexes for sorting

B-tree indexes store their entries in ascending order. Starting in PostgreSQL 8.3, the nulls are also stored in them, defaulting to the last in the table. You can reverse both of those defaults as follows:

```
CREATE INDEX i ON t(v DESC NULLS FIRST);
```

The query planner will use an index that returns rows in sorted order in some cases when `ORDER BY` has been specified. Generally, this only happens when a small number of rows are being returned by the table, which is the opposite of what you might expect.

This is because reading the index blocks is optional. You can always derive the data by sorting the data, and the planner considers index reads to be random rather than sequential access. If a large percentage of the table is being read, directly reading the table--sequentially--and sorting the result will have a lower estimated cost than the redundant disk access of reading both the random index blocks and the data blocks, too. Expecting to see an index used for `ORDER BY`, and instead having all the data read in and sorted, is one way you can end up with considerable unexpected disk activity, in the form of temporary sorting files.

If you are using `ORDER BY` and `LIMIT`, an index is much more likely to be used. In that case, an index might immediately return the limited number of rows needed using the index, while a standard query will have to read all the rows and sort them just to return any of them.

Partial indexes

Indexes do not have to cover the entirety of a table. You can create smaller, targeted indexes that satisfy a particular `WHERE` clause, and the planner will use those when appropriate. For example, consider the case where you are tracking a set of customer accounts, and want to flag a small number of them for special processing; let's just say they are flagged as `interesting` with a Boolean of that name. You could then create a partial index:

```
CREATE INDEX accounts_interesting_index ON accounts
 WHERE interesting IS true;
```

(That's intentionally more verbose than necessary just to demonstrate the syntax.) You'd expect this index to be very small, relative to one that included every account. And it would return the list of interesting accounts quite quickly, because the index will only contain them--not the whole table. As always, this index could also be combined with others using a bitmap index scan.

Expression-based indexes

The classic example for an expression-based index is when you typically search based on a name only after converting it to a particular case:

```
CREATE INDEX i_lower_idx ON t (lower(name));
SELECT * FROM t WHERE lower(name) = 'x';
```

That query will run using that expression index, which turns it into a single value indexed lookup in the table. An index based on the username without converting to lower case first will not be useful in that context.

One non-obvious thing you can implement using partial indexes is forcing uniqueness for an expression, rather than just a combination of fields:

```
CREATE UNIQUE INDEX I ON lower(username);
```

That would only allow one row to exist at any time with that value for the expression. You might want to handle this as a constraint instead though.

Note that every time you insert or update an entry in a table with this sort of index, the function used needs to be called to compute an appropriate value for the index. That makes the insert/update overhead of this sort of index higher than one that just uses a column list. Even more so than regular indexes, the index needs to give you something that's selective against your data for the overhead of building the index to be worthwhile.

Indexing for full-text search

PostgreSQL includes a capable **full-text search** (**FTS**) package, available as a core feature starting in 8.3. Both the GIN and GiST indexes can be used to accelerate text searches done that way. The basic idea is that GIN is better suited for relatively static data, while GiST performs better with frequently updated, dynamic text. The trade-offs involved in choosing between those index types are heavily documented in the PostgreSQL manual chapter devoted to full-text search.

That documentation can also be considered a tutorial on the relative strengths and weaknesses of GIN versus GiST, which can be useful when trying to decide which of them to use for other applications. For example, the hstore key/value implementation available as a `contrib` module can be handy for efficiently storing that sort of data when it doesn't need to have full SQL semantics, an approach popularized by NoSQL recently.

There are both GIN and GiST implementations of hstore available, and choosing between the two is complicated. Knowing more about their relative insert versus query performance characteristics, as seen from the context of FTS, provides a useful vantage point to compare the two that you can extrapolate from.

Summary

Choosing what indexes to add to your tables remains one of those areas where creative tuning work can still trump mechanical attempts to measure and react. There are some index tuning wizards available for other databases, but even the best of them just provide suggestions rather than dependable advice. It's important to be systematic about your indexes though. Because adding an index increases overhead every time you add or change rows in a table, each index needs to satisfy enough queries to justify how much it costs to maintain. There is more information about determining whether the indexes on your system are working effectively in `Chapter 11`, *Database Activity and Statistics*.

Measure actual block reads to determine whether an index is truly effective.

Queries cannot be answered using only the data in an index; the data blocks must be consulted for row visibility information in all cases. An index is only useful if it is selective: it can be used to only return a small portion of the rows in a table. Indexes can be quite large relative to their underlying table. If two useful indexes exist to answer a query, they can be combined, and that's preferable in many cases to using a multicolumn index.

Indexes can require periodic rebuilding to return them to optimum performance, and clustering the underlying data against the index order can also help improve their speed and usability for queries.

A variety of advanced techniques are available to build customized index structures in PostgreSQL, for situations where the standard B-tree index on simple fields isn't powerful enough.

Next we will study the query optimization, which involves explaining queries, working with join, aggregate, and window functions.

10
Query Optimization

For some database administrators and application developers, query optimization is the most important part of database performance tuning. Earlier chapters have wandered some distance from there, because if your fundamental server setup isn't good, no amount of query tuning will help you. But things like selecting good hardware for a database server are rare events. Figuring out why a query is slow and how to improve it is something you can expect to happen all the time. This is particularly true because query plans drift over time, and some aspects of your database will become less efficient. Maintenance to improve the latter problem is covered in the next few chapters. How queries execute and improving that execution is this chapter's topic.

In this chapter, we will be covering the following topics:

- Explanation of the EXPLAIN command
- Query plan node structure
- Cost computation
- Processing nodes
- Scans
- Subquery scan and subplan
- CTE
- Joins
- Statistics

Sample datasets

In order to show you queries to optimize, you need data. So far, randomly generated data for a small number of tables has been good enough for that, but that has its limits and they were reached in the last chapter.

Obtaining sample data of a significant size for benchmarking purposes is a never ending struggle, because the pace of hardware progress means yesterday's massive test database can be today's trivial workload. A listing that's kept up to date with useful new sources for test databases is at `http://wiki.postgresql.org/wiki/Sample_Databases` and some of these are part of a PostgreSQL sample database project, which has several examples worth checking out.

Pagila

The Pagila database is a particularly interesting sample because it showcases many advanced PostgreSQL features. It relies heavily on stored procedures and even partitions some of its data. The main reason it's not used for examples here is because it's very small, only a few megabytes in size. The non-free license on the documentation for the original database it's based on (Sakila, a part of MySQL) makes it problematic to quote from here too.

Dell Store 2

The Dell Store 2 was originally distributed by Dell at `http://linux.dell.com/dvdstore/` as part of an effort to create a vendor neutral comparison test. It includes everything from the database to an e-commerce web application. Their sample store sells DVDs to customers and includes products, orders, inventory, and customer history.

A PostgreSQL port of the database part of the Dell Store application is available from the sample databases project at `http://pgfoundry.org/projects/dbsamples/`.

While the benchmark can theoretically be targeted to create three sizes, only the small (10 MB) version is available for PostgreSQL so far. The data generators for the medium (1GB) and large (100 GB) stores haven't been implemented for PostgreSQL yet. That makes the Dell Store 2 a modest test database for examples here, but not one really representative of a serious workload by modern standards.

Obtaining and installing Dell Store 2 can be this simple:

```
$ wget http://pgfoundry.org/frs/download.php/543/
dellstore2-normal-1.0.tar.gz
$ tar xvfz dellstore2-normal-1.0.tar.gz
$ cd dellstore2-normal-1.0/
$ createdb dellstore2
$ createlang plpgsql dellstore2
$ psql -f dellstore2-normal-1.0.sql -d dellstore2
$ psql -d dellstore2 -c "VACUUM VERBOSE ANALYZE"
```

With indexes and related overhead, the result is actually 21 MB:

```
dellstore2=# SELECT pg_size_pretty(pg_database_size('dellstore2'));
 pg_size_pretty
----------------
 24 MB
(1 row)
```

This means that even the tiny default `shared_buffers` on most systems can hold the whole set of data, but the queries aren't completely trivial. Here are the major tables and indexes:

```
                table                |  size
-------------------------------------+---------
 public.customers                    | 3808 kB
 public.orderlines                   | 2840 kB
 public.cust_hist                    | 2368 kB
 public.products                     | 808 kB
 public.orders                       | 800 kB
 public.inventory                    | 400 kB

            index                    |  size
-------------------------------------+---------
 public.ix_orderlines_orderid        | 1336 kB
 public.ix_cust_hist_customerid      | 1080 kB
 public.ix_cust_username             | 544 kB
 public.customers_pkey               | 368 kB
 public.ix_order_custid              | 232 kB
 public.orders_pkey                  | 232 kB
 public.ix_prod_special              | 192 kB
 public.ix_prod_category             | 192 kB
 public.products_pkey                | 192 kB
 public.inventory_pkey               | 192 kB
```

The examples in this chapter will be using the Dell Store 2 sample store unless otherwise noted.

The structure of this data is easy to understand if you've ever shopped online:

- The store sells a number of products, each of which falls within a category
- The store has customers
- Customers place orders
- Each order has a number of lines to it, each of which references the product being purchased
- A customer history lists all the products that the customer has ever ordered

EXPLAIN basics

If you have a query that's running slowly, the first thing to try is to run it with `EXPLAIN` before the statement executing. This displays a query plan--the list of what's expected to happen when that query is executed. Alternatively, if you use `EXPLAIN ANALYZE` before the statement, you'll get the estimation describing what the planner expected and a description of what actually happened when the query ran. However, note that this form will actually execute the statement as if you run it manually. Consider this statement:

```
EXPLAIN ANALYZE DELETE * FROM t;
```

That is not only going to show you a query plan for deleting those rows, it's going to delete them--that's the only way to know for sure how long actually executing the plan is going to take. It's much harder to get a consistent setup to compare timing of operations that do `INSERT`, `UPDATE`, or `DELETE` using `EXPLAIN ANALYZE` because the act of collecting the data will change what a subsequent execution of the same statement will do.

Timing overhead

Assume you're executing a simple query to count all the customers in the database, and want to time how long this takes:

```
dellstore2=# \timing
Timing is on.
dellstore2=# SELECT count(*) FROM customers;
 count
-------
 20000
(1 row)

Time: 4.254 ms
```

You might then be curious to know what query plan was used to get this result and to know how long it took. The amount of time taken to determine that may shock you:

```
dellstore2=# EXPLAIN ANALYZE SELECT count(*) FROM customers;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=578.29..578.30 rows=1 width=8) (actual time=5.105..5.106 rows=1 loops=1)
   ->  Index Only Scan using customers_pkey on customers  (cost=0.29..528.29 rows=20000 width=0) (actual time=0.019..3.017 rows=20000 loops=1)
         Heap Fetches: 0
 Planning time: 0.077 ms
 Execution time: 5.170 ms
(5 rows)

Time: 6.910 ms
```

This fairly trivial query was picked because it demonstrates something close to a worst-case here: instrumenting the query caused the process to slow dramatically, taking almost 10 times as long. Using `EXPLAIN ANALYZE` is great for getting real times for things, but you shouldn't assume the exact proportions or times will be the same when running the query normally.

Hot and cold cache behavior

The regular version of the preceding query executed in 7.994 milliseconds. This represents "hot" cache behavior, meaning that the data needed for the query was already in either the database or operating system caches. It was left behind in the cache from when the data was loaded in the first place. Whether your cache is hot or cold (not in the cache) is another thing to be very careful of. If you run a query twice with two different approaches, the second will likely be much faster simply because of caching, regardless of whether the plan was better or worse.

You can look at how long a query against the entire table takes as a way to measure the effective transfer rate for that table. In the hot cache case, this gives you an idea of how fast data moves between two sections of memory:

```
dellstore2=# SELECT pg_size_pretty(CAST(pg_relation_size('customers') / 4.254 * 1000 as int8)) AS bytes_per_second;
 bytes_per_second
------------------
 896 MB
(1 row)
```

Since this was run on a simple laptop, getting 896 MB/sec of rows processed by a query is respectable good. In this case, repeatedly running the query takes around the same amount of time each run, which means that the amount cached is staying constant and not impacting results. In this case, it's 100% cached.

Clearing the cache

The method to clear all the caches to get cold cache performance varies based on the operating system. Just stopping the database server isn't enough, because the operating system cache will still have plenty of information cached. In Linux, you can use the `drop_caches` feature to discard everything it has in its page cache. Here's a complete example of cleaning the data out of memory for this database on Linux:

```
$ pg_ctl stop
$ sudo su -
# sync
# echo 3 > /proc/sys/vm/drop_caches
# logout
$ pg_ctl start -l $PGLOG
```

The `sync` here is to try and flush all data to disk before we just blow away the caches. This `drop_caches` feature on Linux is not intended for regular production server use, it is more of a debugging feature that's potentially dangerous.

Rerunning the same benchmark shows a quite different performance:

```
dellstore2=# \timing
Timing is on.
dellstore2=# SELECT count(*) FROM customers;
 count
-------
 20000
(1 row)

Time: 39.509 ms
```

Now, run the following:

```
dellstore2=# SELECT pg_size_pretty(CAST(pg_relation_size('customers') / 39.509 * 1000 as int8)) AS bytes_per_second;
 bytes_per_second
------------------
 96 MB
(1 row)
```

Now you're seeing hard drive sequential read speeds-96MB per second from a laptop hard drive, and this data isn't necessarily even contiguous. It's hard to achieve full drive speed on something so small though.

Repeating the query now returns to the original speed we already know to expect from a hot cache run:

```
dellstore2=# SELECT count(*) FROM customers;
 count
-------
 20000
(1 row)

Time: 3.543 ms
```

Tests against real datasets need to recognize whether their data is already in the cache or not. The usual technique is to run each query three times. If the first is much slower than the second and third, it probably started with a cold cache. If the times are all the same, the cache was likely hot before starting. And if all three vary in some other way, there's probably another variable involved besides the cache. It may take a larger number of queries to extract the pattern for why speeds vary.

Examples in this chapter are all run with a hot cache, so you're seeing processing time, but not disk access time. This makes them slightly unrealistic, but the Dell Store sample available for PostgreSQL is not big enough to be uncached for very long on modern hardware. The larger scale versions haven't been ported to the database yet.

Query plan node structure

Explain output is organized into a series of plan nodes. At the lowest level there are nodes that look at tables, scanning them or looking things up with an index. Higher level nodes take the output from the lower level ones and operate on it. When you run `EXPLAIN`, each line in the output is a plan node.

Each node has several numeric measurements associated with it as well:

```
dellstore2=# EXPLAIN ANALYZE SELECT * FROM customers;
                                                       QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------
 Seq Scan on customers  (cost=0.00..688.00 rows=20000 width=268) (actual time=0.043..3.705 rows=20000 loops=1)
 Planning time: 0.746 ms
 Execution time: 5.108 ms
(3 rows)
```

This plan has one node, a `Seq Scan` node. The first set of numbers reported are the plan estimates, which are the only thing you see if you run `EXPLAIN` without `ANALYZE`:

- `cost=0.00..688.00`: The first cost here is the startup cost of the node. That's how much work is estimated before this node produces its first row of output. In this case, that's zero, because a `Seq Scan` immediately returns rows. A sort operation is an example of something that takes a while to return a single row. The second estimated cost is that of running the entire node until it completes. It may not-a node with a `LIMIT` on it for example may stop long before retrieving all estimated rows.
- `rows=20000`: The number of rows this node expects to output if it runs to completion.
- `width=268`: Estimated average number of bytes that each row output by this node will contain. For this example, 20,000 rows of 268 bytes each means this node expects to produce 5,360,000 bytes of output. This is slightly larger than the table itself (3.8 MB) because it includes the overhead of how tuples are stored in memory when executing in a plan.

The "actual" figures show how well this query really ran:

- `actual time=0.043..3.705`: The actual startup cost wasn't quite zero, it took a small fraction of time to start producing output. Once started, it took 34.489 seconds to execute this plan node in total.
- `rows=20000`: As expected, the plan outputs 20,000 rows. Differences between the expected rows and the number actually produced by a node are one of the most common sources of query problems where the optimizer made a bad decision.
- `loops=1`: Some nodes, such as ones doing joins, execute more than once. In that case, the loop value will be larger than one, and the actual time and row values shown will be per loop, not the total. You'll have to multiply these values by the number of loops in order to get a true total.

Understanding how decisions are made using this data requires more knowledge about how the estimated costs are computed.

Basic cost computation

The job of the query optimizer is to generate many possible plans that could be used to execute a query, and then pick the one with the lowest cost to execute. The cost computations are done using arbitrary units only loosely associated with real-world execution cost:

- `seq_page_cost`: How long it takes to read a single database page from disk when the expectation is you'll be reading several that are next to one another, a sequential read of a section of disk. The rest of the cost parameters are essentially relative to this value being the reference cost of 1.0.
- `random_page_cost`: Read cost when the rows involved are expected to be scattered across the disk at random. This defaults to 4.0.
- `cpu_tuple_cost`: How much it costs to process a single row of data. The default is 0.01.
- `cpu_index_tuple_cost`: Cost to process a single index entry during an index scan. Default is 0.005, lower than what it costs to process a row because rows have a lot more header information (such as the visibility `xmin` and `xmax`) than an index entry does.
- `cpu_operator_cost`: Expected cost to process a simple operator or function. If the query needs to add two numbers together, that's an operator cost, and it defaults to the very inexpensive 0.0025.

It's easier to see in a table how these all numerically compare, showing a speed ratio relative to the reference value for `seq_page_cost`:

Parameter	Default value	Relative speed
`seq_page_cost`	1.0	Reference
`random_page_cost`	4.0	4X slower
`cpu_tuple_cost`	0.01	100X faster
`cpu_index_tuple_cost`	0.005	200X faster
`cpu_operator_cost`	0.0025	400X faster

We can use these numbers to compute the cost shown in the previous example. A sequential scan on `customers` has to read every page in the table and process every resulting row. Look at the statistics the optimizer uses to estimate the pages in the table and the number of rows, then combine them with its internal cost parameters:

```
dellstore2=# SELECT
  relpages,
  current_setting('seq_page_cost') AS seq_page_cost,
  relpages *
    current_setting('seq_page_cost')::decimal AS page_cost,
  reltuples,
  current_setting('cpu_tuple_cost') AS cpu_tuple_cost,
  reltuples *
    current_setting('cpu_tuple_cost')::decimal AS tuple_cost
FROM pg_class WHERE relname='customers';
 relpages | seq_page_cost | page_cost | reltuples | cpu_tuple_cost | tuple_cost
----------+---------------+-----------+-----------+----------------+------------
      488 | 1             |       488 |     20000 | 0.01           |        200
(1 row)
```

Add the cost to read the pages (488) to the cost to process the rows (200) and you get 688.00, exactly the cost shown by the `EXPLAIN` plan for the sequential scan in the previous section. We're not going to show how to compute the costs for every plan shown here, but this example shows you how you might do that yourself for ones you're particularly interested in. Ultimately, every plan node breaks down to into these five operations: sequential read, random read, process a row, process an index entry, or execute an operator. Everything else builds more complicated structures based on these basics.

Estimated costs and real-world costs

Now, if you were paying attention to the disk benchmarking chapters earlier, you might be wondering just why the random reads are only considered 4X as expensive as sequential ones. In reality, that ratio is closer to 50:1. Unfortunately, building a robust query cost model isn't as easy as measuring each of the underlying real-world costs the theoretical units model, then setting the costs based on those. One of the most obvious ways that the planner costs don't reflect reality is that the optimizer doesn't know what pages are in the database or operating system cache. Random page reads often result from index scans. And the data associated with the most used indexes is also the most likely to be in the database and operating system cache. So even though the actual random page cost is much more expensive than 4X a sequential one, the odds of an indexed scan reading a random page finding it in cache are quite high.

In real-world PostgreSQL deployments, you're not going to find people changing `random_page_cost` to a higher value that reflects the real random read speed of their drives relative to their sequential speeds. Instead, you'll probably find them decreasing it. On systems with a lot of RAM where much of the database is known to be in memory, dropping `random_page_cost` to 2.0 or even lower is a very common optimization.

Visual explain

Complicated query plans can be difficult to read, with only the indentation level suggesting how nodes that fit into one another are connected. One way to help visualize complicated plans is to graph them using visual explain, a feature available in the pgAdmin III tool: `http://www.pgadmin.org/`.

Seeing how the nodes fit together for the more complicated structures like multi-level joins is extremely valuable for learning how those work. One useful bit of trivia for the graphic display used: when you see lines connecting nodes in the plan, their thickness is proportional to how costly that section of the plan is. You can get an idea where the parts taking a while to execute are at just by looking for the wide lines, the thin ones aren't contributing as heavily to the query runtime.

Verbose output

If you're interested in what columns are actually being passed around by your queries, an explain plan using `VERBOSE` will show them:

```
dellstore2=# EXPLAIN VERBOSE SELECT customerid,firstname,lastname FROM customers;
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on public.customers  (cost=0.00..688.00 rows=20000 width=22)
   Output: customerid, firstname, lastname
(2 rows)
```

This can be valuable in extremely complicated queries where it's not easy to figure out on your own, or when running queries generated by programs like an **Object Relational Mapper** (**ORM**) that sometimes include more information than they necessarily need to.

Machine readable explain output

Starting in PostgreSQL 9.0, there are several new formats you can produce explain output in. In addition to the old text format, you can output in XML and JSON, which allows analysis using the rich library of tools found in many programming languages for operating on that sort of data. You can also produce plans in `YAML` format, which is interesting because it is both machine parsable and arguably easier to read than the standard format. Consider this relatively simple plan:

```
dellstore2=# EXPLAIN VERBOSE SELECT customerid,firstname,lastname FROM customers  WHERE customerid>1000 ORDER BY zip;
                                QUERY PLAN
-------------------------------------------------------------------------------
 Sort  (cost=2088.30..2135.80 rows=19000 width=26)
   Output: customerid, firstname, lastname, zip
   Sort Key: customers.zip
   ->  Seq Scan on public.customers  (cost=0.00..738.00 rows=19000 width=26)
         Output: customerid, firstname, lastname, zip
         Filter: (customers.customerid > 1000)
(6 rows)
```

The same plan output in `YAML` format is much larger, but quite easy to read too:

```
dellstore2=# EXPLAIN (FORMAT YAML) SELECT customerid,firstname,lastname FROM customers  WHERE customerid>1000 ORDER BY zip;
              QUERY PLAN
---------------------------------------
 - Plan:                              +
     Node Type: "Sort"                +
     Parallel Aware: false            +
     Startup Cost: 2088.30            +
     Total Cost: 2135.80              +
     Plan Rows: 19000                 +
     Plan Width: 26                   +
     Sort Key:                        +
       - "zip"                        +
     Plans:                           +
       - Node Type: "Seq Scan"        +
         Parent Relationship: "Outer" +
         Parallel Aware: false        +
         Relation Name: "customers"   +
         Alias: "customers"           +
         Startup Cost: 0.00           +
         Total Cost: 738.00           +
         Plan Rows: 19000             +
         Plan Width: 26               +
         Filter: "(customerid > 1000)"
(1 row)
```

JSON is one of the most popular formats, and here is the same output in `JSON` format:

```
dellstore2=# EXPLAIN (FORMAT JSON) SELECT customerid,firstname,lastname FROM customers  WHERE customerid>1000 ORDER BY zip;
               QUERY PLAN
-----------------------------------------
 [                                      +
   {                                    +
     "Plan": {                          +
       "Node Type": "Sort",             +
       "Parallel Aware": false,         +
       "Startup Cost": 2088.30,         +
       "Total Cost": 2135.80,           +
       "Plan Rows": 19000,              +
       "Plan Width": 26,                +
       "Sort Key": ["zip"],             +
       "Plans": [                       +
         {                              +
           "Node Type": "Seq Scan",     +
           "Parent Relationship": "Outer",+
           "Parallel Aware": false,     +
           "Relation Name": "customers",+
           "Alias": "customers",        +
           "Startup Cost": 0.00,        +
           "Total Cost": 738.00,        +
           "Plan Rows": 19000,          +
           "Plan Width": 26,            +
           "Filter": "(customerid > 1000)"+
         }                              +
       ]                                +
     }                                  +
   }                                    +
 ]
(1 row)
```

Here is the same output in `XML` format:

```
dellstore2=# EXPLAIN (FORMAT XML) SELECT customerid,firstname,lastname FROM customers  WHERE customerid>1000 ORDER BY zip;
                      QUERY PLAN
------------------------------------------------------------
 <explain xmlns="http://www.postgresql.org/2009/explain">  +
   <Query>                                                 +
     <Plan>                                                +
       <Node-Type>Sort</Node-Type>                         +
       <Parallel-Aware>false</Parallel-Aware>              +
       <Startup-Cost>2088.30</Startup-Cost>                +
       <Total-Cost>2135.80</Total-Cost>                    +
       <Plan-Rows>19000</Plan-Rows>                        +
       <Plan-Width>26</Plan-Width>                         +
       <Sort-Key>                                          +
         <Item>zip</Item>                                  +
       </Sort-Key>                                         +
       <Plans>                                             +
         <Plan>                                            +
           <Node-Type>Seq Scan</Node-Type>                 +
           <Parent-Relationship>Outer</Parent-Relationship>+
           <Parallel-Aware>false</Parallel-Aware>          +
           <Relation-Name>customers</Relation-Name>        +
           <Alias>customers</Alias>                        +
           <Startup-Cost>0.00</Startup-Cost>               +
           <Total-Cost>738.00</Total-Cost>                 +
           <Plan-Rows>19000</Plan-Rows>                    +
           <Plan-Width>26</Plan-Width>                     +
           <Filter>(customerid &gt; 1000)</Filter>         +
         </Plan>                                           +
       </Plans>                                            +
     </Plan>                                               +
   </Query>                                                +
 </explain>
(1 row)
```

Generating machine parsable output. It will also make it far easier to write programs to read EXPLAIN command output, and therefore grow the number of tools that assist with query analysis work.

Planning analysis tools

There are already some tools available on the Web that already know how to do the dirty work to analyze the old text format explain output. The best of the currently available is at `http://explain.depesz.com/` where you can submit a plan and get a version highlighting potential problem areas. For example, if the estimated row counts don't match the actual ones, that jumps right out in the color coded plan output.

Assembling row sets

To understand how to optimize the way a query runs, you have to first understand the options for how it can be executed. Now that you're armed with some basics on how nodes fit together and costs are computed, the next stage to understanding how queries work is to see the options for bottom level plan nodes that are usually selecting rows.

Tuple ID

Each row in the database has a tuple ID, a number visible as the system column named `ctid` in each row. You can use these to lookup a row:

```
dellstore2=# SELECT ctid,customerid FROM customers limit 3;
 ctid  | customerid
-------+------------
 (0,1) |          1
 (0,2) |          2
 (0,3) |          3
(3 rows)

dellstore2=# EXPLAIN SELECT customerid FROM customers WHERE ctid='(0,1)';
                        QUERY PLAN
-----------------------------------------------------------
 Tid Scan on customers  (cost=0.00..4.01 rows=1 width=4)
   TID Cond: (ctid = '(0,1)'::tid)
(2 rows)
```

These `TID` sequences cannot be relied upon as a stable way to access a particular row outside of a transaction, because common operations including `UPDATE` will change them. If you're referring to a row more than once in the same transaction, perhaps in a procedural programming language, the `TID` scan can be a quick way to operate a second time on a row located earlier. The `ctid` value can also be used to distinguish between rows that are otherwise identical, for example, when trying to eliminate duplicate rows.

Object ID

In earlier versions of PostgreSQL, every database row had a unique object identification number called an `OID` that could be used to identify it. The overhead of storing these OIDs was considered too high, and as of PostgreSQL 8.1 they now default to off. You can still include them in a table by specifying `CREATE TABLE... WITH OIDS`, and the system catalog tables include them. Using an `OID` to find a record works the same as any other indexed scan:

```
dellstore2=# SELECT oid,relname FROM pg_class WHERE relname='customers';
  oid  |  relname
-------+-----------
 93528 | customers
(1 row)

dellstore2=# EXPLAIN SELECT relname FROM pg_class WHERE oid = 93528;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using pg_class_oid_index on pg_class  (cost=0.27..8.29 rows=1 width=64)
   Index Cond: (oid = '93528'::oid)
(2 rows)
```

There's little reason to use an `OID` for your own tables when the more portable primary key type can be used instead.

Sequential scan

The preceding examples (and many in the previous index chapter) have shown you plenty of examples of tables being scanned sequentially. You can expect a `Seq Scan` when there isn't a useful index, or when such a large portion of the table is expected to be returned such that using an index would just add needless overhead. They'll also be used when there is only a very small amount of data to access; the index overhead is disproportionately large if the table only takes up a few pages on disk.

Note that a `Seq Scan` must read through all the dead rows in a table, but will not include them in its output. If the table has been badly maintained and is quite bloated with dead rows, it's therefore possible for their execution to take much longer than would be expected to produce all the required output.

Index scan

An index scan is what you want if your query needs to return a value fast. If an index that is useful to satisfy a selective `WHERE` condition exists, you'll get one in a query plan that looks like this:

```
dellstore2=# EXPLAIN ANALYZE SELECT * FROM customers WHERE customerid=1000;
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Index Scan using customers_pkey on customers  (cost=0.29..8.30 rows=1 width=268) (actual time=0.058..0.058 rows=1 loops=1)
   Index Cond: (customerid = 1000)
 Planning time: 0.096 ms
 Execution time: 0.083 ms
(4 rows)
```

The main component to the cost here are the two random page reads (4.0 each making a total of 8.0), the index block, and the one the database row is in. Recall that the data blocks must always be read in PostgreSQL even if the index scan is just being used to determine whether a row exists, because visibility must be checked in the row data. This will however turn into random disk seeks against the actual table, the true overhead of which depends on how much of it is already cached.

Regular index scans are the only type that will return rows that are already sorted. This makes them preferred by queries using a `LIMIT`, where the sort cost might otherwise be proportional to the full table size. There are also some upper node types that need input in sorted order.

Index only scans

In PostgreSQL 9.2 a new performance feature was added: index-only scans. In a case where all the columns in a query are part of an index, the planner selects index-only-scans instead of index scans. Normally PostgreSQL accesses the index and underlying table too, but in some cases it appears unnecessary. If all the columns can be retrieved from the index then why look at the underlying table; index-only-scans avoid this unnecessary lookup. Avoiding unnecessary lookup gives significant performance enhancement. Here is an example:

```
dellstore2=# EXPLAIN ANALYZE SELECT customerid FROM customers WHERE customerid=1000;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
 Index Only Scan using customers_pkey on customers  (cost=0.29..4.30 rows=1 width=4) (actual time=0.009..0.009 rows=1 loops=1)
   Index Cond: (customerid = 1000)
   Heap Fetches: 0
 Planning time: 0.073 ms
 Execution time: 0.025 ms
(5 rows)
```

Bitmap heap and index scans

As mentioned in the previous chapter, PostgreSQL is capable of combining indexes together when two of them are relevant for answering a query. The `customers` table in the Dell Store example has an index on both the `customerid` and the `username`. These are both pretty boring bits of data:

```
dellstore2=# SELECT customerid,username from customers limit 3;
 customerid | username
------------+----------
          1 | user1
          2 | user2
          3 | user3
(3 rows)
```

The following somewhat contrived query only returns two rows, but it can use both indexes to accelerate that, and after tweaking the number of values referenced, it's possible to demonstrate that:

```
dellstore2=# SELECT customerid,username FROM customers WHERE customerid<10000 AND username<'user100';
 customerid | username
------------+----------
          1 | user1
         10 | user10
(2 rows)

dellstore2=# EXPLAIN ANALYZE SELECT customerid,username FROM customers WHERE customerid<10000 AND username<'user100';
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on customers  (cost=5.81..382.36 rows=100 width=13) (actual time=0.014..0.016 rows=2 loops=1)
   Recheck Cond: ((username)::text < 'user100'::text)
   Filter: (customerid < 10000)
   Heap Blocks: exact=1
   ->  Bitmap Index Scan on ix_cust_username  (cost=0.00..5.79 rows=200 width=0) (actual time=0.008..0.008 rows=2 loops=1)
         Index Cond: ((username)::text < 'user100'::text)
 Planning time: 0.160 ms
 Execution time: 0.038 ms
(8 rows)
```

Here the query optimizer thought, based on its statistics, that it might have one or two hundred rows returned by each of the two relevant indexes to sort through, so using the index to find them out of the 20,000 possible rows and then AND-ing the two conditions together was considered the faster approach. It had no way of knowing that just searching on the username would quickly find the only two matching rows.

Bitmap index scans are executed by reading the index first, populating the bitmap, and then reading the table in sequential order. This makes them read sequentially as the data is expected to be laid out of disk, approximately, regardless of what the real underlying row ordering is. Each block is read and then the drive is expected to skip forward to the next block. Because this gives output sorted via physical order on disk, the results can easily require sorting afterwards for upper nodes that expect ordered input.

The index bitmaps built by the scans can be combined with standard bit operations, including either `AND` (return rows that are on both lists) or `OR` (return rows that are on either list) when appropriate. For non-trivial queries on real-world databases, the bitmap index scan is a versatile workhorse that you'll find showing up quite often.

Processing nodes

Once you have a set of rows, the next type of node you'll encounter when using a single table are ones that process that set in various ways. These nodes typically take in a row set and output a different row set of either the same size or smaller (perhaps only a single value).

Sort

Sort nodes can appear when you insert `ORDER BY` statements into your queries:

```
dellstore2=# EXPLAIN ANALYZE SELECT customerid FROM customers ORDER BY zip;
                                                       QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=2116.77..2166.77 rows=20000 width=8) (actual time=15.191..17.020 rows=20000 loops=1)
   Sort Key: zip
   Sort Method: quicksort  Memory: 1706kB
   ->  Seq Scan on customers  (cost=0.00..688.00 rows=20000 width=8) (actual time=0.008..7.813 rows=20000 loops=1)
 Planning time: 0.068 ms
 Execution time: 18.118 ms
(6 rows)
```

`Sort` operations can either execute in memory using the quicksort algorithm, if they're expected to fit, or will be swapped to disk to use what's called an external merge sort-the case in this example. The threshold at which that happens depends on the `work_mem` setting on the server. The preceding example may be surprising because the memory used (352 kB) appears under the default value for that parameter, as used on this test server:

```
SHOW work_mem;
work_mem
----------
 1MB
```

There's a good explanation for that: the size needed for an in-memory sort is bigger than the amount of disk needed for an external one. External disk sorts in PostgreSQL are done by writing a set of sorted files out and then merging the results, and that takes significantly less memory than the quicksort. Watch what happens if the `work_mem` parameter is increased and the statement planned again (which you can do using `SET` for a single query, the server configuration doesn't have to be modified to increase this setting):

```
dellstore2=# SET work_mem='2MB';
SET
dellstore2=# EXPLAIN ANALYZE SELECT customerid FROM customers ORDER BY zip;
                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 Sort  (cost=2116.77..2166.77 rows=20000 width=8) (actual time=8.232..10.361 rows=20000 loops=1)
   Sort Key: zip
   Sort Method: quicksort  Memory: 1621kB
   ->  Seq Scan on customers  (cost=0.00..688.00 rows=20000 width=8) (actual time=0.008..4.411 rows=20000 loops=1)
 Planning time: 0.056 ms
 Execution time: 11.627 ms
(6 rows)
```

This shows why the default wasn't sufficient-doing the sort in memory actually requires 1.3 MB, so it was just rejected at the default value as too big. If you turn on `log_temp_files` to look for how often disk swap sorts are happening, and notice sorting files that are below `work_mem` in size, this might be the reason for that behavior.

Using `ORDER BY` is more important than many newcomers to SQL realize, because there is no row ordering guaranteed when you query unless you request one. You cannot expect you'll get rows back in the order you inserted them, even though that's often the case. A common counter-example is when the synchronized scan features in PostgreSQL 8.3 and later is used. When that kicks in, a second table scan, running against a table that a scan is already running on, will just tag along with the first, starting to return rows from whatever point the original is at. That will get you rows returned offset from a completely random point, only circling back to the beginning once the first scan has finished.

In addition, when you're asking for explicit ordering, it's also possible to see a sort node in situations where another type of node needs its input sorted to operate. Examples include a unique node, some types of joins and grouping, and some set operations.

Limit

Like everything else, query limits are built on top of existing scans that return a set of rows:

```
dellstore2=# EXPLAIN ANALYZE SELECT customerid FROM customers LIMIT 10;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.00..0.34 rows=10 width=4) (actual time=0.010..0.015 rows=10 loops=1)
   ->  Seq Scan on customers  (cost=0.00..688.00 rows=20000 width=4) (actual time=0.009..0.011 rows=10 loops=1)
 Planning time: 0.070 ms
 Execution time: 0.029 ms
(4 rows)
```

Note the actual rows output by the `Seq Scan` here. This shows one of the aspects of query execution that isn't necessarily obvious. The way queries are carried out, the top node in the query plan is started, and it asks its children nodes for things on-demand. It's a top-down execution model, nodes only produce output when the said output is required. In this case, the `Seq Scan` on the customers table could have output as many as 20,000 rows, but because the limit was reached after only 10 rows, that's all the `Seq Scan` node was asked to produce. The action where an upper node asks for a row from one of its children is referred to as it pulling one from it.

`Limit` nodes work a little differently than most because, depending on how large the limit is relative to the total number of rows, the startup cost of the node can become more important than the total cost. If the optimizer knows that only a few rows need to be produced, it can favor a plan that starts quickly to produce rows over a plan that is lower in total cost, but has most of that cost in startup. Putting a small limit on a query biases plans that quickly produce output rows, and whether or not that works well is quite vulnerable to whether the estimates the optimizer is working with are correct or not.

Offsets

When Offset is added to a query it isn't handled by its own node type. It's handled as a different form of limit. Essentially, the first few rows that the underlying scan produces are thrown away. This is easy to see given a variation on the `LIMIT` query:

```
dellstore2=# EXPLAIN ANALYZE SELECT customerid FROM customers OFFSET 10 LIMIT 10;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.34..0.69 rows=10 width=4) (actual time=0.010..0.012 rows=10 loops=1)
   ->  Seq Scan on customers  (cost=0.00..688.00 rows=20000 width=4) (actual time=0.007..0.011 rows=20 loops=1)
 Planning time: 0.066 ms
 Execution time: 0.022 ms
(4 rows)
```

Note how this time, the `Seq Scan` node produced 20 rows of output. The first 10 were skipped by the `OFFSET`, then the next ten satisfied the preceding `LIMIT` node, at which point the query was finished.

Aggregate

Aggregate functions take in a series of values and produce a single output. Examples of aggregates are `AVG()`, `COUNT()`, `EVERY()`, `MIN()`, `MAX()`, `STDDEV()`, `SUM()`, and `VARIANCE()`. To compute an aggregate, all of the rows are typically read, then fed through the aggregate node to compute a result:

```
dellstore2=# EXPLAIN ANALYZE SELECT max(zip) FROM customers;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=738.00..738.01 rows=1 width=4) (actual time=6.016..6.017 rows=1 loops=1)
   ->  Seq Scan on customers  (cost=0.00..688.00 rows=20000 width=4) (actual time=0.005..2.382 rows=20000 loops=1)
 Planning time: 0.326 ms
 Execution time: 6.044 ms
(4 rows)
```

This isn't always the case though, because some values can be computed with indexes instead. Looking for the highest `customerid` in the customers table, where that's the primary key, doesn't have to look at every row:

```
dellstore2=# EXPLAIN ANALYZE SELECT max(customerid) FROM customers;
                                                                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Result  (cost=0.32..0.33 rows=1 width=4) (actual time=0.019..0.019 rows=1 loops=1)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.29..0.32 rows=1 width=4) (actual time=0.017..0.017 rows=1 loops=1)
           ->  Index Only Scan Backward using customers_pkey on customers  (cost=0.29..578.29 rows=20000 width=4) (actual time=0.017..0.017 rows=1 loops=1)
                 Index Cond: (customerid IS NOT NULL)
                 Heap Fetches: 0
 Planning time: 0.117 ms
 Execution time: 0.037 ms
(8 rows)
```

HashAggregate

A versatile node type, the `HashAggregate` node takes in a set of nodes and outputs a series of derived data in buckets. A major reworking in PostgreSQL 8.4 turned several common types of operations into ones that are implemented using a `HashAggregate` node. They are now commonly used to compute distinct values, group by results, and unions. The main value is that hashing this way can avoid needing to sort the values, which is sometimes an expensive step. There are plenty of examples of the `HashAggregate` node type below-in fact, in current PostgreSQL versions, the harder thing to do is not see it used!

Sometimes aggregates that are being computed by `GROUP BY` or even `DISTINCT` will use a `HashAggregate` to compute their output. Consider this example that looks at which category each product belongs to:

```
dellstore2=# EXPLAIN ANALYZE SELECT category,count(*) FROM products GROUP BY category ORDER BY category;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=251.48..251.52 rows=16 width=12) (actual time=9.717..9.718 rows=16 loops=1)
   Sort Key: category
   Sort Method: quicksort  Memory: 25kB
   ->  HashAggregate  (cost=251.00..251.16 rows=16 width=12) (actual time=9.695..9.696 rows=16 loops=1)
         Group Key: category
         ->  Seq Scan on products  (cost=0.00..201.00 rows=10000 width=4) (actual time=0.356..6.852 rows=10000 loops=1)
 Planning time: 1.231 ms
 Execution time: 9.779 ms
(8 rows)
```

In this case, there are only 16 categories involved, easily fitting into a set of hash table buckets. In latest versions of PostgreSQL, you can get a bit more detail about exactly what these various hash types are up to in the `EXPLAIN` plans than on earlier versions.

Unique

A unique node takes a sorted set of rows as an input, and outputs one with all the duplicates removed. It can appear when using `DISTINCT` and when `UNION` is eliminating duplicates in its output. The output will be sorted the same as the input.

As of PostgreSQL 8.4, `DISTINCT` will usually use a `HashAggregate` node instead of a unique one, which on smaller tables can collect the unique values without having to explicitly sort them first:

```
dellstore2=# EXPLAIN ANALYZE SELECT DISTINCT(state) FROM customers;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=738.00..738.51 rows=51 width=3) (actual time=8.120..8.127 rows=52 loops=1)
   Group Key: state
   ->  Seq Scan on customers  (cost=0.00..688.00 rows=20000 width=3) (actual time=0.007..2.641 rows=20000 loops=1)
 Planning time: 0.084 ms
 Execution time: 8.168 ms
(5 rows)
```

To see the old implementation in current PostgreSQL versions with something the size of the Dell Store data, it's easiest to just turn that optimization off:

```
dellstore2=# SET enable_hashagg=off;
SET
dellstore2=# EXPLAIN ANALYZE SELECT DISTINCT(state) FROM customers;
                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Unique  (cost=2116.77..2216.77 rows=51 width=3) (actual time=28.903..31.641 rows=52 loops=1)
   ->  Sort  (cost=2116.77..2166.77 rows=20000 width=3) (actual time=28.902..30.390 rows=20000 loops=1)
         Sort Key: state
         Sort Method: quicksort  Memory: 1529kB
         ->  Seq Scan on customers  (cost=0.00..688.00 rows=20000 width=3) (actual time=0.019..4.674 rows=20000 loops=1)
 Planning time: 0.064 ms
 Execution time: 31.674 ms
(7 rows)
```

Since this earlier implementation of unique had the side-effect of sorting its output, this can be an application breaking change when switching from earlier versions of PostgreSQL to 8.4 or later. This serves as another reminder of why explicit ordering should be requested in your queries, not assumed just because the output you're seeing now seems always to be ordered correctly.

WindowAgg

The SQL Windowing functions added this type of aggregation to PostgreSQL 8.4. An example of it is shown as part of the discussion of that new feature at the end of this chapter.

Result

Sometimes a node just needs to return a result computed by a statement:

```
dellstore2=# EXPLAIN ANALYZE SELECT 1;
                                        QUERY PLAN
-------------------------------------------------------------------------------------
 Result  (cost=0.00..0.01 rows=1 width=4) (actual time=0.001..0.001 rows=1 loops=1)
 Planning time: 0.022 ms
 Execution time: 0.012 ms
(3 rows)
```

Result nodes are basically quick pass-through nodes when individual values are being operated on, rather than sets of rows. They can be used to collapse and therefore optimize sections of a `WHERE` clause that can be computed once:

```
dellstore2=# EXPLAIN ANALYZE SELECT * FROM customers WHERE customerid=(SELECT min(customerid) FROM customers)+1;
                                                
```

This is dramatically easier to read plan than the earlier behavior, which can be forced by turning off the new optimization:

```
SET enable_hashagg=off;
```

We have disabled `hashagg`, now let's `EXPLAIN` the query:

```
dellstore2=# EXPLAIN ANALYZE SELECT customerid,city FROM customers WHERE state='MA' UNION SELECT customerid,city FROM customers WHERE state='MD';
                                                                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Unique  (cost=1497.35..1500.36 rows=401 width=122) (actual time=6.574..6.709 rows=401 loops=1)
   ->  Sort  (cost=1497.35..1498.35 rows=401 width=122) (actual time=6.573..6.606 rows=401 loops=1)
         Sort Key: customers.customerid, customers.city
         Sort Method: quicksort  Memory: 43kB
         ->  Append  (cost=0.00..1480.01 rows=401 width=122) (actual time=0.022..6.403 rows=401 loops=1)
               ->  Seq Scan on customers  (cost=0.00..738.00 rows=214 width=12) (actual time=0.021..3.900 rows=214 loops=1)
                     Filter: ((state)::text = 'MA'::text)
                     Rows Removed by Filter: 19786
               ->  Seq Scan on customers customers_1  (cost=0.00..738.00 rows=187 width=12) (actual time=0.030..2.439 rows=187 loops=1)
                     Filter: ((state)::text = 'MD'::text)
                     Rows Removed by Filter: 19813
 Planning time: 0.160 ms
 Execution time: 6.773 ms
(13 rows)
```

In this case, the external sort is a small one, but in larger types of `UNION` operations the improvement from using the newer implementation can be significant.

