Community Experience Distilled

Clojure High Performance Programming

Understand performance aspects and write high performance code with Clojure

Shantanu Kumar

[PACKT] PUBLISHING open source* community experience distilled

Clojure High Performance Programming

Understand performance aspects and write high performance code with Clojure

Shantanu Kumar

[PACKT] PUBLISHING
open source
community experience distilled
BIRMINGHAM - MUMBAI

Clojure High Performance Programming

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 author, 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: November 2013

Production Reference: 1131113

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

ISBN 978-1-78216-560-6

www.packtpub.com

Cover Image by Duraid Fatouhi (duraidfatouhi@yahoo.com)

Credits

Author
Shantanu Kumar

Reviewers
Jan Borgelin
Mimmo Cosenza aka Magomimmo
Paul Stadig
Miki Tebeka

Acquisition Editors
Sam Birch
Andrew Duckworth

Commissioning Editors
Priyanka Shah
Meeta Rajani
Llewellyn Rozario

Technical Editors
Jalasha D'costa
Monica John

Copy Editors
Alisha Aranha
Roshni Banerjee
Tanvi Gaitonde
Alfida Paiva
Lavina Pereira

Project Coordinator
Amey Sawant

Proofreader
Paul Hindle

Indexers
Hemangini Bari
Mehreen Deshmukh

Graphics
Ronak Dhruv
Yuvraj Mannari

Production Coordinator
Kyle Albuquerque

Cover Work
Kyle Albuquerque

About the Author

Shantanu Kumar is a software developer living in Bangalore, India, with his wife. He started learning programing in 1991, using BASIC on MS DOS when he was at school. There, he developed a keen interest in the x86 hardware and assembly language, and he dabbled in it for a good while. Later, he programmed professionally in various business domains and technologies while working with the Indian Air Force and several IT companies.

In recent years, Shantanu has worked on high performance and distributed systems. Having used Java for a long time, he discovered Clojure in early 2009 and has been a fan ever since. Clojure's pragmatism and fine-grained orthogonality continues to amaze him, and he believes he is a better developer because of this.

When not busy with programming or reading up on technical subjects, he enjoys reading non-fiction, riding his bike, and occasionally just lazing in his free time. Shantanu is an active participant in the Bangalore Clojure users group and develops several open source Clojure projects on GitHub.

Acknowledgments

I would like to thank Rich Hickey for creating Clojure and making it available as open source, and for his awesome talk videos. I would also like to thank Alex Miller for arranging so many Clojure talks and for making their videos accessible to all; and Alex Ott, Michael Klishin, and others from the Clojure community for their hard work in making Clojure documentation aggregated and available.

While I was working at the Bangalore office of Runa (now Staples Lab) earlier, several colleagues shared valuable input about Clojure performance. Most notably, Zach Tellman shared his insight about Clojure and JVM performance, Isaac Praveen and Abhijith Gopal shared a great deal of information about Clojure application behavior under load, and Philippe Hanrigou shared his ideas about high performance API design and queue systems. I want to thank all of them.

This book would not have become a reality without the fine people at Packt Publishing. I would like to thank Ashvini Sharma for contacting me and convincing me to take up writing this book, Anish Ramchandani and Amey Sawant for coordinating the writing process, and the Commissioning Editors Meeta Rajani, Llewellyn Rozario, and Priyanka Shah for shaping up this book as I engaged in my debut writing. Technical Editors Jalasha D'costa and Monica John helped me disambiguate and refine the language in this book. I also owe my gratitude to the technical reviewers Jan Borgelin, Mimmo Cosenza, Paul Stadig, and Miki Tebeka – their feedback made the content so much better. Any errors or omissions, however, are only due to me.

Writing this book has been an arduous task. I want to thank my wife Binita for putting up with me while I was immersed far too many days, nights, and weekends into the book. If not for her support, I would not have been able to do justice to this book.

About the Reviewers

Jan Borgelin is the co-founder and CTO of BA Group Ltd., a Finnish IT consultancy providing services for global enterprise clients. With over 10 years of professional software development experience, Jan has had the chance to work with different technologies and programming languages in international projects where performance requirements have always been critical to the success of the project.

Mimmo Cosenza aka Magomimmo is a programmer and entrepreneur living in Milan, Italy. In the eighties, after graduating in Philosophy of Language, he worked for Rank Xerox and IBM. Then, he joined the Artificial Intelligence lab of ENI S.p.A, the Italian national oil company.

He designed and developed very successful LISP-based applications for the exploration and production departments of ENI. In 1995, after having been in Los Angeles during the rise of the Internet, he founded Sinapsi—an Italian software boutique. In his own country, he is very well known for his involvement in open source communities. In 2012, he founded SmartRM Inc., a startup that applies the Digital Right Management technology for protecting privacy to share confidential information without losing the control of their circulation.

He loves to teach the art of programming. He is the author of *Modern-cljs*, an open source book on the Clojure and ClojureScript programming languages. The book is hosted on `https://github.com/magomimmo/modern-cljs`.

Currently, he is applying machine learning techniques to Big Data by using Clojure on the server-side and ClojureScript on the client-side.

Paul Stadig is a professional software developer living in Crozet, VA, with his wife and three children. He has a B.S. and an M.S. in Computer Science from George Mason University, and he has 16 years of software development experience. He has an insatiable curiosity about the world in general and about programming languages in particular.

He has been involved in the Clojure community since 2008, he was a reveiwer for the first edition of *Programming Clojure*, and he is also a contributor to the language. Since 2010, he has been employed at Sonian, where he builds cloud-based distributed systems in Clojure.

Miki Tebeka has been shipping software for more than 10 years. He has developed a wide variety of products from assemblers and linkers to news trading systems and cloud infrastructures. Miki currently works on the data pipeline at Demand Media. In his free time, Miki is active in several open source communities.

Table of Contents

Preface

Clojure is a remarkably high-performance language despite its dynamic nature. What really strikes you though is the fact that it combines performance with fundamental simplicity and pragmatism, which makes it such a joy to program in. Over the last six years since its first public release, Clojure has been heavily tested and deployed in production by many people and organizations across various domains. Its user base has grown rapidly during this period.