Group

Yet another node type left behind by progress, the Group node was the way earlier PostgreSQL versions implemented `GROUP BY`. It required that the input data was sorted by the grouping column set. As of PostgreSQL 7.4, that's commonly done by a `HashAggregate` instead:

```
dellstore2=# EXPLAIN ANALYZE SELECT state,COUNT(*) FROM customers GROUP BY state;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=788.00..788.51 rows=51 width=11) (actual time=10.179..10.186 rows=52 loops=1)
   Group Key: state
   ->  Seq Scan on customers  (cost=0.00..688.00 rows=20000 width=3) (actual time=0.017..2.233 rows=20000 loops=1)
 Planning time: 0.596 ms
 Execution time: 10.346 ms
(5 rows)
```

You can once again see the old behavior by turning off this optimization:

```
dellstore2=# SET enable_hashagg=off;
SET
dellstore2=# EXPLAIN ANALYZE SELECT state,count(*) FROM customers GROUP BY state;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=2116.77..2267.28 rows=51 width=11) (actual time=24.437..27.309 rows=52 loops=1)
   Group Key: state
   ->  Sort  (cost=2116.77..2166.77 rows=20000 width=3) (actual time=24.398..25.668 rows=20000 loops=1)
         Sort Key: state
         Sort Method: quicksort  Memory: 1550kB
         ->  Seq Scan on customers  (cost=0.00..688.00 rows=20000 width=3) (actual time=0.008..3.933 rows=20000 loops=1)
 Planning time: 0.068 ms
 Execution time: 27.400 ms
(8 rows)
```

Not having to do the external merge sort on disk is the main reason the hash style of plan is preferred now, which improves the runtime of this style of query. You can still see the older `GroupAggregate` execution show up when the input already happens to be ordered, which happens in the merge join in the following example.

Subquery scan and subplan

These two types of nodes are used for shuffling rows around between nodes in some types of `UNION` and subselect queries. They have little performance impact and have also become rare in current PostgreSQL versions due to the `HashAggregate` optimizations.

Subquery conversion and IN lists

A few types of things that you might expect to be executed as subqueries will actually turn into types of joins instead. This happens when using a subquery to find a list of rows to be used for `IN`:

```
dellstore2=# EXPLAIN ANALYZE SELECT * FROM orders WHERE customerid IN (SELECT customerid FROM customers where state='MD');
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Hash Semi Join  (cost=740.34..994.61 rows=112 width=30) (actual time=5.488..15.487 rows=120 loops=1)
   Hash Cond: (orders.customerid = customers.customerid)
   ->  Seq Scan on orders  (cost=0.00..220.00 rows=12000 width=30) (actual time=0.456..8.605 rows=12000 loops=1)
   ->  Hash  (cost=738.00..738.00 rows=187 width=4) (actual time=4.968..4.968 rows=187 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 15kB
         ->  Seq Scan on customers  (cost=0.00..738.00 rows=187 width=4) (actual time=0.047..4.938 rows=187 loops=1)
               Filter: ((state)::text = 'MD'::text)
               Rows Removed by Filter: 19813
 Planning time: 6.107 ms
 Execution time: 15.551 ms
(10 rows)
```

As you can see, this is silently converted into a regular join; the exact type used, a hash semi join, is covered in the following section. There's no subquery node involved there. Performance of PostgreSQL `IN` queries can be quite good because this sort of conversion is always done. Then all the usual join optimization tricks are available to improve the result. This one is stuck doing a `Seq Scan` on each table simply because there are no indexes that are useful here. If they were, the rewritten form might execute quite quickly.

Other types of subqueries will be considered for similar rewrites into joins where doing so is logically equivalent. For example, this slightly odd query:

```
EXPLAIN ANALYZE SELECT * FROM orders WHERE customerid IN (SELECT
  customerid FROM customers WHERE customerid=1000 OR customerid=2000);
```

This query will be rewritten into a nested loop join with inner-index scan in any recent PostgreSQL version.

Set operations

The easiest way to show how a set operation works is to look at a query that uses one:

```
SELECT * FROM customers WHERE state='MD'
INTERSECT
SELECT * FROM customers WHERE zip='21340';
```

This is obviously a trivial example-you could just put both `WHERE` clauses into a single line and get the same result more easily. But there are more complicated types of queries you can build using `INTERSECT`, `INTERSECT ALL`, `EXCEPT`, and `EXCEPT ALL` that are the easiest way to write what you're looking for. `EXCEPT` is shown here for helping determine if two queries give the same output, for example.

In what should be a familiar pattern by now, `SetOp` changed to `HashSetOp` in PostgreSQL 8.4. Here's the new and old types of plans you can get when executing an `INTERSECT` or `EXCEPT`:

```
dellstore2=# EXPLAIN ANALYZE SELECT * FROM customers WHERE state='MD' INTERSECT SELECT * FROM customers WHERE zip='21340';
                                                
```

CTE scan

Common Table Expressions (**CTE**) are another new PostgreSQL 8.4 feature. They add new efficient ways to execute queries that even let you put recursion into SQL. One way they can be used is as a sort of in-line view, which makes for an easy way to demonstrate the feature and the resulting type of plan node:

```
dellstore2=# EXPLAIN ANALYZE WITH monthlysales AS
  (SELECT EXTRACT(year FROM orderdate) AS year,
   EXTRACT(month FROM orderdate) AS month,
   sum(netamount) AS sales
   FROM orders GROUP BY year,month)
SELECT year,SUM(sales) AS sales FROM monthlysales GROUP BY year;
                                                
```

```
postgres =# INSERT INTO foo VALUES(3, 'val3', 'data3');
postgres =# COPY foo TO '/home/postgres/foo.txt';
postgres =# \q
$ cat /home/postgres/foo.txt
1  val1  data1
2  val2    data2
3 val3 data3
```

COPY FROM

The `COPY FROM` command is used to copy data from file to table. This command imports data from a file. Here is an example that imports data from `foo.txt` to a table named bar:

```
postgres=# CREATE TABLE bar ( a int, b  text, c varchar);
postgres=# COPY bar FROM '/usr/local/pg.95/bin/foo.txt';
postgres=# SELECT * FROM bar;
 a |  b   |   c
---+------+-------
 1 | val1 | data1
 2 | val2 | data2
 3 | val3 | data3
(3 rows)
```

COPY FREEZE

PostgreSQL 9.3 included a new performance feature called FREEZE, `http://www.postgresql.org/docs/current/static/sql-copy.html`.

Requests copy the data with rows already frozen, just as they would be after running the `VACUUM FREEZE` command. This is intended as a performance option for initial data loading. Rows will be frozen only if the table being loaded has been created or truncated in the current subtransaction, there are no cursors open, and there are no older snapshots held by this transaction.

Note that all other sessions will immediately be able to see the data once it has been successfully loaded. This violates the normal rules of MVCC visibility and users specifying should be aware of the potential problems that this might cause.

Here is an example of the `copy` command with and without the `FREEZE` option. The timing difference can easily be seen:

```
postgres=# \timing
postgres=# COPY barc from  '/usr/local/pg.95/bin/foo.txt';
COPY 10000000
Time: 7060.007 ms


postgres=# BEGIN;
postgres=# create table barc ( a int, b text, c varchar);
postgres=# COPY barc from  '/usr/local/pg.95/bin/foo.txt' FREEZE;
COPY 10000000
Time: 5458.062 ms
postgres=# END;
```

The `FREEZE` command can be used while the table is created/truncated in a sub-transaction block, otherwise it will throw an "ERROR: cannot perform `FREEZE` because the table was not created or truncated in the current subtransaction" error.

Joins

If all the query planner had to do was decide between index scan types and how to combine them using its wide array of derived nodes, its life would be pretty easy. All the serious complexities in the planner and optimizer relate to joining tables together. Each time another table is added to a list that needs to be joined, the number of possible ways goes up dramatically. If there are, say, three tables to join, you can expect the query plan to consider every possible plan and select the optimal one. However, if there are twenty tables to join, there's no possible way it can exhaustively search each join possibility. Since a variety of techniques are available to join each table pair, that further expands the possibilities. The universe of possible plans has to be pruned somehow.

Fundamentally, each way two tables can be joined together gives the same output. The only difference between them is how efficient the result is to execute. All joins consider an outer and inner table. These alternately might be called the left and right if considering a query plan as a tree or graph, and that usage shows up in some of the PostgreSQL documentation internals documentation.

Nested loop

Use the nested loop if you need every possible row combination joined together. In most other cases, it's probably not the best option. The classic pseudocode description of a nested loop join looks like this:

```
for each outer row:
for each inner row:
if join condition is true:
output combined row
```

Both the `inner` and `outer` loops here could be executing against any of the scan types: sequential, indexed, bitmap, or even the output from another join. As you can see from the code, the amount of time this takes to run is proportional to the number of rows in the outer table multiplied by the rows in the inner. It is considering every possible way to join every row in each table with every other row.

It's rare to see a real nested loop without an inner-index scan, the type covered in the next section. Joining using merges and hashes is normal for real-world data that tends to be indexed or has a clear relationship between tables. You can see one if you just forget to put a `WHERE` condition on a join though, which then evaluates the cross product and outputs a ton of rows. Here's one that produces by far the longest runtime of a query in this chapter:

```
dellstore2=# EXPLAIN ANALYZE SELECT * FROM products,customers;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..2500914.00 rows=200000000 width=317) (actual time=0.035..48407.527 rows=200000000 loops=1)
   ->  Seq Scan on customers  (cost=0.00..688.00 rows=20000 width=268) (actual time=0.019..7.598 rows=20000 loops=1)
   ->  Materialize  (cost=0.00..251.00 rows=10000 width=49) (actual time=0.000..0.558 rows=10000 loops=20000)
         ->  Seq Scan on products  (cost=0.00..201.00 rows=10000 width=49) (actual time=0.004..1.000 rows=10000 loops=1)
 Planning time: 1.996 ms
 Execution time: 57699.377 ms
(6 rows)
```

That's a 19-minute runtime, which sounds like a lot until you realize it's producing 200 million output rows. That works out to 175K row/second being processed, not really that shabby.

Note that a `Nested Loop` is the only way to execute a `CROSS JOIN`, and it can potentially be the only way to compute complicated conditions that don't map into either a useful merge or hash join instead.

Nested loop with inner-index scan

The standard situation you'll see a nested loop in is one where the inner table is only returning back a limited number of rows. If an index exists on one of the two tables involved, the optimizer is going to use it to limit the number of rows substantially, and the result may then make the `inner * outer` rows runtime of the `Nested Loop` acceptable. Consider the case where you're looking for a single order via a field with no index (so every order must be scanned), but then joining with its matching order lines:

```
dellstore2=# EXPLAIN ANALYZE SELECT * FROM orders,orderlines WHERE orders.totalamount=329.78 AND orders.orderid=orderlines.orderid;
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.29..265.43 rows=5 width=48) (actual time=0.235..2.020 rows=9 loops=1)
   ->  Seq Scan on orders  (cost=0.00..250.00 rows=1 width=30) (actual time=0.019..1.802 rows=1 loops=1)
         Filter: (totalamount = 329.78)
         Rows Removed by Filter: 11999
   ->  Index Scan using ix_orderlines_orderid on orderlines  (cost=0.29..15.38 rows=5 width=18) (actual time=0.212..0.214 rows=9 loops=1)
         Index Cond: (orderid = orders.orderid)
 Planning time: 3.224 ms
 Execution time: 2.081 ms
(8 rows)
```

The main way this type of scan can go wrong is if the optimizer's sensitivity guess for the inner relation here is quite wrong. This inner-index scan was expecting five rows, but nine actually came out; a reasonable error that wouldn't have changed the plan type. It's possible for that guess to be extremely wrong instead. Being off a factor of 1,000 or more is quite possible due to bad table statistics or limitations in the optimizer's cost model. Then the resulting join will be very inefficient, and could potentially have been executed better another way.

Another way this plan might be avoided even though it's an otherwise good candidate is because `effective_cache_size` is too small to cope with the inner-index scan. See the section on that parameter below for more information.

A `Nested Loop` can also show up when both the inner and outer scan use an index:

```
dellstore2=# EXPLAIN ANALYZE SELECT * FROM orders,orderlines WHERE orderlines.orderid=1000 AND orders.orderid=orderlines.orderid;
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.57..23.73 rows=5 width=48) (actual time=0.023..0.027 rows=6 loops=1)
   ->  Index Scan using orders_pkey on orders  (cost=0.29..8.30 rows=1 width=30) (actual time=0.014..0.014 rows=1 loops=1)
         Index Cond: (orderid = 1000)
   ->  Index Scan using ix_orderlines_orderid on orderlines  (cost=0.29..15.38 rows=5 width=18) (actual time=0.006..0.009 rows=6 loops=1)
         Index Cond: (orderid = 1000)
 Planning time: 0.141 ms
 Execution time: 0.072 ms
(7 rows)
```

This form should execute even faster, as in this case. But it is even more dangerous, because a sensitivity mistake on either the inner or outer relation can cause this node to do much more work than expected.

Merge join

A `Merge Join` requires that both its input sets are sorted. It then scans through the two in that sorted order, generally moving forward a row at a time through both tables as the joined column values change. The inner table can be rescanned more than once if the outer one has duplicate values. That's where the normally forward scan on it goes backwards, to consider the additional matching set of rows from the duplication.

You can only see a `Merge Join` when joining on an equality condition, not an inequality or a range. To see one, let's ask for a popular report: how much net business was done by each customer?

```
dellstore2=# SET enable_hashagg=off;
SET
dellstore2=# EXPLAIN ANALYZE SELECT C.customerid,sum(netamount) FROM customers C, orders O WHERE C.customerid=O.customerid GROUP BY C.customerid;
                                                
```

PostgreSQL 9.0 introduced a model where rescans have a different cost model than when the rows are initially scanned, to try and accurately estimate when this materialization is a saving. As such, the situations in which you will and won't see `Materialize` nodes show up in that version are very different from earlier versions.

Hash joins

The primary alternative to a `Merge Join`, a `Hash Join` doesn't sort its input. Instead, it creates a hash table from each row of the inner table, scanning for matching ones in the outer. The output will not necessarily be in any useful order.

A query to find all the products that have at one point been ordered by any customer shows a regular `Hash Join`:

```
dellstore2=# EXPLAIN ANALYZE SELECT prod_id,title FROM products p WHERE EXISTS (SELECT 1 FROM orderlines ol WHERE ol.prod_id=p.prod_id);
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=1358.43..1694.30 rows=9736 width=19) (actual time=28.735..32.720 rows=9973 loops=1)
   Hash Cond: (p.prod_id = ol.prod_id)
   ->  Seq Scan on products p  (cost=0.00..201.00 rows=10000 width=19) (actual time=0.011..1.314 rows=10000 loops=1)
   ->  Hash  (cost=1236.73..1236.73 rows=9736 width=4) (actual time=28.706..28.706 rows=9973 loops=1)
         Buckets: 16384  Batches: 1  Memory Usage: 479kB
         ->  HashAggregate  (cost=1139.38..1236.73 rows=9736 width=4) (actual time=26.176..27.352 rows=9973 loops=1)
               Group Key: ol.prod_id
               ->  Seq Scan on orderlines ol  (cost=0.00..988.50 rows=60350 width=4) (actual time=0.010..13.517 rows=60350 loops=1)
 Planning time: 1.601 ms
 Execution time: 33.311 ms
(10 rows)
```

Whether a `Hash Join` is better or worse than the other possibilities depends on things such as whether input is already sorted (in which case a `Merge Join` might be inexpensive) and how much memory is required to execute it. The hash tables built for the inner scan here require enough memory to store all the rows, which can be large.

To keep that under control, larger `Hash Join` executions will split the output into multiple batches. In this example, only one was required to hold 10K rows. This type of join is particularly popular when the inner relation is small and building the hash is therefore inexpensive.

Hash semi and anti joins

One of the hash improvements in PostgreSQL 8.4 is introducing hash semi and anti joins. A semi join is used when the optimizer needs to confirm that a key value exists on one side of the join, but, doesn't particularly care what that value is beyond that. The opposite, an anti join, looks specifically for entries where the key value doesn't exist. The most common way to see these two types of join is when executing `EXISTS` and `NOT EXISTS`.

The inverse of the query in the previous section is finding all the products that have never been ordered by any customer, and this executes most efficiently with an anti join:

```
dellstore2=# EXPLAIN ANALYZE SELECT prod_id,title FROM products p WHERE NOT EXISTS (SELECT 1 FROM orderlines ol WHERE ol.prod_id=p.prod_id);
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Hash Anti Join  (cost=1742.88..2044.57 rows=264 width=19) (actual time=19.343..21.852 rows=27 loops=1)
   Hash Cond: (p.prod_id = ol.prod_id)
   ->  Seq Scan on products p  (cost=0.00..201.00 rows=10000 width=19) (actual time=0.007..0.674 rows=10000 loops=1)
   ->  Hash  (cost=988.50..988.50 rows=60350 width=4) (actual time=19.164..19.164 rows=60350 loops=1)
         Buckets: 65536  Batches: 1  Memory Usage: 2634kB
         ->  Seq Scan on orderlines ol  (cost=0.00..988.50 rows=60350 width=4) (actual time=0.006..9.393 rows=60350 loops=1)
 Planning time: 0.198 ms
 Execution time: 21.889 ms
(8 rows)
```

The main reason to restructure the plan this way is just for efficiency. It can require considerably less index reading to work this way than the earlier approach for executing queries like this, which would do a pair of index scans and combine them with a filter.

This example uses more memory for hashing than the similar query in the last section. Two batches are used to hold 60K rows. The way batches are processed internally requires keeping some in memory while others are flushed to disk, and the optimizer eventually works through them all.

There is actually an interesting optimization in that section of code, one that happens with all hash joins (not just the semi and anti variations). If the outer relation in the join isn't uniform, if it has some **Most Common Values** (**MCV**) that represent a significant portion of the table, the execution will keep batches containing those in memory to process first. If you suspect a skewed distribution of values in your dataset, with many MCV and this type of join, you may want to check statistics targets and possibly increase them until the optimizer recognizes this fact. This can improve the odds that you'll hit the optimized path available.

As for examples of standard semi joins, there are two elsewhere in this chapter. *Subquery conversion and IN lists* preceding has a simple one, while the one appearing in *Avoiding plan restructuring with Offset* below gives a more complicated one to review.

Join ordering

Since overall query costs go up exponentially as the number of joins increases, controlling that complexity is a major component to both query tuning and to ongoing improvement in the PostgreSQL optimizer. The differences between a sequential scan or using an index on a table will be magnified as those results are then joined to additional tables.

Forcing join order

Consider this three-way join that takes the `custhist` table and joins it to its matching product and customer keys:

```
SELECT * FROM cust_hist h INNER JOIN products p ON (h.prod_id=p.prod_id)
INNER JOIN customers c ON (h.customerid=c.customerid);
```

This join is identical to an implementation that uses an implicit join and a `WHERE` clause:

```
SELECT * FROM cust_hist h,products p,customers c
WHERE h.prod_id=p.prod_id AND h.customerid=c.customerid;
```

In either case, the query optimizer is free to choose plans that execute these joins in several orders. It could join `cust_hist` to `products`, then to `customers`, or it could join to `customers` and then to `products`. The results will be identical, and the cheapest one will be used.

However, this doesn't have to be the case. Doing these searches for optimal plans is time consuming, and it has the potential to make a bad decision. If you are sure of the right way to join the tables efficiently and want to reduce planning time, you can force the optimizer to use the order you specified when doing a series of explicit `JOIN` operations. Reduce the `join_collapse_limit` parameter from its default of `8` to do so; typically the useful value to prevent all join reordering is to lower it to `1`. So, this example:

```
SET join_collapse_limit = 1;
SELECT * FROM cust_hist h INNER JOIN products p ON (h.prod_id=p.prod_id)
INNER JOIN customers c ON (h.customerid=c.customerid);
```

Is going to start with `cust_hist`, join to products, and then to customers in every case. Alternate plans won't be considered. This can be useful in two main contexts. If query planning time is large for a particular complicated join, discovering the usual order that will be executing and making it explicit in the query can save significant planning time. In cases where the optimizer's selected order was poor, this is a hint you can provide to it on what the right thing to do is.

There is a similar parameter named `from_collapse_limit` that controls how much freedom the query optimizer has to merge subqueries into upper ones. Generally, both these values are set to the same value, so behavior is easier to predict. There may be cases for setting them to different values if you are being quite careful about tweaking for planning time versus query execution time.

Join removal

A new feature for PostgreSQL 9.0 will remove joins in circumstances where they aren't really required. The following join would be common to many queries running against this data that wanted to match up a product with its matching inventory data:

```
SELECT * FROM products LEFT JOIN inventory ON
products.prod_id=inventory.prod_id;
```

You might even put this into a view as a shorthand for the combined result, then filter it down from there, so that not everyone had to remember how to do the join. However, some queries using that general form might not even need to use the inventory data, such as this one that only references product fields:

```
dellstore2=# EXPLAIN ANALYZE SELECT products.title FROM products LEFT JOIN inventory ON products.prod_id=inventory.prod_id;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on products  (cost=0.00..201.00 rows=10000 width=15) (actual time=0.011..10.048 rows=10000 loops=1)
 Planning time: 0.513 ms
 Execution time: 12.043 ms
(3 rows)
```

Note that there's no join at all here; it has been optimized away. There are three sensible requirements before this logic can kick in:

- There is a `LEFT JOIN`
- A unique index exists for the join columns
- None of the information in the candidate table to be removed is used elsewhere

Because of the current restrictions on this optimization, if it's possible for you to use `LEFT JOIN` instead of `INNER JOIN` when you write a query, that would be preferable if you also suspect removable joins may crop into your execution plans. Removing `INNER` ones may eventually be supported in PostgreSQL, but it doesn't know how to do that yet.

While this particular mistake is obvious here, in complicated database setups where the physical table structure is abstracted away by views and programmatically generated queries, having the optimizer throw out the redundant joins can be a useful improvement.

Genetic query optimizer

Once the number of joins gets large enough, there is really no hope that the query optimizer can evaluate all of them exhaustively and still return plans in a reasonable period of time. When this happens, the **genetic query optimizer** (**GEQO**) is called in to work on the problem instead. It numbers each of the tables to be joined as an integer starting with one, so a possible plan for a five table join would be encoded as 1-2-3-4-5. GEQO starts by creating a number of such plans at random. It then evaluates each of these for fitness, specifically how large the execution cost is for each of them. The best plans are kept, the worst ones dropped, some changes to the plans made ("mutation"), and the process repeats for a number of generations.

Be warned that because there's so much randomization involved in this process, you can't expect the plans that come out of GEQO to be the same every time, even given the same input query and statistics. Starting in PostgreSQL 9.0, it's possible to control that for more consistent plans, by fixing the random seed used to generate them to always be the same value. See the PostgreSQL documentation for more information.

Whether GEQO comes into play or not depends on the `geqo_threshold`, which defaults to 12. Since there aren't even that many tables in the sample database used for this chapter, it's certainly not something that can be easily demonstrated. If you have that many tables to join, expect to spend some time learning how GEQO works and about its tunable parameters. There's little information available on doing so available beyond what's included in the PostgreSQL documentation.

Collecting statistics

The least appreciated part of query optimization is the collection of database statistics. Often when questions like "why isn't the optimizer using my index?" are asked, it is poor statistics that are really to blame.

Statistics are collected for each column in every table in a database when `ANALYZE` is executed against the table. If you're running with `autovacuum` turned on, it will usually run often enough to keep accurate statistics available. Unlike a complete vacuum cleanup, which can take quite some time on large tables, analyzing a table should take only a few seconds at any table size. It doesn't take anything other than a read lock while running either.

Viewing and estimating with statistics

The statistics information collected for each table is easiest to see using the `pg_stats` view. The amount of information it returns is a bit overwhelming though, and not well suited to the standard display format. The following script is named `table-stats.sh` in the book's file set, and it tries to display the statistics information for a particular table (with an optional database too) in a way that makes the information as readable as possible:

```
#!/bin/bash
if [ -z "$1" ]; then
  echo "Usage: table-stats.sh table [db]"
  exit 1
fi
TABLENAME="$1"
if [ -n "$2" ] ; then
  DB="-d $2"
fi
PSQL="psql $DB -x -c "
$PSQL "
SELECT
  tablename,attname,null_frac,avg_width,n_distinct,correlation,
  most_common_vals,most_common_freqs,histogram_bounds
FROM pg_stats
WHERE tablename='$TABLENAME';
" | grep -v "\-\[ RECORD "
```

A typical usage would be:

```
./table-stats.sh customers dellstore2
```

Note that the script as currently written doesn't show the inherited information necessary to sort out statistics on tables using inheritance, such as ones that are partitioned. An alternative way to display some of the data displayed horizontally in the preceding query is to use `array_to_string` to break it into rows, like this:

```
SELECT attname,inherited,array_to_string(most_common_vals, E'\n')
 AS most_common_vals FROM pg_stats;
```

Here's some examples of the information about the Customer table returned by `table-stats.sh` with some of the less interesting pieces (table name and null_frac) removed:

```
attname           | customerid
avg_width         | 4
n_distinct        | -1
correlation       | 1
most_common_vals  |
most_common_freqs |
histogram_bounds  | {1,200,400,600,800,1000,... 19600,19800,20000}
```

Since this is the primary key on the table, the number of distinct values doesn't have to be estimated by probing the data, thus the `-1` for that value. It can be computed based on the running estimate for total row count kept updated as part of the table statistics, rather than relying just on what the last `ANALYZE` noticed.

The histogram divides the range of data sampled into buckets of equal frequency. In order to compute how selective a range query will be, the optimizer figures out where the value selected fits into the histogram. Let's say you were running a query that grabbed the first 300 rows of data here. This would cover all of the first bucket's range (1-200) and half the second bucket range (200-400). The formula used looks like this:

```
selectivity = (1 + (value - bucket[2].min)/
 (bucket[2].max - bucket[2].min)) / num_buckets
selectivity = (1 + (300 - 200)/(400 - 200))/(20000 / 200)
selectivity = 0.015
rows = total row count * selectivity
rows = 20000 * 0.02 = 300
```

That's how the optimizer would correctly guess that this query will return 300 rows:

```
EXPLAIN ANALYZE SELECT * FROM customers WHERE customerid<=300;
QUERY PLAN
----------
Index Scan using customers_pkey on customers  (cost=0.00..20.50
rows=300 width=268) (actual time=0.041..0.670 rows=300 loops=1)
Index Cond: (customerid <= 300)
Total runtime: 1.192 ms
```

The preceding computation is probably indecipherable to you on a quick first read. Study this for a bit and try to understand how it works. A basic working knowledge of how histogram buckets help the query optimizer make decisions will put you way ahead of many people who try to optimize queries without understanding how statistics drive the process.

This simulation does depend on collecting enough buckets of information to make sure the histogram tracked the true distribution of the data. If only a small amount of data was sampled, you could end up with a very misleading perspective on how the data was distributed.

PostgreSQL even tries to improve on this estimate to note the end of indexed values when available, which helps improve estimates when you normally insert data at the start or end of the column's range (a very common situation). If your query needs data from either the first or last bucket, and the corresponding left or right edge can be determined by a quick index lookup, the optimizer will do that to correctly set its edge boundary. That's a guaranteed real minimum or maximum, while the histogram data that `ANALYZE` put there may be out of date.

For text data, instead of a histogram you get common values and frequencies:

```
attname            | country
avg_width          | 5
n_distinct         | 11
correlation        | -0.613603
most_common_vals   |
{US,Chile,Australia,Russia,Canada,Germany,China,UK,Japan,France,"South
Africa"}
most_common_freqs |
{0.5,0.05235,0.0517,0.0506,0.0502,0.0502,0.0501,0.0501,0.04945,0.04855,0.04
675}
histogram_bounds   |
```

The most common value and frequency data is paired up taking a matching entry from each. So that translates into the following:

Country	% of data
US	50%
Chile	5.235%
Australia	5.17%
Russia	5.06%

...and so on, for all 11 of the countries listed. If there were more distinct countries than the number of saved statistics buckets, only the most popular ones would be saved this way, and the remainder estimated more roughly.

The income column is an integer. There's little enough data in there that it also includes a complete distribution rather than a histogram in its statistics:

```
attname            | income
avg_width          | 4
n_distinct         | 5
correlation        | 0.198876
most_common_vals   | {20000,80000,40000,60000,100000}
most_common_freqs  | {0.2033,0.2032,0.2027,0.1961,0.1947}
histogram_bounds   |
```

This data gets combined as additional conditions are added to a `WHERE` clause, presuming that the individual columns are independent of one another. For example, if you were filtering on country and income the selectivity would be combined like this:

```
country = 'US':   50% selectivity
income=80000':  20.32% selective
country = 'US' AND income=80000':  50% * 20.32% = 10.16%
rows = total row count * selectivity
rows = 20000 * 10.16% = 2032
```

And sure enough, 2,032 is the estimated row count that the query planner computes for that query:

```
dellstore2=# EXPLAIN ANALYZE SELECT * FROM customers WHERE country='US' AND income=80000;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on customers  (cost=0.00..788.00 rows=2032 width=268) (actual time=0.021..15.284 rows=2039 loops=1)
   Filter: (((country)::text = 'US'::text) AND (income = 80000))
   Rows Removed by Filter: 17961
 Planning time: 0.313 ms
 Execution time: 15.512 ms
(5 rows)
```

If the planner has no idea how selective something is because it's missing data on it, the guess is it selects 0.5% of the rows.

Statistics targets

It should be obvious from the preceding section that the number of buckets used to hold common values or histogram samples significantly influences how accurate query row estimates will be. The number of them that `ANALYZE` aims to collect is called its target. The `postgresql.conf` parameter, `default_statistics_target`, sets the standard target for tables in each database. The default target was 10 up to PostgreSQL 8.3, increasing to 100 in 8.4. This means that 8.4 has significantly more statistics to work with by default, which normally results in better queries. The cost for that is slightly poorer query planning time, as well as significantly longer `ANALYZE` processing (that second part not being something most people care very much about).

If you only have simple queries to plan and query planning time is large in a later PostgreSQL version where the target starts at 100, it's possible to reduce `default_statistics_target` back to the older starting value of 10 and reanalyze the whole database. This can give about a few percent performance boost to running trivial queries in newer versions.

Adjusting a column target

The maximum number of target buckets is either 1,000 or 10,000 depending on the version. You can confirm which you have by using the following:

```
SELECT name,max_val FROM pg_settings WHERE
name='default_statistics_target';
```

Setting an extremely large value here does incur overhead on every query that is planned; and, as mentioned previously, just the increase from 10 to 100 measurably detunes trivial query runtime in PostgreSQL 8.4. There are, however, some table columns where it takes substantially larger settings in order to get sufficient statistics about the data to make accurate estimates. It's not unheard of in a large data warehouse setting to increase the target to 1,000 or more in order to sample enough of a giant table to represent it. However, you don't want to pay that penalty for every query, just the ones that need it.

Luckily, the target is set per column, so you can override the value just on that basis for those that need it rather than for every table and query run. The per-column value starts at `-1`, which means to use the system default. You can adjust it like this:

```
ALTER TABLE t ALTER COLUMN k SET STATISTICS 1000;
```

And then run a new `ANALYZE` against that table.

Distinct values

Starting from PostgreSQL 9.0, it's possible to override the estimates `analyze` makes for the number of distinct values in a table with your own values in situations where it's not making a good estimate on its own:

```
ALTER TABLE t ALTER COLUMN k SET (n_distinct = 500);
```

In addition to just putting a value in there, you can also set this to a negative value, at which point it's used as a multiplier on the number of rows in the table. That lets you set a distinct estimate that scales along with your data based on current and future expectations about it. There is also a `n_distinct_inherited` parameter that can be set on a parent that multiple tables inherit from.

Difficult areas to estimate

If no data has ever been collected about a table via `ANALYZE`, the planner has to make up completely arbitrary estimates. That's the worst sort of statistics issue to have.

Another area where statistics can fail is situations where the query optimizer just doesn't know how to estimate something. You can watch it utterly fail to estimate this simple query correctly:

```
dellstore2=# EXPLAIN ANALYZE SELECT * FROM customers WHERE customerid=customerid;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Seq Scan on customers  (cost=0.00..738.00 rows=100 width=268) (actual time=0.009..10.129 rows=20000 loops=1)
   Filter: (customerid = customerid)
 Planning time: 0.059 ms
 Execution time: 12.061 ms
(4 rows)
```

It's obvious that all 20,000 rows will be returned. What's happening is that, because it has no idea how selective the test here is, the optimizer has to make a wild guess. The standard estimate it uses in this situation is 0.5% of the rows, which is where the figure of 100 expected rows comes from here.

When row counts are underestimated, this can result in an index scan instead of the correct `Seq Scan`, or in running a long `Nested Loop` instead of a `Hash Join` or `Merge Join`. When row counts are overestimated, you can get a `Seq Scan` instead of an index scan, and a `Merge Join` or `Hash Join` instead of a `Nested Loop`.

Other query planning parameters

We've covered all the ways a query executes. Now, let's look at some of the more obscure query planning parameters.

effective_cache_size

Defaulting to 128 MB, `effective_cache_size` is used to represent approximately how much total disk space is available for caching the database. This is normally set to the total of `shared_buffers` plus the size of the operating system disk buffer cache after the database is started. That turns out to be greater than half of total system memory on a typical dedicated database server. This setting does not allocate any memory itself, it simply serves as an advisory value for the planner about what should likely be available.

The only thing this is used for is estimating whether an index scan will fit into memory, with the alternative being a sequential scan. `Nested Loop` joins that are using an inner-index scan are particularly impacted by this setting. As you reduce `effective_cache_size`, it's less likely that it will be considered an effective query execution plan.

While the scope where this parameter comes into play is limited, these use cases do pop up regularly. The penalty for deciding an index can't be processed in memory is getting a sequential scan or a poor join type choice, which can make for a rather expensive mistake. And the default value is this parameter is quite small relative to the amount of RAM in modern servers.

The main thing to be careful about here is that, like `work_mem`, PostgreSQL isn't tracking client related memory resources in any way that limits them across the whole database. If you have two clients executing queries that require scanning an extremely large index to work well at the same time, because each thinks it can use half or more of the RAM in the server for that purpose, you may discover your operating system cache fighting to retain any data but those indexes. That may be fine if it makes the large queries execute much faster because they're using a selective index, but there is a risk here if you set this value too high. Rather than strictly looking at total system RAM to set this parameter, you might instead do a sanity check by seeing just how large the indexes you might expect to participate in `Nested Loop` with inner-index scan queries might be. Then set `effective_cache_size` to something large enough to fix those comfortably.

Since `effective_cache_size` is a client-side parameter, another alternative approach that isn't used very often is to treat its sizing more like how `work mem` is typically managed. You can set a moderate, but not too high value, in the main `postgresql.conf`. Then, only in queries that really need a high setting to execute, do you increase it. That approach will reduce odds of the potential problem occurring where multiple really large index scans expected to fit in cache actually turn out to exhaust cache when running simultaneously. But it will still allow an increase for queries where you know that's the only good way to approach things.

work_mem

If you are sorting data, `work_mem` determines when those sorts are allowed to execute in memory and when they have to swap to disk instead. Starting in PostgreSQL 8.3, you can turn on `log_temp_files` and see all the cases where `work_mem` was not large enough, and the `external merge Disk sort` is used instead. Note that you may be confused to see such usage appear in the logs even though the value shown is smaller than `work_mem`. An example and the reason behind why that happens is explained in the preceding *Sort* section.

Setting a value for this parameter is tricky. Ideally you'd like it to be large on a system with plenty of memory, so that sorts happen quickly. But every client can use this much memory for each sort node in a query it's busy executing. Standard sizing will therefore put an upper bound at around:

```
work_mem = Total RAM / max_connections / 4
```

On the assumption that half of system memory could be used for other purposes, and that it's unlikely every client will be doing more than two sorts at a time, this is actually a quite aggressive setting. A safer setting would be:

```
work_mem = Total RAM / max_connections / 16
```

A good tuning strategy here is to pick a number in between those two values and then monitor total RAM used by PostgreSQL processes, free memory, and sorts that still exceed the settings that are logged as going to disk. If there appears to be plenty of memory unused at peak times and disk sorts are happening, increase `work_mem`. If memory use looks high, try lowering it instead.

Note that you can set `work_mem` for a single query before running it, which is the preferred technique for situations such as an occasional large report that runs each day. If you know only a small number of clients can be running a query at a time, you can be much more aggressive about how much memory is allocated just for those.

`work_mem` is also used for some other sizing decisions in the query optimizer, particularly in PostgreSQL 8.4 and later where it's become an increasingly important parameter. Hash table construction and materialization creation will also use this value as a guide to how much memory they can use. This does increase the concern, with newer versions, that it's hard to predict how many simultaneous operations might happen using `work_mem` worth of memory. You can normally estimate the number of sorts in a query just by looking at it or its `EXPLAIN` plan, but hash and materialization use isn't so obvious from the query itself, and is more subject to shifting around as the table changes. In 8.4 and particularly 9.0, it's better to err on the side of caution, using a smaller `work_mem` setting initially. Be careful about using a large value here until you have an idea how aggressively clients are allocating working memory for all of its possible purposes.

constraint_exclusion

If you are using partitioned tables that use constraints, enabling `constraint_exclusion` allows the planner to ignore partitions that can't have the data being searched for when that can be proven. This parameter originally defaulted to `off`, meaning that, unless toggled on, partitioned tables will not work as expected. Starting in PostgreSQL 8.4, this was improved such that a new default value, `partition`, will do the right thing in most cases without adding the overhead to the ones it isn't necessary for. It is theoretically possible to see a tiny performance improvement by turning this feature off in newer versions where it defaults to `partition` in cases where there are no partitions on your system.

cursor_tuple_fraction

When you start a query using a cursor rather than a regular statement, PostgreSQL doesn't know for sure how many rows you'll then retrieve. To bias towards both the possibility that you will only access a subset of them, and the fact that cursor-based programs tend to be sensitive to latency, `cursor_tuple_fraction` allows you to lower the expected number of rows that cursor originated queries are expected to return. If set to 1.0, cursors will work the same as regular queries. At its default of 0.1, the query optimizer biases towards query plans that quickly return rows, assuming that only 10% of the total rows will be requested. This works similarly to when a `LIMIT` is used on a query.

Executing other statement types

The same basic execution logic is used to handle all four of the basic SQL query types: `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. For example, when doing an `UPDATE`, the rows to update are identified in a familiar way and then fed to an update node:

```
dellstore2=# EXPLAIN ANALYZE UPDATE customers SET state=state WHERE customerid=1;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Update on customers  (cost=0.29..8.30 rows=1 width=274) (actual time=6.911..6.911 rows=0 loops=1)
   ->  Index Scan using customers_pkey on customers  (cost=0.29..8.30 rows=1 width=274) (actual time=0.008..0.010 rows=1 loops=1)
         Index Cond: (customerid = 1)
 Planning time: 0.082 ms
 Execution time: 7.120 ms
(5 rows)
```

`UPDATE` and `DELETE` can execute efficiently by keeping track of the tuple ID, uniquely identifying each of the rows they are then called to operate on. Note that the preceding `UPDATE` doesn't actually do anything-the values it's setting the state field to are what it already contains. PostgreSQL doesn't look at what you're updating to figure out if the change being made by an `UPDATE` does something or not; it will always execute it instead. It can be a useful optimization to add a `WHERE` clause to prevent this. To show this with an example that doesn't make any business sense, let's say every customer were to suddenly be relocated to Maryland. The appropriate query to do that would be:

```
UPDATE customers SET state='MD' WHERE NOT state='MD';
```

This will avoid adding a redundant row for the situation where the state was already set to that value. Avoiding updates this way can save a lot of unnecessary writes and vacuum cleanup in some applications.

Improving queries

Once you understand what a query is doing, getting it to do something better can be a difficult road to walk down. There are some common techniques that mesh with the areas PostgreSQL is known to be good and bad at.

Note that your first steps to improve queries should be to check if the statistics the optimizer is working with seem reasonable. If those are bad, you might fix the problem by rewriting the query. However, you can expect that future queries will run into the same class of issue, so fixing it from a statistics standpoint is the preferred way, and the one that's more likely to continue reflecting future trends in the data. Even so, improving the statistics about your data can only go so far some days.

Optimizing for fully cached datasets

A basic assumption of the query optimizer is that data is not cached in memory, and therefore all access to an index or table might require some disk activity to retrieve. The planner parameters `seq_page_cost` and `random_page_cost` being very high relative to `cpu_index_tuple_cost` reflects this pessimism.

If in fact the data you are reading is expected to be fully cached in memory, it can be appropriate to dramatically lower these parameters in recognition of that fact. In some cases, it might be appropriate to go so far as to make index and table lookups appear no more expensive than the CPU cost of looking at a single row:

```
SHOW cpu_index_tuple_cost;
cpu_index_tuple_cost
----------------------
 0.005

SET seq_page_cost=0.005;
SET random_page_cost=0.005;
```

It's unlikely that you want to set values this low in your `postgresql.conf` for every query, unless your entire database has proven to be quite small compared to system RAM. Since you can adjust these parameters at session time before executing a query, you can tweak them down for the queries that benefit from knowing the completely cached nature of the data they are operating on.

Testing for query equivalence

In one of the hash join examples, a complicated query using `EXISTS` was used to determine whether each product had ever been ordered. There's theoretically another way to figure that out: the inventory information for each product includes a sales count. If those are complete - every product is included in that inventory, even if it's never been sold - then a query looking for zero sales of an item should give the same results:

```
SELECT prod_id FROM inventory WHERE sales=0;
```

That looks like the same list, but it's long enough that comparing every entry would be tedious. You can easily compare the output from two queries to see if they produce the same rows using the `EXCEPT` construct:

```
SELECT prod_id FROM products p WHERE NOT EXISTS (SELECT 1 FROM orderlines
ol WHERE ol.prod_id=p.prod_id)
EXCEPT
SELECT prod_id FROM inventory WHERE sales=0;
```

This is particularly useful to know when working on optimizing queries by rewriting them. It's easy to break a query when doing that so that it doesn't quite produce the same output anymore. When you have a rewritten query that appears to give the same results as an earlier one but runs faster, constructing a regression test showing no rows come out of an `EXCEPT` construct including the two versions, is a handy way to prove that.

Disabling optimizer features

Sometimes the optimizer doesn't do what you want, and you may want some tools to either force it to change its mind or to see what alternatives it's considering but rejecting. Consider this simple query that searches for one order and joins with its matching customer record:

```
dellstore2=# EXPLAIN ANALYZE SELECT C.customerid,O.orderid FROM customers C,orders O WHERE c.customerid=o.customerid AND o.orderid=10000;
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.57..12.62 rows=1 width=8) (actual time=0.951..0.954 rows=1 loops=1)
   ->  Index Scan using orders_pkey on orders o  (cost=0.29..8.30 rows=1 width=8) (actual time=0.794..0.795 rows=1 loops=1)
         Index Cond: (orderid = 10000)
   ->  Index Only Scan using customers_pkey on customers c  (cost=0.29..4.30 rows=1 width=4) (actual time=0.030..0.030 rows=1 loops=1)
         Index Cond: (customerid = o.customerid)
         Heap Fetches: 0
 Planning time: 0.296 ms
 Execution time: 0.992 ms
(8 rows)
```

This is a nice efficient query. But the type of join used is very sensitive to just how many rows are expected to be returned from the orders table, ones that then have to be matched against customers. This style of query continues to be preferred all the way up to one that selects 195 customers:

```
dellstore2=# EXPLAIN ANALYZE SELECT C.customerid,O.orderid FROM customers C,orders O WHERE c.customerid=o.customerid AND o.orderid BETWEEN 10555 AND 120000;
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.57..748.36 rows=1445 width=8) (actual time=0.018..4.002 rows=1446 loops=1)
   ->  Index Scan using orders_pkey on orders o  (cost=0.29..65.19 rows=1445 width=8) (actual time=0.011..0.495 rows=1446 loops=1)
         Index Cond: ((orderid >= 10555) AND (orderid <= 120000))
   ->  Index Only Scan using customers_pkey on customers c  (cost=0.29..0.46 rows=1 width=4) (actual time=0.002..0.002 rows=1 loops=1446)
         Index Cond: (customerid = o.customerid)
         Heap Fetches: 5
 Planning time: 0.317 ms
 Execution time: 4.144 ms
(8 rows)
```

Make the query just a bit less selective, so one more row is considered, and the join type changes completely. When there are a fairly large number of matches expected, the planner uses a `Merge Join` with an inner-index scan for the order key, while matching against the entire customers table. It builds a hash table holding the relevant information from all 20,000 rows rather than using a `Nested Loop`:

```
dellstore2=# EXPLAIN ANALYZE SELECT C.customerid,O.orderid FROM customers C,orders O WHERE c.customerid=o.customerid AND o.orderid BETWEEN 10554 AND 120000;
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------------------
 Merge Join  (cost=141.42..741.08 rows=1446 width=8) (actual time=1.139..6.331 rows=1447 loops=1)
   Merge Cond: (c.customerid = o.customerid)
   ->  Index Only Scan using customers_pkey on customers c  (cost=0.29..528.29 rows=20000 width=4) (actual time=0.011..2.464 rows=19992 loops=1)
         Heap Fetches: 66
   ->  Sort  (cost=141.10..144.72 rows=1446 width=8) (actual time=1.123..1.310 rows=1447 loops=1)
         Sort Key: o.customerid
         Sort Method: quicksort  Memory: 116kB
         ->  Index Scan using orders_pkey on orders o  (cost=0.29..65.20 rows=1446 width=8) (actual time=0.010..0.529 rows=1447 loops=1)
               Index Cond: ((orderid >= 10554) AND (orderid <= 120000))
 Planning time: 0.296 ms
 Execution time: 6.454 ms
(11 rows)
```

That's exactly where the threshold is on this particular copy of the Dell store database: the expected 195 records gets a `Nested Loop` join. This gives you an idea of just how fast a query plan can completely change on you in production. If the execution time on the `Merge Join` version of this was really terrible, that one extra record that showed up would kill the performance of your server. Unless you were logging query plan `EXPLAIN` data, it's unlikely you'd have any idea what hit you.

What if the whole table is being scanned? That's even less selective, and the `Hash Join` is now preferred:

```
dellstore2=# EXPLAIN ANALYZE SELECT C.customerid,O.orderid FROM customers C,orders O WHERE c.customerid=o.customerid AND o.orderid BETWEEN 1 AND 12000;
                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=778.29..1223.29 rows=12000 width=8) (actual time=19.473..25.640 rows=12000 loops=1)
   Hash Cond: (o.customerid = c.customerid)
   ->  Seq Scan on orders o  (cost=0.00..280.00 rows=12000 width=8) (actual time=0.010..2.148 rows=12000 loops=1)
         Filter: ((orderid >= 1) AND (orderid <= 12000))
   ->  Hash  (cost=528.29..528.29 rows=20000 width=4) (actual time=19.429..19.429 rows=20000 loops=1)
         Buckets: 32768  Batches: 1  Memory Usage: 960kB
         ->  Index Only Scan using customers_pkey on customers c  (cost=0.29..528.29 rows=20000 width=4) (actual time=0.012..9.982 rows=20000 loops=1)
               Heap Fetches: 75
 Planning time: 0.357 ms
 Execution time: 26.237 ms
(10 rows)
```

Because query plan changes can be so disastrous to a production server, some database administrators like to provide what are called optimizer hints to the server. These are strong suggestions to the optimizer that it executes a query a particular way. PostgreSQL doesn't explicitly support hints, based on two observations. Firstly, it makes users better motivated to provide feedback to the database developers on what the optimizer does badly, so that they can continually improve it. Secondly, many plan changes are actually the right thing to do. The original hint could have been based on statistics that are now out-of-date compared with what's actually in the table now.

There is a way to simulate something like an optimizer hint though. PostgreSQL allows individual queries to turn off particular optimizer features. If you disable the type of plan you'd prefer not to see, that's effectively hinting towards the plan you want. The list of features you can toggle off as of PostgreSQL 9.0 are:

- enable_bitmapscan
- enable_hashagg
- enable_hashjoin
- enable_indexscan
- enable_mergejoin
- enable_nestloop
- enable_seqscan
- enable_sort
- enable_tidscan
- enable_material

Note that turning these off doesn't actually disable the feature, it just increases its estimate cost so it's much less likely to be executed. There are some queries that it's only possible to execute with a `Nested Loop`, so even if you turn off `enable_nestloop` you'll still get one. And if you have no index on something, you can turn off `enable_seqscan`, but a `Seq Scan` will nonetheless happen.

Let's see what happens if `Hash Joins` are disabled on this query:

```
dellstore2=# SET enable_hashjoin=off;
SET
dellstore2=# EXPLAIN ANALYZE SELECT C.customerid,O.orderid FROM customers C,orders O WHERE c.customerid=o.customerid AND o.orderid BETWEEN 1 AND 12000;
                                                
```

```
EXPLAIN ANALYZE SELECT * FROM inventory,products WHERE
inventory.prod_id=products.prod_id;
SET enable_mergejoin = off;
EXPLAIN ANALYZE SELECT * FROM inventory,products WHERE
inventory.prod_id=products.prod_id;
SET enable_bitmapscan = off;
EXPLAIN ANALYZE SELECT * FROM inventory,products WHERE
inventory.prod_id=products.prod_id;
```

I find it particularly amusing to watch how creative the optimizer gets at using increasingly less efficient index scans once the better join types go away. In particular, the last join will be much slower than the other ones once all access to the indexes is made impractical; here are some sample runtimes:

Join type	Execution time (milliseconds)
Hash join	119
Merge join	126
Nested loop with inner-index scan	171
Nested loop with inner bitmap index scan	219
Nested loop with inner `Seq Scan`	292,491

Playing with these optimizer options to work around a query problem is tempting, because it can feel like an easy fix to a bad plan issue. It's dangerous though, because the plan that makes sense today can be totally wrong when either the amount or distribution of your data changes in the future. The reason why good tutorials about database queries try to explain how queries execute, including details like how statistics and costs fit together, is because understanding that theory is what can give you the background to make difficult decisions in this area.

Working around optimizer bugs

Only if you can't get the statistics or cost data to work as expected should you then take drastic measures like disabling optimizer features. Knowing that flags, such as `enable_hashjoin` and `enable_seqscan`, exist can be valuable for another reason altogether. Sometimes, bad optimizer behavior can result from a database bug or limitation. If you know or suspect that, toggling optimizer features off can help you determine which code path might be involved in your issue.

Also, there's nothing wrong with temporarily working around a bad plan by putting one of these optimizer hacks into place as a triage measure, to buy yourself time to figure out what's really wrong. Resist doing that in the server configuration though - these sort of changes belong in the code that executes the individual problem queries, not server-wide.

Avoiding plan restructuring with OFFSET

Another technique that shows up sometimes to work around bad plan issues is using `OFFSET 0` strategically in a query. Having no offset from when results return doesn't change the query result, but it is considered a volatile change that prevents some planned node changes, sometimes in a good way, despite what the planner predicts.

A good example of how this can work comes from a potential performance regression from certain types of joins that was introduced in PostgreSQL 8.4, and also still exists in 9.0. It's an interesting case study in how progress in improving the PostgreSQL optimizer tends to happen. The problem is described by Andres Freund using a test case and associated query plans at `http://archives.postgresql.org/pgsql-hackers/2010-05/msg00889.php`, but not with actual data. It's possible to recreate the basic setup using the Dell store data. Let's say you have a small set of orderline data you're searching for. You want to confirm the existence of matching customer and order rows that go along with this fairly small set of orderlines. That doesn't have a strong real-world meaning, given the existence of those rows is guaranteed by foreign key constraints. But a query doing that and showing the problem behavior looks like this:

```
dellstore2=# EXPLAIN ANALYZE
SELECT
  l.prod_id
FROM
    orderlines l
WHERE
    EXISTS (SELECT * FROM customers JOIN orders USING (customerid) WHERE orders.orderid = l.orderid) AND l.orderdate='2004-12-01';
                                                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop Semi Join  (cost=0.57..1732.23 rows=156 width=4) (actual time=7.241..8.197 rows=183 loops=1)
   ->  Seq Scan on orderlines l  (cost=0.00..1139.38 rows=156 width=8) (actual time=7.207..7.638 rows=183 loops=1)
         Filter: (orderdate = '2004-12-01'::date)
         Rows Removed by Filter: 60167
   ->  Nested Loop  (cost=0.57..3.79 rows=1 width=4) (actual time=0.003..0.003 rows=1 loops=183)
         ->  Index Scan using orders_pkey on orders  (cost=0.29..3.46 rows=1 width=8) (actual time=0.001..0.001 rows=1 loops=183)
               Index Cond: (orderid = l.orderid)
         ->  Index Only Scan using customers_pkey on customers  (cost=0.29..0.32 rows=1 width=4) (actual time=0.001..0.001 rows=1 loops=183)
               Index Cond: (customerid = orders.customerid)
               Heap Fetches: 3
 Planning time: 0.337 ms
 Execution time: 8.278 ms
(12 rows)
```

What the database is doing here is constructing a hash table that includes every customer and order entry, on the assumption that it can then efficiently join against the matching order lines that meet the criteria. It is even guessing correctly how many order lines will come out here: there are 183 rows returned, and it expected 155. However, the setup time to build this whole hash structure is high just to join against such a small number of rows. The optimization that tries to do that with a semi-join instead was introduced in 8.4. A semi-join is one where the optimizer only outputs outer rows that join to at least one inner row, and only once each.

This particular type of setup - starting with a selective query against a low-level table, then searching for the existence of matching records in ones that join to it - is not an uncommon one. I have run into this issue on a live customer system and have seen this sort of plan give a dramatic slowdown compared to earlier PostgreSQL versions.

What can you do about it? If the existence checks include a volatile bit of data the planner can't assume it can transform around, because it can't do this optimization. One such change you can easily use is `OFFSET 0`, which is a volatile change that the planner thinks changes query results in a way it can't predict, but it really doesn't. Watch what happens when that is judiciously inserted into the preceding query:

```
dellstore2=# EXPLAIN ANALYZE
SELECT
  l.prod_id
FROM
    orderlines l
WHERE
    EXISTS (
        SELECT *
        FROM customers
        JOIN orders USING (customerid) WHERE orders.orderid = l.orderid OFFSET 0) AND l.orderdate='2004-12-01';
                                                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on orderlines l  (cost=0.00..1004005.50 rows=78 width=4) (actual time=7.244..8.331 rows=183 loops=1)
   Filter: ((orderdate = '2004-12-01'::date) AND (SubPlan 1))
   Rows Removed by Filter: 60167
   SubPlan 1
     ->  Nested Loop  (cost=0.57..16.62 rows=1 width=294) (actual time=0.003..0.003 rows=1 loops=183)
           ->  Index Scan using orders_pkey on orders  (cost=0.29..8.30 rows=1 width=30) (actual time=0.001..0.001 rows=1 loops=183)
                 Index Cond: (orderid = l.orderid)
           ->  Index Scan using customers_pkey on customers  (cost=0.29..8.30 rows=1 width=268) (actual time=0.001..0.001 rows=1 loops=183)
                 Index Cond: (customerid = orders.customerid)
 Planning time: 0.282 ms
 Execution time: 8.381 ms
(11 rows)
```

Now, instead of joining against the big hash structure, the planner just finds the 183 order lines it's interested in, and executes an index lookup for each of them probing into the orders and customers table (note the rare SubPlan node type making an appearance as part of that). This was what PostgreSQL 8.3 and earlier versions always did, and by defeating the ability to use the newer optimization that behavior is back again. Note how the cost for this is believed to be dramatically higher, when in fact the actual runtime is much faster.

While there is the occasional unfortunate regression like this one, for the most part the PostgreSQL query optimizer gets better every version. Now that the type of query that executes badly here is well understood, and reproducible test cases have appeared, the performance regression seen by this type of join issue is on the roadmap for improvement in PostgreSQL 9.1.

Much like faking optimizer hinting by turning off features, the need to use tricks like `OFFSET 0` to prevent bad planner behavior should continue to become less necessary in the future. However, it is a handy trick to have in the toolbox when trying to force a query structure you know is better than the one the optimizer prefers. Seeing what happens to plans when you limit the optimizer's ability to restructure them like this is educational, too.

External trouble spots

It's also possible to have a query run slowly for reasons that have nothing to do with the query plan itself.

One such situation is where you have triggers running against a table being referenced, particularly in cases where it's an `UPDATE` or `INSERT` running instead of just a regular query. The overhead of triggers can be high, particularly if there aren't any indexes on the underlying foreign keys.

Another issue you can run into is that dead rows can clog a query from executing as fast as expected. This is particularly true if you are frequently updating a popular set of rows without running `VACUUM` often enough. Doing anything with those rows can then require large amounts of wasted time, reading the non-visible data, even when doing an index scan.

SQL limitations

SQL is good at a number of things; and there are some things it's known to be poor at, most of which result from returned rows having no knowledge of one another.

Numbering rows in SQL

Let's say you wanted to know who your top five customers are. That's an easy enough query to write:

```
dellstore2=# SELECT customerid,sum(netamount) as sales FROM orders GROUP BY customerid ORDER BY sales DESC LIMIT 5;
 customerid |  sales
------------+---------
      15483 | 1533.76
       9315 | 1419.19
      15463 | 1274.29
      10899 | 1243.14
       3929 | 1196.87
(5 rows)
```

Here's a challenge for you: how would you write this query to also include that sales ranking for each customer, from 1 to 5, as part of this list? It's not a simple problem.

One of the subtle things about SQL queries is that each row is its own independent piece. It doesn't have any notion of how many other rows exist in the result set, where it stands in relation to the others; it's just a row of data. There are a couple of ways to solve this problem by using temporary tables, joining the output from `generate_series` against a primary key when one exists, and processing it in a stored procedure. There's even a portable SQL way to do this using something like the following downright twisted subquery, which joins each entry in the table with itself and counts how many entries are higher than it to get a rank:

```
dellstore2=# SELECT
  (SELECT COUNT(*) FROM
      (SELECT
         sum(netamount) as sales
       FROM orders GROUP BY customerid
      ) AS orderers WHERE orderers.sales>=leaders.sales
  ) AS rank,
  leaders.*
FROM
  (SELECT
     customerid,sum(netamount) as sales
   FROM orders GROUP BY customerid ORDER BY sales DESC LIMIT 5
   ) AS leaders;
 rank | customerid |  sales
------+------------+---------
    1 |      15483 | 1533.76
    2 |       9315 | 1419.19
    3 |      15463 | 1274.29
    4 |      10899 | 1243.14
    5 |       3929 | 1196.87
(5 rows)
```

Not only is this terribly complex, while it looks fine on this subset, it falls apart the minute two customers with matching order balances show up. It's possible to improve that using `DISTINCT` to note when that happens and to add a tie-breaker, but then you'd be looking at an even crazier query.

This example shows the sort of fundamental limitations you can run into with SQL. You don't know where rows are in relation to one another; and you can't reference what data is in the row before or after the one you're currently referencing either. Another fun query to try and write would be to ask for the difference in sales volume between each of these top five customers. That requires a similarly painful self-join or dropping into a procedural language.

Using window functions for numbering

PostgreSQL 8.4 introduces support to handle this sort of problem using features added to newer SQL standards. These add the concept of a window over which you can get additional data. For example, you can make your window in this type of query to line up with each customer, and then use the `row_number()` function to find out the ranking:

```
dellstore2=# SELECT row_number() OVER (ORDER BY sum(netamount) DESC) AS rank,customerid,sum(netamount) as sales FROM orders GROUP BY customerid ORDER BY sales DESC LIMIT 5;
 rank | customerid |  sales
------+------------+---------
    1 |      15483 | 1533.76
    2 |       9315 | 1419.19
    3 |      15463 | 1274.29
    4 |      10899 | 1243.14
    5 |       3929 | 1196.87
(5 rows)
```

That's not only a whole lot cleaner; it's way more efficient too.

Using window functions for cumulatives

As a second example, consider the case where you're computing sales for each month orders were placed:

```
dellstore2=# SELECT EXTRACT(year FROM orderdate) AS year, EXTRACT(month FROM orderdate) AS month,sum(netamount) as sales FROM orders GROUP BY year,month ORDER BY year,month;
 year | month |   sales
------+-------+-----------
 2004 |     1 | 199444.50
 2004 |     2 | 200269.86
 2004 |     3 | 194516.68
 2004 |     4 | 200131.18
 2004 |     5 | 197131.77
 2004 |     6 | 199919.21
 2004 |     7 | 190953.70
 2004 |     8 | 194102.55
 2004 |     9 | 196798.83
 2004 |    10 | 195798.80
 2004 |    11 | 199882.86
 2004 |    12 | 202769.80
(12 rows)
```

What if you wanted a cumulative total at each point during the year? Since that problem also requires referencing things that are in other rows, it's again not something you can do in the standard SQL model easily. Try it out; it's another fun challenge. Possible solutions include building a complicated query that joins the preceding output with itself, or bypassing SQL limitations altogether by using a function.

But you can easily wrap that with a window function instead too:

```
dellstore2=# SELECT *,SUM(sales) OVER (ORDER BY year,month) as cumulative
        FROM (SELECT EXTRACT(year FROM orderdate) AS year,
        EXTRACT(month FROM orderdate) AS month,sum(netamount) as sales
        FROM orders GROUP BY year,month ORDER BY year,month) as m
        ORDER BY year,month;
 year | month |   sales   | cumulative
------+-------+-----------+------------
 2004 |     1 | 199444.50 |  199444.50
 2004 |     2 | 200269.86 |  399714.36
 2004 |     3 | 194516.68 |  594231.04
 2004 |     4 | 200131.18 |  794362.22
 2004 |     5 | 197131.77 |  991493.99
 2004 |     6 | 199919.21 | 1191413.20
 2004 |     7 | 190953.70 | 1382366.90
 2004 |     8 | 194102.55 | 1576469.45
 2004 |     9 | 196798.83 | 1773268.28
 2004 |    10 | 195798.80 | 1969067.08
 2004 |    11 | 199882.86 | 2168949.94
 2004 |    12 | 202769.80 | 2371719.74
(12 rows)
```

That's not only simpler code to maintain, it's going to be much faster than either of the other solutions too. The plan for it looks like this:

```
dellstore2=# EXPLAIN SELECT *,SUM(sales) OVER (ORDER BY year,month) as cumulative
       FROM (SELECT EXTRACT(year FROM orderdate) AS year,
       EXTRACT(month FROM orderdate) AS month,sum(netamount) as sales
       FROM orders GROUP BY year,month ORDER BY year,month) as m
       ORDER BY year,month;
                                                
```

Summary

A major goal of this chapter was not only to show you how a variety of queries execute, but also to demonstrate, using many examples, how to set up a query testing playground for exploring that yourself. You shouldn't just read this chapter. You should load the Dell Store 2 data into your system and experiment with the queries yourself.

Try to understand how the queries are actually executed in each of these cases, based on the statistics available. Adjust the statistics collected and see if anything changes. Write new queries and see if the plans you get match what you expected. Watch how the optimizer degrades as you take away its advanced features. That sort of practice is the only way to really become an expert at query optimization. Once you see how to read query plans and understand how each of the underlying node types work on this relatively simple database, then you should be armed to wrestle the queries on a production database. The variation between estimated and actual rows can cause major planning issues. Looking into the underlying statistics is one way to determine why that's happening; at a minimum, make sure you have recent statistics from an `ANALYZE`. Consider portions of the query that actually had the longest execution time, and see if they had an appropriate matching cost. If not, this may also suggest a statistics problem.

SQL Window queries can be extremely efficient in situations where knowing more about a row's context and data in surrounding rows simplifies the logic.

This chapter slightly touches the statistics, but the next chapter will cover the database activity and statistics in detail.

11
Database Activity and Statistics

PostgreSQL includes a subsystem named the statistics collector that allows monitoring various aspects of the database internals. Each of the processes in the server sends statistics messages to the collector, which then totals the results and periodically publishes the results to where you can see them. By default (the interval is a compile-time option) statistics updates are published every half second. Each of the statistics values is available via a set of functions that return the current value, all documented at `http://www.postgresql.org/docs/current/static/monitoring-stats.html`.

Topics we will cover in this chapter are:

- PostgreSQL statistics views
- Table and Index statistics
- Database level statistics
- Locks
- Background writer statistics

Statistics views

In practice, a series of views that organizes this data is how most people access the statistics collector's output. Reading the SQL source code to these views is quite informative for learning how they can be used. If you have a PostgreSQL source code tree, the file that creates these views is `src/backend/catalog/system_views.sql`; in an installed database binary tree you'll find them at `share/system_views.sql`, and alternately you can download the source code at `http://anoncvs.postgresql.org/cvsweb.cgi/pgsql/src/backend/catalog/system_views.sql`.

A simple view from there that shows how these are commonly constructed is the `pg_stat_bgwriter` view:

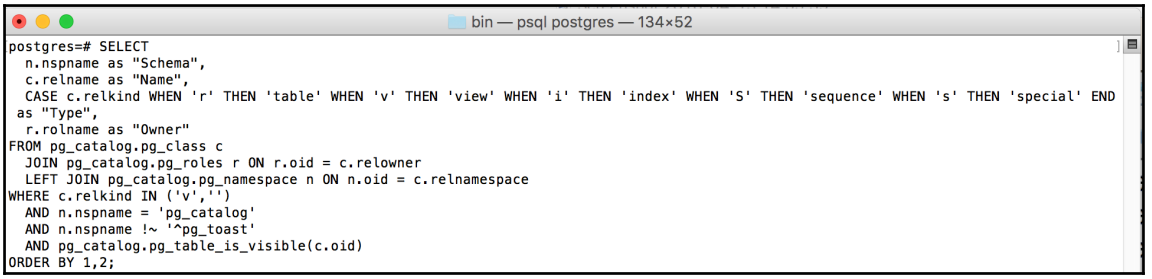
```
postgres=# CREATE VIEW pg_stat_bgwriter AS
    SELECT
        pg_stat_get_bgwriter_timed_checkpoints() AS checkpoints_timed,
        pg_stat_get_bgwriter_requested_checkpoints() AS checkpoints_req,
        pg_stat_get_bgwriter_buf_written_checkpoints() AS buffers_checkpoint,
        pg_stat_get_bgwriter_buf_written_clean() AS buffers_clean,
        pg_stat_get_bgwriter_maxwritten_clean() AS maxwritten_clean,
        pg_stat_get_buf_written_backend() AS buffers_backend,
        pg_stat_get_buf_alloc() AS buffers_alloc;
```

Here is the result of the `pg_stat_bgwriter`:

```
postgres=# SELECT * FROM pg_stat_bgwriter;
-[ RECORD 1 ]---------+------------------------------
checkpoints_timed     | 84
checkpoints_req       | 1
checkpoint_write_time | 2318
checkpoint_sync_time  | 4
buffers_checkpoint    | 23
buffers_clean         | 0
maxwritten_clean      | 0
buffers_backend       | 0
buffers_backend_fsync | 0
buffers_alloc         | 372
stats_reset           | 2017-05-18 13:18:28.094152+05
```

As you can see, this view is just a thin wrapper around the set of functions that each return the individual statistics counters in this area. The rest of the statistics views are similarly constructed. Systems views are a useful way to learn how to build queries that execute against the system catalogs to dig information out of them. The exact way the statistics ones work is valuable to understand, because sometimes, the view doesn't provide exactly the information you'd like to see, and you may have to go directly to the counters to get what you really want. This was the case in the *Autovacuum triggering* example given in `Chapter 7`, *Routine Maintenance*. It was impractical to use the `pg_stat_user_tables` view because it didn't quite have all the information needed, and the view itself abstracted away critical information needed to join against the other system catalogs needed.

You can find out all the available catalog views using the following query:



This is exactly the same query `psql` uses to show you view information when executing `\dv` (you can see that by running `psql -E`), except that where that version excludes `pg_catalog` entries, this version only shows views belonging to that schema.

You can actually get the same data out of a much simpler query:

```
SELECT schemaname,viewname FROM pg_views
 WHERE schemaname='pg_catalog' ORDER BY viewname;
```

But that's a less useful and educational base to build more complicated catalog queries on.

Cumulative and live views

There are two types of information you can monitor from the database. The main statistics are stored into counters. These counters start at 0 when you create a new database cluster, increasing with all activity related to that statistic. Counters in this category include `pg_stat_database`, `pg_stat_bgwriter`, and all of the other views whose names start with `pg_stat`.

The exact way that you reset these counters back to 0 again varies quite a bit based on your PostgreSQL version:

- `8.1`: `pg_stat_reset()` resets all statistics. Enabling `stats_reset_on_server_start` in the `postgresql.conf` file will reset everything each time the server is started.
- `8.2`: `pg_stat_reset()` resets just block and row level statistics. Enabling `stats_reset_on_server_start` allows resetting all statistics, including the database and cluster wide ones, each time the server is started.

- 8.3, 8.4: pg_stat_reset() resets all statistics just for the current database. There is no way to reset cluster wide statistics such as pg_stat_bgwriter.
- 9.0: pg_stat_reset() resets all statistics just for the current database. pg_stat_reset_shared('bgwriter') can be used to reset pg_stat_bgwriter. pg_stat_reset_single_table_counters() and pg_stat_reset_single_function_counters() can be used to reset individual table, index, or function statistics.
- 9.1: pg_stat_reset() resets all database level statistics.

It's important to note which fields in a statistics view are cumulative and which are fixed with information like the table name. In order to make sense of the cumulative ones, you should capture that data regularly, with a timestamp, and note how much the values have changed during that time period. An example of how to do this manually for the pg_stat_bgwriter data is given at the end of this chapter. Recommended practice here is to track this information with a monitoring/trending tool that knows how to work with cumulative counters, which is covered in the next chapter.

In addition to the straight statistics counters, there are some live views of the database activity that give a snapshot of what's happening right now. These views give transitory information, and while you can compute statistics from them, that isn't their primary purpose. Views in this category include pg_stat_activity, pg_locks, and pg_prepared_xacts. Note that prepared transactions, the thing monitoring by pg_prepared_xacts, are used for two-phase commit - there is no relation to the much more common use of prepared statements in database applications. Monitoring prepared transactions isn't covered here; you can find basic documentation about it at http://www.postgresql.org/docs/current/static/view-pg-prepared-xacts.html if you are using PREPARE TRANSACTION in your application.

Table statistics

Basic statistics about each table in your database is available in the view pg_stat_all_tables. Since you probably don't want that cluttered by the many system catalog tables, the data in there is split into two additional views: pg_stat_sys_tables, which only shows the system tables, and pg_stat_user_tables which as you might expect only shows your tables. In most cases pg_stat_user_tables is the one you'll want to look at.

The first useful thing you can use this data for is monitoring how well vacuum is working on your database. You get estimated live and dead tuple counts and timestamps for when vacuum and autovacuum last processed the table. Information about how to monitor that data is covered in the `Chapter 7`, *Routine Maintenance*.

You can use this view to determine whether tables are being accessed by sequential or index scans:

```
pgbench=# SELECT schemaname, relname, seq_scan, idx_scan, cast(idx_scan AS numeric) / (idx_scan + seq_scan) AS idx_scan_pct
FROM pg_stat_user_tables WHERE (idx_scan + seq_scan)>0
ORDER BY idx_scan_pct;
 schemaname |     relname      | seq_scan | idx_scan |      idx_scan_pct
------------+------------------+----------+----------+------------------------
 public     | pgbench_accounts |        0 |   400000 | 1.0000000000000000000
 public     | pgbench_tellers  |        0 |   200000 | 1.0000000000000000000
 public     | pgbench_branches |        0 |   200001 | 1.0000000000000000000
(3 rows)
```

In this tiny `pgbench` database, access to `pgbench_branches` and `pgbench_tellers` is being done by sequential scan because all of the data fits into a single data page. The `pgbench_accounts` table is large enough that `SELECT` statements on it are using an index to look up values.

What is more interesting is exactly how many tuples/rows were actually processed by these scans. This example omits the schema name to make it easier to fit here, and note that it and the above have been rounded for better display; they will show more numeric precision when you run these queries:

```
pgbench=# SELECT relname, seq_tup_read, idx_tup_fetch, cast(idx_tup_fetch as numeric) / (idx_tup_fetch + seq_tup_read) AS idx_tup_pct
FROM pg_stat_user_tables WHERE (idx_tup_fetch + seq_tup_read) > 0 ORDER BY idx_tup_pct;
     relname      | seq_tup_read | idx_tup_fetch |      idx_tup_pct
------------------+--------------+---------------+------------------------
 pgbench_accounts |            0 |        200778 | 1.0000000000000000000
 pgbench_tellers  |            0 |        200000 | 1.0000000000000000000
 pgbench_branches |            0 |        200000 | 1.0000000000000000000
(3 rows)
```

Here you can see that each of the 4 sequential scans shown in the above example likely accessed 100,000 tuples each to reach this total. They could easily have had more of an impact on the database than the 144,345 index scans, even though there were only four of them. Both the number of scans and how many total tuples each fetched are interesting numbers to monitor, as they suggest both how many queries run and the total volume executed by them.

A similar query worth monitoring is how often HOT is being used to update rows, instead of a less efficient regular update:

```
pgbench=# SELECT relname, n_tup_upd, n_tup_hot_upd,
CAST(n_tup_hot_upd AS NUMERIC) / n_tup_upd AS hot_pct
FROM pg_stat_user_tables
WHERE n_tup_upd > 0
ORDER BY hot_pct;
     relname      | n_tup_upd | n_tup_hot_upd |        hot_pct
------------------+-----------+---------------+------------------------
 pgbench_tellers  |    200000 |        198624 | 0.99312000000000000000
 pgbench_accounts |    100389 |         99984 | 0.99596569345246989212
 pgbench_branches |    200000 |        199824 | 0.99912000000000000000
(3 rows)
```

These numbers are a bit different in that `n_tup_upd` is already a total that `n_tup_hot_upd` is a subset of; you don't need to add them together. In this example, almost all of the updates being done are using HOT, which is what you'd like to see for best `UPDATE` performance. This percentage is particularly valuable to monitor when doing the initial testing of your database, to confirm whether or not you are satisfying the conditions HOT kicks in for.

A final useful derived view is to consider the insert/update/delete characteristics of your tables:

```
pgbench=# SELECT relname,cast(n_tup_ins AS numeric) / (n_tup_ins + n_tup_upd + n_tup_del)
AS ins_pct,cast(n_tup_upd AS numeric) / (n_tup_ins + n_tup_upd + n_tup_del)
AS upd_pct, cast(n_tup_del AS numeric) / (n_tup_ins + n_tup_upd + n_tup_del)
AS del_pct
FROM pg_stat_user_tables
ORDER BY relname;
     relname      |          ins_pct           |          upd_pct           |          del_pct
------------------+----------------------------+----------------------------+----------------------------
 pgbench_accounts | 0.00000000000000000000000000 |     1.0000000000000000000 | 0.00000000000000000000000000
 pgbench_branches | 0.00000000000000000000000000 |     1.0000000000000000000 | 0.00000000000000000000000000
 pgbench_history  |     1.0000000000000000000 | 0.00000000000000000000000000 | 0.00000000000000000000000000
 pgbench_tellers  | 0.00000000000000000000000000 |     1.0000000000000000000 | 0.00000000000000000000000000
(4 rows)
```

This confirms that `pgbench_history` is what's sometimes called an append-only table: one that is inserted into but never updated or deleted from. It also suggests that `pgbench_branches` and `pgbench_tellers` are being heavily thrashed around with updates relative to insertions, which can be valuable for determining what tables are likely to need periodic `REINDEX` operations. You can use the deletion percentage figure for a similar purpose, finding tables that are likely to have a large quantity of sparse data or index blocks that you might need an operation like `CLUSTER` to fully clean up after.

Table I/O

In addition to the operation statistic counts, there are also a set of counters in the database that concentrate on physical I/O in `pg_statio_user_tables` (with all/system variations too). The first pair of fields monitors use of the `shared_buffers` structure that caches table data, which is called the heap in this context. When a read happens, the database distinguishes between whether that read could be satisfied using a block already in the database buffer cache (a heap block hit), or whether it required an operating system read to satisfy:

```
pgbench=# SELECT relname,cast(heap_blks_hit as numeric) / (heap_blks_hit + heap_blks_read)
AS hit_pct,heap_blks_hit,heap_blks_read
FROM pg_statio_user_tables
WHERE (heap_blks_hit + heap_blks_read)>0
ORDER BY hit_pct;
     relname      |        hit_pct         | heap_blks_hit | heap_blks_read
------------------+------------------------+---------------+----------------
 pgbench_accounts | 0.67070668480283503881 |        203646 |          99983
 pgbench_history  | 0.98463323124414524169 |        202863 |           3166
 pgbench_tellers  | 0.99892918611250899047 |        434717 |            466
 pgbench_branches | 0.99956967107633705667 |        485466 |            209
(4 rows)
```

This is a well-cached database; almost none of the reads require actual disk I/O.

Not all heap block reads actually turn into physical disk I/O - they might instead read the data from the operating system cache. Currently the database doesn't have any idea which happens. But it is possible to monitor this using the Dtrace on operating systems that support it, using the probes available in PostgreSQL.

A similar query shows all of the disk I/O for every index on this table:

```
pgbench=# SELECT relname,cast(idx_blks_hit as numeric) / (idx_blks_hit + idx_blks_read)
AS hit_pct,idx_blks_hit,idx_blks_read
FROM pg_statio_user_tables
WHERE (idx_blks_hit + idx_blks_read)>0
ORDER BY hit_pct;
     relname      |        hit_pct         | idx_blks_hit | idx_blks_read
------------------+------------------------+--------------+---------------
 pgbench_accounts | 0.86322298262516195475 |      1039355 |        164685
 pgbench_branches | 0.99999759866485766086 |       416434 |             1
 pgbench_tellers  | 0.99999760536398467433 |       417599 |             1
(3 rows)
```

You don't see `pgbench_history` here because it has no indexes.

The table I/O statistics also include counters for I/O related to oversized data stored using the database's TOAST scheme. If your system includes TOASTed data, you need to account for those reads and writes when computing the total read load associated with a table and its indexes. The main three components to reading blocks in the table (as opposed to the index) in this situation--the table blocks, the TOAST blocks, and the TOAST index blocks - should be aggregated together into a total:

```
SELECT *,
 (heap_blks_read + toast_blks_read + tidx_blks_read)
      AS total_blks_read,
 (heap_blks_hit + toast_blks_hit + tidx_blks_hit)
      AS total_blks_hit
FROM pg_statio_user_tables;
```

The hit ratio for that total should be the figure you follow. Since that approach works fine for non-TOAST tables - the TOAST parts will just be zero--always including the TOAST figures is a good habit to adopt.

PostgreSQL does not allow a single tuple to span around multiple pages (by default 8K). That's mean its impossible to add tuple with very large column data. To overcome this PostgreSQL divide the tuple into multiple physical rows. This is called TOAST tables (The Oversized-Attribute Storage Technique). The complete detail can be found at official documentation at `https://www.postgresql.org/docs/current/static/storage-toast.html`.

Index statistics

There is a parallel set of statistics to the table ones that break down activity for each individual index. `pg_stat_user_indexes` also has system/all versions available, and it gives information about how many index scans were done and the rows returned by them.

The naming of the fields in this view is more complicated to distinguish among though. Two fields that look similar are subtly different:

- `idx_tup_read`: In this number of index entries returned by index scans.
- `idx_tup_fetch`: In this number of live table rows fetched by simple index scans using that index. This number will often be a bit smaller than the `idx_tup_read` value because it won't include dead or not yet committed rows in its count. In addition, the use of "simple" here means that the index access was not a bitmap index scan. When those execute, there are at least two indexes involved. That makes it impossible to properly account for what index was responsible for a particular row being fetched. The server therefore doesn't include it in this total for any index, instead only incrementing the index tuple fetch figure for the table as a whole instead.

Because there are so many situations where `idx_tup_fetch` isn't incremented, if you want to track how heavily an index is being used from the system loading performance `idx_tup_read` is the better statistic.

One interesting figure you can derive from this data is how many rows this index scan returns on average, given how the index is being used:

```
pgbench=# SELECT indexrelname,cast(idx_tup_read AS numeric) / idx_scan
AS avg_tuples,idx_scan,idx_tup_read
FROM pg_stat_user_indexes
WHERE idx_scan > 0;
     indexrelname      |       avg_tuples       | idx_scan | idx_tup_read
-----------------------+------------------------+----------+--------------
 pgbench_branches_pkey |     1.1194844025779871 |   200001 |       223898
 pgbench_tellers_pkey  |     1.0975750000000000 |   200000 |       219515
 pgbench_accounts_pkey | 0.50598250000000000000 |   400000 |       202393
(3 rows)
```

This shows that `pgbench` is only grabbing a single row most of the time, which is unsurprising for a primary key. The small amount `avg_tuples` here is greater than 1.0, reflecting the occasional time a dead row was scanned. This average can give you an idea how often larger index range scans are being used.

The main thing that the counts in `pg_stat_user_indexes` are useful for is determining which indexes are actually being used by your application. Since indexes add overhead to the system, ones that aren't actually being used to satisfy queries should be removed from the design. The easiest way to find such indexes is to look for low values for `idx_scan`, which will appear at the top of this query:

```
SELECT
  schemaname,
  relname,
  indexrelname,
  idx_scan,
  pg_size_pretty(pg_relation_size(i.indexrelid))
     AS index_size
FROM
  pg_stat_user_indexes i
  JOIN pg_index USING (indexrelid)
WHERE
  indisunique IS false
ORDER BY idx_scan,relname;
```

This specifically excludes indexes that are enforcing a unique constraint, which you likely can't drop even if they are minimally used.

Index I/O

The same basic I/O statistics available for indexes are also available broken down for each index:

```
pgbench=# SELECT indexrelname,cast(idx_blks_hit as numeric) / (idx_blks_hit + idx_blks_read)
AS hit_pct,idx_blks_hit,idx_blks_read
FROM pg_statio_user_indexes
WHERE (idx_blks_hit + idx_blks_read)>0;
     indexrelname      |        hit_pct         | idx_blks_hit | idx_blks_read
-----------------------+------------------------+--------------+---------------
 pgbench_branches_pkey | 0.99999759866485766086 |       416434 |             1
 pgbench_tellers_pkey  | 0.99999760536398467433 |       417599 |             1
 pgbench_accounts_pkey | 0.86322298262516195475 |      1039355 |        164685
(3 rows)
```

Indexes tend to be better cached than their underlying tables. If they're not, that's another hint that you might not be indexing your data properly.

Database wide totals

Many of the same pieces of data available at the table level are also summarized per database. You can get the following information out of `pg_stat_database` and use it in similar ways to `pg_stat_user_tables` and `pg_statio_user_tables`:

```
SELECT datname,blks_read,blks_hit,tup_returned,
 tup_fetched,tup_inserted ,tup_updated,tup_deleted
FROM pg_stat_database;
```

In addition, there are some useful transaction commit statistics available, as well as a count of the total active client backend connections active for each database:

```
SELECT datname,numbackends,xact_commit,xact_rollback
 from pg_stat_database;
```

Connections and activity

`pg_stat_activity` provides a way to get a snapshot of what every client on the server is currently doing. Because it includes a process ID, on UNIX-like systems `pg_stat_activity` is also useful to line up with information collected at the operating system level, by utilities such as `top` or `ps`.

The simplest thing to do with this view is to count how many client backends are currently active:

```
pgbench=# SELECT count(*) FROM pg_stat_activity WHERE NOT pid=pg_backend_pid();
 count
-------
     2
(1 row)
```

Since the query itself will normally appear in the results, that's filtered out by looking up its process id and excluding it. This is a good practice to get into for queries against this view. This total gives you an idea how close you are to reaching the server's `max_connections` at any time, and monitoring a high-water mark for its value is a good practice to let you know when you're likely to exceed it.

You can use `pg_stat_activity` to see how long a backend has been running and whether it's waiting for anything right now:

```
postgres=# SELECT pid,state,current_timestamp - least(query_start,xact_start)
AS runtime,substr(query,1,25)
AS current_query FROM pg_stat_activity
WHERE NOT pid=pg_backend_pid();
 pid  | state |     runtime     |       current_query
------+-------+-----------------+--------------------------
 7559 | idle  | 00:30:45.017303 | SELECT count(*) FROM pg_s
 7658 | idle  | 00:31:02.440805 | SELECT schemaname, relnam
(2 rows)
```

This shows a couple of common tricks to get output from `pg_stat_activity` down to a manageable set. In addition to blocking the query itself, the very large query text is cut down to a reasonable size to fit on the screen.

This example also demonstrates two issues that you can run into that aren't so easy to fix. One is that when a backend client has stopped doing work, it will appear as either `<IDLE>` or `<IDLE> in transaction`. In either case, you cannot tell what that client was doing just before becoming idle, so once it's reached an idle state you have limited ability to figure out how it got there.

The second issue you're much less likely to run into is the negative `runtime` shown in one row here. The PostgreSQL system time functions that `current_timestamp` belongs to return the time of the current transaction for the client asking for the information. If someone else starts a session after yours, but before you run the query against `pg_stat_activity`, they can have a transaction or query start after what your client believes to be the current time. It's possible to get the true system time instead using the `timeofday()` function. Given that that returns a string rather than a true timestamp that you can further manipulate, it normally isn't worth the effort to avoid this rare situation. In most cases, you should be more interested in long-running transactions rather than just started ones anyway.

Locks

The MVCC model used in PostgreSQL makes locks less of an issue than in some other databases. High performance applications still need to be careful to minimize how often they take stronger locks however, and performance tuning work will regularly require looking for situations where locking behavior is a bottleneck.

The details of how locking works in the database is described extensively at `http://www.postgresql.org/docs/current/static/explicit-locking.html`.

The other useful manual section is the description of the `pg_locks` view, which lets you see all the active locks in the database, available at `http://www.postgresql.org/docs/current/static/view-pg-locks.html`.

As there's far too many details involved in locking to cover all of them in this chapter, consider those sections of the documentation required reading in addition to what's here. This section will mainly present how to work with `pg_locks` and examples of what sort of information you'll actually see in there. All of these are locks generated by running the standard and select-only `pgbench` tests.

Virtual transactions

One bit that isn't really clear from the documentation is how the database internally handles assignment of transaction identification numbers, called transaction ids or simply XIDs. And you can't easily decode the `pg_locks` information without knowing some trivia here.

Every statement in PostgreSQL executes within a transaction, either explicit or implied. If you want a transaction to span multiple statements, you put it into a transaction block using the `BEGIN` statement. Giving something an XID in PostgreSQL has some overhead. For example, each one of them is assigned its own space in the database commit logs, stored in the `pg_clog` directory.

As that involves several types of work including an eventual disk write, this is somewhat expensive. And simple read-only statements like many `SELECT`s don't really need a transaction ID anyway. Starting in PostgreSQL 8.3, an optimization was added to avoid this overhead when it's not needed, using what's referred to inside the code as "lazy XID allocation". Transactions are now assigned a real transaction ID, what the `pg_locks` documentation calls a "permanent ID", only when they modify a database row. Read-only transactions are never given one.

However, the transaction ID number is what's used to implement transactions waiting on one another for a lock, the information shown in `pg_locks`. In order to allow this, the lazy allocation scheme also introduced the concept of a virtual transaction ID. These are allocated using a less permanent naming scheme, one that doesn't involve writing the transaction information to disk.

Since many applications have far more read-only `SELECT` statements than ones that modify rows, in addition to reducing disk writes this optimization significantly reduces the rate at which real transaction IDs are used up within the database. The database has to do some extensive transaction ID wraparound cleanup periodically, as described in `Chapter 7`, *Routine Maintenance*. Reducing the "burn rate" of transaction IDs using lazy allocation, preferring virtual XIDs if you can avoid allocating a real one, lowers how often that happens too.

With that background, this somewhat brief statement from the documentation is worth highlighting:

> *'Every transaction holds an exclusive lock on its virtual transaction ID for its entire duration. If a permanent ID is assigned to the transaction (which normally happens only if the transaction changes the state of the database), it also holds an exclusive lock on its permanent transaction ID until it ends. When one transaction finds it necessary to wait specifically for another transaction, it does so by attempting to acquire share lock on the other transaction ID (either virtual or permanent ID depending on the situation). That will succeed only when the other transaction terminates and releases its locks.'*

Understanding the way this mechanism works for waiting on another transaction ID, either permanent or virtual, is critical to finding and resolving locking issues in an application. And the way transactions wait for other transactions isn't always just a simple pairing, even though it's displayed as such by the locks display. You can actually end up with more of a tree structure instead, where transaction A is waiting for B, while B is waiting for C.

Decoding lock information

The information provided by `pg_locks` is very basic. In order to understand what it means in more real-world terms, you're going to want to match its data against several system catalog tables, as well as the activity information for the query. Here's an example showing how to relate this view to the most popular ones to join it against:

```
SELECT
locktype,
virtualtransaction,
transactionid,
nspname,
relname,
mode,
granted,
cast(date_trunc('second',query_start) AS timestamp)
 AS query_start,
```

```
substr(query,1,25) AS query
FROM
pg_locks
LEFT OUTER JOIN pg_class ON (pg_locks.relation = pg_class.oid)
LEFT OUTER JOIN pg_namespace ON (pg_namespace.oid =
 pg_class.relnamespace),
pg_stat_activity
WHERE
NOT pg_locks.pid=pg_backend_pid() AND
pg_locks.pid=pg_stat_activity.pid;
```

Note that the way this query discovers table relation names will only produce results for tables in the currently connected database; they will be blank for locks held against tables in other databases.

You might want to turn the query or transaction start time into a time interval relative to now, similarly to how `runtime` is computed in the `pg_stat_activity` example in the previous section.

There are two extremely common entries you'll see in `pg_locks`. Queries that are accessing a table will obtain a shared lock on that relation, which prevents anyone else from obtaining an exclusive lock on it:

```
locktype           | relation
virtualtransaction | 2/2478876
transactionid      |
nspname            | public
relname            | pgbench_accounts
mode               | AccessShareLock
granted            | t
query_start        | 2010-04-11 22:44:47
query              | SELECT abalance FROM pgbe
```

And queries will lock their own virtual transaction ids, which is what queries that haven't needed to a get a real transaction id yet are assigned:

```
locktype           | virtualxid
virtualtransaction | 2/2478876
transactionid      |
nspname            |
relname            |
mode               | ExclusiveLock
granted            | t
query_start        | 2010-04-11 22:44:47
query              | SELECT abalance FROM pgbe
```

When you run a more complicated query that needs a real transaction ID, like an `UPDATE`, it will be assigned one that it then acquires an exclusive lock on (as the owner of that transaction id) like this:

```
locktype           | transactionid
virtualtransaction | 2/2528843
transactionid      | 328491
nspname            |
relname            |
mode               | ExclusiveLock
granted            | t
query_start        | 2010-04-11 22:47:19
query              | UPDATE pgbench_branches S
```

If another query has to wait for that transaction to end, it would do so by requesting a shared lock on it; one that was then granted after the transaction finished looks like this:

```
locktype           | transactionid
virtualtransaction | 1/1165014
transactionid      | 371417
nspname            |
relname            |
mode               | ShareLock
granted            | t
query_start        | 2010-04-11 22:49:09
query              | UPDATE pgbench_branches S
```

In order to actually change a row, statements like `UPDATE` acquire an exclusive lock on the row they are changing:

```
locktype           | relation
virtualtransaction | 1/1165014
transactionid      |
nspname            | public
relname            | pgbench_tellers
mode               | RowExclusiveLock
granted            | t
query_start        | 2010-04-11 22:49:09
query              | UPDATE pgbench_branches S
```

And then a lock will be acquired on the tuple (the actual data in the row); here's what that looks like before the lock is acquired, showing an example of an ungranted lock that an application is waiting for:

```
locktype           | tuple
virtualtransaction | 4/2526095
transactionid      |
```

```
nspname            | public
relname            | pgbench_branches
mode               | ExclusiveLock
granted            | f
query_start        | 2010-04-11 22:47:19
query              | UPDATE pgbench_branches S
```

Monitoring your application to see what type of locks it acquires, what it acquires them on, and what it ends up waiting on is a great way to get a feel for how the locking mechanism described in the PostgreSQL manual ends up working out in practice.

Table-level Lock Modes

There are several modes of table level locks. Here is the list of table level locks.

- Access Share
- ROW Share
- ROW Exclusive
- Share Update Exclusive
- Share
- Share Row Exclusive
- Exclusive
- Access Exclusive

The complete detail of all the locks is available at PostgreSQL official documentation:

`http://www.postgresql.org/docs/current/static/explicit-locking.html`

Here is the link where you can get the detail example of the locks:

`http://big-elephants.com/2013-09/exploring-query-locks-in-postgres/`

Transaction lock waits

Locking performance issues will often be evident by an excess of clients that are waiting for a lock to be granted. If you join two `pg_locks` entries together with a matching pair of `pg_stat_activity` ones, it's possible to find out information about both the locker process that currently holds the lock, and the locked one stuck waiting for it:

```
SELECT
  locked.pid AS locked_pid,
```

```
  locker.pid AS locker_pid,
  locked_act.usename AS locked_user,
  locker_act.usename AS locker_user,
  locked.virtualtransaction,
  locked.transactionid,
  locked.locktype
FROM
  pg_locks locked,
  pg_locks locker,
  pg_stat_activity locked_act,
  pg_stat_activity locker_act
WHERE
  locker.granted=true AND
  locked.granted=false AND
  locked.pid=locked_act.pid AND
  locker.pid=locker_act.pid AND
  (locked.virtualtransaction=locker.virtualtransaction OR
  locked.transactionid=locker.transactionid);
```

This variation looks for and provides additional information about transaction ID lock waits like this:

```
locked_pid         | 11578
locker_pid         | 11578
locked_user        | postgres
locker_user        | postgres
virtualtransaction | 2/2580206
transactionid      | 534343
locktype           | transactionid
```

These will also show up for virtual transactions, as mentioned before:

```
locked_pid         | 11580
locker_pid         | 11580
locked_user        | postgres
locker_user        | postgres
virtualtransaction | 4/2562729
transactionid      |
locktype           | tuple
```

The preceding examples aren't necessarily representative of common situations; note that the pid is actually the same in each case, and this lock sequence is due to how the `pgbench` transactions execute. You will likely want to display more of the information available when analyzing locks like this, such as showing enough of the query text for each activity row to see what is happening.