Clojure High Performance Programming is all about Clojure running on the Java Virtual Machine. The JVM has a reputation of being a robust platform to develop and deploy applications on. In this book, we take a deeper look at the performance characteristics of various features of Clojure and the underlying environment. We also explore what it takes to build well-performing software. We begin with the performance fundamentals and gradually proceed over to Clojure and other matters you may have to deal with while writing high-performance applications.

Understanding and achieving performance is both an art and a science, just like writing good software. Remember the big picture in the back of your mind but also be prepared to get into the details with measurement tools. More importantly, know how the software works and keenly study the environment in which it runs. I hope this book will help you on that path.

What this book covers

Chapter 1, *Performance by Design*, classifies the various use cases with respect to performance and analyzes how to interpret their performance aspects and needs.

Chapter 2, *Clojure Abstractions*, is a guided tour of various Clojure data structures, abstractions (persistent data structures, vars, macros, and so on), and their performance characteristics.

Chapter 3, Leaning on Java, discusses how to enhance performance by using Java interoperability and features from Clojure.

Chapter 4, Host Performance, discusses how the host stack impacts performance. Clojure being a hosted language, its performance is directly related to the host.

Chapter 5, Concurrency, is an advanced chapter that discusses concurrency and parallelism features in Clojure and the JVM. Concurrency is an increasingly significant way to derive performance.

Chapter 6, Optimizing Performance, discusses the systematic steps that need to be taken in order to obtain good performance.

Chapter 7, Application Performance, discusses building applications. This involves dealing with external subsystems and factors that impact the overall performance.

What you need for this book

You should acquire Java Development Kit Version 7 or higher for your operating system to work through all examples. This book discusses the Oracle HotSpot JVM in specific situations, so you may want to get Oracle JDK or OpenJDK if possible. You should also get the latest Leiningen version (Version 2.3.3 as of the time of writing) from `http://leiningen.org/ and JD-GUI from http://jd.benow.ca/`.

Who this book is for

This book is for intermediate Clojure programmers who are interested to learn how to write high-performance code. If you are an absolute beginner in Clojure, you should learn the basics of the language first and then come back later to this book. You need not be well-versed in performance engineering or Java. However, some prior knowledge of Java would make it much easier to understand the Java-related chapters.

Conventions

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

Code words in text are shown as follows: "If you want Criterium to be available only in the REPL and not as a project dependency, add the following entry to the `~/.lein/profiles.clj` file."