Table lock waits

Clients waiting to acquire a lock on an entire table are something you should aim to avoid. The following will show you when this is happening:

```
SELECT
  locked.pid AS locked_pid,
  locker.pid AS locker_pid,
  locked_act.usename AS locked_user,
  locker_act.usename AS locker_user,
  locked.virtualtransaction,
  locked.transactionid,
  relname
FROM
  pg_locks locked
    LEFT OUTER JOIN pg_class ON (locked.relation = pg_class.oid),
  pg_locks locker,
  pg_stat_activity locked_act,
  pg_stat_activity locker_act
WHERE
  locker.granted=true AND
  locked.granted=false AND
  locked.pid=locked_act.pid AND
  locker.pid=locker_act.pid AND
  locked.relation=locker.relation;
```

Output from this query looks similar to the previous transaction id examples:

```
locked_pid         | 12474
locker_pid         | 12247
locked_user        | postgres
locker_user        | postgres
virtualtransaction | 2/2588881
transactionid      |
relname            | pgbench_accounts
```

You really shouldn't see this form of lock pop up under normal circumstances-this one was generated by doing an explicit `LOCK` on a table and then trying to query against it in another session.

Logging lock information

At this point you'd probably like a more automatic way to log locking problems than to just watch `pg_locks` all day. There is a pair of database parameters that allow doing that, as a by-product of sorts of how the database defends itself against deadlock problems.

Deadlocks

Consider the following sequence:

- Process 1 acquires a lock on object A.
- Process 2 acquires a lock on object B.
- Process 2 tries to acquire a lock on object A. It's now waiting for Process 1 to finish.
- Process 1 tries to acquire a lock on object B.

At this point the two processes are now in what's called deadlock: each is trying to obtain a lock on something owned by the other. They will both wait on each other forever if left in this state. One of them has to give up and release the locks they already have.

To search for this situation and resolve it, PostgreSQL doesn't wait forever for a lock. Instead, it only waits an amount of time determined by the `deadlock_timeout` parameter. After waiting that long, any process trying to acquire a lock will run the deadlock detector algorithm. This logic inside the database looks for deadlock in the form of locks that lead in a circle, either a simple pair or a more complicated chain. If one is found, the process running the deadlock detector, the one waiting for the lock (not the one who already has it), aborts. If it turns out there is no deadlock, and it's just taking a long time to acquire the lock, the process then goes back to sleep and starts over.

When deadlock occurs, this is always logged. If you turn on the `log_lock_waits` parameter in the database, each time the deadlock detector runs and determines there's no deadlock, information about what it's waiting for is also written to the database. You can use this to figure out just what locks are taking a long time to acquire in your application.

The default lock wait time here in the form of `deadlock_timeout` is 1 second. This default is reasonable for deadlock detection. If you want to find and log lock waits, you may want to reduce that timeout, so that you can find smaller ones. There is some additional overhead from that change, in that you'll be running the deadlock detector code more frequently too. This is why the lock timeout doesn't default to something smaller.

Disk usage

The amount of disk space used by tables and indexes in the database is informative in two major ways. Since many database operations have execution times proportional to the size of the table, tracking size over time can help you predict how query time is going to increase in the future. And as described in the database maintenance chapter, tables or indexes whose sizes change in unexpected ways can indicate a problem with the vacuum strategy being employed.

The basic way to find out how much disk space is used by a table or index is to run `pg_relation_size()` on it. This is often combined with `pg_size_pretty()`, which will provide a human-readable version of the size.

Other useful size queries include `pg_column_size()` and `pg_database_size()`.

A quick example of how to query this information across all the tables in the current database.

On a `pgbench` database with a lot of work done on it, and thus a large history table, output from this might look like this:

```
pgbench=# SELECT
  nspname,
  relname,
  pg_size_pretty(pg_relation_size(C.oid)) AS "size"
FROM pg_class C
LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
WHERE nspname NOT IN ('pg_catalog', 'information_schema')
ORDER BY pg_relation_size(C.oid) DESC
LIMIT 20;
 nspname  |        relname        |    size
----------+-----------------------+------------
 public   | pgbench_accounts      | 163 MB
 public   | pgbench_accounts_pkey | 27 MB
 public   | pgbench_history       | 10 MB
 public   | pgbench_tellers       | 480 kB
 pg_toast | pg_toast_2618         | 376 kB
 public   | pgbench_branches      | 248 kB
 public   | pgbench_tellers_pkey  | 168 kB
 public   | pgbench_branches_pkey | 40 kB
 pg_toast | pg_toast_2619         | 24 kB
 pg_toast | pg_toast_2619_index   | 16 kB
 pg_toast | pg_toast_2618_index   | 16 kB
 pg_toast | pg_toast_2606_index   | 8192 bytes
 pg_toast | pg_toast_2396_index   | 8192 bytes
 pg_toast | pg_toast_12509_index  | 8192 bytes
 pg_toast | pg_toast_1255_index   | 8192 bytes
 pg_toast | pg_toast_3596_index   | 8192 bytes
 pg_toast | pg_toast_2620_index   | 8192 bytes
 pg_toast | pg_toast_2964_index   | 8192 bytes
 pg_toast | pg_toast_3592_index   | 8192 bytes
 pg_toast | pg_toast_12514_index  | 8192 bytes
(20 rows)
```

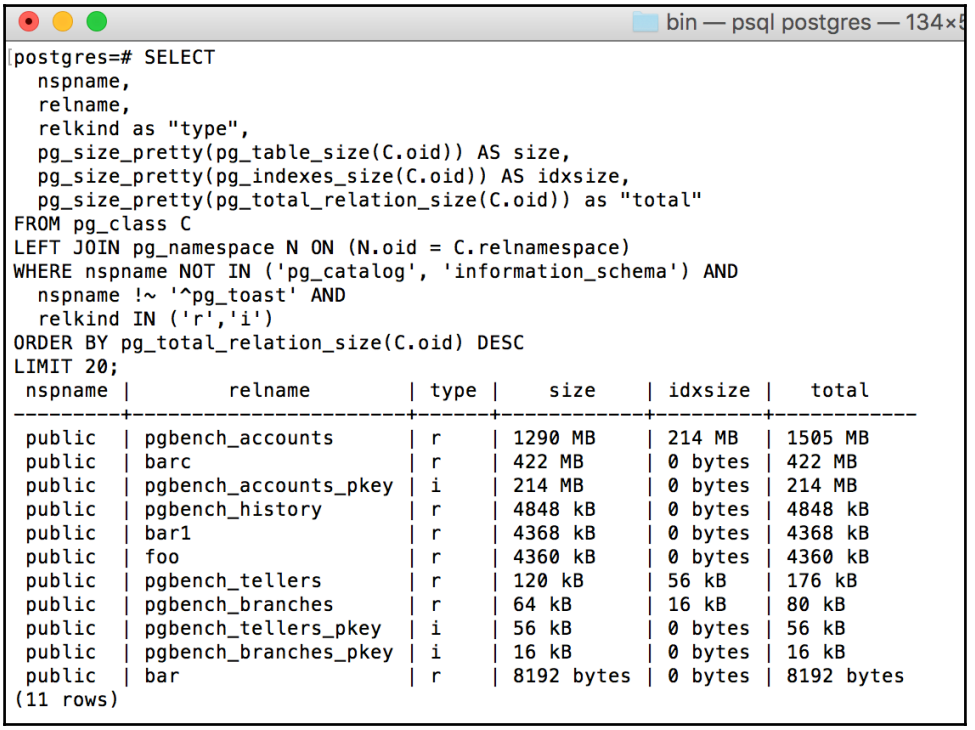
This shows the main issue with this simple query: wide columns that have been moved out of the main table into a TOAST relation are broken out as their own entries. There is a `pg_total_relation_size()` function available too, but it also includes indexes.

As of PostgreSQL 9.0, there is a function named `pg_table_size()` that includes all of the TOAST information in its total, but not the indexes. And `pg_indexes_size()` gives a total size for all the indexes of this table. The actual components break down like this:

```
pg_total_relation_size = pg_table_size + pg_indexes_size
pg_table_size = pg_relation_size + toast table + toast indexes + FSM
```

Starting with PostgreSQL 8.4 the free space map is stored in a separate "relation fork" that takes up some amount of disk space; in earlier versions that isn't a component to the size you need to worry about. There is a way to ask `pg_relation_size()` for the FSM size, but this is ignored in the examples here, as it's not something particularly useful to track.

This query uses the new functions to display the TOAST and index sizes:



In earlier versions, you can compute the TOAST total or the index total manually:

```
bin — psql postgres — 134×52
postgres=# SELECT
  nspname,
  C.relname,
  C.relkind as "type",
  pg_size_pretty(pg_relation_size(C.oid)) AS size,
  pg_size_pretty(
    CASE when C.reltoastrelid > 0 THEN pg_relation_size(C.reltoastrelid) ELSE 0 END +
    CASE when T.indexrelid > 0 THEN pg_relation_size(T.indexrelid) ELSE 0 END
    ) AS toast,
  pg_size_pretty(cast(
    (SELECT sum(pg_relation_size(I.indexrelid))
     FROM pg_index I WHERE I.indrelid = C.oid)
     AS int8)) AS idxsize,
  pg_size_pretty(pg_total_relation_size(C.oid)) as "total"
   FROM pg_class C
LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
LEFT OUTER JOIN pg_index T ON (C.reltoastrelid=T.indexrelid)
WHERE nspname NOT IN ('pg_catalog', 'information_schema') AND
  nspname !~ '^pg_toast' AND
  C.relkind IN ('r','i')
ORDER BY pg_total_relation_size(C.oid) DESC
LIMIT 20;
 nspname |        relname         | type |  size   |  toast   | idxsize |  total
---------+------------------------+------+---------+----------+---------+---------
 public  | pgbench_accounts       | r    | 1290 MB | 0 bytes | 214 MB  | 1505 MB
 public  | pgbench_accounts_pkey  | i    | 214 MB  | 0 bytes |         | 214 MB
 public  | pgbench_history        | r    | 4824 kB | 0 bytes |         | 4848 kB
 public  | pgbench_tellers        | r    | 88 kB   | 0 bytes | 56 kB   | 176 kB
 public  | pgbench_branches       | r    | 32 kB   | 0 bytes | 16 kB   | 80 kB
 public  | pgbench_tellers_pkey   | i    | 56 kB   | 0 bytes |         | 56 kB
 public  | pgbench_branches_pkey  | i    | 16 kB   | 0 bytes |         | 16 kB
(7 rows)
```

There is complete detail of disk usage and some useful queries at `https://wiki.postgresql.org/wiki/Disk_Usage`.

Monitoring Buffer, background writer, and checkpoint activity

Monitoring gross activity in the database buffer cache was very difficult until PostgreSQL 8.3, where the `pg_stat_bgwriter` view was introduced. This allows tracking the general flow of every data page buffer that goes in or out of the cache, along with statistics about the related checkpoint process responsible for much of that. Some questions you can answer with this data include:

- What percentage of the time are checkpoints being requested based on activity rather than by time passing?
- How much data does the average checkpoint write?
- What percentage of the data being written out happens from checkpoints and backends, respectively?

It's possible to compute these numbers easily enough right from the view. The only additional piece needed is the block size needed to hold a single buffer, available as one of the internal settings exposed in `pg_settings` or `current_setting` as the `block_size` parameter:

```
pgbench=# SELECT
  (100 * checkpoints_req) / (checkpoints_timed + checkpoints_req)
    AS checkpoints_req_pct,
  pg_size_pretty(buffers_checkpoint * block_size / (checkpoints_timed + checkpoints_req))
    AS avg_checkpoint_write,
  pg_size_pretty(block_size * (buffers_checkpoint + buffers_clean + buffers_backend)) AS total_written,
  100 * buffers_checkpoint / (buffers_checkpoint + buffers_clean + buffers_backend) AS checkpoint_write_pct,
  100 * buffers_backend / (buffers_checkpoint + buffers_clean + buffers_backend) AS backend_write_pct,
  *
FROM pg_stat_bgwriter,(SELECT cast(current_setting('block_size') AS integer) AS block_size) AS bs;
-[ RECORD 1 ]---------+------------------------------
checkpoints_req_pct   | 1
avg_checkpoint_write  | 45 kB
total_written         | 1878 MB
checkpoint_write_pct  | 0
backend_write_pct     | 86
checkpoints_timed     | 317
checkpoints_req       | 4
checkpoint_write_time | 182613
checkpoint_sync_time  | 30
buffers_checkpoint    | 1817
buffers_clean         | 31256
maxwritten_clean      | 277
buffers_backend       | 207346
buffers_backend_fsync | 0
buffers_alloc         | 802523
stats_reset           | 2017-05-20 22:52:30.041735+05
block_size            | 8192
```

You can see that only 1% of the checkpoints executed are required ones, which suggests the system has not been active enough to run out of WAL segments before hitting the `checkpoint_timeout` value. The average checkpoint write is 2.8MB, a fairly small amount. A full 86% of the buffers written out were handled by backend writes, which suggests this system isn't configured very well - more should go out via checkpoints. It's like that because `shared_buffers` is tiny on this system.

A look at the total amount written shows you where these cumulative totals start to break down. 24 GB sounds like a lot, but without a time period to reference it over you can't draw any conclusions about it. Is that 24 GB per hour? Per month? There's no way to tell.

Saving pg_stat_bgwriter snapshots

In order for this data to really be interesting, you need to save periodic snapshots of it with an associated timestamp. It's easy to create a table for that purpose and put a first snapshot into it:

```
pgbench=# CREATE TABLE pg_stat_bgwriter_snapshot
AS SELECT current_timestamp,* FROM pg_stat_bgwriter;
SELECT 1
pgbench=# INSERT INTO pg_stat_bgwriter_snapshot (
 SELECT current_timestamp,* FROM pg_stat_bgwriter);
```

Now you can wait until some time has passed, with a representative chunk of activity; at least an hour is recommended, preferably a day. Insert a second snapshot into that table:

```
pgbench=# INSERT INTO pg_stat_bgwriter_snapshot (
 SELECT current_timestamp,* FROM pg_stat_bgwriter);
```

And now it's possible to get a difference both in values and in elapsed time between these two. You might even add the above `INSERT` to a query scheduling regime using tools like `cron` or `pgAgent`. Having two snapshots lets you compute averages in units like bytes/second over time. The following giant query shows a full range of interesting statistics you can derive:

```
pgbench=# SELECT
  cast(date_trunc('minute',start) AS timestamp) AS start,
  date_trunc('second',elapsed) AS elapsed,
  date_trunc('second',elapsed / (checkpoints_timed + checkpoints_req)) AS avg_checkpoint_interval,
  (100 * checkpoints_req) / (checkpoints_timed + checkpoints_req)
    AS checkpoints_req_pct,
  100 * buffers_checkpoint / (buffers_checkpoint + buffers_clean + buffers_backend) AS checkpoint_write_pct,
  100 * buffers_backend / (buffers_checkpoint + buffers_clean + buffers_backend) AS backend_write_pct,
  pg_size_pretty(buffers_checkpoint * block_size / (checkpoints_timed + checkpoints_req))
    AS avg_checkpoint_write,
  pg_size_pretty(cast(block_size * (buffers_checkpoint + buffers_clean + buffers_backend) / extract(epoch FROM elapsed) AS int8)) AS written_per_sec,
  pg_size_pretty(cast(block_size * (buffers_alloc) / extract(epoch FROM elapsed) AS int8)) AS alloc_per_sec
FROM
(
SELECT
  one.now AS start,
  two.now - one.now AS elapsed,
  two.checkpoints_timed - one.checkpoints_timed AS checkpoints_timed,
  two.checkpoints_req - one.checkpoints_req AS checkpoints_req,
  two.buffers_checkpoint - one.buffers_checkpoint AS buffers_checkpoint,
  two.buffers_clean - one.buffers_clean AS buffers_clean,
  two.maxwritten_clean - one.maxwritten_clean AS maxwritten_clean,
  two.buffers_backend - one.buffers_backend AS buffers_backend,
  two.buffers_alloc - one.buffers_alloc AS buffers_alloc,
  (SELECT cast(current_setting('block_size') AS integer)) AS block_size
FROM pg_stat_bgwriter_snapshot one
  INNER JOIN pg_stat_bgwriter_snapshot two
    ON two.now > one.now
) bgwriter_diff
WHERE (checkpoints_timed + checkpoints_req) > 0;
```

Below is what the output from this would look like during a relatively busy period. This is from a laptop that was running the standard `pgbench` test for a bit during the period monitored:

```
start                   | 2010-04-09 19:52:00
elapsed                 | 00:17:54
avg_checkpoint_interval | 00:03:34
checkpoints_req_pct     | 80
checkpoint_write_pct    | 85
backend_write_pct       | 14
avg_checkpoint_write    | 17 MB
written_per_sec         | 94 kB
alloc_per_sec           | 13 kB
```

Here 80% of the checkpoints were required, because the system ran out of WAL segments, and that matches an average time between checkpoints of around 3.5 minutes -pretty frequent. Luckily, most buffers written out - 85% - were written by checkpoints, which is good because a checkpoint write is the most efficient type to have. Each checkpoint averages 17 MB written out. But if you sum up all of the write activity, it's only streaming an average of 94 kB/s out to disk, which is still a pretty low amount. On the read side, 13 kB/s of buffers were allocated to satisfy queries. That balance suggests that for the most part, the majority of buffer churn was things persistently saved in cache being repeatedly written out to disk, with little corresponding reading going on. That's typical for the standard `pgbench` test used to generate the load here, because it's always dirtying the same blocks in the tellers and branches tables in particular.

A snapshot for the 46 minutes following this, which didn't have any test running, shows what a mostly idle period looks like:

```
start                   | 2010-04-09 20:10:00
elapsed                 | 00:46:26
avg_checkpoint_interval | 00:05:09
checkpoints_req_pct     | 0
checkpoint_write_pct    | 100
backend_write_pct       | 0
avg_checkpoint_write    | 910 bytes
written_per_sec         | 3 bytes
alloc_per_sec           | 0 bytes
```

Note how the checkpoint interval matches the default `checkpoint_timeout` of 5 minutes, with none of the checkpoints being required - they're all time based ones. And the average read and allocation rates are slowed to almost nothing.

The `alloc_per_second` rate is not exactly the same as the read rate from the OS buffer cache, but it is an interesting number suggesting how often internal buffers are being re-used you can combine a look at actual OS reads with. The `written_per_sec` number is a complete average for writes to the database tables and indexes on disk. That should very closely match averages computed with OS level tools. You do need to recall that other sources for database writes include the WAL, the transaction commit logs, and temporary files used for sorting. This lets you quantify one component of that mixed write set accurately.

Tuning using background writer statistics

Tuning database parameters related to buffering and checkpoints in PostgreSQL is often considered more magic than science. With regular snapshots of the background writer statistics, it's possible to bring a more formal iterative method to tuning adjustments.

1. Start collecting regular `pg_stat_bgwriter` snapshots so you have a performance baseline. Every time you make a change, make sure to generate a new snapshot point after starting the server with the new values:

```
pgbench=# \d pg_stat_bgwriter;
              View "pg_catalog.pg_stat_bgwriter"
        Column         |           Type           | Modifiers
-----------------------+--------------------------+-----------
 checkpoints_timed     | bigint                   |
 checkpoints_req       | bigint                   |
 checkpoint_write_time | double precision         |
 checkpoint_sync_time  | double precision         |
 buffers_checkpoint    | bigint                   |
 buffers_clean         | bigint                   |
 maxwritten_clean      | bigint                   |
 buffers_backend       | bigint                   |
 buffers_backend_fsync | bigint                   |
 buffers_alloc         | bigint                   |
 stats_reset           | timestamp with time zone |
```

2. There will come points where checkpoints occur first, where `checkpoint_segment` worth of files are written or `checkpoint_timeout` occurs. In that case, increase `checkpoint_segments` until most checkpoints are time driven, rather than requested because the segment threshold has been crossed. Eventually, 90% or more should be time based, and the `avg_checkpoint_interval` figure should be close to 5 minutes (or whatever you've set `checkpoint_timeout` to).

In PostgreSQL 9.5 `checkpoint_segments` are replaced with `min_wal_size` and `max_wal_size`. These values can be calculated by this formula:

```
max_wal_size = (3 * checkpoint_segments) * 16 MB
```

3. Increase `shared_buffers` by a large amount - 25% or more. If your current value is less than 1/6 of total RAM in a dedicated server, try a value of at least that amount. Compare a snapshot after that change with one from before.
4. When your system is using the larger buffer space usefully, the percentage of buffers written at checkpoint time (rather than by backends or the background writer) should increase, and the total I/O `written_per_sec` figure should drop. If the last increase in `shared_buffers` is helped by those standards, return to step 3, continuing to iterate with size increases until improvements level off.
5. If the values didn't change significantly, your original `shared_buffers` value was already big enough for this workload. You can either keep the new configuration or revert to the older one; in either case, you're done with increasing its size.
6. You should now have `checkpoint_segments` and `shared_buffers` set to useful sizes. If a large percentage of writes are coming from `buffers_checkpoint`, and the average I/O figure shown in `written_per_sec` seems high (or the system is obviously slow with excess writes), consider increasing `checkpoint_timeout` to further spread the checkpoints out. If you do that, start over again-you probably have to revisit `checkpoint_segments` again:

```
postgres=# SELECT * FROM pg_stat_get_snapshot_timestamp();
pg_stat_get_snapshot_timestamp
-------------------------------
2016-03-31 17:38:23.704994+05
(1 row)
```

In PostgreSQL 9.5 another function is added which gives the timestamp of the snapshot.

The key to this method is to recognize that any change that shifts your system toward doing more writes as part of a widely-spread checkpoint is an improvement over writing the same data from a backend or the background writer. Your buffer cache should be filled with data that stays around because it's used regularly, accumulating a high usage count. You might even observe that using `pg_buffercache`, although that's not essential to this tuning technique. The ideal situation is one where anything that's modified regularly is only written out once per checkpoint, not each time it's dirtied. That's the most efficient approach, and improvements in that direction should correspondingly lower total writes too.

There is one main downside to getting too aggressive about increasing `shared_buffers` and the checkpoint parameters. When checkpoints end, they have to force all the writes done during the checkpoint out to disk. In current PostgreSQL versions, this is done as a fairly tight operation. If the operating system didn't actually write those buffers out in advance, you can cause a checkpoint I/O spike proportional to how much data the OS has cached when the checkpoint ends. That is known to be a problem on Linux systems with large amounts of memory; they can get stuck having gigabytes to synchronize to disk. If you run into this problem, you may have to tune in the opposite direction - less checkpoint writes, more backend and background writer ones - in order to reduce these spikes, and the application side latency they introduce. I have some improvements to this area planned for PostgreSQL 9.1, initially aimed at helping Linux systems using the XFS and ext4 file systems sync writes out more smoothly. It's impossible to do a good job here with ext3, because of limitations in how that filesystem handles cache synchronization, since it will just write out the whole file system cache once the first sync request arrives.

If you're going to try and tune your system as described in this section, make sure you turn on `log_checkpoints`, which is off by default, and monitor what that's writing to the database logs too. If you see the "sync" time listed at the end of the checkpoint rise, you may be running into this problem. If so, reduce the buffer cache and/or make checkpoints more frequent to reduce its impact, even if that makes the buffer cache statistics worse.

Summary

It seems fitting that a database filled with data can also produce its own set of data, via monitoring its internal activity and statistics. While in many cases it's more appropriate to setup monitoring using pre-built external tools, as covered in the next chapter, knowing how to directly query and derive interesting values from the database's activity and statistics collector can prove valuable. And when it comes to troubleshooting system performance issues, monitoring activity and lock data yourself is a must.

The database statistics are exposed via views you can either use all of, or reinterpret by joining with additional system catalog and activity data. Most statistics are cumulative values. You'll either need to track how they change over time with an external tool, or regularly reset the totals. Particularly valuable statistics to monitor include table/index caching and query index usage statistics.

Live activity snapshots can be used on their own, or combined with other process-based monitoring, to correlate what's happening inside the database with the rest of the system. The locking data provided by the database is extensive but very low-level. You'll need to write your own queries to make sense of what's provided, to search for locking issues in database applications. Tracking disk space used is valuable for noting how tables change over time, both for finding disk usage problems and for trending predictions. The statistics provided by the background writer can be used to tune critical checkpoint and buffer cache parameters in a systematic way.

The next chapter will cover the monitoring of database and the operating system. Monitoring the hardware and operating system plays a key role in database performance.

12
Monitoring and Trending

The performance of your database server is directly tied to how well the underlying operating system is working; and performance there is driven by the hardware you're using. To fit all these pieces together - hardware performance, operating system performance, and database performance - you need a good monitoring system. Once you're capturing all the right data, software that graphs the monitoring system is vital to tracking general trends in your servers It can help you predict when you're reaching the capacity limits of your system, and see whether changes made are effective improvements.

Topics we will cover in this chapter are:

- Unix monitoring tools
- Monitoring tools compatible with PostgreSQL

UNIX monitoring tools

The simple performance tools on a UNIX-derived system are straightforward to use. It's easy to show examples of good and bad behavior, and that's the best way to teach how those tools are useful for monitoring. Note that the general background here and the examples of what to look for are still relevant even on a Windows system. The underlying hardware, the way operating systems work, and the resulting performance concepts are no different. A table later section in this chapter shows how to translate between UNIX and Windows monitoring terminology.

Sample setup

The server used here is the same one described in the `pgbench` chapter. For these examples, initially a small `pgbench` database was created with a scale of 100 (one that easily fits into RAM), and the standard mixed test was run:

```
$ createdb pgbench
$ pgbench -i -s 100 pgbench
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
creating tables...
100000 of 10000000 tuples (1%) done (elapsed 0.08 s,
  remaining 8.10 s)
...
...
...
10000000 of 10000000 tuples (100%) done (elapsed 13.68 s,
  remaining 0.00 s)
vacuum...
set primary keys...
done.

$ pgbench -j 4 -c 8 -T 300 pgbench
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 100
query mode: simple
number of clients: 8
number of threads: 4
duration: 300 s
number of transactions actually processed: 1717215
latency average = 1.398 ms
tps = 5723.945824 (including connections establishing)
tps = 5724.085890 (excluding connections establishing)
```

This provided approximately `5,723` transactions per second. Larger tests with lower TPS values will appear as well.

If you're using a PostgreSQL earlier than 9.0, you'll have to leave out the -j 4 part of the preceding code to try this yourself here, and with the `pgbench` examples shown later. Versions before 8.4 won't handle `-T 300` either, and you'll find to find a substitute number of transactions to pass via `-t` in order to make the test last for a while.

The sample server has the following as its disk layout for the database directories:

- `pg_xlog`: `/dev/sdf1`
- `data`: `/dev/md0`. Linux software RAID0; individual drives are `sdc1`, `sdd1`, `sde1`

You'll need this disk layout background to run the following `iostat` examples:

vmstat

If you post a question to the pgsql-performance mailing list that suggests your system might be overloaded, the first thing you'll be asked for is a snapshot of `vmstat` data. It's the most valuable quick summary of what your system is doing. Because it displays a full system snapshot per line, it's even possible to extract short-term trends from staring at a screen full of data.

Since the output from `vmstat` is a bit too wide to fit on the page at once, we've broken it up into a left and right side for now; later examples will include just the interesting columns. Here's the left side showing a few seconds of heavy memory-limited `pgbench` work:

```
$ vmstat 1
procs -----------memory------------ ---swap--
r  b   swpd   free    buff   cache   si   so
8  0      0 2542248 386604 3999148    0    0
3  0      0 2517448 386668 4023252    0    0
1  0      0 2494880 386732 4043064    0    0
7  1      0 2476404 386792 4060776    0    0
```

The explanations for these columns in the manual for `vmstat` are:

r: The number of processes waiting for runtime.
b: The number of processes in uninterruptible sleep.
swpd: The amount of virtual memory used.
free: The amount of idle memory.
buff: The amount of memory used as buffers.
cache: The amount of memory used as cache.
si: Amount of memory swapped in from disk (/s).
so: Amount of memory swapped to disk (/s).

Next, you'll see some examples of how to interpret the `procs` data, and what it looks like when the server runs low on RAM. One thing not shown here is what happens when the server starts using swap. On a database server, if you're using swap at all, you've probably made a configuration error, and should reduce memory usage. Therefore, the main thing to monitor the swap figures for is that any value other than zero for `si` or `so` is a likely problem. On Linux, the swappiness setting (covered in `Chapter 4`, *Disk Setup*) can have a major impact on how this works.

The part of the `vmstat` data that's much more interesting for database performance is that on the right side; these are the other half of the preceding four lines:

```
$ vmstat 1
----io---- --system-- -----cpu------
bi    bo    in    cs us sy id wa st
24 38024  7975 73394 40 18 34  7  0
48 57652 11701 93110 43 16 34  6  0
36 75932 11936 86932 44 15 34  7  0
4 96628 12423 77317 39 17 37  6  0
```

Here's what the manual has to say about these:

bi: Blocks received from a block device (blocks/s).
bo: Blocks sent to a block device (blocks/s).
in: The number of interrupts per second, including the clock.
cs: The number of context switches per second.
us: CPU Time spent running non-kernel code. (user time, including nice time)
sy: CPU Time spent running kernel code. (system time)
id: CPU Time spent idle.
wa: CPU Time spent waiting for IO.
st: CPU Time stolen from a virtual machine.

The various "CPU Time" figures are all given in percentages. By default, the Linux `vmstat` being used here is counting blocks in units of 1,024 bytes, which means that the numbers given are in KB/s. Therefore the first `bo` figure, 38,024, means approximately 38 MB/s of disk writes happened during that time. This may not be true on non-Linux systems; see the following `iostat` section for more background about block sizes.

All of the `vmstat` examples here are produced using a one second time interval, the parameter passed on the command line in the preceding examples. All of the counts in its data (as opposed to the percentages) are averages per second over the given time period, so the interpretation isn't impacted by the collection period. It just changes the resolution of the data you see.

The other thing to note about `vmstat` and `iostat` is that when you run them, the first line they output is a long-term one summarizing all activity since the server was started. The snapshots of a small unit of time start on the second line printed. If you're writing scripts to collect this data and process it, typically you'll need to be careful to always throw away the first line.

As a first example of what bad data looks like, here's a snapshot from the preceding `pgbench` run showing a period where the system became less responsive for about two seconds:

```
procs ----io---- --system--- -----cpu------
r  b   bi    bo    in    cs us sy id wa st
2  2    4 93448 11747 84051 44 19 32  5  0
0  3    0 54156  8888 47518 23 10 53 14  0
0  2    0  6944  1259  1322  1  0 72 27  0
0  2    0 12168  2025  2422  0  0 65 35  0
8  0    0 26916  5090 41152 23  9 47 21  0
2  0    4 57960  9802 54723 31 12 46 11  0
```

Note the dramatic drop in **context switches** (**cs**) for the middle two entries there. Since most completed work executed by the server and the `pgbench` client itself involves a context switch, those low entries represent a period where almost nothing happened. Instead of tens of thousands of things happening during that second, there were only a few thousand. Also note how that corresponds with a jump in the waiting for the I/O (`wa`) category, and the CPUs becoming less active. All these things are characteristics of what a bad performing section of time looks like, when the system is at a bottleneck waiting for the disk drive(s).

iostat

The data `vmstat` gives is total across all devices on the system. If you want totals per disk device instead, you need to use `iostat` for that.

On Linux, `iostat` defaults to slightly different behavior than `vmstat`. When it uses "block", it means a 512 byte chunk of data, not the 1,024 byte chunks `vmstat` uses. You can switch `iostat` to using kilobytes instead with `iostat -k`, or you can just divide all the figures by two in order to get them on the same scale. Here's an example of the same data shown both ways:

```
$ iostat
Device     tps Blk_read/s  Blk_wrtn/s  Blk_read  Blk_wrtn
sda1      0.07       3.29        0.24   1579784    115560
$ iostat -k
Device     tps  kB_read/s   kB_wrtn/s   kB_read   kB_wrtn
sda1      0.07       1.64        0.12    789892     57780
```

Since not all UNIX versions will have the kilobyte option available, the examples here all use the default 512 byte blocks, so just halve the block figures to interpret via kilobyte units.

You'll likely find that you need to average `iostat` data over a slightly longer period of time than `vmstat` data. A single second of `vmstat` data is a summary of all the disks on the system. A PostgreSQL database goes through several common phases:

- Just after a checkpoint:
- Heavy full-page writes to WAL, less writes to database disks because there are less dirty buffer evictions
- Between checkpoints:
- (Previous one finished and the most commonly used buffers have had full pages written): Most equal mix of WAL and database writes
- Checkpoint in progress:
- Small to moderate WAL writes, with increasingly heavy database writes as checkpoint data is written and starts flowing to disk
- Checkpoint sync phase:
- Minimal WAL writes because fewer full page writes are generally happening, heavy writes to database disks as all data is flushed out of the OS cache

If you are looking at the `vmstat` data, or if you don't have the `pg_xlog` WAL data broken out onto a separate disk, you can't see the balance of the data `vs`. WAL writes change; you just see a total. If you're grabbing really short `iostat` snapshots, you're likely to see writes bounce between the WAL and database disks, with the exact pattern depending on where in the checkpoint cycle you're at. You need to combine a few seconds of data (five seconds is used for these examples) in order to have both types of writes usefully averaged out:

```
$ iostat 5
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          42.69    0.00   18.07    6.69    0.30   32.25

Device     tps Blk_read/s  Blk_wrtn/s  Blk_read  Blk_wrtn
sda       0.00        0.00        0.00         0         0
sda1      0.00        0.00        0.00         0         0
sdc      80.80        0.00     1286.40         0      6432
sdc1     80.80        0.00     1286.40         0      6432
sdd      77.80        0.00     1251.20         0      6256
sdd1     77.80        0.00     1251.20         0      6256
sde      69.40        0.00     1086.40         0      5432
sde1     69.40        0.00     1086.40         0      5432
sdf    2348.20        0.00    88262.40         0    441312
sdf1   2348.20        0.00    88262.40         0    441312
md0     311.40        0.00     2491.20         0     12456
```

You can see that much of the valuable information from `vmstat`, such as the CPU statistics, also appear here. However, with so much more data here, it's much harder to track trends on the console with this tool than with `vmstat`.

Since all of the activity relates to the single partition on these disks, there's a lot of redundant data in here. You should also note that many of the statistics for the software RAID volume used here are not very interesting - you have to look at the underlying physical disk devices instead. If you're using hardware RAID, that particular problem will go away, but you won't have any easy way to get actual disk performance information out of that abstraction layer either; you'll just see the summary for the whole logical RAID device. The following examples eliminate all the redundant lines, and place the `md0` array device between its individual components and the device the WAL is on (`sdf1`), for easier readability.

iotop for Linux

It's also possible on Linux to get per-process I/O statistics, so you can see exactly who is reading and writing heavily, using a program named `iotop`. This data isn't available in mainstream Linux until kernel 2.6.20, and the `iotop` program requires Python 2.5 too. RedHat RHEL5 doesn't meet either requirement. Red Hat has been working on getting the I/O statistics backported to their mainstream 2.6.18 kernel and working through the Python issues too. By RHEL 5.7, it may be fully available; see `https://bugzilla.redhat.com/show_bug.cgi?id=557062` to track their progress.

If you're running on a system with a more recent kernel, as may be the case for Debian or Ubuntu users, make sure to try `iotop` out once you've seen heavy I/O wait on a system. It's extremely useful to determine where the reads and writes causing that are coming from.

It's possible to collect similar statistics from older kernels, a topic introduced at `http://www.xaprb.com/blog/2009/08/23/how-to-find-per-process-io-statistics-on-linux/`, and the `blktrace` utility can be used to profile disk I/O as well. Both these are more complicated tools to use than the simple `iotop` program.

Examples of good performance

When busy but not overloaded, `iostat` data for this system looks as follows:

```
$ iostat 5
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          18.54    0.00    9.45   23.49    0.15   48.38

Device     tps Blk_read/s  Blk_wrtn/s  Blk_read  Blk_wrtn
sdc1   1068.80       0.00    15740.80         0     78704
sdd1   1041.80       0.00    15459.20         0     77296
sde1   1028.00       0.00    15377.60         0     76888
md0    5969.20       0.00    47753.60         0    238768
sdf1    989.00       0.00    40449.60         0    202248
```

The `%iowait` figure of 23% is high enough to know the disks are busy, but not completely saturated yet. This is showing 20 MB/s (40449.6 512-byte blocks per second) being written to the WAL and 24 MB/s to the entire database disk array, the latter of which is evenly split as almost 8 MB/s to each of the three drives.

Linux also features an extended `iostat` mode. This produces a large number of statistics derived from the underlying data. Since that's much too wide to display here, the first example showing all of the data here has been transposed to swap the row for columns and vice-versa:

```
$ iostat -x 5
            sdc1     sdd1     sde1      md0     sdf1
rrqm/s         0        0        0        0        0
wrqm/s     411.8    404.6    396.2        0   3975.4
r/s            0        0        0        0        0
w/s        438.6      442    444.2   2461.4   1229.8
rsec/s         0        0        0        0        0
wsec/s    6956.8   6966.4   6915.2 19691.2 41643.2
avgrq-sz   15.86    15.76    15.57        8    33.86
avgqu-sz   67.36    67.09    62.93        0     0.65
await     158.18   158.85   148.39        0     0.55
svctm        1.2      1.2     1.19        0     0.51
%util       52.8    52.88    53.04        0    63.04
```

All of the values here with a "q" in them (most of what's listed on the following bulleted line) represent figures related to the read or write queues on these devices. Since the queue size doesn't correspond with any real-world figure you can benchmark the device against, it's hard to do anything with that data. The number of read and write requests is similarly useless in a database context. The following fields of `iostat -x` data are therefore not that useful here:

```
rrqm/s wrqm/s r/s w/s avgrq-sz avgqu-sz
```

And this won't be discussed in detail. Trimming some of those out also lets the samples fit onto the horizontal space available.

Solaris has a similar extended mode available via `iostat -xc`.

This next example is similar to the `iostat` one given previously:

```
$ iostat -x 5
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          21.51    0.00   11.08   23.75    0.10   43.56

Device rsec/s   wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sdc1     0.00  6956.80    15.86    67.36  158.18   1.20  52.80
sdd1     0.00  6966.40    15.76    67.09  158.85   1.20  52.88
sde1     0.00  6915.20    15.57    62.93  148.39   1.19  53.04
```

```
md0       0.00 19691.20     8.00     0.00    0.00   0.00   0.00
sdf       0.00 41643.20    33.86     0.65    0.55   0.51  63.04
```

That's 21 MB/s written to the WAL and 20 MB/s to the database disks, about 7 MB/s to each one. However, recall that the total disk read or write throughput available depends heavily on how random the workload is, which is normally a hard thing to estimate. The `%util` figure, by far the most valuable of the derived figures shown here, gives you a rough idea of that by noting how congested the device is to achieve that throughput. In this following example, there's minimal database I/O and heavy WAL I/O, typical of the period just after a checkpoint:

```
$ iostat -x 5
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          49.35    0.00   22.00    3.80    0.25   24.60

Device rsec/s    wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sdc1     0.00   2649.10    15.01     0.76    4.31   0.06   1.04
sdd1     0.00   2895.01    14.95     0.90    4.64   0.06   1.12
sde1     0.00   2728.94    15.06     0.82    4.51   0.06   1.04
md0      0.00   8273.05     8.00     0.00    0.00   0.00   0.00
sdf1     0.00 103760.48    38.11     0.23    0.09   0.09  23.47
```

This is happily getting greater than 50 MB/s out of the WAL volume, but it's still only busy 23.5% of the time. That suggests writes to it are being cached by the disk controller and written quite efficiently. One of the reasons to break out the WAL onto its own disk is because it makes it so easy to monitor this balance between WAL and database writes, and to determine if the WAL volume (which only gets sequential writes normally) is keeping up. Since there are techniques to accelerate the WAL writes at the expense of something else, such as switching to an unjournaled filesystem, the `%util` figure can help you determine when the WAL is the system bottleneck and therefore necessary to accelerate that way.

A final example of good performance involves the database disks. There are some operations in PostgreSQL that can bypass writing to the WAL. For example, if you start a transaction that creates a new table and does a `COPY` into it, as long as you don't have PITR archiving turned on, that data is not put through the WAL before being written to disk. The idea is that if the server crashes, the whole transaction will be rolled back anyway, which includes deleting the table data; therefore, whether it's consistent or not at the block level, it doesn't matter.

Here is what the database disks are capable of when running such a `COPY`, which essentially turns into sequential write I/O directly to the database:

```
$ iostat -x 5
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
```

```
           16.39    0.00    6.85   12.84    0.00   63.92

Device rsec/s    wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sdc1    25.60  58710.40   249.09    27.22  115.43   1.19  28.08
sdd1    24.00  58716.80   249.11    27.76  117.71   1.20  28.24
sde1     1.60  58667.20   250.51    28.31  120.87   1.14  26.80
md0     51.20 176094.40     8.00     0.00    0.00   0.00   0.00
sdf1     0.00      0.00     0.00     0.00    0.00   0.00   0.00
```

That's over 29 MB/s being written to each database disk, for a total of 88 MB/s to the RAID0 array. And even that isn't fully utilizing the disks, as shown by the `%util` at about 28%. Given that this is a four-core server and `COPY` is the only process running, a `%user` of 16 means that about 64% of a single CPU is busy here. The CPU and disks are likely each waiting on each other in this situation, and you might have to improve both to significantly speed this up. This example is from a server with a battery-backed RAID controller; without one, it's much easier to run into one of the disk bottlenecks here before the CPU ones.

Final `iostat` hint: on some versions you can switch the output to use megabytes/second as its units, which is often the easiest to read. This syntax:

```
$ iostat -x -m 5
```

Usually makes for a good summary on Linux systems, for example.

Overloaded system samples

To get a more realistic workload, the next few samples use a much larger scale of the `pgbench` database (1000) and more clients (64):

```
$ pgbench -i -s 1000 pgbench
$ pgbench -j 4 -c 64 -T 300 pgbench
```

This gives about 400 TPS on this server. The following snapshot shows one type of problem that you can discover from the `vmstat` data:

```
$ vmstat 1
procs ----io---- --system--- -----cpu------
r  b   bi    bo    in    cs us sy id wa st
3 62 4916 34296 11504 23871  8  6  0 85  0
2 64 7132 35828 13247 32406 11  8  0 81  0
4 63 6120 40756 11313 29722 35  7  0 58  0
0 48 3896 13712  6953 19544 28  3  2 66  1
0 12  400 25564  2222  3417  0  1 25 73  0
0  5   44  3700   818  1020  0  0 39 61  0
1 12   64  8364  1388  1773  6  0 44 50  0
```

```
1 45  704  7204  2133  3883 23  1  3 73  0
5 60 2912 26148  8677 15774 17  3  1 79  0
0 62 2376 15900  5648 12084  3  2  0 95  0
```

As mentioned previously, the spots where the `cs` figures drop dramatically (while the system was under heavy load) represent a drop in total system throughput. This example is a bit different, because the `wa` actually drops too--the system is so overloaded that it isn't even generating a full sized write load. This is typical of when the server is so overloaded that it stops even servicing client work, typically because of lock contention. You can see that from how all the user time has disappeared too. Also, when in a functional state, you can see most of the 64 clients (as well as some other database and system processes) in either the running or sleeping category. During the worst `cs` entry here, a mere five of those client processes (the `b` column) got any run time on the server during the one second interval. This profile is common when the cache on a disk controller has completely filled up, and clients are all stuck waiting for WAL data to flush to disk.

Sometimes the slowdown is purely I/O though, as in this example:

```
$ iostat -x 5
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           5.21    0.00    2.80   91.74    0.25    0.00

Device rsec/s    wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sdc1  2625.60   2057.60    14.67    38.79  101.47   3.14 100.08
sdd1  2614.40   2000.00    14.90    57.96  162.95   3.23 100.08
sde1  2736.00   1963.20    14.64    48.50  135.26   3.12 100.00
md0   7916.80   7206.40    10.60     0.00    0.00   0.00   0.00
sdf1     0.00  22190.40    50.20     0.84    1.79   1.34  59.20
```

The first thing to notice here is that the `%util` figure is rounded badly so it looks like over 100% in spots. That's a result of some sloppy accounting and computation, and is not anything to be worried about. Next, note that the WAL is only getting 11 MB/s of writes to it. Meanwhile, the database disks are 100% utilized, but are actually processing under 8 MB/s in total (an even mix of reads and writes). This is what it looks like when the database has heavy random I/O. These underlying disks are only capable of about 2 MB/s of true random I/O, and sure enough they aren't doing much better than that even with a caching controller sitting in the middle to buffer and sort. This is typical of the checkpoint sync phase, where a large amount of random I/O has been pushed into the write cache and is now being forced out to disk.

To show some really unpleasant performance, now let's crank the scale up to 4,000:

```
$ pgbench -i -s 4000 pgbench
$ pgbench -j 4 -c 64 -T 300 pgbench
```

This managed 250 TPS, but with very intermittent processing. A particularly bad period looks as follows:

```
$ vmstat 1
procs ----io---- --system--- -----cpu-------
r  b   bi    bo    in    cs us sy id  wa st
1 63 5444  9864  8576 18268  4  3  0  92  0
0 42 4784 13916  7969 16716  4  2  3  90  0
0 59  464  4704  1279  1695  0  0 25  75  0
0 54  304  4736  1396  2147  0  0 25  74  0
0 42  264  5000  1391  1796  0  0 10  90  0
0 42  296  4756  1444  1996  1  0 10  89  0
0 29  248  5112  1331  1720  0  0 25  75  0
0 47  368  4696  1359  2113  0  0 23  76  0
1 48  344  5352  1287  1875  0  0  0 100  0
0 64 2692 12084  5604  9474  8  2  0  90  0
1 63 5376  9944  8346 18003  4  3 20  74  0
0 64 5404 10128  8890 18847  4  3 25  67  0
```

That's an eight second long period of seriously degraded performance. Note the low `cs` counts, that the total `procs` figures drop below the number of clients, and that there's no user time. Those should be familiar characteristics of all the clients getting stuck waiting for something by now. Here the cause is fairly obvious: `wa` is high the whole time and even hits a full 100%, showing the server just can't keep up with the disk I/O load here.

You might wonder what that disk load looks like through the extended `iostat` data; here's a similar period:

```
$ iostat -x 5
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           2.35    0.00    1.85   85.49    0.15   10.16

Device rsec/s   wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sdc1  2310.40  1169.60    14.63    39.85  147.59   4.21 100.00
sdd1  2326.40  1216.00    14.53    71.41  264.60   4.10 100.00
sde1  2438.40  1230.40    14.77    47.88  157.12   4.01  99.60
md0   7044.80  4820.80    11.12     0.00    0.00   0.00   0.00
sdf1     0.00 19040.00    70.31     4.20   15.31   2.10  56.88
```

Note how low the read/write rates are here, while still having 100% utilization. This shows another I/O load heavy on random reads and writes. Even without the Linux specific `iostat -x` data, you could also tell that from the combination of extremely high `%iowait` with low throughput:

```
$ iostat 5
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.35    0.00    0.55   88.99    0.10    9.02
```

```
Device     tps Blk_read/s  Blk_wrtn/s  Blk_read  Blk_wrtn
sdc1    252.80    1236.80     2550.40      6184     12752
sdd1    268.80    1377.60     2611.20      6888     13056
sde1    264.40    1312.00     2622.40      6560     13112
md0    1234.20    3924.80     7784.00     19624     38920
sdf1    118.00       0.00     6380.80         0     31904
```

We know that on sequential I/O these disks are capable of much higher throughputs than this. So when the server is at a `%iowait` of 89%, but is only managing to write less than 4 MB/s to the database disks, it's sensible to conclude that they are coping with a mix of random read/write requests instead.

top

If you want a snapshot of what your server is actually doing right now, from the operating system's perspective, `top` is an easy tool to run. By default, it will sort the active processes by their CPU usage, and it's easy to sort other ways as well.

When looking at `top`'s memory accounting (which is similar to what `ps` uses as well), you'll find three memory totals given. Both the `VIRT` and `SHR` figures include shared memory, and accordingly the total memory percentage shown for a PostgreSQL processes is probably inflated by some amount. Basically, any memory touched by the client backend from shared memory will get added to its total. The `RES` figure is the more useful one to monitor for regular client processes.

The most useful `top` trick to know is the fact that PostgreSQL processes update their process information based on what they're doing, essentially modifying what command line they appear to have been started with. Whether this happens or not depends on the `update_process_title` configuration parameter in the database. Both `top` and `ps` are capable of displaying that information. On Linux, you can see it by hitting the C key while `top` is running, or by running `top -c`. On FreeBSD `top -a` does the same thing.

It's also possible to run `top` in a batch mode, where it writes data for every single process on the system out at an interval rather than just showing a screen full of them. Returning to our example with a `pgbench` scale of 4,000 and 64 clients, here's how you might capture 10 seconds worth of `top` data at 1 second intervals to a file, find all the `postgres` backends, and then pick out the first word of what they're doing:

```
$ top -c -b -d 1 -n 10 | tee topdata
$ cat topdata | grep "postgres: postgres" | cut -f 25 -d " "
$ cat topdata | grep "postgres: postgres" | cut -f 25 -d " " |
 wc -l
 640
```

The second line there (output not shown) gets you `COMMIT`, `UPDATE`, or `idle` for each line. There's no `SELECT`s in here just because those happen so fast it's hard to catch any of them, which won't be the case for most other workloads.

With 64 clients and 10 snapshots the expected number of 640 lines are there. You can then count by type:

```
$ cat topdata | grep "postgres: postgres" | cut -f 25 -d " " |
  grep "COMMIT" | wc -l
179
$ cat topdata | grep "postgres: postgres" | cut -f 25 -d " " |
  grep "UPDATE" | wc -l
459
$ cat topdata | grep "postgres: postgres" | cut -f 25 -d " " |
  grep "idle" | wc -l
2
$ cat topdata | grep "postgres: postgres" | cut -f 25 -d " " |
  grep "SELECT" | wc -l
0
```

Now we've learned something interesting that didn't require any database level investigation, just by using `top`: at any particular point in time, this sample snapshot of workload had 28% `COMMIT` and 72% `UPDATE` statements executing. This isn't a substitute for a full query profiler, but when you combine it with the fact that you get process-level CPU and memory information it's a valuable alternative source of information.

Solaris top replacements

Solaris doesn't ship with `top` or with a version of `ps` that shows updated process titles in the default `PATH`. You can use `/usr/ucb/ps` and `/usr/ucb/top` to get versions that are more likely to display this information. In general, getting the process titles to show under Solaris is more difficult, and there are some restrictions on when it happens. There are some additional notes on this subject in the *Standard Unix Tools* section of `http://www.postgresql.org/docs/current/interactive/monitoring.html`.

The standard Solaris monitoring tool recommended instead of `top` is `prstat`. That utility is fine for monitoring overall system activity, but without being able to display the process title updates it's much less useful than a fully functional `top` for monitoring PostgreSQL.

htop for Linux

One entry in the "build a better `top`" category is a program named `htop` (another is `atop`). It's usually available as a package on modern Linux systems. In addition to niceties such as color, two of its features make it a compelling improvement over regular `top` for some users. The first is that it shows CPU usage broken down by individual CPU. This makes it much easier to distinguish the situation where a single process is using all of a CPU from when many processes are using a portion of one. `htop` also displays processes in a hierarchy tree format, based on which they spawned. This is a particularly useful way to display the PostgreSQL processes.

SysStat and sar

After looking at how useful `vmstat` and `iostat` data is, you might be wondering how you can capture it all of the time. The standard UNIX system package for this purpose is named `sysstat`, and it's not installed or activated on many systems by default. On some systems, you have to install this package just to get `iostat`. The easiest way to test out if you have `sysstat` installed and working is to run its user interface, the `sar` program, and see if anything comes back. If not, you will either have to install `sysstat` (if `sar` isn't even there), or enable it (if it's there but has no data).

Once it's running and collecting data, `sysstat` collects up data quite regularly, and gives you a broad perspective of average server performance during each day. The following example comes from the period when all of the benchmark examples described in this chapter were run:

```
$ sar
01:20:02 PM CPU %user %nice %system %iowait %steal  %idle
01:20:02 PM all  0.02  0.00    0.03    0.11   0.01  99.84
01:30:01 PM all 12.18  0.00    5.36   10.74   0.07  71.64
01:40:01 PM all 13.39  0.00    5.71    9.59   0.09  71.22
01:50:01 PM all  1.29  0.00    0.59    0.30   0.01  97.82
02:00:02 PM all 11.22  0.00    5.36   13.68   0.01  69.72
02:10:01 PM all  2.44  0.09    1.59   47.16   0.09  48.62
02:20:01 PM all  5.89  0.00    2.61    4.16   0.01  87.32
02:30:01 PM all 16.09  0.00    6.85   16.58   0.01  60.46
02:40:02 PM all 13.10  0.00    4.52   10.78   0.01  71.58
02:50:02 PM all 11.75  0.00    4.39    9.57   0.01  74.28
03:00:02 PM all  3.48  0.00    6.42   16.63   0.02  73.46
03:10:01 PM all  0.59  0.09    1.19   18.78   0.02  79.34
03:20:01 PM all  1.16  0.00    0.74   40.23   0.08  57.79
03:30:01 PM all  1.92  0.00    1.28   61.33   0.11  35.37
03:40:01 PM all  0.46  0.00    0.38   18.71   0.03  80.41
```

```
03:50:01 PM all  0.01  0.00    0.00    0.00   0.01  99.97
```

You can easily see that the disk subsystem was getting hammered around 3:30 PM, with an average `%iowait` of 61%. When averaged over a full 10 minutes, having an average that high means the server was seriously busy. That period corresponded with the tests on the `pgbench` database with a scale of 4,000, and the time just before that includes building the database and its indexes.

One thing `sar` is quite well suited for is monitoring the memory usage on your server. Here's a sample of its memory report, showing just the left side of the result-the right side shows swap use, which in this case was zero:

```
$ sar -r
01:40:01 PM kbmemfree kbmemused  %memused kbbuffers  kbcached
01:40:01 PM   1726444   6077460     77.88    390696   4831784
01:50:01 PM    420440   7383464     94.61    394620   6093328
02:00:02 PM     14948   7788956     99.81      6388   6971368
02:10:01 PM    375372   7428532     95.19     22916   6603856
02:20:01 PM     16512   7787392     99.79      6788   6989596
02:30:01 PM     14060   7789844     99.82      6972   6999284
02:40:02 PM     15140   7788764     99.81      6604   6980028
02:50:02 PM     16008   7787896     99.79      9960   6982620
03:00:02 PM     15380   7788524     99.80      6176   6997812
03:10:01 PM     14172   7789732     99.82     64312   6866432
03:20:01 PM    352876   7451028     95.48     68824   6646456
03:30:01 PM     27452   7776452     99.65     72428   6600724
03:40:01 PM    356212   7447692     95.44     74944   6552036
```

This shows you clearly that the 8 GB of RAM on the server was under serious memory pressure starting after 1:40 PM. The total RAM free shown by `kbmemfree` went from 1.7 GB to 15 MB in 20 minutes. One of the things Linux does in this situation is shred its buffer cache, which you can certainly see as it shrinks `kbbuffers` from 394 MB to 6 MB in 10 minutes. This also captures one of the known Linux oddities, which is that once the buffer cache has been reduced like that, it's sometimes slow to recover. Eventually, it reallocates to bring that cache back to the 70 MB range, but growth there is gradual. Even 12 hours later, the cache on this system was still at only 119 MB.

Enabling sysstat and its optional features

It's also possible to get disk I/O data out of `sar`, but this is even less likely to be turned on by default due to concerns about collection overhead (specifically disk space used to save the data). This is the common error that you run into:

```
$ sar -d
Requested activities not available in file
```

`sar` works by collecting data through `cron`. On RHEL/CentOS Linux, the configuration file that does so is at `/etc/cron.d/sysstat`; it defaults to collection every 10 minutes, but you can increase or reduce this by tuning it. That calls the `sa1` shell script, which then runs the `sadc` (system activity data collector). `sadc` is the utility that you pass a flag to in order to have it collect disk activity. So to enable disk collection, edit the `sysstat cron` job file on your system and make it look something like this:

```
*/10 * * * * root /usr/lib64/sa/sa1 -d 1 1
```

`-d` turns on disk data collection. Once `cron` has made its next pass over your system, you'll have the same sort of disk activity saved by `iostat` available in the history.

On Debian/Ubuntu Linux systems, installing `sysstat` doesn't start collection. You'll need to edit `/etc/default/sysstat` to change the `ENABLED` entry and then restart the service before data will be collected:

```
$ /etc/init.d/sysstat restart
```

Graphing with kSar

You can get your monitoring data out of in all sorts of ways by using `sar`, which is a very powerful tool. A popular add-on is the `kSar` Java application: `https://sourceforge.net/projects/ksar/` that allows graphing the data `sar` saves. Note that `kSar` can require some local adjustments to work. The outline at `http://arsenicks.wordpress.com/2010/01/28/testing-sar-and-ksar/` adjusts the way raw data is collected to be compatible with it, and that's the most straightforward way to handle its requirements.

It's also possible to convert the files after collection if you can't adjust the server settings, by running it on a new summary file generated with the right location. Something like the following code will output the `sar` data with the right locale for `kSar` to then read it in:

```
export LC_ALL=C
sar -A -f /var/log/sysstat/sa15 > sardata.txt
```

A good general introduction to using `kSar` is at `http://www.linux.com/archive/feature/114224`.

Note that it's pretty easy to find `sar` data that `kSar` just won't parse right. Normally the entries it doesn't like will come from the very first line of data collected during a day. Sometimes those need to be manually trimmed away with an editor before `kSar` will import the file.

Windows monitoring tools

The monitoring tools available on Windows have a slightly different balance of strengths and weaknesses compared to the traditional UNIX ones. You can't easily run them directly from the command line, and there are some additional steps required before any data can be saved. On the upside, the graphing capabilities are fully integrated.

Task manager

The simplest way to monitor what's happening live on a Windows server is to use the task manager, available by right-clicking on the Start bar or by hitting *Control* + *Shift* + *Esc*. This provides a similar view to the UNIX `top` utility.

Sysinternals tools

Another more powerful option is to download the Process Explorer and Process Monitor programs from the Sysinternals toolkit at: `http://technet.microsoft.com/en-us/sysinternals/`. Process Explorer, in particular, lets you monitor the system viewing the process title information, and is considered a must have utility for Windows PostgreSQL servers accordingly.

Windows system monitor

Windows also has a set of tools that provide similar capabilities to the UNIX `vmstat` and `iostat` utilities, through the Windows System Monitor-originally called the Performance Monitor, and the command line to run it is still named `perfmon`. This utility lets you see live views of system activity that includes functional replacements for all the OS-level monitoring tools described previously.

It's possible to disable the performance counters that track disk activity on a Windows server, and they defaulted to off in earlier versions than are common now. To check that they're turned on, run the `diskperf` command. You can enable them by running `diskperf -y`.

Here is a translation table that suggests the Windows counters that correspond to the UNIX ones discussed previously:

Windows Counter	**UNIX Equivalent**
`Processor\% Processor Time`	`100% - vmstat %idle`
`Processor\% Idle Time`	`vmstat %idle`
`Processor\% Privileged Time`	`vmstat %system`
`Processor\% User Time`	`vmstat %user`
`Memory\Pages Input/sec`	`vmstat si`
`Memory\Pages Output/sec`	`vmstat so`
`Memory\Pages/sec`	`vmstat si+so`
`Memory\Available Bytes`	`vmstat free`
`Memory\Cache Bytes`	`vmstat cached`
`Paging File(_Total)\% Usage`	`vmstat swpd`
`System\Context Switches/sec`	`vmstat cs`
`PhysicalDisk\% Disk Time`	`iostat -x %util`
`PhysicalDisk\Avg. Disk Bytes/Read`	`iostat Blk_read/s`
`PhysicalDisk\Avg. Disk Bytes/Write`	`iostat Blk_read/s`

If you are using RAID software or similar abstractions over your disk hardware, you might prefer to watch the `LogicalDisk` figures rather than the `PhysicalDisk` ones listed previously.

As you can see, most of the same data is available, with similar names in many cases. Since the database works in the same basic way in terms of how it uses the CPU and disk, you can look for the same general patterns in things such as context switching and disk throughput, as were shown in the problematic examples section on Windows too. The main concept that doesn't quite translate is how UNIX computes a specific `%iowait` percentage. Windows administrators usually using the average disk queue figures such as `PhysicalDisk/Average Disk Queue Length` instead. However, the total disk utilization percentages are often a better figure to monitor for database workloads on this platform too.

Another useful figure to monitor is `PhysicalDisk / disk\avg sec / read or write`. Measuring how long an individual write took on average is helpful to determining if commits to the WAL are happening at a good pace, for estimating if database writes are likely executing with a lot of random I/O, and guessing if a controller write cache might be completely full.

One thing that is certainly easier to do on Windows is account for performance on a per process basis. The `Process` and `Job Object` counter have some counters that can be used to track processor and I/O statistics per process, using the `Select instances from list` section when adding a counter. This is particularly useful for separating out database activity from that of other processes on the system.

Saving Windows system monitor data

To replace what's done with `sar` on UNIX systems, you can enable logging for the counters system monitor tracks. It's then possible to see historical data for the logs, not just the live charts. The exact procedure varies based on version, Microsoft's guides are at:

- Windows 2000, XP, Server 2003: `http://support.microsoft.com/kb/248345`
- Vista: `http://technet.microsoft.com/en-us/library/cc722173(WS.10).aspx`
- Windows 7,Windows Server 2008 R2: `http://technet.microsoft.com/en-us/library/dd744567(WS.10).aspx`

Another useful trick to know is that once you've set up logging on a server, you can save that setup and even move it onto another system. Just right-click on the trace data and select **Save Settings As...** The result will be an HTML file with a list of all the counters you're logging, along with the server name embedded in it. If you edit that file to reference a different server, or just remove all the `\\servername` references altogether (the default is the local host), you can now load that counter onto another server. See `http://www.pythian.com/news/1349/windows-performance-monitor-perfmon-tip/` for more information.

Trending software

It's important to know how all these low-level tools work in order to use the more complicated ones. When your system is in trouble, there's no substitute for the really detailed, second by second information tools like `vmstat` and `top` deliver. However, most of the time, what people prefer is something more like what `sar` provides: a basic summary every few minutes, but with graphs. And if you can integrate the database statistics, too, so that everything is visible on a unified timeline, then you're really in a good position to note trends in your server's use.

Types of monitoring and trending software

One of the things that make this area of software complicated to sort through is that there are several different things lumped into the category of monitoring. The major requirements are:

- **Monitoring for service interruptions**: Make sure that all the hosts on the network are working, the databases are up, and the applications are running. This category includes monitoring for things like the database disks filling up.
- **Recording performance data**: Track CPU and disk statistics on the server. Take snapshots of database statistics. Here you might also monitor things such as the number of connections to the database server.
- **Alert when problems occur**: You might e-mail out warnings, page administrators, or update some sort of operations dashboard. Alerts might be based on either service issues or performance thresholds being exceeded.
- **Graphing trends**: View the historical and current performance data recorded by the monitoring system.

Each of the monitoring and/or trending solutions you evaluate will provide a different quality of implementation for each of these four major areas. Which software makes sense really depends on how you prioritize each component, and it's common for companies that need best of breed products in multiple categories here to combine two monitoring/trending systems to meet their needs.

Storing historical trend data

One of the implementation details that is useful to understand is how the storage of performance data has progressed. Similarly to how `pg_stat_bgwriter` was analyzed in the last chapter's examples, many of the interesting performance counters you'll want to monitor and then produce trend graphs of constantly increase over time. The data that comes out of such a counter can be thought of as a series of (`timestamp`, `count`) samples. Two important questions to answer are where to store this series of points, and how to query them usefully.

The first generation of trend monitoring software was used to chart network performance data, typically collected via the **Simple Network Management Protocol** (**SNMP**) protocol that better routers and switches published their information via. You might ask SNMP for the number of bytes transferred over a port on a router, and then store the resulting time/count data somewhere for later analysis.

A program named **Multi Router Traffic Grapher** (**MRTG**) became the popular open-source solution to graphing this data. MRTG is a straightforward Perl script that can be configured to store the resulting data in simple plain text log files, and you might even push the results into a database instead. Problems with this approach include how to clean up old files, and how to handle the situation where the graphing time scales of the original collected data are not exactly the same. These are not problems that are easy to solve in standard databases that use query interfaces like SQL. If you tried, you'd discover a constant need to worry about things such as sample rate conversion and concerns about how to deal with missing data points.

After fighting with that class of problems for a while, the author of MRTG built a second generation data storage mechanism named the **Round Robin Database tool** (**RRDtool**). The "round robin" part describes the way storage overflow is managed. Each database is designed to store some amount of history, and once filled the oldest points are replaced with the newest ones, treading the storage as a circular ring. The implementation also provides a good theoretical model to deal with interpolating over missing data and resampling to higher time scales. Many of the current second generation of open source trending tools are built using RRDtool.

Nagios

The most commonly mentioned monitoring/trending software compatible with PostgreSQL is Nagios, which is itself a popular open-source project active for more than 10 years now. You can monitor just about everything-OS, network, database, applications-with either the base Nagios or via one of its many plugins.

Nagios is primarily a monitoring system that looks for common problems and then issues alerts when they happen. You can use it to detect when network hosts have gone down, when the database is doing some unexpected, and e-mail out alerts when those conditions are met. If your requirements include things such as "page people when the database disk is running out of space based on this on-call support schedule", Nagios is the first tool you should consider using for that purpose.

The simultaneous strength and weakness of Nagios is that the core of the program doesn't really know how to monitor or graph anything. It relies on a vast set of plug-ins to provide everything beyond the core. This is really flexible, but hard to get started with.

Nagios calls counter recording "Performance Data". The data comes in via a plug-in and it's written to either flat files or a database such as PostgreSQL/MySQL. And then some other tool, originally MRTG, is hooked into Nagios as a plug-in to draw trend data. Popular graphing plug-ins includes Nagios Grapher and PHP4Nagios.

The main limitations of Nagios compared to the other tools is how much work is needed to set up everything if what you want it for is higher-level trending work. You'll be editing a lot of unforgiving plain text configuration files, along with selecting and configuring plug-ins for monitoring, data storage, and graphing so that they all work together. It's even possible to configure Nagios so it's more like a second-generation tool, storing its data into RRDtool and integrating with the sort of tools it's easier to build on that infrastructure.

If your goal is a single integrated solution, and you need the strong alerting features of Nagios, that might all be worthwhile. Most of the alternatives to Nagios lean more towards trending rather than alerting, and accordingly they are usually easier to set up for that purpose, as well as more powerful in their base installs. You can do just about anything in Nagios, as long as you're willing to go through the trouble of installing and configuring all of the necessary plug-ins required.

Nagios and PostgreSQL

There are a few projects that monitor a PostgreSQL database and output the results in a way compatible with Nagios. The best of these tools in terms of active development and feature set is `check_postgres`, available at `http://bucardo.org/wiki/Check_postgres`.

In addition to the usual database statistics, highlights of what you can watch using this program that aren't necessarily obvious how to monitor directly include:

- How near the database is to running out of transaction IDs, suggesting a vacuum problem
- Whether tables and indexes appear bloated from poor cleanup of dead rows
- If any particularly long queries or idle transactions are running
- Checking if checkpoints are occurring regularly and archived files are being shipped properly to a standby server

Note that `check_postgres` can be run from the command line, too, so it's not just limited to use with Nagios. If you find the monitoring report data it generates useful, but don't want to use all of Nagios, you could script actions by running it directly and acting on the results. That would save quite a bit of time over starting from scratch to develop a similarly high quality of monitoring points for the database. Reading the `check_postgres` source code can be quite educational for learning how to write your own customized monitoring queries.

Nagios and Windows

While you can't install Nagios itself on Windows, if you do have a UNIX-like system it will run on that, and you can use it to collect data from a Windows server. You simply install a Windows data collection agent that reports back to the main Nagios system, such as `NSClient++` or `NC_Net`. Setting up these agents is covered in the Nagios installation guide.

Cacti

The second most popular application you'll see used to monitor PostgreSQL servers is **Cacti**. Cacti relies heavily on web application technology and RRDtool to do its work. This approach, heavily leveraging external libraries but including more features in the core, gives working with Cacti a slightly different feel than using Nagios, from installation to usage.

Cacti is primarily a trending system, and it doesn't do alerting. It generally has a nicer UI and is more powerful for analysis purposes than Nagios. However, its scope is limited. Where Nagios aims to satisfy all sorts of purposes moderately well, Cacti focuses on the monitoring portions that require trend analysis (the strength of its RRDtool core), trying to do a superior job in that area.

Cacti and PostgreSQL

Currently the support for PostgreSQL-specific monitoring using Cacti is still too rough and under development to properly document here. See `http://wiki.postgresql.org/wiki/Cacti` for a live document covering what's available.

Cacti and Windows

Unlike Nagios, it's possible (albeit complicated) to get Cacti running directly on a Windows host. It requires setting up a web server, PHP, a MySQL database server, and various Cacti components. Reports about the quality of the result are generally positive.

Munin

Munin is a smaller and more recent tool than either Nagios or Cacti. It uses the same RRDTool data analysis structure as Cacti, with an emphasis on making it easy to create new visualization plug-ins. Also, it can integrate into the Nagios alerting structure. The `https://www.postgresql.org/` site uses it in that way, combining a Nagios-based alerting system with the more powerful graphing capabilities of Munin. There is a PostgreSQL plug-in for Munin at `http://muninpgplugins.projects.postgresql.org/` that supports the basic server monitoring tasks, but not the sort of complicated derived views that `check_postgres` supports.

Munin is also one of the easiest monitoring projects to get started with and develop additions to. If you set up Munin to monitor hosts and just turn on all of the recommended host checks (CPUs, memory, disks), the data it will collect and graphs it creates will include almost everything needed for basic performance analysis. The same is true of its PostgreSQL monitoring plug-in. Set it up, point it as your database, and the default set of data collected will be great.

From the perspective of getting high quality graphs that combine operating systems and PostgreSQL statistics into one usable set with the minimum of setup work, Munin delivers the best bang for the buck of the open-source packages around. That it's easy to combine with Nagios for alerting purposes helps too.

Other trending packages

In addition to the big three, there are several alternates for PostgreSQL monitoring and trending with various benefits and drawbacks.

pgStatspack

A slightly different spin on database monitoring than the rest of the packages listed here, pgStatspack does exactly what the last chapter's `pg_stat_bgwriter` example did: it just saves periodic snapshots of the PostgreSQL statistics into tables for later analysis. The design was influenced by the very popular statspack utilities for Oracle. The project is hosted at `http://pgstatspack.projects.postgresql.org/` and a good introduction to it is at `http://www.fuzzy.cz/en/articles/db-tuning-using-pgstatspack/`

The main problem with this approach from my personal perspective falls into three categories:

- Database statistics alone are not sufficient to solve complicated performance issues. You have to collect operating system level information too.
- Aggregating snapshots stored in this fashion into different time scales is a harder problem to solve than it appears at first. The reason why programs such as RRDtool exist is because they solve that particular problem well.
- Graphing data over time is the easiest way to find trends. pgStatspack provides a SQL interface to the data.

I suspect that many people who start with pgStatspack will end up needing to add more powerful trending and monitoring tools to the database anyway eventually, which makes me question the value of this intermediate step. However, my perspective is not shared by everyone.

If you're the type of DBA who is focused on the database and want to primarily use tools hosted within it, pgStatspack may be an appropriate tool for you. Similarly, if you're used to using similar Oracle tools, having this available may make the PostgreSQL environment easier to get used to.

Zenoss

Zenoss is a hybrid license product that is a bit slicker to set up and use than most of the tools mentioned here. The Zenoss Core is an open-source monitoring solution, and the company also has a commercial Enterprise version as well. While not particularly popular yet for PostgreSQL use, it is supported, and the rest of the integrated infrastructure around Zenoss can be valuable for some environments. Extensions to Zenoss come bundled into a module called a ZenPack. A recently released PostgreSQL ZenPack is available at `http://community.zenoss.org/docs/DOC-3389`.

Hyperic HQ

Another commercial product with integrated PostgreSQL monitoring is Hyperic HQ: `http://www.hyperic.com/products/postgresql-monitoring`.

One advantage of using Hyperic's products is that they monitor a whole lot of other types of software as well. If you want to monitor your database and application server using the same technology, it may be easier to do so with Hyperic than some of the more OS or PostgreSQL focused tools mentioned here. Similarly, there are even Hyperic plug-ins aimed at monitoring connection poolers such as pgpool. Its real strength is letting you see how these various levels of your application fit together from a performance perspective. On the flip side, PostgreSQL support in Hyperic HQ is not always kept current. At the time of writing, they haven't even added support for PostgreSQL 8.4 yet.

Reconnoiter

After struggling with all the other tools listed here not quite fitting their needs, the Internet application and support company OmniTI built a solution optimized for their needs named Reconnoiter and released the results to the world recently: `http://labs.omniti.com/`.

Compared to Nagios, it's less complicated to set up, is optimized to use less resources, and has built-in trending capabilities. It uses a standard PostgreSQL database to store its data, which provides some synergy if you're already using it for other purposes. Rather than use the RRDtool approach where storage is reused aggressively, Reconnoiter considers using large amounts of disk space for logging historical data at a fine time scale a feature rather than a problem.

As shown in the problem examples earlier, there are often performance situations that really matter to your app that you can only see if you drill down into second-level data, ones that disappear when averaged out over a longer time period. RRDtool, as commonly used in tools such as Cacti, stores limited data at any sort of fine timescale. Instead, the assumption is that if you're going back months instead of hours, you'll be satisfied with minute or hourly level data. Monitoring with Reconnoiter instead would allow finding a similar period in the past and potentially drilling down into all the sub-minute data it still has stored. For applications sensitive to latency, being able to track down performance issues to this resolution scale after the fact can be quite valuable.

Staplr

Staplr was developed by heavy PostgreSQL website `myYearbook.com` for their own trending statistics analysis. Staplr has proven capable of helping to scale up one of the largest sites using the database. It's not in heavy use outside the company yet and documentation is minimal: `http://github.com/myYearbook/staplr`. As a result, so far it still feels like an internal tool rather than a larger community project. It is worth looking at when evaluating PostgreSQL-centric trending systems.

SNMP tools

Many larger companies rely upon SNMP for their enterprise-wide system monitoring. There is a very basic PostgreSQL SNMP MIB available at `http://pgsnmpd.projects.postgresql.org/`, but it hasn't been updated for several years at the time of writing.

Summary

Attempts to improve database and operating system performance are best done using careful observation, not speculation, in order to determine where the system bottlenecks are at. You need to start observations before you have a performance problem, to record baseline information. Most production database servers should consider basic monitoring and trending setup a requirement of their early deployment.

However, it is helpful to know the low-level tools too, because the longer term views provided by most monitoring and trending tools will miss brief problems. With today's breed of database applications, even a pause lasting a few seconds could be a major response time failure, and it's one that you wouldn't even be able to see in data collected on a minute scale. Both short-term and long-term data collection has considerable value, and knowing when to switch to and from the detail level to match the problem at hand is a valuable skill to hone. Watching the total percentage of time a disk is being utilized is more useful than any queue or transfer total figures for determining how close it is to maximum capacity. It even allows you to estimate the random `vs`. sequential balance of the workload that disk is running. Periods with bad database response time will usually have a high waiting for I/O percentage, a drop in context switches, and/or a decrease in the number of processes running. The exact pattern that appears gives you a clue as to where the most likely performance bottleneck is at. Historical data can be collected and reported on with the `sar` utility, albeit at a rough timescale that will average out some performance spikes into invisibility.

Windows has functional replacements for all these UNIX utilities, particularly if you install a small number of useful monitoring tools. A variety of true trending graph analysis packages are available with various degrees of PostgreSQL monitoring support: Cacti and Munin are the most popular focused ones available.

The next chapter describes pooling and caching techniques, and tools available for pooling and caching.

13
Pooling and Caching

One of the unfortunate limitations of any database software, and one that's particularly apparent in PostgreSQL, is that there is a lot of overhead you're paying for even when executing really simple queries. If you have a complicated join happening, you should be happy to pay the expense of query optimization. If you're just reading a simple table, the overhead of opening a database connection and waiting for that query to execute can be higher than you'd like.

There are two common approaches for reducing that overhead, pooling and caching, and both are introduced in this chapter. This sort of software is external to the database, relatively complicated to set up, and its use is very application dependent. Accordingly, the focus here is on general theory rather than trying to show a working example. It's unlikely any one example would translate into your environment very well anyway.

We will be covering the following topics in this chapter:

- A connection pooling technique using different tools like pgBouncer and pgPool-II
- Database caching using "`memcached`" and "`pgmemceche`".

Connection pooling

PostgreSQL does not highly prioritize making the act of connecting to the database one that happens quickly. On each connection, PostgreSQL creates a new process to talk to the client. This is a very expensive operation because the operating system creates a new process and copies all the data. Things like the `pg_hba.conf` authentication implementation are optimized for security and flexibility, even if it's at the expense of speed. The presumption is that users will run things for a long time relative to how long it takes to connect.

When this isn't true, which can be the case for web applications in particular, connection pooling is one approach to reducing this overhead. The connection pool sits between your application and the database. It makes a fixed number of connections to the database, typically under 100, and keeps them open all the time. As incoming requests come in, those connections in the pool are re-used. The `DISCARD ALL` command can be used in a PostgreSQL session to make it fresh for a new connection. When clients disconnect, the pooler resets the session without dropping the database connection, leaving it ready for a new connection to use.

Pooling connection counts

The fundamental idea behind sizing a connection pool is that you should have enough connections to use all of the available resources, but not significantly more than that. The right size to saturate the server in most cases depends on the number of CPU cores, how much of the database is cached in memory, and the speed of the underlying disks. Once you've moved beyond the point where the server is busy all the time, adding more connections only serves to reduce efficiency, forcing the server to swap among multiple tasks when it would be better served with a smaller number.

It's hard to predict how many pooled connections to the database are necessary to keep it fully loaded without overloading it. Many users report optimal pooling counts to be between two and three times the number of cores on the system, perhaps more on a system with a large number of drives. A standard iterative technique is to start with 100 connections and then tune the number down from there if the server load seems too high. Prepare to be surprised at how low the optimal count really is; it's probably lower than you'd expect. Even though you might think that your server needs lots of connections to service a heavy load, in reality once you've saturated all of the CPUs on the server, trying to execute more things at once just decreases efficiency. Almost all of the time, you'll be better off queuing the connections so that they wait without causing contention and then execute on a lightly loaded server.

There are a few rules of thumb for how many connections are definitely too many. On typical UNIX derived operating systems, such as Linux, the point at which adding additional connections become really ineffective is generally between 500 and 1,000 active connections. If many of your connections spend a good chunk of their time `IDLE` (as shown by `pg_stat_activity`), you can discount their average overhead to some extent.

```
postgres=# SELECT PID, APPLICATION_NAME, STATE FROM pg_stat_activity;
 pid  | application_name | state
------+------------------+--------
 6501 | psql             | idle
 6596 | psql             | idle
 6608 | psql             | active
(3 rows)
```

Generally fewer connections than that is optimal, but you don't necessarily want to add the overhead of maintaining a pool until the server is really swamped. If your connections count is well into the thousands of active sessions, you definitely want to use a pool rather than direct connections.

Windows systems don't scale to as many connections in general, and there's a hard limit you'll run into in most cases too. If you are running the PostgreSQL server as a service, rather than directly from the command line, it will typically be assigned 512 KB of "Desktop Heap" space to work with. Since each connection takes approximately 3.2 KB of space, expect your server to run out of space in the heap and therefore stop accepting new connections after approximately 125. It's possible to increase the heap size, but there's a potential that your system will not boot if you set it too high. See the *I cannot run with more than about 125 connections at once* entry at `http://wiki.postgresql.org/wiki/Running_%26_Installing_PostgreSQL_On_Native_Windows` for more information about this limitation and possible workarounds. Generally, the best workaround is to use a connection pooler that limits the number of connections to below this threshold, as that will improve server efficiency too.

From a monitoring perspective, connection pooling is likely to help if you have hundreds or more connections and you see that most of your system processors are being fully utilized. Connection overhead will show up as a mix of user and system/kernel time, so expect to reduce both with a pooler in front of the database. If your system spends most of its time waiting for disk I/O instead, it's unlikely a pooler will help you out. Caching might however.

pgpool-II

The oldest of the PostgreSQL compatible packages used for connection pooling is pgpool. The original version of pgpool had many fewer features, but now it is in good shape. The pgpool-II improves on the original pgpool in a variety of ways, and detail can be found at `http://www.pgpool.net/mediawiki/index.php/Main_Page`.

Its primary purpose is not just connection pooling: it also provides load balancing and replication related capabilities and a parallel query feature which is dropped in pgpool version 3.4.

Connection pooling

The pool in pgpool is primarily to handle multiple servers, with the program serving as a proxy server between the clients and a number of databases. The pgpool sits between PostgreSQL and client, saves the connection, and reuses the connection afterwards. This reduces the connection overhead of with PostgreSQL. Brief details can be also be found at `http://www.enterprisedb.com/postgres-plus-edb-blog/ahsan-hadi/little-respect-pgpool-ii`.

Replication

pgpool-II also provides the replication by managing the multiple PostgreSQL servers. Details about the replication can found at `http://www.pgpool.net/pgpool-web/contrib_docs/simple_sr_setting2_3.3/`.

Watch-dog

pgpool-II introduce the problem of a single point of failure. pgpool-II failure can cause complete database service shutdown and make the whole database system unavailable. To overcome this problem, pgpool-II provides the watchdog feature for high availability. To avoid a single point of failure, install two or more pgpool-IIs and enable the watchdog feature. Full details of this feature can be found at `http://www.pgpool.net/pgpool-web/contrib_docs/watchdog/en.html`.

Failover

Pgpool also provides the failover feature. If one or more PostgreSQL servers fail, then pgpool lets other remaining PostgreSQL takeover for the uninterrupted service. A brief description of pgPool's failover can be found
here: `http://www.archaeogeek.com/blog/2012/06/19/failover-with-pgpool/`.

Load balancing

There are a few limitations to the pgpool-II setup serving as a connection pooler. One is that each connection is setup as its own process, similar to the database only re-used. The memory overhead of that approach, with each process using a chunk of system RAM, can be significant. Pcp system; pgpool-II is not known for having powerful monitoring tools either. But the main drawback of the program is its queuing model. Once you've gone beyond the number of connections that it handles, additional ones are queued up at the operating system level, with each connection waiting for its network connection to be accepted. This can result in timeouts that depend on the network configuration, which is never a good position to be in. It's a good idea to proactively monitor the *waiting for connection* time in your application and look for situations where it's grown very large, to let you correlate that with any timeouts that your program might run into.

Because of its replication and load balancing related features, for some purposes pgpool-II is the right approach even though it's not necessarily optimal as just a connection pool. pgpool-II supports what it calls master/slave mode, for situations where you have a master database that handles both reads and writes as well as a number of replicated slaves that are only available for reading.

The default replication software it assumes you're using, and the only one available in older versions of the software, requires you have a set of databases all kept in sync using the Slony-I replication software. A common setup is to have a pgpool-II proxy in front of all your nodes, to spread the query load across them. This lets you scale up a read-only load in a way that's transparent to the application, presuming every node is qualified to answer every query.

Starting in pgpool-II 3.0, you can use this feature with the PostgreSQL 9.0 streaming replication and Hot Standby capabilities too. The read-only slaves will still be subject to the limitations of Hot Standby described in `Chapter 14`, *Scaling with Replication*. However, within those, pgpool-II will handle the job of figuring out which statements must execute on the master and which can run against slaves instead.

As with the Slony case, it does that by actually parsing the statement that's executing to figure out how to route it. The way it makes that decision is covered in the pgpool-II documentation. This is one of the reasons pgpool-II is slower than pgBouncer, because it's actually interpreting the SQL executing. However, as it enables the intelligent routing capability, too, that may be worth doing.

pgpool-II can be used for connection pooling, but needs to be carefully monitored for network timeout issues. pgpool-II allows load balancing connections among multiple replicated slaves to scale read-only traffic usefully, when combined with Slony-I or the streaming replication.

pgBouncer

pgBouncer is a connection pooler: the application connects with pgBouncer and pgBouncer uses an already created connection or creates a new connection with PostgreSQL. The pgBouncer project originated as part of the database scaling work done by Skype. A detailed description of pgBouncee can be found at PostgreSQL wiki page: `https://wiki.postgresql.org/wiki/PgBouncer`.

Designed to be nothing but a high-performance connection pooler, it excels at solving that particular problem. pgBouncer runs as a single process, not spawning a process per connection. The underlying architecture, which relies on a low-level UNIX library named `libevent`, was already proven for this purpose in the field--the memcached program uses the same approach. The internal queue management for waiting connections is configurable so that it's easy to avoid timeouts.

When it comes time to monitoring the pool itself, it displays its internal information by a database interface you can even send commands to, serving to both provide information and provide a control console. Simply connect to the `pgbouncer` database on the port pgBouncer is running on, using the standard `psql` tool, and you can use the `SHOW` command to get a variety of information about the internal state of the pool. The console interface accepts commands like `PAUSE` and `RESUME` to control the operation of the pool.

Another neat feature of pgBouncer is that it can connect to multiple underlying database servers. You can have databases on different hosts look like different databases on the single host the pool is running on. This allows a form of partitioning for scaling upwards if your system's load is split among many databases. Simply move each database to its own host and merge them together using pgBouncer as the intermediary, and your application won't even need to be changed.

If you have hundreds or thousands of connections and are out of CPU time, pgBouncer should be your first consideration as a way to reduce the amount of processor time being used. The main situations where pgpool-II works better at this point are ones where its load-balancing features mesh well with the replication approach being used.

To look at what's new in pgBouncer, visit this link: `http://blog.2ndquadrant.com/pgbouncer-1-6/`.
pgBouncer is the preferred PostgreSQL connection pooler if you just need pooling.

Application server pooling

Depending on the application you're running, you may not need to use a database-level connection pooler. Some programming models include what's referred to as an application server, an idea popularized by Java. Popular application servers for Java include Tomcat, JBoss, and others. The Java database access library, JDBC, includes support for connection pooling. Put those together, and you might get efficient database connection pooling without adding any more software to the mix. Tomcat calls this its **Database Connection Pool** (**DBCP**). A longer list of open source pooling software is available at `http://java-source.net/open-source/connection-pools` and commercial vendors selling application servers might include their own pooler.

There are also application poolers available for some other programs, too. It's not an idea unique to Java application servers. If you have such an application level pooling solution available, you should prefer it for two main reasons, beyond just reducing complexity. Firstly, it's probably going to be faster than passing through an additional layer of software just for pooling purposes. Secondly, monitoring of the pool is likely integrated into the application server already. You'll still need to monitor the database underneath the pool.

Database caching

The fastest type of database query is the one that doesn't happen at all. There are ways to avoid actually passing queries through to the database that can seem like a silver bullet for scaling at first, even if they're only effective at reducing certain types of load. You need to understand the limitations of the underlying technology to make sure that the non-database semantics introduced by that change don't cause your application any issues.

What is memcached

Memcached is a distributed memory caching system. Memcached starts up with a block of memory it's allowed to use for caching, and one that it manages quite efficiently. Like pgBouncer, incoming connections are handled using the `libevent` library, so that any number can be serviced without timeout issues.

The interface to the program lets you define key/value pairs of information that are then stored. Keys can only be up to 250 characters long, and values are limited to 1 MB. The program's popularity comes from how easily it can be used to cache content in a web site. Returning to the idea that the best database query is the one that doesn't have to happen, the fastest way to return a database-driven web page a second time is to send a cached copy of it. That avoids both the database query and the page rendering.

If you can figure out how to map your incoming requests into keys without spaces in them (which is easy for most web site URLs), and how to save the result into the amount of space the cache can hold, memcached will do all the dirty work of keeping the cache optimally filled with useful data. Key generation doesn't have to be hard either. The robust way to generate those in a general way is to use a hash code generation scheme, such as the PostgreSQL `hashtext` implementation discussed in the partitioning chapter; MD5 and SHA are other popular hashing approaches. That will let you turn arbitrarily long strings into smaller ones with minimal chance of a collision in the generated keys.

While not directly a database technique, using memcached to cache higher-level content can avoid page rendering, which would have otherwise required a database lookup. You just need to make sure you have a way to invalidate those pages when the underlying content changes. That's the main non-obvious complexity you'll run into with any non-trivial memcached deployment.

To have more information on this, please visit this link: `https://memcached.org/` is the official site of memcached and `https://en.wikipedia.org/wiki/Memcached` is the wiki page of memcached.

pgmemcache

If you can resolve the page invalidation problem in a more general way, it's possible to use memcached as an intermediary for database queries too. When a user on a web site is grabbing their own data over and over, but not modifying it, you should be able to figure out how to cache database lookups related to their data usefully. Then you just need to be careful to throw those pages out if the data is modified.

Prior to version 9.3, PostgreSQL was missing materialized view, a technique that lets you run an expensive query and save its output for use by later clients who want the same data. You can implement a simple version of that technique using pgmemcache.

The way to do that in PostgreSQL is quite straightforward: simply add a pair of triggers for `AFTER UPDATE ON` and `AFTER DELETE ON` for any table you are storing in the cache. If you take care of that, you can use the pgmemcache program from `http://pgfoundry.org/projects/pgmemcache/` or from `https://github.com/ohmu/pgmemcache` in order to handle the routine chores of setting up memcached for storing database information.

There is a second program with the confusingly similar name of pgmemcached that was a development effort related to PostgreSQL/memcached integration. That project was completed and abandoned in 2008. pgmemcache is now recommended as the program to use here even by the original author of pgmemcached.

This technique is particularly well suited to serving key/value data from a PostgreSQL database efficiently, simply by allocating some memory for pgmemcache to cache the most popular parts. This combination gives the fast response times of a cache for read operations and the integrity and complicated query capabilities of a database, with the downside of a small window for write race conditions while the update/delete trigger is modifying the value in the cache.

Recent releases of pgmemcache have broken compatibility with PostgreSQL versions before 8.4. This is covered in the `README` file distributed with the program, and a presentation noting that and other useful trivia about the program is available at `http://2ndquadrant.com/media/pdfs/talks/char10/pgmemcache.pdf`.

Memcached can be used to cache anything you can represent as a key/value pair, most popularly web pages. The pgmemcache combined with database triggers allows caching database lookups with little loss in transactional behavior.

Summary

Large deployments of PostgreSQL systems go through several common phases as the number of database clients increases. If you can size the system properly so that the database is mainly returning information that's in RAM, rather than being dependent on disk speeds, it's quite easy to move onto a new bottleneck: simply not being able to keep up with the relatively high overhead of creating a database connection and asking it for data.

When reaching that point, there are two major approaches to consider. You can reuse database connections with pooling, or try and cache database activity outside the database. The best part is that these two approaches both stack on top of one another. You can, for example, use pgmemcache to reduce database reads, while also using pgBouncer to reduce the connection overhead for the reads that still happen. Separately or as part of that, you might also scale upwards by replicating the database or partitioning your data set across multiple systems. The subject is covered in more detail over the next two chapters. Opening a new PostgreSQL connection is a fairly expensive operation in terms of CPU used. Connection poolers keep some of those connections open all the time to reduce that overhead. The connection pool should be sized to just use all the system resources, typically less than 100 connections. Systems with greater than 500 clients are very likely to benefit from a pool. Windows systems will rarely go greater than 125 clients before running into resource limits, making pooling an even higher priority.

The next chapter comprises PostgreSQL scaling and replication techniques. The chapter elaborates the new PostgreSQL replication techniques and discusses the external tools that can be used for replication.