A block of code is set as follows:

```
(import 'java.util.concurrent.Callable)
(import 'java.util.concurrent.Future)
(def ^ExecutorService e (Executors/newSingleThreadExecutor))
(def ^Future f (.submit e (cast Callable #(reduce + (range
10000000)))))
(.get f)  ; blocks until result is processed, then returns it
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Clicking on the **Next** button moves you to the next screen."

> 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 may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via 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 on `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.

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 would 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright 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

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Performance by Design

Clojure is a functional programming language that brings great power and simplicity to the user. Clojure is also dynamically typed and has very good performance characteristics. Naturally, every activity performed in a computer has an associated cost. What constitutes acceptable performance varies from one use case and workload to another. In today's world, performance is the determining factor for several kinds of applications. We will discuss Clojure (which runs on the Java Virtual Machine) and its runtime environment in the context of performance, which is the goal of this book.

The performance of Clojure applications depends on various factors. For a given application, understanding its use cases, design and implementation, algorithms, resource requirements and alignment with the hardware, and underlying software capabilities are essential. In this chapter, we will study the basics of performance analysis which includes the following:

- A whirlwind tour of how the application stack impacts performance
- Classifying performance anticipations by use cases types
- Outlining the structured approach to analyze performance
- A glossary of terms commonly used to discuss performance aspects
- Performance numbers every programmer should know

Use case classification

Performance requirements and priority vary across different kinds of use cases. We need to determine what constitutes acceptable performance for various kinds of use cases. Hence, we classify them to identify their performance model. When it comes to details, there is no sure fire performance recipe for any kind of use case, but it certainly helps to study their general nature. Note that in real life, the use cases listed in this section may overlap each other.

User-facing software

The performance of user-facing applications is strongly linked to the user's anticipation. The difference of a good number of milliseconds may not be perceptible by the user, but at the same time, a wait of more than a few seconds may not be taken kindly. One important element to normalize the anticipation is to engage the user by providing duration-based feedback. A good idea to deal with such a scenario would be to start the task asynchronously in the background and poll it from the UI layer to generate duration-based feedback for the user. Another way could be to incrementally render the results to the user to even out the anticipation.

Anticipation is not the only factor in user-facing performance. Common techniques such as staging or pre-computation of data and other general optimization techniques can go a long way to improve the user experience with respect to performance. Bear in mind that all kinds of user-facing interfaces fall into this use case category: web, mobile web, GUI, command-line, touch, voice-operated, and gestures.

Computational and data-processing tasks

Non-trivial compute-intensive tasks demand a proportional amount of computational resources. All of the CPU, cache, memory, efficiency, and parallelizability of the computation algorithms would be involved in determining the performance. When the computation is combined with distribution over a network, or when reading from / staging to disk, I/O bound factors come into play. This class of workloads can be further subclassified into more specific use cases.

CPU bound

A CPU bound computation is limited by the CPU cycles spent on executing it. Processing arithmetic in a loop, small matrix multiplication, determining whether a number is *Mersenne Prime*, and so on would be considered CPU bound jobs. If the algorithm complexity is linked to *N*, such as O(*N*) and O(N^2), then performance depends on how big *N* is and how many CPU cycles each step takes. For parallelizable algorithms, performance of such tasks may be enhanced by assigning multiple CPU cores to the task. On virtual hardware, performance may be impacted if CPU cycles are available in bursts.

Memory bound

A memory bound task is limited by the availability and bandwidth of a computer memory; examples include large text processing, list processing, and so on. Note that higher CPU resources cannot help when memory is in the bottleneck and vice versa. Lack of availability of memory may force you to process smaller chunks of data at a time, even if you have enough CPU resources at your disposal. If the maximum speed of your memory is X and your algorithm on single CPU-core accesses memory at a speed of X/3, the multicore performance of your algorithm cannot exceed 3 times the current performance, no matter how many CPU cores you assign to it. Memory architecture, for example SMP and NUMA, contributes to the memory bandwidth in multicore computers. Performance with respect to memory is also subject to page faults.

Cache bound

A task is cache bound when its speed is constrained by the amount of cache available. When a task retrieves values from a small number of repeated memory locations, for example small matrix multiplication, the values may be cached and fetched from there.

> Typically, CPUs have multiple layers of cache, and the performance will be at its best when the processed data fits in the cache. Processing will still happen, albeit slower, when the data does not fit into the cache . These will be covered in greater details in *Chapter 4*, *Host Performance*.

It is possible to make the most of the cache using cache-oblivious algorithms. A higher number of concurrent cache / memory bound threads than CPU cores is likely to flush the instruction pipeline, as well as the cache, at the time of a context switch.

Input/Output (I/O) bound

An I/O bound task would go faster if the I/O subsystem it depends on goes faster. Disk or storage as well as network are the most commonly used I/O subsystems in data processing. Other I/O devices are serial ports, a USB-connected card readers, and so on. An I/O bound task may consume very few CPU cycles. Depending on the speed of the device, connection pooling, data compression, asynchronous handling, caching, and so on may help in performance. One notable aspect of I/O bound tasks is that the performance is usually dependent on the time spent waiting for connection (or disk seek) and the amount of serialization we do, but hardly on the other resources.

In practice, many data processing workloads are usually a combination of CPU bound, memory bound, cache bound, and I/O bound tasks. The performance of such mixed workloads effectively depends on the even distribution of CPU, cache, memory, and I/O resources over the duration of the operation. While all system resources are finite, some I/O resources may be particularly limited in bandwidth and latency. A bottleneck situation arises only when one resource gets too busy to make way for another.

Online transaction processing (OLTP)

OLTP systems process business transactions on demand. It could work as a backend system for a user-facing ATM machine, a point-of-sale terminal, a network-connected ticket counter, an ERP system, and so on. OLTP systems are characterized by low latency, availability, and data integrity. OLTP systems run day-to-day business transactions. Any interruption or outage is likely to have a direct and immediate impact on the sales or service. Such systems are expected to be designed for resiliency rather than delayed recovery from failures. When the performance objective is unspecified, you may want to consider graceful degradation as a strategy.

It is a common mistake to ask OLTP systems to answer analytical queries, something that they are not optimized for. It is desirable of an informed programmer to know the capability of the system and suggest design changes as per the requirements.

Online analytical processing (OLAP)

OLAP systems are designed to answer analytical queries in a short time. They typically get data from OLTP operations and their data model is optimized for querying. OLAP systems basically provide for consolidation (roll-up), drill-down, and slicing and dicing of data for analytical purposes. They often use specialized data stores that can optimize ad-hoc analytical queries on the fly. It is important for such databases to provide pivot-table-like capability. Often, an OLAP cube is used to get faster access to analytical data.

Feeding OLTP data into OLAP systems may entail workflows and multistage batch processing. The performance concern of such systems is to efficiently deal with large quantities of data while also dealing with inevitable failures and recovery.

Batch processing

Batch processing is the automated execution of predefined jobs. These are typically bulk jobs and are executed during off-peak hours. Batch processing may involve one or more stages of job processing. Often, batch processing is clubbed with workflow automation, where some workflow steps are executed offline. Many of the batch processing tasks work on staging and preparing data for the next stage of processing to pick up.

Batch jobs are generally optimized for the utmost utilization of computing resources. Since there is little to moderate demand to lower latencies of particular subtasks, these systems tend to optimize for throughput. A lot of batch jobs involve large I/O processing, and they are often distributed over a cluster. Due to distribution, data locality is preferred when processing the jobs; that is, data and processing should be local in order to avoid network latency in reading/writing data.

Structured approach for performance

In practice, the performance of non-trivial applications is rarely a function of coincidence or prediction. For many projects, performance is not an option but rather compulsory, which is why this is even more important today. Capacity planning, determining performance objectives, performance modeling, measurement, and monitoring are crucial to achieving performance..

Tuning a poorly-designed system to perform as well as a system that is a well-designed system from the ground up is significantly hard, if not practically impossible. In order to meet a performance goal, **performance objectives** should be known before the application is designed. Performance objectives are stated in terms of latency, throughput, resource utilization, and workload. These terms are discussed in the *Performance vocabulary* section in this chapter.

The resource cost can be identified in terms of application scenarios, such as browsing of products, adding products to the shopping cart, and checkout. Creating workload profiles that represent users performing various operations is usually helpful.

Performance modeling is a reality check of whether the application design would support the performance objectives. It includes performance objectives, application scenarios, constraints, measurements (benchmark result), workload objectives, and, if available, the performance baseline. It is not a replacement of measurement and load testing, rather, the model is validated using these. The performance model may include performance test cases to assert the performance characteristics of the application scenarios.

Deploying an application to production almost always needs some form of **capacity planning**. It has to take into account the performance objectives for today and the foreseeable future. It requires an idea of application architecture and an understanding of how the external factors translate into internal workload. It also requires informed expectations about the responsiveness and the level of service to be provided by the system. Often, capacity planning is done early in a project to mitigate the risk of provisioning delays.

Performance vocabulary

There are several technical terms that are heavily used in performance engineering. It is important to understand them as they form the cornerstone of performance related discussions. Collectively, these terms form a performance vocabulary. Performance is usually measured in terms of several parameters where every parameter has roles to play; such parameters are part of the vocabulary.

Latency

Latency is the time taken by an individual unit of work to complete a task. It does not imply successful completion of a task. Latency is not collective; it is linked to a particular task. If two similar jobs, j1 and j2, took 3ms and 5ms respectively, their latencies would be treated as such. If j1 and j2 were dissimilar tasks, it would have made no difference. In many cases, **average latency** of similar jobs is used in performance objectives, measuring, and monitoring results.