14
Scaling with Replication

Sometimes when trying to improve database performance, the most practical approach is to add more copies of the data and spread out the load over them all. There are a variety of PostgreSQL projects that replicate data across multiple nodes that you might be able to use for that purpose. Also, high performance systems tend to come with high availability requirements, with their respective overhead too. Learning how to coexist with your replication software may impact your performance tuning work, even if the copies are not being used actively for load distribution.

Note that none of the mature tools mentioned in this chapter support scaling up for high write volume usefully. The best-explored solution in that category is using pl/Proxy, as described in `Chapter 15`, *Partitioning Data*. Write volume scaling is also a goal of Postgres-XC. In general, when we're talking about scaling a database upwards by adding nodes in this chapter, that means allowing more reads to happen against the same data, but without an improvement in the write speed.

Topics we will look at in this chapter are:

- Hot standby
- Streaming and synchronous replication
- Replication queue managers
- Different replication tools

Hot standby

PostgreSQL has shipped with integrated standby features since version 8.2. This allows creating a master/standby pair of nodes, where the standby regularly receives a log of database updates from the master. With the currently common warm standby configuration, introduced in PostgreSQL 8.2, this allows a high-availability setup where, if the master is lost, the slave can be brought up quickly to replace it.

Starting with PostgreSQL 9.0, it's possible to run queries against the standby server or servers as well. This hot standby feature lets you scale up read workload by using a pool of multiple read-only servers. You just need to architect your application so writes load go to the master node. The details of PostgreSQL's hot standby can be found at `https://www.postgresql.org/docs/current/static/hot-standby.html`.

Terminology

There are several terms you need to know in order to follow any discussion of warm or hot standby setups:

- **Write-Ahead Log** (**WAL**): PostgreSQL writes information to a series of write-ahead log files, in segments 16 MB in size, before making corresponding changes to the database itself. If you start with an identical pair of databases, and apply the same WAL files to them, the resulting pair will also be identical, since the WAL contains all changed data.
- **Base Backup**: A complete backup of the database in a way that includes everything. It also includes the crash recovery data and all the changes made during the backup process. This requires using the `pg_start_backup` and `pg_stop_backup` commands, as well as making backup copies of both the entire database and the WAL files archived during the period between when those commands executed.
- **Point-in-Time Recovery** (**PITR**): If you have a base database and a series of WAL files, you can apply only some of WAL files and then stop recovering information from those WAL files. This allows the Point-in-Time Recovery feature. This even allows complicated recovery situations like alternate timelines, where a rebuilt database diverges from its original history--perhaps multiple times in search of the best place to recover to--before the server is started.

- **File-Based Log Shipping**: If you make a base backup of a server, and then "ship" all of the new WAL files, it archives to another server--that new server can be kept in sync with the original.
- **Standby**: A system with a complete base backup and a stream of file-based logs shipped to it can be a standby: a server with exactly the same data as the original. A standby is kept up to date as enough new transactions appear (or time passes) and WAL files are shipped to it. A warm standby continuously applies new WAL files as they appear.
- **Failover**: Taking the standby out of recovery mode and turning it into an active server, using what's called a "trigger" file. In intentional failover situations, you can stop the primary first and make sure all its data has been flushed. In a true failure of the primary situation, this will not be possible. Some recent transactions (the ones not yet shipped to the standby) committed on the primary may not be available on the standby that's now promoted to being a primary.

- The important thing to realize here is that the WAL is a stream of block-level changes to the database. You can't filter the results by database or any other mechanism; you can only make a perfect copy of an entire PostgreSQL cluster. The transport mechanism used doesn't know anything about the transactions involved; it just knows how those turned into changes to the database on disk.

Setting up WAL shipping

The exact way you set up replication using WAL shipping varies based on PostgreSQL version, and a fair amount of additional scripting is required.

- `archive_mode`: When `archive_mode` is enabled, WAL files are sent to the archived folder using the `archived_command`. The `archived_command` is based on `archived_mode`. The `archived_command` is only invoked when `archived_mode` is enabled.
- `archive_command`: The shell command to execute to archive a WAL file.
- `archive_command = 'copy "%p" "/home/postgresql/%f"' # Windows`.
- The `%f` will be replaced with the name of WAL file and `%p` will be replaced by the path of WAL file

- `archive_timeout`: The main tunable parameter to note with basic WAL shipping is `archive_timeout`. This determines how long a WAL file can sit on the master before it gets shipped to the standby, regardless of whether it's full or not. If you want to reduce the window of potential data loss, you might reduce this value to a much lower value. But be careful: this can significantly increase the overhead of processing, network use, and disk use from WAL shipping. The 16 MB WAL files pile up surprisingly quick if you try to do something like minimize lost transactions by setting the archive timeout to only a few seconds; that really doesn't work.

Streaming replication

The other major replication related feature in PostgreSQL 9.0 is streaming replication. Once the standby server is synchronized with the master, this new feature will asynchronously send new WAL data over the network shortly after it's committed, close to real-time if things go well. This replication style is sometimes referred to as semi-synchronous replication. It's not fully synchronous, but the delay between commit and replication will be quite short in most cases; typically under a second, something impossible to accomplish with earlier versions.

Cascading replication

In asynchronous replication, standby sever can also stream WAL record--this is called cascade replication. So the primary server sends WAL records to the standby server and the standby-server streams WAL record to the next server and continue. There is no limit on cascade. Let's look at the following:

- **cascade server**: The server, which can receive and send WAL records is called the cascade server
- **upstream server**: The server directly connected to the primary server is called the upstream server
- **downstream server**: The other server not directory connected to the primary server is called the downstream server

Synchronous replication

As we discussed, PostgreSQL's streaming replication is asynchronous; this has the potential of data loss. If a transaction commits on the primary server and crashes before committing to the standby server, it causes data loss. This is because of transaction delay: the more the transaction delays, the more the data loss. But in case of synchronous replication commits, wait till commits have been written on the primary and standby servers. This technique reduces the chance of data loss if both standalone and primary servers crash at the same time, because it is waiting for confirmation from both the primary and standby server. This provides a high level of durability but introduces response time delay, because of confirmation from both servers. But this response time delay happens only in the case of write only top--level transactions and is not applicable on read-only and sub-transactions. Therefore, it requires planning to avoid such delays, even if it's only for write transactions. It's a bad plan to specify a synchronous replication option for each and every transaction. Admin should categorize the data and specify the synchronous replication option only for the most important information.

Tuning Hot Standby

Imagine the case where the standby is replaying the block changes resulting from a `DROP TABLE` on the master. If you're using hot standby to run queries on the standby, it's possible there's one using that table already running when these changes arrive. The server has two choices at this point: it can either apply the changes, or it can continue to let the query run--it can't do both.

While getting rid of a table is a rare situation, this is the easiest example to understand from a class of problems where hot standby cannot replay some blocks without cancelling queries that expect them to be there. One of the most common causes of such changes is when you vacuum the master database.

What the server should do when this happens really depends on the priorities of your standby server. There are three fundamental things you might want to do with your standby server:

- **Prioritize keeping the standby current**: Your goal is to have a seamless, quick failover if the master goes down.
- **Prioritize being able to run long queries on the slave**: One popular use for this form of standby is to offload large batch reports that would otherwise need to be run on the master to instead run on the standby.

- **Prioritize minimizing the impact of the standby server on the master**: Ideally the standby would be completely decoupled from the master, but doing that introduces some potential issues.

The nature of the problems here are such that you can only do two of these three things at a time. The good news is that you can pick which two. There are a few tunable parameters and some techniques available that let you decide exactly how to skew your implementation in regards to these three goals. The parameters are:

- `max_standby_archive_delay` and `max_standby_streaming_delay`: Controls how long a block change from the two possible sources here (a shipped archive file and streaming replication) that will interrupt a query will be allowed to wait before it causes that query to be cancelled. Increasing these values prioritizes running longer queries at the expense of keeping the standby current.
- `vacuum_defer_cleanup_age`: Increase this parameter on the master in order to delay vacuuming for this number of transactions. This prioritizes running long queries without cancellation, but there will be higher overhead on the master because vacuum will lag behind its normal activity level. While this only improves the situation with cancellation from vacuum activity, since that is the most common source for conflicts in most normal database use (where tables aren't being modified, only the data in them is), it helps quite a bit.

The expected evolution of the hot standby design in future versions of PostgreSQL will let the standby export information about what queries are running to the master. That approach lets the master work similarly to how MVCC prevents this sort of problem from happening on a single server: the system will keep old versions of any row that might still be needed for a running query from being cleaned up. It's possible to do this with the current PostgreSQL 9.0 by opening a connection from the standby to the master and starting a transaction before running a long query, perhaps using the `dblink contrib` module. The best way of doing that is still a topic of research at the point this is being written.

One way to work around the fact that standby servers can't be tuned universally is to simply add more than one of them. There's nothing to prevent you from having two standby systems--one optimized for high-availability with low standby delay parameters, while a second is optimized for long reports.

Replication queue managers

At the other end of the spectrum, some replication software does statement-based replication. Rather than sending blocks over, the list of transactions made to each table is extracted from the database, typically with triggers. Then those statements can be saved to a queue, shipped to some number of slave nodes, and then executed there. This introduces some non-trivial overhead on the master server, because the overhead of the triggers, queue management, and statement shipping is moderate.

However, the resulting copies are then completely independent of the master server. And unlike WAL shipping approaches, you can pick and choose exactly which tables do and don't get shipped to the standby. In addition, since high-level statements are being shipped, the servers don't even need to match perfectly. This form of replication is therefore useful for doing PostgreSQL version upgrades. You can bring a server running a newer version of the database software up, add it as a replica, and let it stream the data from the master at whatever speed you want until it catches up. Once it's current, a standard failover to that new node will allow you to bring the database up and running newer version.

It's not unusual for developers to think that they can just ship statements between database servers themselves, rather than use one of these packages. One hidden complexity in statement level replication is coping with non-deterministic statements, where the exact result will vary if you execute the same thing on a different server. Three sources for such data are: calls to generate random numbers, sequence generation, and statements that incorporate a server timestamp. You can expect the latter in particular to haunt the poor developer who decides to reinvent this particular wheel, as such statements are very common in database applications. Getting statement order to match exactly is another extremely tricky problem. It's unlikely you'll do a better job in the end than these mature solutions already available. There are good reasons that all these replication packages end up being more complicated than you might think is required for this sort of work--they can't be any simpler and still work correctly.

Slony

One of the oldest and most mature PostgreSQL replication solutions, Slony is both a complex and feature rich replication program which you will find at `http://slony.info/`.

It's written in C and highly optimized for the form of replication it does. Much of the development work on Slony is done by Afilias, who use Slony for both redundancy and load scaling for a significant portion of the Internet Domain Name Service. It aims at that reliability level.

Slony is a complicated program, which is reflected everywhere from initial setup complexity to extensive monitoring requirements. Expect to devote a significant amount of study to any Slony deployment. The return on that time investment is a fairly powerful replication solution, with many years worth of thought put into its feature set.

One of the limitations of Slony to be aware of is that every table it replicates needs to have a primary or similar key field for it to operate over.

There are multiple reasons why Slony is a better choice than streaming replication because Slony works well in a heterogonous environment with different version of PostgreSQL Streaming replication replicates everything, but Slony replicates only what you really want to replicate.

Londiste

Londiste is a more recent queue-based replication software driven by triggers, that's part of the SkyTools project set, available at `http://pgfoundry.org/projects/skytools/`.

It's one of the database applications written to support scaling up database operations at Skype Relative to Slony, Londiste has two highly visible advantages. The first is that it targets a smaller feature set and is therefore less complicated to set up. It has also abstracted away the concept of transporting changes from the master to the slaves a bit more cleanly. The Londiste design is based on a generic queue library that is less coupled to the underlying data transport mechanism used than Slony.

Another advantage for Londiste is that it requires minimal locking in order to replicate. Slony has a few situations you might run into where the exclusive table locks are required. An example is moving a replication set from one node to another.

Given the way open source software works, where it's expected that you might end up needing to read and perhaps change the source code to add-on software you use, Londiste can also be easier to work with. If you need to get familiar with its internals or make changes to Londiste, it's simpler to do so than it is to gain similar competence with the Slony source code.

Read scaling with replication queue software

When using either Slony, Londiste, or similar statement-based replication software, or even streaming replication plus hot standby, each of your slaves is going to lag a bit behind the master. If your application can support running queries against that slightly out of date information, it's straightforward to put a load balancing program such as pgpool-II in front of the master and a set of slave nodes in order to scale up reads.

This replication approach is also very good for maintaining autonomous remote servers. You might put servers in multiple cities and direct application reads against the closest copy of the data, as a way to decrease load on any one server.

More information about how pgpool-II works for this purpose was covered in the previous chapter.

Special application requirements

All of the solutions covered so far are master/slave solutions that, while asynchronous, expect fairly tight contact between all the database nodes. For example, the master in a hot standby deployment will eventually run out of disk space and crash if it can't ship WAL files to the slaves it's feeding.

Bucardo

One of the things most PostgreSQL replication solutions have in common is that they have a single master node. Bucardo is instead a multi-master replication solution, also implemented with triggers, available at `http://bucardo.org/`.

Anytime you have more than one master, this introduces a new form of replication problem. What do you do if more than one node modifies the same record? Solving that is called conflict resolution in replication terminology. Bucardo provides a hook for your application to figure out what to do in that case, after it does the heavy lifting work on the main replication chore.

Bucardo is also very asynchronous. Instead of expecting nodes to be in constant contact with one another as most database replication software does, Bucardo allows a disconnected master to execute an arbitrary amount of work and then resynchronize when connected to the rest of the network again. This makes it particularly well suited to scenarios such as databases hosted on a mobile system. A laptop might run the database server, synchronize before leaving for a trip, then update in both directions with all the changes made when it returns to the office.

Bucardo probably isn't the right solution for scaling upwards using load balancing, and it's not a replication solution appropriate for most high-availability failover requirements. But it can be appropriate for splitting load among servers in multiple locations, particularly if the links between them are unreliable--a situation that warm standby and Slony/Londiste don't handle particularly well.

The detailed differences between Bucardo, Slony and Londiste can be found at `https://wiki.postgresql.org/wiki/BDR_Comparisons`.

pglogical

pglogical is a PostgreSQL extension providing logical replication. pglogical is as flexible as trigger-based replication and as efficient as log-based replication. pglogical replication uses a publish/subscribe model for replication.

Further details of pglogical can be found at `https://2ndquadrant.com/en/resources/pglogical/`.

xDB

xDB is an EnterpriseDB propriety product providing multi-master replication. In multi-master replication, xDB is one of the best propriety solutions. Further details of xDB can be found at `https://www.enterprisedb.com/products-services-training/products-overview/xdb-replication-server-multi-master`.

pgpool-II

The replication features of pgpool and pgpool-II work quite differently to the rest of the programs covered here. They are implementations of statement-based middleware. That means that they serve as an intermediate proxy server between your application and the database, translating the statements executed into the appropriate ones for each of the destinations. They can provide synchronous replication as just one of their benefits. For more information visit `http://pgpool.projects.postgresql.org/`.

pgpool-II works in one of several modes that include connection pooling, replication, master/slave, and parallel query, some of which can be combined to be active at the same time.

In the latest version of pgpool-II, the parallel query feature drops due to some issues.

pgpool-II provides a useful toolkit of database scaling features, while suffering a bit in complexity from its lack of focus. Other replication solutions that have a more targeted feature set tend to be easier to set up and work with than pgpool-II.

Other interesting replication projects

There is a wide perception that since PostgreSQL hasn't ever shipped with a complete replication solution (even the streaming replication in 9.0 requires some external code support), its replication options are therefore weak. The opposite is true: because the development community isn't focused on a single replication solution, there are actually so many viable options that just describing them all is a chore. The programs listed in this section don't have the general developer mindshare that the ones already covered do. But most of them have a very specific type of replication problem that they solve better than any of the other alternatives.

- **Mammoth Replicator**: Rather than derive statement changes from triggers, Mammoth instead modifies the PostgreSQL source code to generate them. The need for a customized PostgreSQL makes it inappropriate for some uses. If you need statement replication but can't afford the overhead and complexity of needing triggers to derive them, Mammoth is a mature solution to that problem, available at `http://www.commandprompt.com/products/mammothreplicator/`.

- **PgCluster**: Synchronous master-master replication solution developed using customized PostgreSQL code. Applications that need true synchronous transactions, where commits either happen on both nodes or not at all, might find PgCluster appropriate for that need. It requires modifications to the PostgreSQL source code, and the open-source release (there's also a commercial version) usually isn't very current with the main PostgreSQL project's release cycles, available at `http://pgcluster.projects.postgresql.org/`.
- **Postgres-XC**: Most replication solutions fail to address scaling up write performance. It's rarely as important as scaling up reads and is a much harder to deal with. Postgres-XC attacks that problem head-on by building a fully distributed cluster of nodes that provides near linear write scaling. The software is both very complicated and still fairly new, and at the point this is being written, it isn't either feature complete nor considered stable for generic workloads yet; it only handles relatively simple work. It is available at `http://sourceforge.net/projects/postgres-xc/`.
- **Rubyrep**: This is similar to Bucardo in that it's an asynchronous replication solution that can handle multiple masters. As a fairly new piece of software, it still has a fairly simple learning curve and is being actively developed, but isn't yet very mature outside of its originally targeted application types. It is available at `http://www.rubyrep.org`.
- **Golconde**: A queue-based data distribution system, Golconde aims to scale upward database workloads by handling the chores normally done by replication software at the application level instead. Rather than derive transactions from what's in the database itself, it aims to be a database aware transaction passing mechanism, using one of the many generic message queue solutions available. Find it at `http://code.google.com/p/golconde/`.

The number of external projects is expected to shrink following the release of PostgreSQL 9.0, due to its internal streaming replication with associated hot standby query taking over as the obvious built-in way to handle some types of replication.

Replications solution comparison

Replication is very much an area where there is no one-size fits all solution. You need to match the solution chosen to what your specific problems and prioritizations are. The main replication options break down based on what method they use to replicate and whether that replication is synchronous or not:

Program	Replication Method	Synchronization
WAL Shipping	Master/Slave	Asynchronous
Slony	Master/Slave	Asynchronous
Londiste	Master/Slave	Asynchronous
Mammoth	Master/Slave	Asynchronous
Bucardo	Master/Slave or Master/Master	Asynchronous
Rubyrep	Master/Slave or Master/Master	Asynchronous
PgCluster	Master/Master	Synchronous
Postgres-XC	Master/Master	Synchronous
pgpool-II	Statement-based middleware	Synchronous

Summary

Hot standby allows us to run queries against the type of standby databases used in early versions only for high availability failover. The Write-Ahead Log used for creating standby databases logs block changes, and can only be used to make a complete copy of a database cluster, not a subset. Using log-shipping replication will result in some archiving lag between the standby and the master, but this is resolved in synchronous replication. Replicating database statements instead is more flexible and allows features such as version upgrades, but some statements don't replicate easily and this approach has higher overhead than write-ahead log shipping. Slony, Londiste, and Bucardo are mature solutions for replications. pgpool-II can be used to do load balancing across multiple read-only slaves, as well as some forms of synchronous replication. There are many additional replication tools for PostgreSQL available that target specific feature sets not covered by the more popular packages.

The next chapter will cover the partitioning concepts in PostgreSQL. If a table becomes huge, it becomes necessary to divide it into smaller tables. PostgreSQL provides the portioning concept where, without changing the application, the table can be divided into smaller tables.

15
Partitioning Data

The previous chapter elaborated on replication and scalability; however, as databases grow, it's common to have a table or two become unmanageably large. If the table itself is much larger than physical memory, and even its indexes stop fitting comfortably, query execution times will escalate. One way you can deal with large tables is to partition them, which breaks the table into a series of smaller, related tables instead. You don't have to change your application, just keep querying the same table as before. But when the query can be answered by just using a subset of the data, that optimization can occur, rather than scanning the whole thing. This is purely a performance improvement feature, and does not require any change to the application or queries. It only works well in case of large or, one can say, huge tables.

Partition is only beneficial in cases when the size of a table exceeds the size of the physical memory of the PostgreSQL.

In this chapter we will be covering these topics:

- Table inheritance
- List and range partitioning
- Horizontal partitioning

Table inheritance

Inheritance is an object-oriented concept when a subclass inherits the properties of its parent class. PostgreSQL adopts this concept for object-oriented databases. In this case, a table inherits from its parent table. In this example, there is a parent table called `vehicle`, which has some properties represented in columns. The child table `car` inherits the table `vehicle` because it has the same properties and an additional property (passenger capacity). The query on the `car` table shows not only the column of table `car` but also the columns of its parent table `vehicle`:

```
postgres=# CREATE TABLE vehicle (reg_no TEXT, model CHAR(4), company TEXT);
CREATE TABLE
postgres=# CREATE TABLE car (passenger INTEGER) INHERITS (vehicle);
CREATE TABLE
postgres=# INSERT INTO car VALUES('1234-ML', '1999', 'FORD', 2);
INSERT 0 1
postgres=# INSERT INTO car VALUES('4567-ML', '2000', 'FORD', 4);
INSERT 0 1
postgres=# INSERT INTO car VALUES('8901-ML', '2001', 'FORD', 6);
INSERT 0 1
postgres=# \d+ vehicle
                          Table "public.vehicle"
 Column  |     Type      | Modifiers | Storage  | Stats target | Description
---------+---------------+-----------+----------+--------------+-------------
 reg_no  | text          |           | extended |              |
 model   | character(4)  |           | extended |              |
 company | text          |           | extended |              |
Child tables: car

postgres=# \d+ car
                             Table "public.car"
  Column   |     Type      | Modifiers | Storage  | Stats target | Description
-----------+---------------+-----------+----------+--------------+-------------
 reg_no    | text          |           | extended |              |
 model     | character(4)  |           | extended |              |
 company   | text          |           | extended |              |
 passenger | integer       |           | plain    |              |
Inherits: vehicle

postgres=# SELECT * FROM car;
 reg_no  | model | company | passenger
---------+-------+---------+-----------
 1234-ML | 1999  | FORD    |         2
 4567-ML | 2000  | FORD    |         4
 8901-ML | 2001  | FORD    |         6
(3 rows)
```

dellstore2 database

Returning to the dellstore2 example database used in the query optimization chapter, consider the structure of the `orders` table:

```
dellstore2=# \d orders;
                            Table "public.orders"
   Column    |     Type      |                         Modifiers
-------------+---------------+------------------------------------------------------------
 orderid     | integer       | not null default nextval('orders_orderid_seq'::regclass)
 orderdate   | date          | not null
 customerid  | integer       |
 netamount   | numeric(12,2) | not null
 tax         | numeric(12,2) | not null
 totalamount | numeric(12,2) | not null
Indexes:
    "orders_pkey" PRIMARY KEY, btree (orderid)
    "ix_order_custid" btree (customerid)
Foreign-key constraints:
    "fk_customerid" FOREIGN KEY (customerid) REFERENCES customers(customerid) ON DELETE SET NULL
Referenced by:
    TABLE "orderlines" CONSTRAINT "fk_orderid" FOREIGN KEY (orderid) REFERENCES orders(orderid) ON DELETE CASCADE
```

Imagine that, after many years of operation, the store has received so many orders that queries against this table are unwieldy. The usual rule of thumb thresholds for considering table partitioning are when an individual table is larger than the total amount of memory in the server, or when it's reached 100 million rows.

Partitioning in PostgreSQL

Partitioning in PostgreSQL is based on the database's table inheritance feature. This allows a table to have children that inherit all of its columns. The current version of PostgreSQL supports only two types of partitioning--range and list partitioning.

Range partitioning

In range partitioning, the table is partitioned into ranges. Each non-overlapped range consists of single or multiple columns representing a single table. For this example, a partition is needed for each quarter that inherits from the main `orders` table:

```
dellstore2=# CREATE TABLE orders_2004_q1
(CHECK ( orderdate >= DATE '2004-01-01' and orderdate < DATE '2004-04-01'))
INHERITS (orders);
CREATE TABLE
dellstore2=# CREATE TABLE orders_2004_q2
(CHECK ( orderdate >= DATE '2004-04-01' and orderdate < DATE '2004-07-01'))
INHERITS (orders);
CREATE TABLE
dellstore2=# CREATE TABLE orders_2004_q3
(CHECK ( orderdate >= DATE '2004-07-01' and orderdate < DATE '2004-10-01'))
INHERITS (orders);
CREATE TABLE
dellstore2=# CREATE TABLE orders_2004_q4
(CHECK ( orderdate >= DATE '2004-10-01' and orderdate < DATE '2005-01-01'))
INHERITS (orders);
CREATE TABLE
```

However, only the column structure is inherited. You'll need to add indexes, constraints, and adjust permissions on each individual partition to match the master table. The output from `psql\d` for the table, as shown for the preceding `orders` table, can be a helpful guide as to what all of those are.

Each partition needs the same primary key:

```
dellstore2=# ALTER TABLE ONLY orders_2004_q1
             ADD CONSTRAINT orders_2004_q1_pkey PRIMARY KEY (orderid);
ALTER TABLE
dellstore2=# ALTER TABLE ONLY orders_2004_q2
             ADD CONSTRAINT orders_2004_q2_pkey PRIMARY KEY (orderid);
ALTER TABLE
dellstore2=# ALTER TABLE ONLY orders_2004_q3
             ADD CONSTRAINT orders_2004_q3_pkey PRIMARY KEY (orderid);
ALTER TABLE
dellstore2=# ALTER TABLE ONLY orders_2004_q4
             ADD CONSTRAINT orders_2004_q4_pkey PRIMARY KEY (orderid);
ALTER TABLE
```

That will create an index by `orderid` as well. A manual index on `customerid` is also needed:

```
dellstore2=# CREATE INDEX ix_orders_2004_q1_custid ON orders_2004_q1 USING btree (customerid);
CREATE INDEX
dellstore2=# CREATE INDEX ix_orders_2004_q2_custid ON orders_2004_q2 USING btree (customerid);
CREATE INDEX
dellstore2=# CREATE INDEX ix_orders_2004_q3_custid ON orders_2004_q3 USING btree (customerid);
CREATE INDEX
dellstore2=# CREATE INDEX ix_orders_2004_q4_custid ON orders_2004_q4 USING btree (customerid);
CREATE INDEX
```

Each order also contains a foreign key constraint to ensure the customer referenced is valid. Those need to be applied to each partition:

```
dellstore2=# ALTER TABLE ONLY orders_2004_q1
             ADD CONSTRAINT fk_2004_q1_customerid FOREIGN KEY (customerid)
             REFERENCES customers(customerid) ON DELETE SET NULL;
ALTER TABLE
dellstore2=# ALTER TABLE ONLY orders_2004_q2
             ADD CONSTRAINT fk_2004_q2_customerid FOREIGN KEY (customerid)
             REFERENCES customers(customerid) ON DELETE SET NULL;
ALTER TABLE
dellstore2=# ALTER TABLE ONLY orders_2004_q3
             ADD CONSTRAINT fk_2004_q3_customerid FOREIGN KEY (customerid)
             REFERENCES customers(customerid) ON DELETE SET NULL;
ALTER TABLE
dellstore2=# ALTER TABLE ONLY orders_2004_q4
             ADD CONSTRAINT fk_2004_q4_customerid FOREIGN KEY (customerid)
             REFERENCES customers(customerid) ON DELETE SET NULL;
ALTER TABLE
```

The other constraint involved here is actually against the orderliness table, confirming that each order exists. So long as we're careful to never remove an order while working on the partitioning, that constraint can stay in place without modifications. If the table were being dumped and reloaded, you'd have to drop that constraint while that was going on, lest the constraint be violated and cause a problem.

Determine a key field to partition over

There are two potential ways that you could split this data into smaller pieces. The first would be to partition the table into sections based on the `orderid` field. This is probably what a real-world deployment would need to do here, because if the `orders` table is too large, the `orderlines` table would be even larger. Both tables could usefully be partitioned by `orderid`.

However, imagine that orders are only kept for a period of time, perhaps a couple of years. Older orders could therefore be deleted in that situation. The problem with mass deletion in PostgreSQL is that it leaves a lot to be cleaned up afterwards. First, you'll need to vacuum the table to mark the deleted rows as dead. This might return space to the operating system, but it's quite possible it will not. You can also end up with bloated indexes from this usage pattern if data from various time periods is mixed together in the same data block.

If the `orders` table were partitioned by the `orderdate` field, there's another way to deal with removing old orders. You can just `DROP` the partitions containing the old data instead. Because this particular pattern is so common--timestamp ordered data where the old sections are dropped to keep access times reasonable to newer ones--the example we'll walk through here will partition by the date the orders were placed.

Another important part of partitioned design is that in order to benefit from splitting the table into pieces, queries need to run against a useful subset of the data. You should consider what fields are being used in `WHERE` clauses for queries against the table as part of deciding which field is the right one to split it over.

Sizing the partitions

A useful initial step to breaking a table into partitions is to figure out the range of data it contains, relative to the candidate field, and how large it is:

```
dellstore2=# SELECT min(orderdate),max(orderdate) FROM orders;
    min     |    max
------------+------------
 2004-01-01 | 2004-12-31
(1 row)

dellstore2=# SELECT relname,relpages FROM pg_class WHERE relname LIKE 'orders' ORDER BY relname;
 relname | relpages
---------+----------
 orders  |      100
(1 row)
```

This is obviously much too small to be worth partitioning, but for a demonstration sample it's large enough to demonstrate how splitting the table would look. Since there's a year of data here, breaking that into month-sized pieces would be appropriate.

List partitioning

The example here, and the pattern you're most likely to deploy, uses range partitioning. This is where you provide a non-overlapping range of values that each partition holds uniquely. It's also possible to partition based on an explicit list of values that direct which partition a value goes into. For example, if you had queries against the customers table that were routinely targeted at specific states, you might partition based on that:

```
CHECK ( state IN ( 'AK', 'AL', 'AR','AZ' ))
CHECK ( state = 'CA' )
```

Such that populous states like **California** (**CA**) have their own partition.

Note whether a partition is a range or a list is a choice of descriptive terminology. There's no syntax difference between the two; and to the query planner, they're all just constraints that a value is or isn't inside of.

Redirecting INSERT statements to the partitions

Now that the structure is present, the next step is to make rows inserted into the parent table instead go into the appropriate partition. The recommended way to do that is with a trigger function:

```
dellstore2=# CREATE OR REPLACE FUNCTION orders_insert_trigger()
RETURNS TRIGGER AS
$$
BEGIN
    IF ( NEW.orderdate >= DATE ('2004-10-01') AND NEW.orderdate < DATE ('2005-01-01') ) THEN
       INSERT INTO orders_2004_q4 VALUES (NEW.*);
    ELSIF ( NEW.orderdate >= DATE ('2004-07-01') AND NEW.orderdate < DATE ('2004-10-01') ) THEN
       INSERT INTO orders_2004_q3 VALUES (NEW.*);
    ELSIF ( NEW.orderdate >= DATE ('2004-04-01') AND NEW.orderdate < DATE ('2004-07-01') ) THEN
       INSERT INTO orders_2004_2 VALUES (NEW.*);
    ELSIF ( NEW.orderdate >= DATE ('2004-01-01') AND NEW.orderdate < DATE ('2004-04-01') ) THEN
       INSERT INTO orders_2004_q1 VALUES (NEW.*);
    ELSE
RAISE EXCEPTION 'Error in orders_insert_trigger():  date out of range';
     END IF;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
CREATE FUNCTION
```

Note how the function starts with rows at the end of the range of last quarter Q4. Starting with the latest defined partition is recommended practice, because in most business scenarios with a split by date, that's the partition you are most likely to be inserting new data into.

Once the function is created, it needs to be called each time a row is inserted:

```
CREATE TRIGGER insert_orders_trigger
  BEFORE INSERT ON orders
  FOR EACH ROW EXECUTE PROCEDURE orders_insert_trigger();
```

You will probably end up needing to update the actual function here to reflect new partitions at some point, but that doesn't require creating the trigger again. It will use the new function automatically once it's replaced.

Dynamic trigger functions

The `orders_insert_trigger()` function shown above is static: the statements it executes are the same every time. As you might expect from reading the code, the actual execution time will vary based on which partition you are inserting into. Maintaining that trigger code is both monotonous and error prone. It's possible to remove the maintenance chore by just directly computing the partition required:

```
dellstore2=# CREATE OR REPLACE FUNCTION orders_insert_trigger()
RETURNS TRIGGER AS $$
DECLARE
ins_sql TEXT;
BEGIN
ins_sql := 'INSERT INTO orders_' || to_char(NEW.orderdate, 'YYYY') ||
           '_q' || date_part('quarter', DATE(NEW.orderdate)) ||
           '(orderid,orderdate,customerid,net_amount,tax,totalamount) VALUES ' ||
           '('|| NEW.orderid || ',' || quote_literal(NEW.orderdate) || ',' ||
          NEW.customerid || ' ' ||  NEW.netamount ||','|| NEW.tax || ',' ||
          NEW.totalamount || ')';
                          EXECUTE ins_sql;
RETURN NULL;
END
$$ LANGUAGE plpgsql;
CREATE FUNCTION
```

The execution time of this version is constant and predictable, whereas the static version's runtime will depend on how many comparisons happen before it finds the right partition to insert into. In return for that, and getting rid of the need to keep the more verbose static version up to date, there are a few downsides to this approach.

You have to be careful to handle quoting and `NULL` values here. Because of how the statement is constructed and then executed as text, this will be slower than the static version, at least when the static one has the active partition as the beginning of its comparison list.

A more subtle downside is that dynamically generating these statements will then accept bad data pointing toward partitions that don't exist, without the clear error message the static version gives in that case. Before using this approach on a production system, you should rewrite this example procedure to catch insertion errors, because rows inserted into partitions that don't exist are still going to fail.

Partition rules

There is actually another way to implement the partition redirection being done in the trigger here. PostgreSQL has a feature called **rules** that allows the substitution of an alternative command for one you want to change. Here is an example:

```
dellstore2=# CREATE RULE orders_2004_q1_insert AS
ON INSERT TO orders
WHERE (  orderdate >= DATE '2004-01-01'
AND   orderdate < DATE '2004-04-01' )
DO INSTEAD
    INSERT INTO orders_2004_q1 VALUES (NEW.*);
CREATE RULE
dellstore2=# CREATE RULE orders_2004_q2_insert AS
ON INSERT TO orders
WHERE (  orderdate >= DATE '2004-04-01'
AND   orderdate < DATE '2004-07-01' )
DO INSTEAD
    INSERT INTO orders_2004_q2 VALUES (NEW.*);
CREATE RULE
dellstore2=# CREATE RULE orders_2004_q3_insert AS
ON INSERT TO orders
WHERE (  orderdate >= DATE '2004-07-01'
AND   orderdate < DATE '2004-10-01' )
DO INSTEAD
    INSERT INTO orders_2004_q3 VALUES (NEW.*);
CREATE RULE
dellstore2=# CREATE RULE orders_2004_q4_insert AS
ON INSERT TO orders
WHERE (  orderdate >= DATE '2004-10-01'
AND   orderdate < DATE '2005-01-01' )
DO INSTEAD
    INSERT INTO orders_2004_q4 VALUES (NEW.*);
CREATE RULE
```

Using a rule has the potential upside that bulk inserts can be processed more efficiently than using a trigger. As such, benchmarks that focus on that particular measurement might suggest it's the faster approach. However, the rule approach has higher overhead for an individual insert, which is what you're probably going to do more of over the life of the database. And since the number of rules and therefore the rules overhead is proportional to the number of partitions, with no dynamic approach available, the performance of a rules-base partitioning implementation can become quite high once you have lots of partitions.

A final blow against using rules is that `COPY` doesn't follow them. If you're using `INSERT` triggers instead, those work fine. There's very little to recommend rules over partitions, and several reasons you might want to avoid them. The main reason I bring them up at all is to point out their issues, and to suggest you be skeptical of any recommendation in favor of using them in PostgreSQL.

Empty partition query plans

At this point, the partitions are all in place, but the data is still in the parent table. Since these are new tables the optimizer doesn't have any statistics on them until `ANALYZE` is run. Let's do that and see what a query against the `orders` table looks like now:

```
dellstore2=# ANALYZE;
ANALYZE
dellstore2=# EXPLAIN ANALYZE SELECT * FROM orders;
                                                
```

Date change update trigger

One of the optional things usually left out of PostgreSQL partitioning examples is an update trigger. Consider the case where you update a row and change the date. That could require relocating it to another partition. If you want to allow for that case, you need to install a trigger into each partition:

```
dellstore2=# CREATE OR REPLACE FUNCTION orders_2004_q1_update_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.orderdate != OLD.orderdate ) THEN
      DELETE FROM orders_2004_q1
        WHERE OLD.orderid=orderid;
      INSERT INTO orders values(NEW.*);
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
CREATE FUNCTION
dellstore2=# CREATE TRIGGER update_orders_2004_q1
     BEFORE UPDATE ON orders_2004_q1
     FOR EACH ROW
     EXECUTE PROCEDURE orders_2004_q1_update_trigger();
CREATE TRIGGER
```

As in the insert trigger case, you could instead write a dynamic version of this that constructed both the `DELETE` and `INSERT` statement. Then you can attach the same function to every partition. This has the same potential benefits and concerns as in the dynamic `INSERT` trigger shown earlier.

Starting in PostgreSQL 9.0, it's possible to write `UPDATE` triggers that are tied to a specific column changing. This technique could be used to slim down and speed up this example, by writing something that only executes when `orderdate` is modified.

When showing the rest of the examples below, it's assumed that you did not install this optional form of trigger. Few applications update data that impacts their primary key afterwards as it's dangerous, and you can always manually direct data to a specific partition yourself if you need to make such a change.

Live migration of a partitioned table

If you executed the partition creation, index, and trigger functions in this chapter against a live `dellstore2` installation, you've now got all of the actual data in the parent `orders` table, with a number of empty partitions. This is typically how a live migration to partitions would preferably happen, too. The other option is to dump the table, create the partition structure, then load the data back in again--which involves some downtime.

There is another way though--consider the update trigger again. What if you installed one of those against the parent table? You could then migrate to the partitioned structure just by updating every row. Here's what the code looks like for that:

```
dellstore2=# CREATE OR REPLACE FUNCTION orders_update_trigger()
RETURNS TRIGGER AS $$
BEGIN
    DELETE FROM orders WHERE OLD.orderid=orderid;
    INSERT INTO orders values(NEW.*);
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
CREATE FUNCTION
dellstore2=# CREATE TRIGGER update_orders
     BEFORE UPDATE ON orders
     FOR EACH ROW
     EXECUTE PROCEDURE orders_update_trigger();
CREATE TRIGGER
```

When doing this, the paranoid approach is to wrap all of the changes into a transaction block with `BEGIN`/`COMMIT`. That way, if there's a problem you can execute `ROLLBACK` and make all the changes revert. You'll either migrate all the data successfully or not do anything. This approach was used to debug the preceding code; each early attempt that didn't result in something that looked like sane data was just rolled back. Here's a sample session that succeeds:

```
dellstore2=# BEGIN;
BEGIN
dellstore2=# SELECT count(*) FROM orders;
 count
-------
 12000
dellstore2=# SELECT count(*) FROM orders_2004_01;
 count
-------
     0
```

```
dellstore2=# SELECT count(*) FROM orders_2004_12;
 count
-------
     0
dellstore2=# UPDATE orders SET orderid=orderid;
UPDATE 0
dellstore2=# SELECT count(*) FROM orders_2004_01;
 count
-------
  1000
dellstore2=# SELECT count(*) FROM orders_2004_12;
 count
-------
  1000
dellstore2=# SELECT count(*) FROM orders;
 count
-------
 12000
dellstore2=# COMMIT;
COMMIT
```

Counting the number of rows is a good way to confirm all of the data was migrated over as expected. You can see that the stating configuration had 12,000 rows in the parent table and none in the partitions, while the ending one has 1,000 rows in each of the 12 partitions. One small quirk to be aware of: the form of `UPDATE` trigger being used here doesn't actually report that it did anything. Note how it shows 0 rows processed, even though it obviously executed against all of them.

After the commit is done and any active sessions from before then have ended, you'll want to remove all the dead rows related to the original data. `VACUUM` isn't enough to clean this up, because all it will do is mark those rows as free space. That doesn't help you if you don't ever expect to add more rows to this parent table again. One easy way to fully clean the old parent is with `CLUSTER`, which should produce a new, tiny table from any data that's left behind (zero in this case), and then drop the original version with all the dead rows. Once this conversion is done and the data verified, this trigger is redundant and mainly apt to introduce confusion. It's better to drop it:

```
CLUSTER orders;
 DROP TRIGGER update_orders ON orders;
 DROP FUNCTION orders_update_trigger();
```

Than to keep it around once it's not needed anymore.

Partitioned queries

Now that the data is in the partitions, it's a good idea to update the statistics. It's also important to confirm that the `constraint_exclusion` feature is active:

```
ANALYZE;
SHOW constraint_exclusion;
constraint_exclusion
----------------------
partition
```

Constraint exclusion allows the query planner to avoid including partitions in a query when it can prove they can't provide useful rows to satisfy it. In PostgreSQL 8.4 and later, the default value of `partition` turns this `on` when partitioned tables are referenced, and `off` otherwise. In earlier versions, it defaults to `off`. You will have to turn the parameter `on` once you start using partitions in earlier versions if you expect them to work properly.

Now for an example that shows the value of partitions. Consider a query that only looks for orders on a particular day.

Instead of executing against every order, as this query would have before, now it only has to consider ones in the quarter 4 partition (`orders_2004_q4`) and a brief scan of the empty parent table. The optimizer can prove that none of the others contain rows with this value by considering their respective `CHECK` constraints:

```
dellstore2=# EXPLAIN ANALYZE SELECT * FROM orders WHERE orderdate='2004-11-16';
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Append  (cost=0.00..88.50 rows=36 width=30) (actual time=0.188..0.474 rows=35 loops=1)
   ->  Seq Scan on orders  (cost=0.00..0.00 rows=1 width=30) (actual time=0.004..0.004 rows=0 loops=1)
         Filter: (orderdate = '2004-11-16'::date)
   ->  Seq Scan on orders_2004_q4  (cost=0.00..88.50 rows=35 width=30) (actual time=0.183..0.461 rows=35 loops=1)
         Filter: (orderdate = '2004-11-16'::date)
         Rows Removed by Filter: 2965
 Planning time: 0.280 ms
 Execution time: 0.500 ms
(8 rows)
```

In this dataset, much like many real-world ones, the `orderid` key is highly correlated with the creation date. Watch how the following `orderid` range scan is executed:

```
dellstore2=# EXPLAIN ANALYZE SELECT * FROM orders WHERE orderid<2000;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------
 Append  (cost=0.00..354.00 rows=2004 width=30) (actual time=0.040..2.285 rows=1999 loops=1)
   ->  Seq Scan on orders  (cost=0.00..0.00 rows=1 width=30) (actual time=0.004..0.004 rows=0 loops=1)
         Filter: (orderid < 2000)
   ->  Seq Scan on orders_2004_q1  (cost=0.00..88.50 rows=2000 width=30) (actual time=0.036..0.611 rows=1999 loops=1)
         Filter: (orderid < 2000)
         Rows Removed by Filter: 1001
   ->  Seq Scan on orders_2004_q2  (cost=0.00..88.50 rows=1 width=30) (actual time=0.432..0.432 rows=0 loops=1)
         Filter: (orderid < 2000)
         Rows Removed by Filter: 3000
   ->  Seq Scan on orders_2004_q3  (cost=0.00..88.50 rows=1 width=30) (actual time=0.537..0.537 rows=0 loops=1)
         Filter: (orderid < 2000)
         Rows Removed by Filter: 3000
   ->  Seq Scan on orders_2004_q4  (cost=0.00..88.50 rows=1 width=30) (actual time=0.376..0.376 rows=0 loops=1)
         Filter: (orderid < 2000)
         Rows Removed by Filter: 3000
 Planning time: 0.238 ms
 Execution time: 2.494 ms
(17 rows)
```

The optimizer isn't smart enough to prove that no rows matching the `orderid` value appear in the higher numbered partitions. It does estimate only one row might match from those though, which is enough to have it do an index scan on those partitions, to quickly exclude them from the output. Meanwhile, the partitions that do have useful data are sequentially scanned.

In the original unpartitioned table, the optimizer would have needed to choose between a sequential or index scan for the entire data set. Neither one would have produced a very good plan, given this query returns about 16% of the data. You might recall from the query chapter that the transition point between the two types of scans is around 20%, which means a query returning 16% is going to be expensive to execute relative to the number of rows produced with either option. This is a common improvement when partitioning data: it lets the portions of the data set reduce in size enough that a sequential scan becomes a more viable and efficient means to query subsets of the data.

Creating new partitions

When you have live data coming in, you can expect that you'll one day need to add new partitions. The example here is only good for 2004 data. Once 2005 starts, inserts into the `orders` table with an `orderdate` from that year are going to fail.

You need to consider two things when the active partition is about to move forward into the next value. The first is whether your trigger function will support it. In cases where that's a static block of code, with a series of hard-coded comparisons, you may need to update it. The second thing is that the partition needs to be created and all its attributes properly set.

Scheduled creation

The simplest way to deal with creating the new partitions is to write a program that adds them, then make sure you're always at least one partition ahead of what's needed. On UNIX derived systems this is typically scripted into a `cron` job that executes more frequently than the partition changes, for example, run weekly given a monthly partitioning scheme. The Windows Task Scheduler could be used similarly.

Dynamic creation

At that point, you might be thinking that there's no need for scheduling partition creation. Why not just create them dynamically as needed? Indeed, that is an option. However, there are few elements of not necessarily obvious complexity to be wary of.

When two sessions both notice a partition is missing at the same time, it's extremely easy to end up with a race condition in the code, where both try to create it, and transaction deadlock is a common side effect of problems in this area.

Sometimes bad data will come into your table. It might come from far in the past or the future relative to right now. You might not want to create a partition for all of those situations.

There are many pieces to get right for partitioning to work. In addition to creating the partition, you need to add the right indexes, constraints, and permissions to the table. Allowing users, who are inserting data, enough power to set up all those things also introduces a class of security concerns better avoided.

Partitioning advantages

There are quite a few performance related improvements that partitioning this way provides. The average number of index blocks you'll have to navigate in order to find a row goes down, because the first level split there is pushed toward being the query planner's job instead. Also, as mentioned already, having smaller blocks of your data might alter when the database can consider a sequential scan of a range a useful technique.