Latency is an important indicator of the health of a system. A high performance system often thrives on low latency. Higher than normal latency can be caused due to load or a bottleneck. It helps to measure the **latency distribution** during a load test. For example, if more than 25 percent of similar jobs under a similar load have significantly higher latency than others, it may be an indicator of a bottleneck scenario worth investigating.

When a task, j1, consists of smaller tasks, say j2, j3, and j4, the latency of j1 is not necessarily the sum of latencies of each of the j2, j3, and j4 tasks. If any of the subtasks of j1 are concurrent with another, the latency of j1 will turn out to be less than the sum of the latencies of j2, j3, and j4. I/O bound tasks are generally more prone to higher latency. In network systems, latency is commonly based on the **roundtrip** to another host, including latency from source to destination and then back to source.

Throughput

Throughput is the number of successful tasks or operations performed in a unit of time. The top-level operations performed in a unit of time are usually of a similar kind but with potentially different latencies. So, what does throughput tell us about the system? It is the rate at which the system is performing. When you perform load testing, you can determine the maximum rate at which a particular system can perform. However, this is not a guarantee of conclusive overall maximum rate of performance.

Throughput is one of the factors that determine the scalability of a system. Throughput of a higher level task depends on the capacity to spawn off multiple such tasks in parallel and also depends on average latency of the tasks. Throughput should be measured during load testing and performance monitoring to determine **peak measured throughput** and **maximum sustained throughput**. These factors contribute to the scale and performance of a system.

Bandwidth

Bandwidth is the raw data rate over a communication channel measured in a certain number of bits per second. This includes not only the payload but all the overhead necessary to carry out the communication. A few examples are Kbits/sec, Mbits/sec, and so on. An uppercase B in KB/sec denotes 'Bytes', as in Kilo Bytes per second. Bandwidth is often compared to throughput. While bandwidth is the raw capacity, throughput for the same system is the successful task completion rate that usually involves a roundtrip. Note that throughput is for an operation which involves latency. To achieve maximum throughput for a given bandwidth, the communication/protocol overhead and operational latency should be minimal.

For storage systems (such as hard disks and solid-state drives), the predominant way to measure performance is **IOPS** (**Input-output per second**), which is multiplied by the transfer-size and represented as Bytes-per-second, or further into MB/sec, GB/sec, and so on. IOPS is usually derived for sequential and random workloads for read/write operations.

Mapping the throughput of a system to the bandwidth of another may lead to dealing with the impedance mismatch between the two. For example, an order processing system may transact to the database on disk and post results over the network to an external system.

Depending on the bandwidth of the disk subsystem, the bandwidth of the network, and the execution model of the order, processing the throughput may depend not only on the bandwidth of the disk subsystem and network, but also on how loaded they currently are. Parallelism and pipelining are common ways to increase throughput over a given bandwidth.

Baseline and benchmark

Performance baseline, or simply baseline, is the reference point including measurements of well characterized and understood performance parameters for a known configuration. Baseline is used to collect performance measurements for the same parameters which we may benchmark later for another configuration. For example, collecting "throughput distribution over 10 minutes at a load of 50 concurrent threads" is one such performance parameter we can use for baseline and benchmarking. A baseline is recorded together with the hardware, network, OS, and system configuration.

Performance benchmark, or simply benchmark, is the recording of performance parameter measurements under various test conditions. A benchmark can be composed as a performance test suite. A benchmark may collect a small to large amount of data, and may take a varying duration depending on use cases, scenarios, and environment characteristics.

Baseline is a result of a benchmark that was conducted at one point of time; however, benchmark is independent of baseline.

Profiling

Performance profiling, or simply profiling, is the analysis of the execution of a program at its runtime. A program can perform poorly for a variety of reasons. A profiler can analyze and find out the execution time of various parts of the program. It is possible to interleave statements in a program manually to print execution time of blocks of code, but this gets very cumbersome as you try to refine the code iteratively. A profiler is of great assistance to the developer.

Going by how profilers work, they are of three major kinds: instrumenting, sampling, and event-based. The event-based profilers work only for selected language platforms, and they provide a good balance between overhead and results; for example, Java supports event-based profiling via the JVMTI interface. Instrumenting profilers modify code at either compile time or runtime to inject performance counters. They are intrusive by nature and add significant performance overhead. However, you can profile regions of code very selectively using instrumenting profilers. Sampling profilers pause the runtime and collect its state at 'sampling intervals'. By collecting enough samples, it gets to know where the program spends most of its time. For example, at a sampling interval of 1ms, the profiler would have collected 1000 samples in a second. A sampling profiler also works for code that executes faster than the sampling interval, as the frequency of pausing and sampling is proportional to the overall execution time of any code.

Profiling is not meant only for measuring execution time. Capable profilers can provide a view of memory analysis, garbage collection, threads, and so on. A combination of such tools is helpful to find memory leaks, garbage collection issues, and so on.

Performance optimization

Simply put, optimization is minimizing a program's resource consumption after performance analysis. The symptoms of a poorly performing program are observed in terms of high latency, low throughput, unresponsiveness, instability, high memory consumption, and high CPU consumption. During performance analysis, you may profile the program in order to identify bottlenecks and tune the performance incrementally by observing performance parameters.

Better and suitable algorithms are an all-round good way to optimize code. CPU bound code can be optimized with computationally cheaper operations. Cache bound code can try using less memory lookups to keep a good hit ratio. Memory bound code can use adaptive memory usage and conservative data representation to store in memory for optimization. I/O bound code can attempt to serialize as little data as possible, and can batch operations to make the operation less chatty for better performance. Parallelism and distribution are other overall good ways to increase performance.

Concurrency and parallelism

Most of the computer hardware and operating systems we use today provide concurrency. On the x86 architecture, hardware support for concurrency can be traced as far back as the 80286 chip. Concurrency is the simultaneous execution of more than one process on the same computer. In older processors, concurrency was implemented using a context switch by the operating system kernel. When concurrent parts are executed in parallel by the hardware instead of merely switching context, it is called parallelism. Parallelism is the property of the hardware, though the software stack must support it in order for you to leverage it in your programs. You must write your program in a concurrent way to exploit the parallelism features of the hardware.

While concurrency is a natural way to exploit hardware parallelism and speed up operations, it is worth bearing in mind that having significantly higher concurrency than the parallelism your hardware can support is likely to schedule tasks to varying processor cores, thereby lowering branch prediction and increasing cache misses.

Low level processes/threads, mutexes, semaphores, locking, shared memory, and inter-process/thread communication are used for concurrency. The JVM has excellent support for these concurrency primitives and inter-thread communication. Clojure builds upon the JVM features to provide both low and higher level concurrency primitives that we will discuss in the concurrency chapter.

Resource utilization

Resource utilization is the measure of the server, network, and storage resources consumed by an application. Resources include CPU, memory, disk I/O, network I/O, and so on. The application can be analyzed in terms of CPU bound, memory bound, cache bound, and I/O bound tasks. Resource utilization can be derived by means of benchmarking by measuring the utilization at a given throughput.

Workload

Workload is the quantification of how much work there is in hand to be carried out by the application. It is measured in total numbers of users, concurrent active users, transaction volume, and data volume. Processing a workload should take into account the load conditions, such as how much data the database currently holds, how filled up are the message queues, and the backlog of I/O tasks after which the new load will be processed.

Latency numbers every programmer should know

Hardware and software have progressed over the years. Latencies for various operations put things into perspective. The latency numbers for 2013 are as shown in the following table. (Reproduced with the permission of Aurojit Panda and Colin Scott of Berkeley University: `http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html`)

Operation	Time taken as of 2013
L1 cache reference	1 ns (nano second)
Branch mis-predict	3 ns
L2 cache reference	4 ns
Mutex lock/unlock	17 ns

Operation	Time taken as of 2013
Compress 1KB with Zippy (`http://code.google.com/p/snappy/`)	2 µs (1000 ns = 1 µs : micro second)
Send 2000 bytes over commodity network	500 ns (that is, 0.5 µs)
SSD random read	16 µs
Roundtrip in same datacenter	500 µs
Read 1,000,000 bytes sequentially from SSD	200 µs
Disk seek	4 ms (1000 µs = 1 ms)
Read 1,000,000 bytes sequentially from disk	2 ms
Packet roundtrip CA to Netherlands	150 ms

Summary

We learned about the basics of what it is like to think deeper about performance. We saw the common performance vocabulary and also saw the use cases by which performance aspects might vary. We concluded by looking at the performance numbers of different hardware components, which is how the performance benefits reach our applications. In the next chapter, we will dive into performance aspects of various Clojure abstractions.

2
Clojure Abstractions

Clojure has four founding ideas. Firstly, Clojure was set out to be a functional language. It is not pure (as in purely functional), but it emphasizes immutability. Secondly, Clojure is a dialect of Lisp; Clojure is malleable enough that users can extend the language without waiting for the language implementers to add new features and constructors. Thirdly, Clojure was built to leverage concurrency for a new generation of challenges. Fourthly, Clojure is designed to be a hosted language. As of today, Clojure implementations exist for the JVM, CLR, JavaScript, Python, Ruby, and Scheme. Clojure blends seamlessly with its host language.

Clojure is rich in abstractions. Though the syntax itself is very minimal, the abstractions are finely grained, mostly composable, and precise to tackle a wide variety of concerns in the least complicated way. In this chapter, we will discuss the following topics:

- Performance characteristics of non-numeric scalars
- Immutability and the epochal time model, paving the way for performance by isolation
- Persistent data structures and their performance characteristics
- Laziness and its impact on performance
- Transients as a high-performance, short-term escape hatch
- Other abstractions, such as tail recursion, protocols/types, multimethods, and many more

Non-numeric scalars and interning

Strings and characters in Clojure are the same as in Java. String literals are implicitly interned. **Interning** is a way of storing only unique values in the heap and sharing the reference wherever required. Depending on the provider and the version of Java you use, the interned data may be stored in a string pool, Permgen, ordinary heap, or some special area in the heap marked for interned data. Interned data is subject to garbage collection when not in use, just like ordinary objects. Take a look at the following code:

```
user=> (identical? "foo" "foo")  ; literals are automatically interned
true
user=> (identical? (String. "foo") (String. "foo"))  ; created string is not interned
false
user=> (identical? (.intern (String. "foo")) (.intern (String. "foo")))
true
user=> (identical? (str "f" "oo") (str "f" "oo"))  ; str creates string
false
user=> (identical? (str "foo") (str "foo"))  ; str does not create string for 1 arg
true
user=> (identical? (read-string "\"foo\"") (read-string "\"foo\""))  ; not interned
false
user=> (require '[clojure.edn :as edn])  ; introduced in Clojure 1.5
nil
user=> (identical? (edn/read-string "\"foo\"") (edn/read-string "\"foo\""))
false
```

Note that `identical?` in Clojure is the same as `==` in Java. The benefit of interning a string is that there is no memory allocation overhead for duplicate strings. Commonly, applications on the JVM spend quite some time on string processing. So, it makes sense to have them interned whenever there is a chance of duplicate strings being simultaneously processed. Most of the JVM implementations today have an extremely fast intern operation; however, you should measure the overhead for your JVM if you have an older version.

Another benefit of string interning is that when you know that two string tokens are interned, you can compare them for equality faster using `identical?` than non-interned string tokens. The equivalence function `=` first checks for identical references before conducting a content check.

Symbols in Clojure always contain interned string references within them, so generating a symbol from a given string is nearly as fast as interning a string. However, two symbols created from the same string will not be identical:

```
user=> (identical? (.intern "foo") (.intern "foo"))
true
user=> (identical? (symbol "foo") (symbol "foo"))
false
user=> (identical? (symbol (.intern "foo")) (symbol (.intern "foo")))
false
```

Keywords are, on the basis of their implementation, built on top of symbols and are designed to work with the `identical?` function for equivalence. So, comparing keywords for equality using `identical?` would be faster, just like with interned string tokens.

Clojure is increasingly being used for large volume data processing that includes text and composite data structures. In many cases, the data is either stored as JSON or EDN (`http://edn-format.org`). When processing such data, you can save memory by interning strings or using symbols/keywords. Remember that string tokens read from such data would not be automatically interned, whereas the symbols and keywords read from EDN data would invariably be interned. You may come across such situations when dealing with relational or NoSQL databases, web services, CSV or XML files, log parsing, and so on.

Interning is linked to JVM **Garbage Collection** (**GC**), which, in turn, is closely linked to performance. When you do not intern the string data and let duplicates exist, they end up being allocated on the heap. More heap usage leads to GC overhead. Interning a string has a tiny, but measurable and upfront performance overhead, whereas GC is often unpredictable and unclear. GC performance, in most JVM implementations, has not increased in a similar proportion to the performance advances in hardware. So, often, effective performance depends on preventing the GC from becoming the bottleneck, which in most cases means minimizing it.

Identity, value, and epochal time model

One of the principal virtues of Clojure is its simple design which results in malleable, beautiful composability. Using symbols in place of pointers is a programming practice that has existed for several decades now. It has found widespread adoption in several imperative languages. Clojure dissects that notion in order to uncover the core concerns that need to be addressed. The following subsections illustrate this aspect of Clojure.

We program using logical entities to represent values. For example, a value, 30, means nothing unless it is associated with a logical entity, let us say *age*. The logical entity *age* is the *identity* here. Now, even though *age* represents a *value*, the *value* may change with *time*; this brings us to the notion of *state*, which represents the *value* of the *identity* at a certain *time*. Hence, *state* is a function of *time* and is causally related to what we do in the program. Clojure's power lies in binding an *identity* with its *value* that holds true at the *time* and the *identity* remains isolated from any new *value* it may represent later. We will discuss state management in *Chapter 5*, *Concurrency*.

Variables and mutation

If you have previously worked with an imperative language (C/C++, Java, and so on), you may be familiar with the concept of a **variable**. A variable is a reference to a block of memory. When we update its value, we essentially update the *place* in memory where the value is stored. The variable continues to point to the place where the older version of the value was stored. So, essentially a variable is an alias for the place of storage of values.

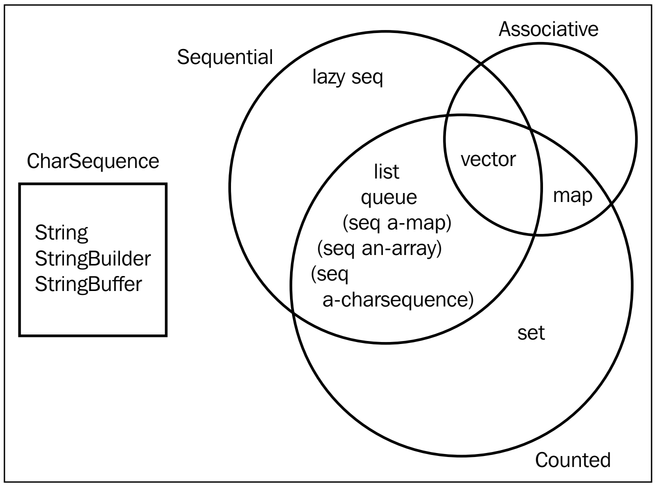
A little analysis would reveal that variables are strongly linked to the processes that read or mutate their values. Every mutation is a state transition. The processes that read/update the variable should be aware of the possible states of the variable to make sense of the state. Can you see a problem here? It conflates identity and state! It is impossible to refer to a value or a state in time when dealing with a variable—the value could change at any time unless you have complete control over the process accessing it. The mutability model does not accommodate the concept of time that causes its state transition.

The issues with mutability do not stop here. When you have a composite data structure containing mutable variables, the entire data structure becomes mutable. How can we mutate it without potentially undermining the other processes that might be observing it? How can we share this data structure with concurrent processes? How can we use this data structure as a key in a hash-map? This data structure does not convey anything. Its meaning could change with mutation! How do we send such a thing to another process without also compensating for the time, which can mutate it in different ways?

Immutability is an important tenet of functional programming. It not only simplifies the programming model, but also paves the way for safety and concurrency. Clojure supports immutability throughout the language. Clojure also supports fast, mutation-oriented data structures as well as thread-safe state management via concurrency primitives. We will discuss these topics in the forthcoming sections and chapters.

Collection types

There are a few types of collections in Clojure categorized based on their properties. The following Venn diagram (adapted with permission from Tim McCormack from `http://www.brainonfire.net/files/seqsandcolls/main.html`) depicts this categorization on the basis of whether the collections are counted (so that `counted?` returns true) or associative (so that `associative?` returns true) or sequential (so that `sequential?` returns true):



Persistent data structures

As we've noticed in the previous section, Clojure's data structures are not only immutable, but can also produce new values without impacting the old version. Operations produce these new values in such a way that old values remain accessible; the new version is produced in compliance with the complexity guarantees of that data structure and both the old and new versions continue to meet the complexity guarantees. The operations can be recursively applied to nested data structures and can still meet the complexity guarantees. Such immutable data structures as the ones provided by Clojure are called **persistent data structures**. They are *persistent* in that when a new version is created, both the old and new versions *persist* in terms of both the value and complexity guarantee. They have nothing to do with storage or durability of data. Making changes to the old version doesn't impede working with the new version and vice versa. Both versions persist in a similar way.

Among the publications that have inspired the implementation of Clojure's persistent data structures, two of them are well known. Chris Okasaki's *Purely Functional Data Structures* has influenced the implementation of persistent data structures and lazy sequences/operations. Clojure's persistent queue implementation is adapted from Okasaki's *Batched Queues*. Phil Bagwell's *Ideal Hash Tries*, though meant for mutable and imperative data structures, was adapted to implement Clojure's persistent map/vector/set.

Constructing less-used data structures

Clojure supports a well-known literal syntax for lists, vectors, sets, and maps. Shown in the following list are some less-used methods for creating other data structures:

- Map (`PersistentArrayMap` and `PersistentHashMap`):