There are some maintenance advantages too. You can `DROP` an individual partition, to erase all of the data from that range. This is a common technique for pruning historical data out of a partitioned table, one that avoids the `VACUUM` cleanup work that `DELETE` leaves behind. If you have monthly data, and you only keep a certain number of months at a time, once a new month starts you just `DROP` the oldest partition.

Another advantage is that REINDEX operations will happen in a fraction of the time it would take for a single giant index to build. If you only have limited windows of time when a REINDEX is practical, you might be able to squeeze in rebuilding an index or two during each of them, and eventually make your way through all of them. If you're finding that a maintenance window big enough to REINDEX your whole table just isn't available anymore, partitioning might get you back to where you can do it again, one subset at a time.

Common partitioning mistakes

There are enough partitioning problems that people run into repeatedly that it's worth mentioning the most common:

- Not turning on constraint_exclusion and therefore always including every partition.
- Failing to add all the same indexes or constraints to each partition that existed in the parent.
- Forgetting to assign the same permissions to each child table as the parent.
- Writing queries that don't filter on the partitioned key field. The WHERE clause needs to filter on constants. A parameterized query will therefore not work right at all; and functions that can change their value, like CURRENT_DATE, will also optimize poorly. In general, keep the WHERE clauses as simple as possible, to improve the odds that the optimizer will construct the exclusion proof you're looking for.
- Query overhead for partitioning is proportional to the number of partitions. It's barely noticeable with only 10 or 20 partitions, but if you have hundreds you can expect it to be significant. Keep the number of partitions to the two digit range for best performance.
- When you manually vacuum/analyze, those will not cascade from the parent. You need to specifically target each partition with those operations.
- Failure to account for out of range dates in the insert trigger. Expect that bad data will show up one day with a timestamp either far in the past or the future relative to what you have partitioned right now. Rather than throwing an error, some prefer to redirect inserted rows from outside of the partitioned range into a holding pen partition dedicated to suspicious data.

Another thing worth mentioning is that some people with **big database** backgrounds tend to partition whether or not they need to, just because that's the mindset their previous database work left them in. You have to put a fair amount of work into a partitioned PostgreSQL deployment. Make sure there's a true need for the partitions and a useful expected payback for the work to justify spending time on it first.

Horizontal partitioning with PL/Proxy

If splitting data among several sub-tables on a single server improves performance, then surely splitting it similarly among multiple servers would be even better, right? That's the theory behind **PL/Proxy**, a procedural language specifically designed to make that easier: `http://pgfoundry.org/projects/plproxy/`.

PL/Proxy was designed to fit the database scaling needs of Skype, which includes a target of serving a billion users at once. When you have that kind of user base, you just can't fit everyone on a single server.

The basic premise of PL/Proxy is that you first insulate access to the database behind database functions (also known as stored procedures). Let's say you want to grab a username field that uniquely identifies a user. Rather than selecting it from a table, instead you'd call a function that returns a username. This style of database design is popular for other reasons, too, because it allows the refactoring of the design of a database with minimal application impact. So long as the function call API remains stable, what that maps to in the database can be changed more easily.

The next thing to do is determine a way to break your workload up into pieces, similarly to how that decision is made for single server partitioning. The main difference is that you just need to identify a key field in most cases, not figure out how to map that to the underlying partitions; there is a standard, generic approach that will likely work for you there.

Getting PL/Proxy up and running requires a few software installation steps that are outside of PostgreSQL itself, and this chapter doesn't try and provide a complete walkthrough of those. The goal here is more to provide you with background about how the program works sufficiently to know when it could be considered useful to your scaling efforts, not to detail exactly how to set it up. An initial starter tutorial for the details is available at the project's home page: `http://plproxy.projects.postgresql.org/doc/tutorial.html`.

Hash generation

The standard practice for most PL/Proxy implementations is to partition based on the `hashtext` of the field you're splitting on, which allows splitting among a number of nodes fairly without knowing the distribution of the dataset in advance. `hashtext` is a PostgreSQL provided internal function that takes in any text input and generates an integer as output with a hash code. If you `AND` the result at the bit level, to only take the lowest few bits, that turns out to be a very quick and fair way to distribute load among multiple systems. For this to work, the number of partitions needs to be a power of 2 (2, 4, 8, 16, etc.) and then you bitwise `AND` against one less than that number. So for two partitions that's `& 1`, for four partitions it's `& 3`, and so on. Here's what the output from that looks like for the first ten integers hashed into four partitions:

```
SELECT s,hashtext(s::text) & 3 AS hash
FROM generate_series(1,10) AS s;
 s  |  hash
----+------
  1 |     1
  2 |     2
  3 |     0
  4 |     0
  5 |     2
  6 |     0
  7 |     1
  8 |     0
  9 |     0
 10 |     3
```

This shows you that there's a fairly equal distribution among the four possible partition values (`0,1,2,3`) available in this subset. If you look at the count of the various hash values for a larger subset:

```
SELECT hash,count(*) FROM (SELECT s,hashtext(s::text) & 3 AS hash
FROM generate_series(1,10000) AS s) AS q
GROUP BY hash ORDER BY hash;
 hash | count
------+-------
    0 |  2550
    1 |  2411
    2 |  2531
    3 |  2508
```

This shows you how using a few bits from the `hashtext` can provide a very equal split among partitions without any specific knowledge of your data.

The values returned by the `hashtext` function changed between PostgreSQL 8.3 and 8.4. If you're running version earlier than 8.4, you'll see different output from the above. Upgrading such a system to a newer PostgreSQL version also needs to respect that change--the resulting partition layout will be different by default.

The only reason this approach isn't more popular for splitting up key ranges in single node PostgreSQL partitioning is that the planner doesn't know how to do constraint exclusion based on the `hashtext` function.

Scaling with PL/Proxy

Given that hash-based distribution allows splitting among an arbitrary power of two partitions with relatively even loads, PL/Proxy makes it straightforward to split large independent tables among multiple nodes in a way that allows almost unbounded scalability.

The key word here is *independent*. If in fact your queries commonly cross partition boundaries, the query optimizer will not be able to help you generate good plans for those. PL/Proxy calls can be configured to run on any available node or on every node. If you need data from all of them, that's going to turn into a number of independent queries that need to be assembled with `UNION ALL` in order to get the full result. That may not execute as efficiently as if all the data were on a single node, with a unified set of statistics.

If your application is such that it can be split into distinct independent datasets, one of the unique aspects of PL/Proxy scaling is that it works on both read and write workloads. Many database scaling techniques only improve read scalability. The fact that PL/Proxy-based scaling has worked for database expansion for heavy PostgreSQL users, like Skype and myYearbook, shows how powerful a scaling technique it can allow. If you believe your application may become large enough to need this sort of approach, the main thing you need to get started on early is to practice insulating database access through functions. Once using that technique, whether the result is then executed on a single server, or multiple ones via PL/Proxy, will be mostly transparent to your application.

Sharding

Sharding is a form of horizontal partitioning, and can be implemented in PostgreSQL using PL/Proxy. Its main distinction is that sharding puts emphasis on placing shared data structures on every node, so that each of them can operate independently of one another. Typically the data you'd want on every system are what's normally referred to as "dimension", "lookup", or "fact tables" for your application: the pieces that are common to every set of data. The idea is that you should approach a shared-nothing architecture, where each sharded partition is self-sufficient for answering queries related to the data it contains.

The complexity of a sharding style design includes things like how to coordinate update to those shared structures. The approach that makes sense there is very application specific. Using a standard replication approach to keep all the copies up to date is a common design choice. The component in Skype's scaling stack to handle that is their Londiste replication program. Like any asynchronous replication solution, and like most sharded database deployments, this introduces an element of database inconsistency, where data is present in one database but not the rest, and you need to be aware of this. It's easy to introduce a race condition into such a design if you're not careful. Also, if an individual client moves between two copies of the database, they can run into a disconcerting situation where they are moving forward and backwards in time, relative to changes made to the shared structures. You may need to code something in your application to prefer a particular database's copy of the shared data, once you've used it once in a session.

Sharding style database scaling is a useful refinement on standard horizontal partitioning, but any time you have independent nodes sharing data there's additional complexity and potential for application bugs. The sort of consistency guarantees you eventually get are very different from the transactional ones a database normally provides.

Scaling with GridSQL

Another way you might be able to approach hash-based distribution among nodes is using GridSQL: `http://www.enterprisedb.com/community/projects/gridsql.do`.

Rather than putting the logic for what node to execute against directly into the database, GridSQL instead provides a client that speaks to a node coordinator process that handles that job. This process is all relatively transparent to you. The main catch is that not all PostgreSQL features are available, and you need a client driver compatible with GridSQL.

GridSQL uses its own parser, planner, and optimizer, accepting ANSI standard SQL but not all of PostgreSQL's syntax. The driver interface for your applications to talk to the database can look like a standard PostgreSQL one, but not everything a full PostgreSQL application expects will be supported. In stark contrast to PL/Proxy style scaling, rather than relying heavily on database functions, GridSQL doesn't support running them at all.

Right now the primary driver available is a Java JDBC level one. If you have a Java application you want to scale across multiple nodes, GridSQL allows a parallel query against multiple servers, or even against multiple database instances on a single server with many cores. Getting such a deployment working will be relatively little work, compared to building something similarly complicated using most other solutions for this problem. This architecture is also well suited for a sharded configuration, where dimension tables are replicated across nodes to make it easy for a single node to fully answer common queries.

This parallel query feature makes GridSQL most appropriate for building data warehouse style database applications, where a long query time is the main performance issue. It's not aimed to accelerate write-heavy transactional loads.

If you have a straightforward Java application and don't need database functions, GridSQL can provide a useful way to scale data warehouse style loads onto multiple cores or nodes. However, if you're using other client programming languages, or doing sophisticated database side programming, it would be much harder to convert your application to using its approach.

Summary

Every form of partitioning in PostgreSQL currently requires a moderate amount of manual setup. Accordingly, it's not something you want to do by default. There needs to be sufficient justification in terms of expected improvement in scalability before partitioning your tables will make sense. When it is effective, partitioning can dramatically speed up some types of queries against the database, improving overall system efficiency. Partitioning large tables normally makes sense when they, or the active portion of a database as a whole, exceed the total amount of memory in the server.

The choice of which key field to partition over needs to include the careful review of all queries made against the table, to make sure they are selective against that field. Individual partitions need to be created manually, as well as any constraints, indexes, or permissions against the parent applied to them.

Trigger functions are the easiest way to redirect `INSERT`, `UPDATE`, or `DELETE` statements to move the resulting row to the correct partition.

Partitioning improves queries by reducing the number of rows considered, lowering index sizes, and allowing sequential scans where only an index scan would have been appropriate before. Some database maintenance operations might be improved by partitioning. You'll get quicker individual REINDEX times and the ability to DROP old partitions to remove them almost instantly, rather than using `DELETE` against all of the old rows. You need to be vigilant in creating partitions before they are used. This is normally a scheduled operation, because trying to create the partitions at insert time is tough to do correctly. The upper limit on the number of partitions you can use effectively is tens of them. Anything using significantly more than 100 partitions can expect to suffer some from the overhead of needing to evaluate so many partition constraints whenever a query executes.

The next chapter is about the common database mistakes while loading and selecting from the database.

16
Database Profiling

Sometimes figuring out why your code isn't working properly is best done by diving into the database itself looking for bottlenecks in its code. There are a few techniques available for this.

We discuss the different profiling tools in this chapter:

- Profiling using gprof,
- Profiling using OProfile
- Debugging using Valgrind
- Profiling using DTrace

Profiling using gprof

The standard GNU profiler, `gprof`, is available for most UNIX-like systems. If you compile the PostgreSQL server using the `--enable-profiling` option, it will produce a `gmon.out` file that can be given to `gprof` for verbose profiling of the server internals.

The main issue with `gprof` profiling and PostgreSQL is that it's known to have problems when tracing into functions in loadable libraries. Also, it's hard to combine the profiling information with a look at what the underlying operating system is doing. These two limitations combine to make `gprof` only suitable for profiling fairly simple and pure database operations.

Download the PostgreSQL code and compile the code with profiling. You can always download PostgreSQL code from using git:

```
# git clone git://git.postgresql.org/git/postgresql.git
# cd postgresql
```

```
$./configure --enable-profiling
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
checking which template to use... linux
checking whether to build with 64-bit integer date/time support... yes
checking whether NLS is wanted... no
checking for default port number... 5432
checking for block size... 8kB
checking for segment size... 1GB
checking for WAL block size... 8kB
checking for WAL segment size... 16MB
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking whether gcc supports -Wdeclaration-after-statement... yes
checking whether gcc supports -Wendif-labels... yes
checking whether gcc supports -Wmissing-format-attribute... yes
checking whether gcc supports -Wformat-security... yes
```

Run `gprof` to get the information, the `result.out` file has very detailed level information. Here is an example of some information from the `grprof` output:

```
$gprof -a postmaster data/gprof/38202/gmon.out >result.out
$
$more result.out
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00    33020     0.00     0.00  LWLockInitialize
  0.00      0.00     0.00     6721     0.00     0.00  SHMQueueInsertBefore
  0.00      0.00     0.00     5280     0.00     0.00  RestoreTransactionSnapshot
  0.00      0.00     0.00     4979     0.00     0.00  pg_wchar_strlen
  0.00      0.00     0.00     1868     0.00     0.00  SHMQueueInit
  0.00      0.00     0.00     1381     0.00     0.00  GUC_yylex
  0.00      0.00     0.00     1002     0.00     0.00  AtProcExit_LocalBuffers
  0.00      0.00     0.00      782     0.00     0.00  load_tzoffsets
  0.00      0.00     0.00      709     0.00     0.00  pg_tolower
  0.00      0.00     0.00      568     0.00     0.00  BufferShmemSize
  0.00      0.00     0.00      567     0.00     0.00  LockBufHdr
  0.00      0.00     0.00      567     0.00     0.00  ReadBufferWithoutRelcache
  0.00      0.00     0.00      567     0.00     0.00  finish_spin_delay
  0.00      0.00     0.00      504     0.00     0.00  ShmemAlloc
```

`OProfile` is a well-respected tool for profiling everything that happens on a Linux server. You need to have debugging information for your kernel, which can be tricky to obtain depending on your distribution. However, the basic data collection is simple after that, and contains everything from the OS to the highest level database operations. This can be download from `http://oprofile.sourceforge.net/download/`.

`OProfile` is the preferred tool for many who work on PostgreSQL to find bottlenecks and hotspots in the code under Linux. An excellent introduction to it, that also covers additional tools to extend its analysis power, is available at `https://wiki.postgresql.org/wiki/Profiling_with_OProfile`.

Debugging using Valgrind

Valgrind is a dynamic analysis tool to profile the code to detect memory and threading errors. Valgrind can be downloaded from `http://valgrind.org/`, and details can be found at `https://wiki.postgresql.org/wiki/Valgrind`. Here are some example how to use the Valgrind on PostgreSQL.

You can't use `gperf` and Valgrind at the same time. You need to compile the PostgreSQL source code without profiling:

```
$ valgrind --leak-check=no --gen-suppressions=all --time-stamp=yes --log-file=$HOME/pg-valgrind/%p.log --trace-children=yes  /usr/local/pgsql/bin/postg
res --log_line_prefix="%m %p " --log_statement=all --shared_buffers=64MB 2>&1 | tee $HOME/pg-valgrind/postmaster.log
2017-04-11 02:22:05.856 PDT 50476 LOG:  database system was interrupted; last known up at 2017-04-11 02:20:42 PDT
2017-04-11 02:22:05.988 PDT 50476 LOG:  database system was not properly shut down; automatic recovery in progress
2017-04-11 02:22:06.005 PDT 50476 LOG:  invalid record length at 6/330001E8: wanted 24, got 0
2017-04-11 02:22:06.006 PDT 50476 LOG:  redo is not required
2017-04-11 02:22:06.071 PDT 50476 LOG:  MultiXact member wraparound protections are now enabled
2017-04-11 02:22:06.118 PDT 50435 LOG:  startup process (PID 50476) was terminated by signal 27: Profiling timer expired
2017-04-11 02:22:06.126 PDT 50435 LOG:  aborting startup due to startup process failure
2017-04-11 02:22:06.134 PDT 50435 LOG:  database system is shut down
```

Here is another example usage of Valgrind:

```
$ valgrind --leak-check=no --gen-suppressions=all --time-stamp=yes --log-file=$HOME/pg-valgrind/%p.log --trace-children=yes  /usr/local/pgsql/bin/postg
res --log_line_prefix="%m %p " --log_statement=all --shared_buffers=64MB 2>&1 | tee $HOME/pg-valgrind/postmaster.log
2017-04-11 10:57:48.560 PDT 64287 LOG:  database system was interrupted; last known up at 2017-04-11 02:22:06 PDT
2017-04-11 10:57:48.678 PDT 64287 LOG:  database system was not properly shut down; automatic recovery in progress
2017-04-11 10:57:48.695 PDT 64287 LOG:  invalid record length at 6/33000258: wanted 24, got 0
2017-04-11 10:57:48.695 PDT 64287 LOG:  redo is not required
2017-04-11 10:57:48.754 PDT 64287 LOG:  MultiXact member wraparound protections are now enabled
2017-04-11 10:57:48.806 PDT 64285 LOG:  database system is ready to accept connections
2017-04-11 10:57:48.871 PDT 64291 LOG:  autovacuum launcher started
2017-04-11 10:59:08.957 PDT 64312 LOG:  statement: CREATE TABLE foo(a int);
2017-04-11 10:59:28.458 PDT 64312 LOG:  statement: INSERT INTO foo values(generate_series(1,1000));
```

The output of Valgrind looks like this:

```
$ more 64323.log
==00:00:00:00.000 64323== Memcheck, a memory error detector
==00:00:00:00.000 64323== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==00:00:00:00.000 64323== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==00:00:00:00.000 64323== Command: /bin/cp pg_xlog/000000010000000600000033 /home/ibrar/edb/backup/pg/archived_wals/000000010000000600000033
==00:00:00:00.000 64323== Parent PID: 64322
==00:00:00:00.000 64323==
==00:00:00:01.072 64323==
==00:00:00:01.072 64323== HEAP SUMMARY:
==00:00:00:01.072 64323==     in use at exit: 2,176 bytes in 2 blocks
==00:00:00:01.072 64323==   total heap usage: 47 allocs, 45 frees, 146,322 bytes allocated
==00:00:00:01.072 64323==
==00:00:00:01.072 64323== For a detailed leak analysis, rerun with: --leak-check=full
==00:00:00:01.072 64323==
==00:00:00:01.072 64323== For counts of detected and suppressed errors, rerun with: -v
==00:00:00:01.072 64323== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Visual Studio

Starting with PostgreSQL 8.4, on Windows the database can be built using Visual Studio. This means that it's possible to use Visual Studio tools, including the debugger and profiler in order to analyze the performance of the database code.

Profiling using DTrace

Sun's Solaris 10 release in 2004 introduced DTrace, perhaps the most comprehensive toolkit for profiling available on any operating system. The probes required to instrument code with DTrace have minimal overhead when they are not enabled, and the way they're inserted makes it possible to instrument a production system very safely. Making this level of power available to regular user level processes is also quite useful.

PostgreSQL must have been compiled with the `--enable-dtrace` option for its DTrace probes to be available. You could write an entire book just on how to use DTrace with PostgreSQL, one I'd happily write in the future. The main PostgreSQL manual section at `http://www.postgresql.org/docs/current/interactive/dynamic-trace.html` covers all the probe points available as of the current version. Links to additional utilities and documentation are available at `http://wiki.postgresql.org/wiki/DTrace`.

The DTrace implementation on Apple's Mac OS X is also very complete, including surprisingly good details about what's happening into the operating system itself. In addition to the command-line tools, Apple provides a good GUI interface to the DTrace data, called **Instruments**, as part of their XCode development tools.

DTrace on FreeBSD

As of the current FreeBSD 8.0, DTrace implementation in that operating system has two major missing features; The probes into the operating system itself aren't very comprehensive yet, and, so far, only the root can use DTrace. Both these limitations are being worked on, and FreeBSD may move closer to Solaris in terms of how capable it is for doing DTrace work in future releases.

Linux SystemTap emulation of DTrace

SystemTap is a Linux utility that allows dynamic tracing similar to DTrace. Starting in SystemTap V0.9, it includes header files that expose the same user space capabilities that DTrace does. This means that on a Linux system with that version of SystemTap or later, you can compile PostgreSQL using `--enable-dtrace` and it will hook into SystemTap using the DTrace probes.

Support for SystemTap varies quite a bit based on Linux distribution. The Fedora Core versions starting with FC10 include support for the PostgreSQL DTrace probe emulation, and that support has been included as part of RHEL 6 too. These RedHat systems are the most reliable source for SystemTap on Linux.

On Debian and Ubuntu, SystemTap with appropriate headers may be available, but support for the matching kernel debugging symbols has been very sporadic. It's not clear yet whether this will be considered a priority to resolve in future releases.

From a database administrator perspective, the amount of kernel level work required to make SystemTap install and instrument a running system makes it a less appropriate tool for use on a production server than DTrace. It just isn't as well designed for that purpose, particularly because it's easier to misuse in a way that's dangerous to the server.

Summary

Profiling is a good way to find the bottleneck of performances, and there are multiple tools available to profile PostgreSQL and the operating system. Directly profiling functions and other activities in the database, including snooping into what the operating system is doing under the hood, can find unexpected performance issues that no simple analysis would locate.

The next chapter is about common performance problems and how to tackle these problems.

17
Avoiding Common Problems

The previous chapter elaborated on the profiling tools in detail, which helps to detect the problem in our code. But there are some common problems we need to discuss here. Beyond the expected day-to-day setup and design issues, there are some recurring issues PostgreSQL administrators and developers run into regularly that don't have really obvious answers. PostgreSQL has Frequently Asked Questions (FAQ) lists available at `http://wiki.postgresql.org/wiki/Category:FAQ`, and you may find those interesting reading even though some of those topics are covered in this chapter too.

In this chapter, we will be covering following topics:

- Bulk loading
- Common performance problems

Bulk loading

There are two slightly different types of bulk data loads you might want to do. The first type, and the main focus of this section, is when you're initially populating an empty database: sometimes you also need to do later bulk loads into tables that are already populated. In that case, some of the techniques here, such as dropping indexes and constraints, will no longer be applicable. Also, you may not be able to get quite as aggressive in tuning the server for better loading speed when doing incremental loading. In particular, options that decrease integrity for the whole server, like disabling `fsync`, are something that only makes sense when starting with a blank system.

Loading methods

The preferred path to get a lot of data into the database is to use the `COPY` command. This is the fastest way to insert a set of rows. If that's not practical, and you have to use `INSERT` instead, you should try to include as many records as possible per commit, wrapping several into a `BEGIN/COMMIT` block. Most applications find between 100 and 1,000 rows per commit gives the majority of the performance possible in doing batch inserts, but it's still not going to be as fast as `COPY`.

From slowest to fastest, this is the usual ranking:

1. `INSERT` a single record at once.
2. `INSERT` larger blocks of records at one time.
3. Use a single `COPY` at a time--this is what a standard `pg_restore` does.
4. Multiple `INSERT` processes using larger blocks in parallel.
5. Multiple `COPY` processes--parallel `pg_restore` does this.

Note that doing multiple processes doing `INSERT` or `COPY` at once can be harder than expected in some programming languages. Process or threading limitations in the programming language can end up being the bottleneck, rather than the database itself.

Also, be aware that if using a regular programming language for this job, you may need to be very explicit about tuning off per-transaction autocommit in order to get proper batch insert behavior. It's also useful in some cases to use `PREPARE` to set up the `INSERT` statement, just to lower parsing overhead. Unlike a query, there's little potential for the plan to degrade when you do that.

External loading programs

Another thing to be aware of is that `COPY` will abort all its work if it finds any error in the input data. This can be frustrating if the bad row is near the end of a large input. If you're importing from an external data source (for example, a dump out of a non-PostgreSQL database), you should consider a loader that saves rejected rows while continuing to work anyway, such as pgloader (`http://pgfoundry.org/projects/pgloader/`). pgloader will not be as fast as `COPY`, but it's easier to work with on dirty input data, and it can handle more types of input formats too.

Another loader that is useful for some special cases is pg_bulkload (`http://pgbulkload.projects.postgresql.org/`). This can be even faster than straight COPY. This is due to very aggressive performance features like bypassing the WAL, which is possible in some cases, with regular `COPY` (covered in a later section of this chapter), and it also has some features for handling bad data. The `pg_bulkload` loader is a fairly invasive loader that requires the loading of custom functions into the database, and as such it's a little more intense to work with than the simpler, external only pgloader. Which one to use, or whether straight `COPY` is fine, really depends on the cleanliness of your data and exactly how fast you must have it loaded.

Tuning for bulk loads

The most important thing to do in order to speed up bulk loads is to turn off any indexes or foreign key constraints on the table. It's more efficient to build indexes in bulk and the result will be less fragmented. Constraint checks are much faster to check in bulk too.

There are a few `postgresql.conf` values that you can set outside of their normal range specifically to accelerate bulk loads:

1. `maintenance_work_mem`: Increase to a much larger value than you'd normally run your server with. 1 GB is not unreasonable on a server with more than 16 GB of RAM nowadays. This speeds up `CREATE INDEX` and `ALTER TABLE ADD FOREIGN KEY`, presuming you've followed the preceding advice to do those in a batch after loading.
2. `checkpoint_segments`: Much higher values than would normally be safe are acceptable here to spread out checkout I/O, because crash recovery time and disk space aren't so important before the server goes live. 128-256 are common values for bulk loading. Similarly, increasing `checkpoint_timeout` can also be advisable, with values of 30 minutes not being unreasonable in this context.
3. `autovacuum`: Having periodic `autovacuum` does little but get in the way of a standard bulk load, it's recommended that you turn this off.
4. `Wal_buffers`: This can easily turn into a bottleneck on a bulk load. Setting it to the practical upper limit of 16 MB is the recommended practice.
5. `synchronous_commit`: If your bulk load is an all or nothing affair, there's little downside to turning off synchronous commits, so that they are done in larger batches instead. If you crash and lose some transactions as a result, it's normally possible to just wipe out the last table being loaded and reload it from scratch.

6. `fsync`: The only time that it is reasonable to consider turning off fsync is when you are initially loading a database. In that situation, the main downside of no fsync, which is that a database crash can lead to corruption, is normally not fatal; you just start over with the loading again. Disabling fsync was particularly valuable on PostgreSQL 8.2 and earlier, before `synchronous_commit` was available, and on hardware that doesn't include a non-volatile write cache. It can make the difference between only loading a hundred records per second and loading thousands.

7. `vacuum_freeze_min_age`: Lowering this parameter to 0 is appropriate if you're going to manually `VACUUM` the database before putting it into production. It will maximize the amount of database cleanup that `VACUUM` can do.

Some of the preceding points presume you're doing a fairly dumb loading routine. If your loading process involves any table cleaning procedure that does joins and the like, you may need to keep `autovacuum`, in particular, on in order for those to work efficiently. It might be beneficial to increase `work_mem` as well in that situation. All of these memory parameters can potentially be set to much higher values than normal during loading, so long as you're not allowing clients to connect while loading is going on.

Skipping WAL acceleration

The purpose of the **write-ahead log** (**WAL**) is to protect you from partially committed data being left behind after a crash. If you create a new table in a transaction, add some data to it, and then commit at the end, at no point during that process is the WAL necessary. If the commit doesn't happen, the expectation is that the entire table has gone anyway. Accordingly, in recent PostgreSQL versions, this particular case is accelerated.

A bulk load like the following will not generate any WAL records by default, and can therefore execute quicker. You can get the same acceleration with a table that exists if you use `TRUNCATE` to clear it out first:

```
BEGIN;
CREATE TABLE t ...
COPY t FROM ...
COMMIT;
```

However, if you have turned on the `archive_command` facility to feed WAL segments to a PITR standby, this optimization is disabled. The WAL is in that case the only way to get the data over to the standby system, so it can't be skipped. In PostgreSQL 9.0, some settings of `wal_level` can disable this optimization too. Since archiving is often off or can be delayed until after initial bulk loading of a database, this trick can still be useful in the early stages of a standby configuration.

Recreate indexes and add constraints

Presuming you dropped indexes and constraints before loading data in, you'll now need to recreate them again. Index rebuild in particular can be very processor, memory, and disk intensive. If you have a large disk array and you're CPU limited, it's quite reasonable to consider running two or more such rebuilds in parallel. It's recommended that you have high values for `shared_buffers`, `checkpoint_segments`, `checkpoint_timeout`, and `maintenance_work_mem` to make indexing and constraint checking perform well.

Parallel restore

PostgreSQL 8.4 introduced an automatic parallel restore that lets you allocate multiple CPU cores on the server to their own dedicated loading processes. In addition to loading data into more than one table at once, running the `parallel pg_restore` will even usefully run multiple index builds in parallel. You just need to tell it how many "jobs" it can run at once, and once it's finished the basic table setup, it will try to keep that many things active at all times. You do have to use the custom database dump in just the right format for the parallel restore to use it.

`COPY` in PostgreSQL requires a pretty large amount of CPU to run, and it's actually quite easy to discover a single core is maxed out running it without keeping the disks busy. However, it is possible that parallel restore may just turn disk bound and not see much of a speed increase. The magnitude of the improvement is proportional to how fast your disks are. Also, if the bulk of your loading time is spent dealing with one very large table, this won't benefit very much from a parallel load either. The ideal case for it is where you have many similarly sized tables to load and/or index.

If you have a PostgreSQL 8.4 install available, it's possible to use it to dump and restore against older database versions too. You might connect to an 8.2 server, dump in the format parallel restore expects, and then use parallel restore against an 8.3 server. All of the improved smarts here are handled on the client side; they don't require specific server support. There have been reports of this working fine, but since it's normally preferable to dump with the same client version you expect to restore with though it's a bit risky. Consider using parallel `pg_restore` against older versions of something you want to carefully test before you rely on it for a production quality migration.

Post load clean up

Your data is loaded, your indexes recreated, your constraints active. There are two maintenance chores you should consider before putting the server back into production. The first is a must-do: make sure you run `ANALYZE` against all the databases. This will ensure you have useful statistics for them before queries start running.

You could also consider running a database-wide `VACUUM`, if you can stand the additional downtime during the load. The reason for this is that it will set the hint bit data for each page of the database, as described in *Unexplained writes*. You can even combine the two with `VACUUM ANALYZE` and get everything ready for good initial performance. It's possible to get `autovacuum` to chew away on that problem instead, but it will considerably slow the database until that's finished every table. Another potential improvement recommended here is making this wide `VACUUM` more aggressive about freezing old transactions, as the things you're loading are unlikely to get rolled back later. This is the spot when reducing the `vacuum_freeze_min_age` parameter in the `postgresql.conf` to 0, in order to make any manual post-load `VACUUM` as aggressive as possible, would be useful.

Once you've done as much of this maintenance as you can tolerate, make sure you also revert any loading specific optimizations you made to the postgresql.conf file before making the database live again. That may include turning back on `autovacuum` and changing your relative memory allocations to something tuned for clients, rather than loading.

Common performance issues

Many performance issues come from bad design or implementations that just don't fundamentally work well with PostgreSQL. There are a few areas where the problem is not so bad, it's more of a quirk with known workarounds. This section covers some of the more common problems new PostgreSQL users run into from that category.

Counting rows

It's not unusual to find an application that does the following to determine how many rows are in a table:

```
SELECT count(*) FROM t;
SELECT count(*) FROM t;
```

In some databases other than PostgreSQL, this executes very quickly, usually because that information is kept handy in an index or similar structure. Unfortunately, because PostgreSQL keeps its row visibility information in the row data pages, you cannot determine a true row count without looking at every row in the table, one at a time, to determine if they are visible or not. That's a sequential scan of the full table, and it's pretty slow; it even turns out to be an effective way to benchmark sequential read speed on a table to count its rows this way!

If your statement is fairly selective instead, as in the following example, this can execute quickly, presuming that the index scan on k is only returning a few rows. How fast a count runs is proportional to the number of rows returned:

```
SELECT count(*) FROM t WHERE k>10 and k<20;
```

If you need a row count and can't afford to wait that long, there are two alternative approaches. For situations that just need an approximate count, without taking every last bit of row visibility information into count, one is computed each time `ANALYZE` is run, either manually or via `autovacuum`. You can find the row estimate it computes, which is usually quite accurate unless your rows vary wildly in size, like this:

```
SELECT reltuples FROM pg_class WHERE relname='t';
```

If you were to have more than one table with the same name but a different namespace, this form is required:

```
SELECT reltuples
 FROM pg_class C
 LEFT JOIN pg_namespace N ON (N.oid = C.relnamespace)
 WHERE c.relname='t' and nspname = 'public';
```

Again, these values are only as accurate as your last analyze, so make sure those happen regularly if you expect this form of count to be useful.

The other approach you can consider is adding a trigger onto the table that tracks the change in count each time a row is inserted or deleted, then store the total that way. This total will always be exact, except for the usual fuzziness about whether the most recent transactions are visible or not. There is a list of code examples to build this onto your own app at `http://wiki.postgresql.org/wiki/Slow_Counting` that is worth looking at. At a somewhat higher level, maintaining a count can be considered a very simple materialized view, and some of the techniques useful for that can also be used here.

Unexplained writes

There are several situations where you can find yourself executing statements that only read against the database, yet find significant write volume happening on the database disk.

Parts of the buffer cache must be dirty before writes happen, so if any of that work is left hanging around it could be involved. Flushing everything out with a manual checkpoint is one way to assure that's not the cause of the writes.

A second small source of writes are the access time updates many operating systems do every time you read from a file. It's suggested these get turned off in `Chapter 4`, *Disk Setup*, and the actual volume of writes from that is pretty low anyway.

If the volume of writes is substantial, and this data was recently created, what you're mostly likely to run into are updates to the hint bits in the database. Hint bits are two bits stored in each row that indicate whether that row's associated transaction has committed or rolled back. If they aren't set, the database's commit log data (in `pg_clog` and possibly `pg_subtrans`) has to be consulted to determine whether the row is visible. Once going to all that trouble, the row is then updated with the correct hint bit information, so that expensive computation doesn't have to be done again. The result is now a dirty page that needs to be written out again, even though all you did was read it.

Executing something that checks every row in a table, including a full table `SELECT`, `COUNT`, or running `VACUUM` against it, will perform this visibility check and write out final hint bits for each row. Until that's happened, expect some steady write activity consisting of hint bit updates any time you're selecting against recently committed data. See `http://wiki.postgresql.org/wiki/Hint_Bits` for more information about the internals involved.

Don't forget about some simpler causes for writing when executing queries too: `autovacuum` running in the background after being triggered by recent activity can be writing things; and if your query is doing a sort operation or using temporary tables that spill over to disk, that will cause writes too.

Slow function and prepared statement execution

Prepared statements are a valuable technique that reject SQL injection into your applications as well as allowing you to bypass regular query parsing in order to save overhead. You might use one like this:

```
PREPARE getvals (int) AS SELECT * FROM t WHERE t.v=$1;
 EXECUTE getvals(5);
```

This returns everything in the rows with a matching v value. The `PREPARE` saves the output from the query parsing and planning stage. You might think that this will always be a win over directly executing the query if it's being executed more than once, because that overhead will then be amortized over more statements.

However, this isn't necessarily true. When a statement is prepared, the query optimizer can only produce a generic query plan for it, not knowing anything about the actual values that are going to be requested. When you execute a regular query, the statistics about each column available to the optimizer can sometimes do much better than this. Consider the case where the value v=5 only appears in 1% of the rows in the table, which the optimizer could know based on the statistics collected about this table. Had you executed a regular statement using that restriction in a `WHERE` clause, the optimizer would have strongly considered an index scan to only retrieve those small numbers of rows. But in a prepared statement context, where it had to choose the query plan at preparation time, it could only choose a generic plan that couldn't take advantage of statistics so detailed--it didn't know the exact value to match against yet.

In PostgreSQL 8.4, it's possible to use the new `QUERY EXECUTE...USING` syntax in order to get a newly planned query while still being safe from SQL injection. See `http://okbob.blogspot.com/2008/06/execute-using-feature-in-postgresql-84.html` for background about this new feature. In older versions, you can construct statements as strings, `quote_literal` to resist SQL injection, and then manually `EXECUTE` those to work around this issue; see `http://blog.endpoint.com/2008/12/why-is-my-function-slow.html` for an example.

PL/pgSQL benchmarking

PostgreSQL 8.4 added a feature to allow track statistics on how long each database function written in PL/pgSQL and similar languages take to execute. To enable this, edit your postgresql conf:

```
track_functions = all
```

The other options here is pl, which only tracks the pl/* language functions rather than all of them (i.e. not the ones written in C). Once that's done, you've executed some functions, and you've waited a moment for those to show up in the statistics collector, the following query will show you what executed:

```
SELECT * FROM pg_stat_user_functions;
```

In earlier versions, it was possible to build your own test harness using the technique described at `http://www.justatheory.com/computers/databases/postgresql/benchmarking_function.html`.

If, instead, you'd like to figure out which individual statements are responsible for the overhead, rather than just finding out how long the entire function takes, you'll need something more like a profiling tool for that. EnterpriseDB has released a PL/pgSQL debugger that includes profiling features. It comes bundled with their PostgreSQL installer, and you also may be able to install it separately on top of community PostgreSQL. There is also an interesting, albeit intrusive, way to build your profiling by hand, described at `http://www.depesz.com/index.php/2010/03/18/profiling-stored-proceduresfunctions/`, which is applicable to any PostgreSQL install.

High foreign key overhead

When you commit a transaction, if there's a foreign key involved, that commit is fairly expensive. As a percentage of total processing time, this is particularly bad if you're doing updates one at a time. To help improve this situation for batch updates, you can mark foreign key constraints as `DEFERRABLE`, either during `CREATE TABLE` or in a later `ALTER TABLE` to adjust it. The default is `NOT DEFERRABLE`. Note that this only impacts foreign keys and similar constraints implemented as triggers. It doesn't do anything for `CHECK` and `UNIQUE` restrictions, which cannot be deferred, and instead are always processed immediately, at least before PostgreSQL 9.0 that is. See the performance notes for v.9.0 at the end of this chapter for more information about additional options available in that version.

For deferrable constraints, there are again two options. If the check is labelled `INITIALLY IMMEDIATE`, it still happens immediately rather than at the end of the transaction. You can also modify the constraint so the default is instead `INITIALLY DEFERRED`, making the check only at transaction end.

In the case where a check is `DEFERRABLE` and it's `INITIALLY IMMEDIATE`, you can alter what a particular session does using `SET CONSTRAINTS`. Standard practice for foreign keys is to make them deferrable but default to the more proactive immediate behavior, then make batch updates process them in a deferrable transaction like this:

```
BEGIN;
SET CONSTRAINTS ALL DEFERRED;
[update or insert statements]
COMMIT;
```

This will batch all of the processing related to deferrable constraint checks into one block, which is much more efficient. There is a potential downside to be aware of here though: if your batch size is large when using this technique, the `COMMIT` can block for quite some time while chugging through all that work; and it's easy for other transactions to get stuck behind it. Overly large foreign key checks bunched together by deferring them is something to keep an eye out for, as a potential source for latency spikes in query activity.

Trigger memory use

If you've used some of the suggestions here to group your database commits into larger chunks, you might notice an expected downside to that: running out of memory at commit time. The issue is that if the table you're changing has an `AFTER` trigger on it, each statement due to trigger is inserted into an in-memory list. If there are a large number of them, perhaps due to a bulk loading job, it's possible to run out of the amount of memory on the system (or reach the limit for your user account) before the transaction can complete. Be careful when combining `AFTER` triggers with batch commits.

Heavy statistics collector overhead

One of the background database processes that shouldn't occupy too much of your system is the statistics collector, which looks like this from the perspective of ps:

```
Postgres 32450 32445 0 13:49 ? 00:00:00 postgres: stats
 collector process
```

Under normal circumstances, the statistics collection shouldn't take more than a small percentage of CPU time, even under heavy activity. There are occasional versions of PostgreSQL where this is not true. For example, 8.2.0 through 8.2.3 had a serious bug where statistics wrote far too often. However, even if you're running 8.2, you should never be using such an old minor release of it anyway.

Sometimes circumstances aren't normal enough, and the statistics collector takes up a lot more time than it normally does to run. This is usally associated with the statistics collector file itself becoming very large; here's a small one:

```
$ ls -l $PGDATA/pg_stat_tmp
-rw------- 1 postgres postgres 28961 2010-06-01 13:49 pgstat.stat
```

If your statistics collector is running slowly, and particularly if that's accompanied by the `pgstat.stat` file being very large, you should try resetting your database statistics. It's hard to say what **large** is here without knowing the size of your database; tracking the file size for a bit after reset may give you an idea what's appropriate for your system.

You might want to save interesting statistics such as `pg_stat_user tables` before resetting, just so that cumulative data there isn't lost. Statistics are reset like this:

```
$ psql -c "select pg_stat_reset();"
```

Once reset, the file should be small again. It should grow for a while then hit a steady state size. If it continues to grow without bound again, this is behavior you can consider reporting to one of the PostgreSQL mailing lists, as it's a problem for some users recently that has been difficult to replicate.

Targeted statistics resets

Starting in PostgreSQL 9.0, it's also possible to reset the `pg_stat_bgwriter` view using the following command:

```
$ psql -c "select pg_stat_reset_shared('bgwriter');"
```

However, this will never cause heavy statistics overload as it's not stored in a file that will get large. It's also possible to reset single statistics values in 9.0 using `pg_stat_reset_single_table_counters` and function stats with `pg_stat_reset_single_function_counters`, but if your issue is high statistics collector overhead you should really reset the whole thing instead.

Materialized views

One of the most effective ways to speed up queries against large data sets that are run more than once is to cache the result in a materialized view, essentially a view that is run and its output stored for future reference. Work on adding these into the core database is in progress, but like counting, it's possible to build them right now using features such as triggers. See `http://wiki.postgresql.org/wiki/Materialized_Views` for comments on the state of the art here, and pointers to code samples. If you have views derived from large tables in the database, few techniques are as effective as materializing those views and optimizing their updates for improving query performance.

Summary

One of the advantages of an open source community like the one around PostgreSQL is that it's easy to see what other people struggle with. Watch enough of that, and some trouble spots that aren't too difficult to avoid become obvious. Some of the biggest problems people run into are quite fundamental to PostgreSQL--getting `VACUUM` to work properly, making sure your application acquires locks properly--there's a long list of major things you need to get right. However, sometimes the little details can trip you up instead. The tips covered in this chapter, from bulk loading to profiling, might provide the right background trivia to make your use of the database simpler when dealing with the smaller problems, too. Sometimes simply using a newer version of PostgreSQL, where it's been engineered out of the database, is the answer to your problem.

Bulk loading works well using `COPY`, but alternative techniques, such as bulk inserts or even add-on tools, may be appropriate when dealing with less than perfect input data. You should also consider tuning your database server a bit differently when running a bulk load than you would for regular operations. PostgreSQL doesn't have any reliable way of knowing how many rows are visible in a table to accelerate counting them. Applications that expect that to be fast may need to use an alternative approach.

The next chapter will cover the performance features introduced in different releases of PostgreSQL.

18
Performance Features by Release

Since the start, PostgreSQL has provided performance features. A guide to the major changes in each PostgreSQL version starting with 7.4 is available at the Feature Matrix page: `http://www.postgresql.org/about/featurematrix`.

Since many performance-related changes are more internal, you won't necessarily see them there. The details are only in the release notes, which you can read the latest version of at `http://www.postgresql.org/docs/current/static/release.html`.

This section will help guide which features and settings aren't available in older (or are removed in newer) versions of PostgreSQL in a more compact form, targeted specifically at performance-related ones. First a digression into why this is more important than you might think is in order.

Beyond the expected day-to-day setup and design issues, there are some recurring issues PostgreSQL administrators and developers run into regularly that don't have really obvious answers. PostgreSQL has Frequently Asked Questions lists available at `http://wiki.postgresql.org/wiki/Category:FAQ`, and you may find those interesting reading even though some of those topics are covered in this chapter too.

In this chapter, we will cover following topics:

- Performance features introduced in different versions of PostgreSQL
- Brief description of performance features

Aggressive PostgreSQL version upgrades

One habit some database deployments adopt that is counterproductive with PostgreSQL is the idea that the version of the software used should be frozen forever once the system is validated as working. This is a particularly troublesome viewpoint for systems that are running into performance issues to adopt. The performance increases you'll find just from upgrading from older to newer PostgreSQL versions can be far larger than anything you can do just by tweaking the older version. Similarly, if you're running into a problem with how a specific query is executing, don't be surprised to find that's fixed in a later version of PostgreSQL, and upgrading to it is really your only option to obtain that fix.

There are a few reasons for the feature by version list given below. One is to make you aware of what your version of PostgreSQL can and can't do from a performance perspective, information that can be hard to accumulate on your own (the release notes are not verbose). Knowing what was improved in a later version can be a clue as to what's wrong with earlier ones. And in cases where your problem sounds like something resolved in a later version, that should at least raise the possibility to you that a database version upgrade is worth considering.

To give a very real example of how this regularly plays out, there are people quite happily running PostgreSQL 7.4, a popular version because it shipped with Red Hat Enterprise 4. And that version has continued to receive critical bug fixes and security patches up until very recently, even though it's 7-years old at this point. But if you're trying to optimize the performance of a 7.4 installation, something the PostgreSQL community regularly sees people trying to do with little success, that's time you're probably wasting. The effort would be better spent upgrading to a newer version. The same thing is true for PostgreSQL 8.0, and even 8.1 and 8.2 look pretty slow from the perspective of the current version at this point.

Performance features in version 8.1

PostgreSQL version 8.1 introduced some performance features which include partitioned tables performance, improvement in GiST, R-tree and some improvement in optimizer. This section describes only the performance features. For a full list of features introduced in version 8.1, look at the release notes of PostgreSQL:

```
https://www.postgresql.org/docs/8.1/static/release-8-1.html
```

- The shared buffer cache used to hold database data was redesigned for better concurrent use. This change is the major reason 8.1 is the earliest PostgreSQL version with good scalability.
- Bitmap scans allow combining multiple single-column indexes for queries that reference more than one column. This decreases the need to create computationally expensive multi-column indexes on tables.
- Autovacuum, formerly a module that had to be explicitly loaded, was integrated into the main server (but not started by default).
- Table partitioning is improved to allow skipping partitions that can't possibly satisfy a query.
- Indexes are automatically used for `MIN()` and `MAX()` queries.

Performance features in version 8.2

PostgreSQL version 8.2 introduced some performance features, such as query level performance and sorting performance and improvement in vacuuming. This section describes only the performance features. For a full list of features introduced in version 8.2 look at the release notes of PostgreSQL: `https://www.postgresql.org/docs/8.2/static/release-8-2.html`

Let's look at the features:

- Memory settings in the `postgresql.conf` file can be specified in standard units such as 1 MB, rather than as simply raw integers whose settings depends on the server units.
- Indexes can be created concurrently, without blocking writes to the table.
- You can watch the autovacuum daemon work in `pg_stat_activity`, and it records what it has done in views such as `pg_stat_user_tables`.
- Tables and indexes can have a `FILLFACTOR` that allows better clustering when inserting data out of order.
- The `seq_page_cost` configuration parameter was added, allowing easier fine tuning of how expensive the query planner thinks sequential and random I/O are.
- Several features were improved allowing more useful warm-standby server setups for disaster recovery failover.
- `COPY` can dump the results of an arbitrary query out, allowing that high-performance output path more targeted data replication capabilities. A faster binary `COPY` was also added.

- Initial support for DTrace was added. The number of things you can monitor with it is limited in this version though.

Performance features in version 8.3

PostgreSQL version 8.3 introduced some performance features, this section describes the only the performance features. For a full list of features introduced in version 8.3 look at the release notes of PostgreSQL: `https://www.postgresql.org/docs/8.3/static/release-8-3.html`

Let's look at the features in version 8.3:

- **Heap-Only Tuples** (**HOT**) allow quicker re-use of the dead space left behind when a row is updated or deleted, as long as that doesn't touch its indexed columns.
- Autovacuum is turned on by default, and multiple autovacuum processes can run at once.
- You can turn off `synchronous_commit` in order to speed up database commits, at the expense of introducing some potential for lost transactions.
- Checkpoints can be spread over a longer period of time, adjusted using the `checkpoint_completion_target` setting.
- The background writer in earlier versions was easy to configure so that it decreased performance rather than improving it. The new just-in-time background writer in this version is largely automatic and has fewer parameters to configure.
- `pg_stat_bgwriter` allows monitoring both the background writer and the writes done by the spread checkpoints.
- Sequential scans use an improved scheme that reduces their memory footprint, so that they're less likely to eject useful pages from the buffer cache as they execute. And if multiple scans are executing against the same table, they will synchronize in order to reduce duplication of reading. This is another reason to make sure you explicitly `ORDER` things that need to be sorted, because when synchronized scans kick in, the second client joining the scan will get rows starting with wherever the first one happens to be reading against.

- Read-only transactions are allocated virtual transaction ID numbers instead of real ones, reducing their overhead. The rate of transaction use and associated snapshot information can be inspected using a set of added functions that include `txid_current()` and `txid_current_snapshot()`.
- Temporary files can be moved using the `temp_tablespaces` parameter.
- Several types of activity that can correlate with performance issues can be more easily logged--autovacuum, checkpoints, and temporary file usage.
- All log activity can also be saved in the **Comma Separated Values** (**CSV**) output format, which makes them easier to process with external tools.
- If you comment out a previously set parameter in the `postgresql.conf`, starting in this version that will return it to the default; in earlier ones that just left it unchanged.
- Live and dead tuples (a tuple is essentially what most people think of as a row) are tracked in `pg_stat_user_tables`.
- Long-running transactions are easier to find in `pg_stat_activity`.
- The `archive_mode` parameter makes it easier to toggle on and off saving write-ahead log segments for a standby server.
- The `CLUSTER` command is improved to where it's a viable way to clean up after some situations where `VACUUM FULL` was the only alternative before.
- Performance of the Windows port was improved by building it and using memory in a more efficient manner for that platform.
- The `pg_buffercache` module can now display page usage counts, allowing more useful introspection of the buffer cache.
- The `pgbench` utility now saves latency information with a full timestamp for each transaction, enabling analysis such as graphing of the results.
- Indexing of `NULL` values is greatly improved, including the ability to control where they are placed relative to the rest of the table (beginning or end).

Performance features in version 8.4

PostgreSQL version 8.4 introduced some performance features, this section describes only the performance features. For a full list of features introduced in version 8.4, look at the release notes of PostgreSQL: `https://www.postgresql.org/docs/8.4/static/release-8-4.html`

Let's look into the features now:

- The free space map used to track space that used to belong to now deleted data is tracked on disk instead of in shared memory, which removes any limits on how large it can be. This removes needing to worry about the `max_fsm_pages` and `max_fsm_relations` parameters, which are removed in this version. It also makes it much less likely systems that are not being vacuumed frequently enough will end up in a situation where it's difficult to recover from that situation.
- The default value of the `default_statistics_target` parameter was increased from 10 to 100. This makes query planning time longer, and is responsible for a significant portion of the sometimes measured degradation in performance between 8.4 and 8.3 seen in many benchmarks. This mainly impacts trivial queries, however, in more real-world ones, having improved statistics to work with will improve performance by making it less likely the query planner will pick a particularly bad execution plan. It's always possible to reduce this parameter for 8.4 systems that don't need to plan more complicated queries.
- In earlier versions, queries running against partitioned tables would only know how to limit their search to individual partitions if the `constraint_exclusion` parameter was changed from its default value, because that overhead would otherwise apply to every query. A new `partition` setting for this parameter, now the default, allows partitioned queries to be supported without impacting the performance when a partition isn't involved.
- Bitmap index scans, often used when combining multiple relevant indexes used by a query, can do read-ahead set by the `effective_io_concurrency` parameter to improve performance. This feature was tested mainly on Linux. It can have a significant benefit on systems that support more than one disk operation at once, such as when a striped RAID volume is available.
- SQL `WINDOW` functions allow writing new types of queries far more efficiently than before, such as queries that need an idea of what the contents of nearby rows are.
- `WITH` clauses such as `WITH RECURSIVE` allow **Common-Table Expression** (**CTE**) programming. A common use of CTEs is to execute recursive query programs, which greatly simplifies and speeds up handling structures such as trees stored in the database.
- `pg_restore` can now do some operations in parallel to improving loading speed on systems that are limited there by CPU speed rather than disk throughput.

- The `auto_explain` feature makes it easy to log `EXPLAIN` output only for queries that take longer to run.
- A new `pg_stat_statements` add-in module allows for the tracking of execution statistics of all SQL statements executed by a server. This replaces several situations where parsing the database log files was needed to gather statement profiling information needed for query tuning.
- User-defined functions can be monitoring using `pg_stat_user_functions`.
- `EXPLAIN VERBOSE` gives a more useful set of information than before.
- SQL standard `LIMIT` and `OFFSET` queries are now supported, and they can use sub-selects.
- Autovacuum parameters can now be easily adjusted per table using the `CREATE TABLE` and `ALTER TABLE` storage parameter mechanism.
- `VACUUM` is more likely to make progress on systems with really long-running transactions. Partial `VACUUM` was also added, which allows processing to skip sections of a table that it knows cannot have anything to clean up left on them.
- It's possible to adjust some of the server block size parameters at build time using parameters passed to the `configure` script, rather than needing to edit the source code.
- The database statistics file, which is both written to often and not necessarily critical for all situations, can be run against a different location such as a RAM disk using the `stats_temp_directory` parameter. When this is enabled, a normal startup/shutdown procedure will save the data changes to disk, but some statistics could be lost in an unplanned crash.
- More DTrace probes were added, and now they work on platforms other than Solaris (such as Mac OS X).
- `pgbench` can be run for a period of time rather than just for a number of transactions. The names of the tables it uses now are all prefixed with `pgbench_` to avoid confusion with real user tables.
- Much smaller sizes of data might be compressed using the TOAST scheme.

Performance features in version 9.0

PostgreSQL version 9.0 introduced some performance features, this section describes only the performance features. There were so many changes to 9.0, they're being broken up by category for this section. As with all of the version summaries here, this is focusing on performance-related features, which represent only a fraction of the changes made in this version. Some major areas renovated in 9.0, such as big changes to database function languages PL/pgSQL, PL/Perl, and PL/Python, are completely ignored here. The goal here is to provide a quick (compared to the release notes at least) summary of the big performance-related pieces in only a few pages. For a full list of features introduced in version 9.0, look at the release notes of
PostgreSQL: `https://www.postgresql.org/docs/9.0/static/release-9-0.html`

Replication

Following is list of changes in replication area in this release:

- Asynchronous master/slave Streaming Replication is built-in to the database, without requiring as many external scripts to be configured.
- Replicated systems can be set up as Hot Standby ones that allow read-only queries to execute against them.
- The control over WAL archiving has been split. In addition to the existing `archive_mode` and `archive_command` settings, a new `wal_level` setting allows fine-tuning of how much information is written to the WAL.
- Monitoring of the replication progress made by a standby server that accepts Hot Standby queries can be done using the `pg_last_xlog_receive_location()` and `pg_last_xlog_replay_location()` functions.
- A new tool for removing older WAL archive files has been provided as `contrib/pg_archivecleanup`. This is expected to be called when using the new `archive_cleanup_command` parameter to maintain standby archive log directories. The program is always written so that it can be copied to and compiled for earlier PostgreSQL versions too, where this problem already existed, and was usually handled with less robust techniques than this program uses.

Queries and EXPLAIN

Here is the query and `EXPLAIN` related changed in this release:

- `EXPLAIN` plans can be output in XML, JSON, or YAML formats, to make it easier to analyze them with external tools (or to write such tools yourself).
- `EXPLAIN (BUFFERS)` allows monitoring the actual I/O done by a particular query, rather than just showing its costs.
- `EXPLAIN` output shows more information about how the increasingly useful hash nodes in query plans.
- Window functions have some new options such as `PRECEDING`, `FOLLOWING`, and starting with `CURRENT ROWS`, all of which allow writing some new types of windowed queries.
- Outer joins that reference tables not actually used to satisfy the query can now be removed from the query plan to improve their performance. Queries with that issue are commonly created by **Object-Relational Mapper (ORM)** software. This can happen when queries use only part of the data in a complicated view too.
- Each tablespace can be assigned its own sequential and random page costs via `ALTER TABLESPACE`. This allows tweaking those settings for disks known to be particularly fast or slow relative to others. SSD and similar memory-based storage where random page lookup is nearly as fast as sequential are an example.
- The statistics for how many distinct values are in a column can be manually adjusted using `ALTER TABLE`, to improve cases where they've been estimated badly.
- Queries using `IS NOT NULL` can now use indexes to help compute their results. This particularly accelerates `MIN()` and `MAX()` lookups when your index has many `NULL` values.
- The **Genetic Query Optimizer** (**GEQO**) can now be forced to use a fixed random seed value. While query plans will still be random, any given query will always get the same optimizer plan. This makes reviewing and troubleshooting bad GEQO plans much easier to do. Before it was impossible to be sure how a query using this feature was optimized and executed if you didn't log it at the time.

- Estimation for values using greater or less than comparisons are improved, particularly for situations where data is constantly being inserted into a table at one end of a column's range. This is common for time series and many other types of data. If the relevant statistics histogram bucket for the comparison is the first or last one available, and there is an index available, it's cheap to determine the true minimum or maximum value of that column. That min/max will be used instead of what's in the histogram bucket to find the left or right edge of the values in the column, rather than the possibly out-of-date information `ANALYZE` last put into the relevant histogram bucket.
- The query optimizer is much more aggressive about using Materialize nodes when it plans queries, if it believes that it will perform better than earlier approaches for executing that plan. This may result in queries to use a higher multiple of `work_mem` than they did before as a result, to hold those Materialize nodes. This feature can be disabled by turning off the `enable_material` parameter. Like any new optimizer feature, this is useful for troubleshooting whether the implementation is working as expected when you see more of these nodes appear in query plans.
- GIN indexes are created using self-balancing Red-Black trees, which particularly improves their performance when data is inserted in index order-a situation particularly common when loading a database.

Database development

General database related changes of the release are:

- The PL/pgSQL language used for writing many types of functions is now installed in all databases by default. It can still be removed if desired.
- Triggers can be attached specifically to a column, eliminating unnecessary calls to trigger function that aren't needed unless specific columns are altered. It's also possible to restrict when a trigger fires using a `WHEN` clause, further cutting down on unnecessary calls to the trigger function.
- The `LISTEN`/`NOTIFY` mechanism for passing messages between database clients is faster and allows sending a "payload" string message. This allows building some better cross-client messaging systems using the database itself than was possible before.

- Applications can set an `application_name` parameter, which is shown in the `pg_stat_activity` information the database displays. Consistent use of this feature allows better monitoring of activity based on what the application is doing. In older versions, assigning dedicated user roles to certain application functions was normally done to approximate tracking at this level. While that still works too, this mechanism gives a more fine-grained way to track what the application is doing.
- The error message when a constraint uniqueness violation occurs now includes the value involved. This makes it much easier to figure out what caused the problem.
- Constraint checks can now be marked as `DEFERRABLE`. Previously, constraints were checked on a per-row basis. This new approach allows checking for constraint violations only at the end of a statement or transaction. Some types of changes that were previously difficult to execute, such as mass renumbering of the primary key in a table, are much easier to execute if you defer all key checking until commit of every change.
- A new class of constraints similar to uniqueness ones are available, called exclusion constraints. These allow constraints to enforce things such as ranges that must not overlap, instead of just checking for duplicate values. This can be particularly useful for applications that work with date ranges. A simple example is preventing items on a calendar from overlapping.
- The `contrib/hstore` module, which adds a relatively low overhead key/value store type, has been upgraded to be faster and more capable. It's much more practical to add a high performance simple key/value store to a PostgreSQL database using this module now. This allows that sometimes more practical storage approach to be combined with traditional relational queries, using whichever of the two is more appropriate for each type of data you need to store.

Configuration and monitoring

Configuration and monitoring related changes in this release:

- Server parameters can be adjusted per user/database combination
- When the `postgresql.conf` file is reloaded via `pg_ctl reload` or sending the server a `SIGHUP` signal, the server log file notes which parameters were changed as a result
- Background writer statistics can now be reset using `pg_stat_reset_shared('bgwriter')`

- Table and function counters can be individually reset using `pg_stat_reset_single_table_counters()` and `pg_stat_reset_single_function_counters()` functions
- The `log_temp_files` parameter is now specified in kilobyte units, and its documentation corrected (the description in earlier versions was misleading)
- The `log_line_prefix` option can now track the `SQLSTATE` value set for messages such as errors, allowing logs to be more carefully analyzed by their exact error codes
- The somewhat confusing relation size functions include new variations named `pg_table_size()` and `pg_indexes_size()`, which don't require knowing as much about database storage trivia to interpret usefully

Changes related to tools:

- `pgbench` can now run multiple benchmarking processes/threads at once. Earlier versions could easily become limited by the speed of the `pgbench` client coordination part of the program itself instead.
- The `contrib/auto_explain` output displays the query being executed, in addition to its plan.
- Query log data collected by `contrib/pg_stat_statements` includes counts for how much buffer cache activity was associated with each statement.

Let's look at changes in the internals:

- A utility named `contrib/pg_upgrade` allows upgrading to 9.0 from either version 8.3 or 8.4 without needing to dump and reload the database, what's normally referred to as an "in-place upgrade".
- A 64-bit Windows version is released, which allows the server to use more memory than earlier versions. It's still not expected that extremely large settings for `shared_buffers` will be useful on that platform, however. But this does allow you to be more aggressive with settings such as `work_mem` than you were able to before.
- `VACUUM FULL` has been changed to use the same rebuild mechanism that `CLUSTER` uses in earlier versions. This requires more disk space, but is normally faster and it avoids the index bloat issues that `VACCUM FULL` used to introduce in many situations.

- You can use `CLUSTER` on any system catalog tables that are per database, but still not ones that are shared by all databases. However, shared system catalogs can now have `REINDEX` executed on them safely. Previously, that was only possible in the special database maintenance standalone mode, and even there it was dangerous.
- Some variations of `ALTER TABLE` require making a whole new copy of the table that includes the added/changed information, in a similar way to how `CLUSTER` rewrites a table. Since the copy in progress is not important unless it is complete, the block changes to create it are no longer written to the WAL. This is the same way that new and truncated tables avoid WAL logging when you write to them in a transaction that clears them. If the transaction doesn't complete, the partial copy is just destroyed. In this new case, as with the older `CREATE TABLE`/`TRUNCATE` ones, if you have WAL archiving enabled to send data to a standby server this optimization can't be used.
- When compiling the database, the default is now to include the feature that used to require requesting `--enable-thread-safety`. That option allows multi-threaded client programs to be compiled using the database client libraries and run safely. In earlier versions, this was disabled by default, and required recompiling the database to fix.
- The database has a better idea how to deal with the Linux **Out of Memory** (**OOM**) killer process. With the appropriate matching changes to server startup scripts, this allows making the main database server process resistant to the killer. Individual clients will instead be targeted if their memory usage goes out of control. In earlier versions, it was usually the main server process that was killed in that situation.
- Several commands now allow a new syntax where options can be passed using a list in a `()` block rather than more traditional SQL syntax. `EXPLAIN`, `COPY`, and `VACUUM` all have this alternate syntax available for some parameters. In some cases it's the only way to use newer features added to those commands.

Performance features in version 9.1

This PostgreSQL release has added many features, which are unique to PostgreSQL. Synchronous replication is a major feature added in that release. Here is the list of performance features added in the 9.1 release. Complete details can be found at `https://www.postgresql.org/docs/9.1/static/release-9-1.html`

Let's look at the features:

- Performance improvement of `commit_siblings` to reduce overhead.
- A new feature added in CREATE TABLE, which is unlogged table, in which table data is not written in WAL. This gives a performance while writing to the table, but these are not crash safe.
- To provide chance for query optimization FULL OUTER JOIN to be implemented as a hash joins. This allow either side of LEFT/RIGHT OUTER JOIN to be hashed.
- Improve the memory requirement of ispell.
- Inheritance table return sorted result, which improves performance of ORDER BY LIMIT and MIN/MAX.
- Some improvements have been done for cost estimation of aggregates, window functions, and GIN index scan.
- In heavy write load, fsync becomes an issue. This release supports merging of duplicate fsync requests.

Performance features in version 9.2

The focus of this PostgeSQL's release is performances. Many small and big-level performance features such as index-only scans and SP-GiST were added in this release. Here are some of the performance features, for complete listing please visit: `https://www.postgresql.org/docs/9.2/static/release-9-2.html`

Let's look at the changes here:

- Index-only scan introduced in this release, retrieve data from index and avoid heap. This is a major performance featured added in this release.
- The visibility map was also made crash-safe.
- A new index access method **SP-GiST** (**Space-Partitioned GiST**) was added, which supports unbalanced partitioned search structures rather than balanced trees.
- Improved group commit performance under heavy load.
- Improved internal lock management.
- Reduced overhead for shared invalidation cache messages.
- Improved performance of COPY command.

- Improved performance and build time of GiST index.
- Allow hint bits to be set sooner for temporary and unlogged tables.
- Improved performance of checkpointer's `fsync-request`.
- Proper management of open file description on Windows, to allow Windows to use more open file description.
- Create a separate process for background writer for dirty page writing and checkpointing.
- Improve the idle and sleep time of bgwriter, walwriter, checkpointer, statistics collector, log collector, and archiver background processes.
- New optimization MIN/MAX index on Boolean column introduced.
- Generate custom plans for parameter values for prepared statements, instead of single generic plan.
- Improve the planner's ability to use nested loops with inner index scans.
- Improvement in planning API of Foreign data wrapper introduced in this release.
- Fix planner to handle indexes with duplicated columns more reliably.
- Flush WAL to disk when WAL buffer filled, this is another trigger for WAL flushing to disk along with `wal_writer_delay`. This improves the asynchronous commit.

Performance features in version 9.3

Here is the list of all performance features but you can always get the complete listing from here: `https://www.postgresql.org/docs/9.3/static/release-9-3.html`

- A new option FREEZE added in COPY command to avoid later overhead of marking tuple frozen
- Improvement of session waiting for `commit_delay`
- Improve VACUUM to recheck the visibility, to enhance the chance that page beaning mark all visible
- Add a cache called per-resource-owner useful for `pg_dump`
- Add `SP-GiST` (`https://www.postgresql.org/docs/9.3/static/rangetypes.html#RANGETYPES-INDEXING`) support for range data types
- Allow `GiST` indexes to be unlogged, and improve the performance of GiST index

- Improve concurrency of hash index operations
- Improve optimizer's cost estimation for index access
- Avoid scanning the entire relation cache at commit of a transaction that creates a new relation
- Suppress no-op Result and Limit plan nodes
- Reduce optimizer overhead by not keeping plans on the basis of cheap startup cost when the optimizer only cares about total cost overall

Performance features in version 9.4

This release improves parallel write performance by allowing multiple backends to insert into WAL buffer concurrently and improve performance of aggregate function and so on. Here is the list of some performance features but you can always get the complete listing from here:

`https://www.postgresql.org/docs/9.4/static/release-9-4.html`

Let's look at the features:

- Concurrent writing into WAL by multiple backend support added this release.
- Improved writing process to WAL, therefore only modified portion can be updated to WAL.
- Auto-resize the catalog caches. This reduces memory consumption for sessions accessing only a few tables, and improves performance for sessions accessing many tables.
- Aggregate function used as widow function is a performance boost. Also, improves the performance of aggregate function using numeric state values.
- Improved speed of `COPY` command.
- Improved speed of accessing many different `sequences` in the same session.
- Reduce memory allocated by PL/pgSQL `DO` blocks.

Performance features in version 9.5

This release improves the performance of multi CPU machines, performance of sorting, which is a big win, and performance of hash joins. Here is the list of all performance features but you can always get the complete listing from here:

`https://www.postgresql.org/docs/9.5/static/release-9-5.html`

Let's look at the features:

- Improve the performance of sorting using abbreviated keys
- Improve performance of hash joins and bitmap index
- Improve concurrency of shared buffer replacement
- Improve locks in case of system running on multiple CPU sockets
- Allow the optimizer to remove unnecessary references to left-joined subqueries
- Teach the planner to use statistics obtained from an expression index on a boolean-returning function, when a matching function call appears in WHERE
- Allow restriction pushdown to subquery in case of Window function
- Improve CRC calculation and switch to CRC-32C
- Improve the speed of index creation by removing unnecessary memory copies
- Make per-backend tracking of buffer pins more memory-efficient

Performance features in version 9.6

This PostgreSQL release focuses on one of the major performance features which is called parallelism of existing features. This release adds parallelism in sequential scans, joins, and aggregates. This also added new `postgres_fdw` remote joins, sorts, UPDATEs, and DELETEs. Here is the list of some performance features but you can always get the complete listing from here:

`https://www.postgresql.org/docs/9.6/static/release-9-6.html`

Let's look at the features:

- Allow old MVCC snapshots to be invalidated after a configurable timeout
- Index-only scan can be used even if restriction clause column not indexed
- Perform checkpoint writes in sorted order, instead of random order
- Improve the performance of aggregate-function by dividing the calculation across multiple aggregates

- Allow tuple hint bits to be set sooner than before
- Improve performance of short-lived prepared transactions, resource owners with many tracked objects, and memory context destruction
- Improve the performance of timestamp, date and time data type's output function
- Increase the number of clog buffers for better scalability
- Avoid reducing the `SO_SNDBUF` setting below its default on recent Windows versions

Summary

PostgreSQL performance and features have increased so much in recent versions that time spent trying to optimize an older version might be better spent planning an upgrade. Minor PostgreSQL releases focus on bug fixes, and can be effective at removing performance issues caused by bugs too.

At last this book has covered all the aspects of the performance of PostgreSQL. Including, performance improvement from hardware benchmarking to PostgreSQL benchmarking and how to detect the performance bottleneck using different tools and resolve that issue. Therefore this is a complete handbook for developers and database administrators.

Index

A

activity 326, 327
Adaptive Replacement Cache (ARC) 95
advanced index
  expression-based indexes 251
  multicolumn indexes 249
  partial indexes 250
  used, for full-text search (FTS) 251
  used, for sorting 249
  uses 249
aggressive PostgreSQL version upgrades 443
allowed change context, live configuration
    interaction 136, 137
application scaling life cycle
  about 17, 18
  path database servers 17
atomicity, consistency, isolation, durability (ACID)
    35
auto_explain module
  reference 144, 195
autoanalyze 183
Automated Teller Machine (ATM) 208
autovacuum
  about 145, 174
  issues 178
  logging 175
  monitoring 176
  per-table adjustments 178
  triggering 176

B

background writer statistics
  used, for tuning 343, 344, 345
bad results from pgbench, sources
  about 220
  developer PostgreSQL builds 221
base backup 388
battery-backed write cache (BBWC) 37
benchmarking
  about 18
  reference 45
Block Range Indexes (BRIN)
  reference 245
bonnie++
  about 58
  Bonnie++ 2.0 60
Bonnie++ZCAV
  about 61
  seek-rate test 65
  sysbench 63
bottleneck 18
Bucardo
  about 395, 396
  reference 395
buffer cache contents
  analyzing 128
  queries, inspecting 128
buffer cache queries inspection
  about 128
  buffer contents summary, with percentages 131
  buffer usage count distribution 133
  summarizing, by usage count 131
  top relations 130
  used, for sizing feedback 133
bulk loading
  about 428
  constraints, adding 432
  indexes, recreating 432
  methods 429
  tuning 430, 431
  WAL acceleration, skipping 431, 432

C

Cacti
  about 371
  and PostgreSQL 372
  and Windows 372
cascading replication 390
check_postgres
  reference 371
checkpoint, live configuration interaction
  about 147
  checkpoint_completion_target 147, 148
  checkpoint_timeout 147
  wal_buffers 148
  wal_sync_method 148, 149
checkpoint
  heading to 126
  processing, basics 119
  reference 119
  spikes 122
  spreading 122
  timings 120, 121
checkpoint_completion_target parameter 147, 148
checkpoint_timeout parameter 147
CLUSTER command 187
commented-out settings 139
commit rate
  about 51
  INSERT rate 52
  PostgreSQL test_fsync 52
  Windows commit rate 52
Common Table Expressions (CTE) scan 282
Completely Fair Scheduler (CFS) 222
complicated disk benchmarks
  iozone, reference 67
  pgiosim, reference 67
configuration file
  commented-out settings 139
  server-wide settings 140
connection pool
  about 378
  sizing 378
connection pooling 377
connections 326, 327
constant angular velocity (CAV) 50
constraint_exclusion parameter 153
context switches (cs) 351
contrib modules
  about 13
  extensions, using 16
  finding, on system 13, 14
  installing, from source 14
  reference 13
  using 15
copy command
  about 282
  COPY FREEZE 283
  COPY FROM 283
  COPY TO 282
  reference 282
COPY FREEZE command
  reference 283
cost computation
  arbitrary units, using 261
  estimated cost 262
  real-world costs 262
count(*) 248
CPU benchmarking
  about 47
  slow memory and processors, sources 47
crash recovery
  and buffer cache 118
CSV logging
  reference 193
cumulative view 318, 319

D

data definition language (DDL) 143
database block
  dirty block write paths 123
  lifecycle 122
database buffer cache
  versus operating system cache 124
Database Connection Pool (DBCP)
  about 383
  reference 383
database connections, live configuration interaction
  listen_addresses 140
  max_connections 141
  reference 140

database directory
  base 100
  global 100
  pg_clog 100
  pg_multixact 100
  pg_stat_tmp 100
  pg_subtrans 100
  pg_tblspc 100
  pg_twophase 100
  pg_xlog 100
  temporary files 101
database hardware
  balancing 21
  disk controllers 30
  disks 24
  hard drive reliability studies 35
  write-back caches 36
database server configuration, for pgbench
  sample server configuration 210, 211
Database Test (dbt) project
  reference 224
database wide totals 326
database/table
  vacuuming 187
date change update trigger 410
DDR RAM 44
deadlocks 335
debugging
  with Valgrind 425
default_statistics_target parameter 146, 147
defaults, live configuration interaction 136
Dell Store 2
  reference 254
dellstore2 database 402
detailed data
  monitoring 189
difficult queries, logging
  auto_explain 195
  reference 195
Direct Attached Storage (DAS) 33
direct memory access (DMA) 30
disk benchmarking tools
  about 52
  bonnie++ 58
  disk throughput, sequential read and write 57
  HD Tune, using 53, 54
  unpredictable performance 57
disk controllers
  about 30
  attached storage 33
  hardware and software RAID 30
  MegaRAID 31
  Network Attached Storage (NAS) 33
  PowerEdge RAID Controller (PERC) 31
  recommended disk controllers 31
  Storage Array Network (SAN) 33
disk drives
  reference 35
disk layout
  about 101
  database directory tree 100
  disk arrays 101
  guidelines 104
  RAID 101
  symbolic links 99
  tablespaces 99
disk performance expectation
  slow disk and array performance 71
disk result
  about 68
  performance expectation 70
disks
  about 24
  drive error, handling 27
  drive firmware 28
  RAID 25
  SSDs 29
doubly cached data
  about 125
  OS cache, inspecting 126
DTrace
  Linux SystemTap emulation 427
  on FreeBSD 427
  reference 426
  used, for profiling 426
dump/restore 187
dynamic trigger functions 407, 408

E

effective_cache_size parameter 150, 151
empty partitions query plans 409
Enterprise Postgres Query Analyze (epqa)
  reference 201
error recovery control
  reference 27
error reporting and logging
  reference 192
EXPLAIN
  about 256
  cache, clearing 258
  hot and cold ache behavior 257
  time, checking 257
ext4
  reference 81
Extensible Firmware Interface (EFI) 76
Extreme Memory Profile (XMP) 48

F

failover 389
Fake RAID 32
FAT32 97
File Allocation Table (FAT)
  reference 97
File-Based Log Shipping 389
filesystem
  crash recovery 76
  maximum sizes 75
Force Unit Access (FUA) 85
free space map (FSM)
  about 142
  setting, reference 142
FreeBSD UFS2 94
FreeBSD
  about 92
  reference 96
Frequently Asked Questions list, PostgreSQL
  reference 428, 442
fsync parameter 153, 154
full_page_writes parameter 154

G

Generalized Inverted Index (GIN) 244
Generalized Search Tree (GiST) 245
gnuplot software 61
Golconde
  reference 398
gprof
  used, for profiling 423, 424
graphing
  with kSar 364, 365
GridSQL
  about 420
  reference 420
  used, for scaling 421
GUID Partition Table (GPT)
  reference 76

H

hash joins
  about 288
  anti joins 288
  hash semi 288
HD Tune
  used, for measuring IOPS 55
HighPoint 32
historical trend data
  storing 369
horizontal partitioning, with PL/Proxy
  about 417
  hash generation 418, 419
  scaling 419
hot standby
  about 388
  reference 388
  tuning 391, 392
htop
  for Linux 362
Hyperic HQ 374

I

in-place upgrading 11
index bloat
  fixing 186
  measuring 185

reference 186
Index I/O 325
Index Only Scans
reference 247
index page
monitoring 189
index statistics 323, 324, 325
index
about 226
clustering 241
example 227
fillfactor 242
indexes
block statistics, working 227
buffer counts, using 239
clustering against 237
combining 235
concurrent index, creating 241
creating 233, 239
example, running 229
full table scans 232
inefficient index, using 234
maintaining 239
plan changes, planning 236
query disk, measuring 227
sample data setup 229
simple index lookups 231
unique indexes 240
inheritance 401
INSERT statements
redirecting, to partitions 406, 407
IOPS
measuring, HD Tune used 55
reference 50
iostat
about 352, 353
example of good performance 354, 355
overloaded system samples 357
iotop
for Linux 354
issues, autovacuum
constant running 179
disruptive 180
free space map (FSM) exhaustion 181
long-running transactions 181
major problem recovery 182
not keeping up, on busy server 180

J

join order
forcing 290
genetic query optimizer (GEQO) 292
removing 291
joins
about 284
hash joins 288
merge join 287
nested loop 285
ordering 289
journaling filesystems
about 77
reference 77
Junction utility
reference 97
Just a Bunch of Disks (JBOD) 40

K

kernel resources
reference 108
key field
determining, to partition over 404, 405
kSar
reference 364, 365
used, for graphing 364, 365

L

latency analysis 217
limiting factor 18
Linux filesystems
btrfs 84
ext4 81
JFS 83
ReiserFS 83
second extended filesystem (ext2) 78
tuning operation 86
XFS 82
Linux SystemTap emulation
of DTrace 427
list partitioning 406
listen_addresses 140

live configuration interaction
about 135
allowed change context 136, 137
checkpoints 147
configuration file, reloading 138
database connections 140
defaults 136
logging setup 142
PITR and WAL replication 150
reset values 136
shared memory settings 142
statistics 144
tunables to avoid 153
vacuuming 144
live view 318, 319
load balancing 381
lock information, logging
about 335
background writer 339
buffer, monitoring 339
checkpoint activity 339
deadlocks 335
disk usage 336, 337, 338
lock information
decoding 331
locking clause
reference 165
locks
about 327
information, decoding 329
reference 328, 332
log file analysis
Enterprise Postgres Query Analyze (epqa) 201
mk-query-digest 202
normalized query fingerprints 196
pg_stat_statements 197
pgFouine 199
PostgreSQL System Impact (PGSI) 202
Practical Query Analysis (pqa) 201
log_line_prefix parameter 143
log_min_duration_statement 144
log_statement
options 143
logging, live configuration interaction
about 142
log_line_prefix 143
log_min_duration_statement 144
log_statement 143
Logical Volume Manager (LVM) 85
Londiste
about 394
advantage 394
reference 394

M

maintainance_work_mem parameter 146
Mammoth Replicator
reference 397
Master Boot Record (MBR)
reference 76
materialized views
about 440
reference 440
max_connections 141
max_prepared_transactions parameter 154
MegaRAID 31
memcached
about 383, 384
reference 384
memory benchmarking
about 43
Memtest86+ 43
STREAM memory testing 44
Memtest86+
about 43
reference 43
methods, bulk loading
external loading programs 429, 430
mk-query-digest
reference 202
monitoring
types 368
Multi Router Traffic Grapher (MRTG) 369
multicore processors
reference 22
multiversion concurrency control (MVCC)
advantages 167
disadvantages 168
reference 161
transaction ID wraparound 168

used, for handling transaction visibility 161
Munin
about 372
reference 372

N

Nagios
about 370
and PostgreSQL 371
and Windows 371
limitations 370
nested loop
about 285
and Merge Join materialization 287
inner-index scan, using 286
NetApp
reference 28
Network Attached Storage (NAS)
advantages 33
drawbacks 34
NTFS
about 97
mounting behavior, adjusting 98

O

object identifier (OID)
about 115
reference 115
Object Relational Mapper (ORM) 263
OmniTI
reference 374
Online Transaction Processing (OLTP) 224
OProfile
about 425
download link 425
reference 425
options, hard disk
parallel ATA, reference 25
SCI 25
serial ATA, reference 25
solid-state drive 25
options, log_statement
all 143
ddl 143
mod 143
none 143
orders_insert_trigger() function 407

P

parallel restore 432, 433
partition rules 408, 409
partitioned table
live migration 411
partitioning
advantages 415
in PostgreSQL 402
issues 416
partitions
creating 414
dynamic creation 415
INSERT statements, redirecting to 406, 407
scheduled creation 415
sizing 405
performance features, PostgreSQL 8.1
about 443, 444
reference 443
performance features, PostgreSQL 8.2
about 444, 445
reference 444
performance features, PostgreSQL 8.3
about 445, 446
reference 445
performance features, PostgreSQL 8.4
about 447, 448
reference 446
performance features, PostgreSQL 9.0
about 449
configuration 452
database development 451, 452
EXPLAIN 450, 451
internals 453, 454
monitoring 453
queries 450, 451
reference 449
replication 449
tools 453
performance features, PostgreSQL 9.1
about 455
reference 454
performance features, PostgreSQL 9.2

changes 455, 456
reference 455
performance features, PostgreSQL 9.3
about 456, 457
reference 456
performance features, PostgreSQL 9.4
about 457
reference 457
performance features, PostgreSQL 9.5
about 458
reference 458
performance features, PostgreSQL 9.6
about 458, 459
reference 458
performance issues
about 434
heavy statistics collector overhead 439
high foreign key overhead 438
materialized views 440
PL/pgSQL benchmarking 437
prepared statement execution 436
rows, counting 434, 435
slow function 436, 437
targeted statistics resets 440
trigger memory use 439
unexplained writes 435, 436
performance tuning 18, 19
pg_bulkload
reference 430
pg_hba.conf file
reference 140
pg_locks view
reference 328
pg_settings
reference 136
pg_stat_bgwriter snapshots
saving 340, 341, 342
pg_stat_statements
reference 197
pg_upgrade
reference 11
pgbench custom tests
about 223
Insert speed test 223
pgbench default tests
about 204
database server, configuring 209
query script definition 207, 209
scale 207
scale detection 206
table definition 205, 206
pgbench program
limitations 222
pgbench-tools
configuring 212, 213
reference 212
results, graphing with 212
pgbench
reference 8
pgBouncer
about 382
application server pooling 383
database caching 383
reference 382
PgCluster
about 398
reference 398
pgfincore
reference 24
pgFouine
reference 198, 200
pgFoundry
about 16
reference 16
pgloader
reference 429
pglogical
about 396
reference 396
pgmemcache
about 384
reference 385
pgPool's failover
reference 380
pgpool-II
about 379, 397
limitations 381
reference 379
replication, providing 380
pgpool

about 380
failover feature 380
reference 380
pgStatspack
about 373
reference 373
pgtune program
about 158
reference 158
physical disk performance
commit rate 51
measuring 49
random access 49
sequential access 50
ZCAV 50
PITR and WAL replication, live configuration interaction
per-client settings 150
PITR and WAL replication
constraint_exclusion 153
effective_cache_size 150, 151
random_page_cost 152
synchronous_commit 151
work_mem 152
PL/pgSQL benchmarking
about 437
reference 437
PL/Proxy
about 417
reference 417
Point-in-Time Recovery (PITR) 388
post load clean up 433
Postgres-XC
about 398
reference 398
PostgreSQL 9.0
reference 8
PostgreSQL 9.6.3 Documentation
reference 135
PostgreSQL bench
reference 9
PostgreSQL log setup
CSV logging 193
multi-line queries 192
syslog, used for log messages 192
PostgreSQL manual section
reference 426
PostgreSQL SNMP MIB
reference 375
PostgreSQL System Impact (PGSI)
reference 201
PostgreSQL tools
about 12
additional related software 17
contrib 13
pgFoundry 16
PostgreSQL, Feature Matrix page
reference 442
postgresql.conf file
memory units 107
PostgreSQL
about 7
application scaling life cycle 17
comparing, with another database 12
deploy version, selecting 9
disk layouts 98
errors, reference 108
historical releases, performance 9
minor version updates 11
partitioning 402
reference 11, 111, 237
tools 12
upgrading, to new major version 10, 11
PowerEdge RAID Controller (PERC) 31
Practical Query Analysis (pqa)
reference 201
processing nodes
about 271
aggregate functions 274
append node 277
Group node 278
HashAggregate node 275
IN lists 280
materialize node 281
query limits, building 273
query limits, offset 273
result, returning 276
set operations 280
sort nodes 271
subplan 279

subquery conversion 279
subquery scan 279
unique node 275
profiling
about 18
with DTrace 426
with gprof 423, 424
Promise 32
pull the cord test 150

Q

queries
equivalence, testing 304
external trouble spots 311
improving 303
optimizer bugs 308
optimizer features, disabling 304
optimizing, for fully cached datasets 303
plan restructure avoiding, OFFSET used 309
query enable parameters
about 155
dedicated server guidelines 155, 156, 157
new server tuning 155
pgtune 158
shared server guidelines 157
query logs, monitoring
about 189
difficult queries, logging 194
log file analysis 196
PostgreSQL log setup 190
query plan node structure
about 259
analysis tool, planning 266
cost computation 261
machine readable explain output 264
verbose, using 263
visual explain, using 263
query planning parameters
constraint_exclusion 301
cursor_tuple_fraction 302
effective_cache_size 299
work_mem 300

R

RAID 5
and RAID 6, comparison link 26
RAM timing
reference 48
random read seek time 49
random_page_cost parameter 152
range partitioning 403, 404
read scaling
with replication queue software 395
Reconnoiter 374, 375
recovery processing 119
Redundant Array of Inexpensive Disks (RAID)
about 25
RAID (stripping) 25
RAID 1 (mirroring) 26
RAID 10 (1+0) 26
RAID 5 (striped with parity) 26
RAID 6 26
reference 26
reindexing 188, 242
release notes, PostgreSQL
reference 442
replication projects
Golconde 398
Mammoth Replicator 397
PgCluster 398
Postgres-XC 398
Rubyrep 398
replication queue managers
Bucardo 395, 396
Londiste 394
pglogical 396
pgpool-II 397
Slony 393, 394
xDB 396
replication
solution comparison 398
RESET command
reference 136
reset values, live configuration interaction 136
results
graphing, with pgbench-tools 212
rite-back caches

  disk controller monitoring 38
  drive write cache, disabling 39
Round Robin Database tool (RRDtool) 369
row lock conflicts
  serialization 165
row sets
  assembling 266
  bitmap heap 269
  index only scans 269
  index scan 268, 269
  sequential scan 267
  tuple ID 266
RRDtool 375
Rubyrep
  about 398
  reference 398

S

sample database
  Dell Store 2 254
  Pagila 254
  reference 254
sample datasets 254
sample pgbench test results
  about 214
  latency analysis 217, 218, 219, 220
  SELECT-only test 214
  TPC-B-like test (Read/Write) 215, 216
sar 362
scaling
  with GridSQL 421
scricomplicated disk benchmarks
  fio, reference 67
second extended filesystem (ext2)
  reference 78
seek-rate test
  about 65
  fsync commit rate 67
  test files, removing 66
SELECT-only test 214
Serial ATA (SATA) 24
Serial Attached SCSI (SAS) 24
serial presence detect (SPD) 47
server-wide settings 140
sharding 420
share/system_views.sql
  reference 316
shared memory, live configuration interaction
  free space map settings 142
  shared_buffers 142
shared_buffers 142
short stroking tests 51, 54
Simple Network Management Protocol (SNMP)
  369
size guidelines
  implementing 126
  shared server 127
  versions 127
  workload limitations 127
Slony
  about 393, 394
  limitations 394
  reference 10
SMART protocol 27
SNMP tools 375
Solaris 92
Solaris top replacements 361
Solaris UFS 92
Solid State Drive (SSD)
  reference 29
sources, write-back cache
  disk controller write cache 37
  disk drive write cache 38
  operating system write cache 37
space-partitioned GiST
  reference 245
spindle 49
SQL
  limitations 311
  rows, numbering 312
  windows functions, used for cumulatives 313
  windows functions, used for numbering 313
standby 389
Staplr
  about 375
  reference 375
statement logging 144
statement types
  executing 302
statistics targets

about 297
column target, adjusting 297
distinct values 298
statistics views 316, 318
statistics, live configuration interaction
default_statistics_target 146, 147
statistics
collecting 292
complicated estimation areas 298
targets 297
used, for estimation 293
viewing 293
Storage Array Network (SAN)
advantages 33
disadvantages 34
stream-scaling
reference 44
STREAM
and Intel versus AMD 45
memory testing 44
reference 44
streaming replication 390
stripes 72
symlink 99
synchronous replication 391
synchronous_commit parameter 151
Sysinternals tool
reference 365
sysstat
about 362
enabling 364
optional features 364
system information functions
reference 169
SystemTap 427

T

table I/O statistics 322, 323
table inheritance 401
table lock waits 334
table statistics 319, 320, 321
table-level lock
modes 332
template database
reference 115
third extended file system (ext3)
about 79
levels 79
TOAST tables 323
top 360
TPC-B-like test (Read/Write) 215, 216
TPC-B
reference 204
TPC-C
reference 224
transaction isolation
reference 166
transaction lock waits 332
Transaction Processing Performance Council (TPC)
about 204
benchmarks 224, 225
transaction visibility
computation internals 161
deletions 166
handling, with multiversion concurrency control (MVCC) 161
row lock conflicts 164
updates 162
trending packages
Hyperic HQ 374
pgStatspack 373
Reconnoiter 374
Staplr 375
Zenoss 374
trending software 368
TrustedInstaller process 57
tunables to avoid, live configuration interaction
about 153
commit_delay 154
commit_siblings 154
fsync 153, 154
full_page_writes 154
max_prepared_transactions 154
query enable parameters 155
tuning operations, Linux filesystem
file access times 87
I/O scheduler elevator 90
read caching 88
read-ahead 86

swapping 88
write cache sizing 89
Tuning Your PostgreSQL Server article
reference 135
tuning, for bulk loads 430, 431
tuple ID
about 266
object ID 267
two-phase commit (2PC) 154
types, index
about 242
B-tree 243
Block Range Indexes (BRIN) 245
Generalized Inverted Index (GIN) 244
Generalized Search Tree (GiST) 245
hash index type 244
Index Only Scans 247
space-partitioned GiST 245
text operator classes 243

U

unique node
WindowAgg function 276
Unix File System (UFS) 92
UNIX monitoring tools
about 347
iostat 352, 353
sample setup 348, 349
sar 362
sysstat 362
top 360
vmstat 349, 351
UNIX shared memory parameters
database cache, inspecting 111
database disk layout 113
dirty blocks, writing to disk 118
increasing, for larger buffer sizes 108
kernel semaphores 109
new block, creating in database 117
pg_buffercache, installing into database 112
shared memory allocation, estimating 110

V

vacuum
autovacuum 174
autovacuum logging 175
cost-based vacuuming 173
free disk space, returning 171
full vacuum 171
HOT (Heap Only Tuples) 172
implementation 170
issues 178
regular vacuum 170
tuple timeout, removing 172
vacuuming, live configuration interaction
about 144
autovacuum 145
autovacuum, enabling 145
maintainance_work_mem 146
Valgrind
about 425
example 425
output 426
used, for debugging 425
virtual transactions 328, 329
visibility map
about 248
reference 248
Visual Studio
using 426
vmstat 349, 350, 351

W

wal_buffers 148
wal_sync_method 148, 149
watchdog feature
about 380
reference 380
white-box vendor system 32
Windows monitoring tools
about 365
task manager 365
Windows system monitor
about 366, 367
data, saving 367, 368
work_mem parameter 152
write barriers
about 84
drive support 84
Write-Ahead Log (WAL)

about 388, 431
reference 119
shipping, setting up 389
write-back caches
about 36
performance impact 40
sources 37

X

xDB
about 396
reference 396
XFS 82

Z

ZCAV
reference 51
Zenoss
about 374
reference 374
ZenPack 374
ZFS 95