  ```
  {:a 10 :b 20}  ; array-map up to 8 pairs, try (class {:a 10 :b 20})
  {:a 1 :b 2 :c 3 :d 4 :e 5 :f 6 :g 7 :h 8 :i 9}  ; hash-map for 9
  or more pairs
  ```

- Sorted map (`PersistentTreeMap`):

  ```
  (sorted-map :a 10 :b 20 :c 30)  ; (keys ..) should return sorted
  ```

- Sorted set (`PersistentTreeSet`):

  ```
  (sorted-set :a :b :c)
  ```

- Queue (`PersistentQueue`):

  ```
  (import 'clojure.lang.PersistentQueue)
  (reduce conj PersistentQueue/EMPTY [:a :b :c :d])  ; add to queue
  (peek queue)  ; read from queue
  (pop queue)  ; remove from queue
  ```

As you can see, abstractions such as `TreeMap` (sorted by key), `TreeSet` (sorted by element), and `Queue` should be instantiated by calling their respective APIs.

Complexity guarantee

The following tables give a summary of the complexity guarantees (using the big O notation; all logarithms have base 2 unless mentioned otherwise) of various kinds of persistent data structures in Clojure:

Operation	PersistentList	PersistentHashMap	PersistentArrayMap
`count`	O(1)	O(1)	O(1)
`conj`	O(1)		
`first`	O(1)		
`rest`	O(1)		
`doseq`	O(n)	O(n)	O(n)
`nth`	O(n)		
`last`	O(n)		
`get`		O(<7)	O(1)
`assoc`		O(<7)	O(1)
`dissoc`		O(<7)	O(1)
`peek`			
`pop`			

Operation	PersistentVector	PersistentQueue	PersistentTreeMap
`count`	O(1)	O(1)	O(1)
`conj`	O(1)	O(1)	
`first`	O(<7)	O(<7)	
`rest`	O(<7)	O(<7)	
`doseq`	O(n)	O(n)	
`nth`	O(<7)	O(<7)	
`last`	O(n)	O(n)	
`get`	O(<7)	O(<7)	O(log n)
`assoc`	O(<7)		O(log n)
`dissoc`	O(<7)		O(log n)
`peek`	O(1)	O(1)	
`pop`	O(<7)	O(1)	

A **list** is a sequential data structure. It provides constant time operation for everything regarding the first element only. For example, `conj` adds the element to the head and guarantees O(1) complexity. Similarly, `first` and `rest` provide O(1) guarantee too. Everything else provides O(n) complexity guarantee.

Persistent hash-maps and vectors use the bit-partitioned trie data structure with a branching factor of 32 under the hood. So, even though the complexity is O($\log_{32}$ N), only 2^{32} hash-codes can fit into the trie nodes. Hence, $\log_{32} 2^{32}$, which turns out to be 6.4 and is less than 7, is the worst-case complexity and can be considered near-constant time. As the trie grows large, the portion to copy gets tiny in proportion due to structure sharing. Persistent hash-set implementation is also based on hash-maps; hence, the hash-sets share the characteristics of the hash-maps. In a persistent vector, the last incomplete node is placed at the tail, which is always directly accessible from the root. This makes using `conj` to the end to be a constant time operation.

Persistent tree-map and tree-set are basically sorted maps and sets respectively. Their implementation uses *red-black trees* and is generally more expensive than hash-maps and hash-sets. Persistent queue uses a persistent vector under the hood for adding new elements. Removing an element from a persistent queue takes off the head from a seq, which is created from the vector where new elements are added.

The complexity of an algorithm over a data structure is not an absolute measure of its performance. For example, working with hash-maps involves computing the hash-code, which is not included in the complexity guarantee. Our choice of data structures should be based on the actual use case. For example, when should we use a list instead of a vector? Probably when we need sequential or **last-in-first-out** (**LIFO**) access, or when constructing an **abstract-syntax-tree** (**AST**) for a function call.

Concatenation of persistent data structures

While persistent data structures have excellent performance characteristics, the concatenation of two persistent data structures has been a linear time O(N) operation, except for some recent developments. The `concat` function, as of Clojure 1.5, still provides linear time concatenation. Experimental work on **Relaxed Radix Balanced** (**RRB**) trees is going on in the **core.rrb-vector** contrib project (`https://github.com/clojure/core.rrb-vector`) that may provide logarithmic time O(log N) concatenation. Readers interested in the details should refer to the following links:

- The RRB-trees paper at `http://infoscience.epfl.ch/record/169879/files/RMTrees.pdf`
- Phil Bagwel's talk at `http://www.youtube.com/watch?v=K2NYwP90bNs`
- Tiark Rompf's talk at `http://skillsmatter.com/podcast/scala/fast-concatenation-immutable-vectors`

Sequences and laziness

"A seq is like a logical cursor."

– Rich Hickey

Sequences (commonly known as **seqs**) are a way to sequentially consume a succession of data. Like Java iterators, they let a user begin consuming elements from the head and proceed realizing one element after another. However, unlike Java iterators, sequences are immutable. Also, since sequences are only a view of the underlying data, they do not modify the storage structure of the data.

What makes sequences stand apart is they are not data structures per se; rather, they are a data abstraction over a stream of data. The data may be produced by an algorithm or a data source connected to an I/O operation. For example, the function `resultset-seq` accepts a JDBC `java.sql.ResultSet` instance as an argument and produces lazily-realized rows of data as a seq.

Clojure data structures can be turned into sequences using the `seq` function. For example, `(seq [:a :b :c :d])` returns a sequence. Calling `seq` over an empty collection returns `nil`.

Sequences can be consumed by the following functions:

- `first`: returns the head of the sequence
- `rest`: returns the remaining sequence, even if it's empty, after removing the head
- `next`: returns the remaining sequence or nil, if it's empty, after removing the head

Laziness

Clojure is a mostly strict (as in, the opposite of *lazy*) language where one can can choose to explicitly make use of laziness when required. Anybody can create a lazily-evaluated sequence using the `lazy-seq` macro. Some Clojure operations over collections, such as `map`, `filter`, and more, are intentionally lazy.

Laziness simply means that the value is not computed until actually required. Once the value is computed, it is cached so that any future reference to the value need not re-compute it. The caching of the value is called **memoization**. Laziness and memoization often go hand in hand.

Laziness in data structure operations

Laziness and memoization together form an extremely useful combination to keep the single-threaded performance of functional algorithms comparable to its imperative counterparts. For an example, consider the following Java code:

```
List<String> titles = getTitles();
int goodCount = 0;
for (String each: titles) {
    String checksum = computeChecksum(each);
    if (verifyOK(checksum)) {
        goodCount++;
    }
}
```

As is clear from the preceding snippet, it has a linear time complexity, that is, O(N), and the whole operation is performed in a single pass. The comparable Clojure code is as follows:

```
(->> (get-titles)
  (map compute-checksum)
  (filter verify-ok?)
  count)
```

Now, since we know `map` and `filter` are lazy, we can deduce that the Clojure version also has linear time complexity, that is, O(N), and finishes the task in one pass with no significant memory overhead. Imagine for a moment that `map` and `filter` are not lazy—what would be the complexity then? How many passes would it make? It's not just that `map` and `filter` both would have taken one pass, that is, O(N), each; they would each have taken as much memory as the original collection in the worst case due to storing the intermediate results.

It is important to know the value of laziness and memoization in an immutability-emphasizing functional language such as Clojure. They form a basis for **amortization** in persistent data structures, which is about focusing on the overall performance of a composite operation instead of microanalyzing the performance of each operation in it; the operations are tuned to perform faster in those operations that matter the most.

Another important bit of detail is that when a lazy sequence is realized, the data is memoized and stored. On the JVM, all the heap references that are reachable in some way are not garbage collected. So, as a consequence, the entire data structure is kept in memory unless you lose the head of the sequence. When working with lazy sequences using local bindings, make sure you don't keep referring to the lazy sequence from any of the locals. When writing functions that may accept lazy sequence(s), take care that any reference to the lazy seq does not outlive the execution of the function in the form of a closure or such.

Constructing lazy sequences

Now that we know what lazy sequences are, let us try to create a retry counter that should return true only as many times as the retry can be performed. This is shown in the following code:

```
(defn retry? [n]
  (if (<= n 0)
    (cons false (lazy-seq (retry? 0)))
    (cons true (lazy-seq (retry? (dec n))))))
```

The `lazy-seq` macro makes sure that the stack is not used for recursion. We can see that this function would return endless values. Hence, in order to inspect what it returns, we should limit the number of elements as shown in the following code:

```
user=> (take 7 (retry? 5))
(true true true true true false false)
```

Now, let us try using it in a mock fashion:

```
(loop [r (retry? 5)]
  (if-not (first r)
    (println "No more retries")
    (do
      (println 'Retrying)
      (recur (rest r)))))
```

As expected, the output should print `Retrying` five times before printing `No more retries` and exiting as follows:

```
Retrying
Retrying
Retrying
Retrying
Retrying
No more retries
nil
```

Let us take another simpler example of constructing a lazy sequence, which gives us a countdown from a specified number to zero:

```
(defn count-down [n]
  (if (<= n 0)
    '(0)
    (cons n (lazy-seq (count-down (dec n))))))
```

We can inspect the values it returns as follows:

```
user=> (count-down 8)
(8 7 6 5 4 3 2 1 0)
```

Lazy sequences can loop indefinitely without exhausting the stack and can come in handy when working with other lazy operations. To maintain a balance between space-saving and performance, consuming lazy sequences results in the chunking of elements by a factor of 32. That means lazy seqs are realized in a chunk-size of 32, even though they are consumed sequentially.

Custom chunking

The default chunk size 32 may not be optimum for all lazy sequences—you can override the chunking behavior when you need. Consider the snippet below (adapted from Kevin Downey's public gist here: https://gist.github.com/hiredman/324145):

```
(defn chunked-line-seq
  "Returns the lines of text from rdr as a chunked[size] sequence of
  strings. rdr must implement java.io.BufferedReader."
  [^java.io.BufferedReader rdr size]
  (lazy-seq
   (when-let [line (.readLine rdr)]
     (chunk-cons
      (let [buffer (chunk-buffer size)]
        (chunk-append buffer line)
        (dotimes [i (dec size)]
          (when-let [line (.readLine rdr)]
            (chunk-append buffer line)))
        (chunk buffer))
      (chunked-line-seq rdr size)))))
```

As per the previous snippet, the user is allowed to pass a chunk size that is used to produce a lazy sequence of text. A larger chunk size may be useful when processing large text files, such as when processing CSV or logfiles.

Macros and closures

Often times, we define a macro so as to turn the parameter body of code into a closure and delegate it to a function. See the following example:

```
(defmacro do-something
  [& body]
  `(do-something* (fn [] ~@body)))
```

When using such code, if the body binds a local to a lazy sequence it may be retained longer than necessary, likely with bad consequences on memory consumption and performance. Fortunately, this can be easily fixed:

```
(defmacro do-something
  [& body]
  `(do-something* (^:once fn [] ~@body)))
```

Notice the `^:once` hint, which makes the Clojure compiler clear the closed-over references, thus avoiding the problem. Readers interested in the details should refer to `http://cljme.cgrand.net/2013/09/11/macrosclosuresandunexpectedobjectretention/`.

Transients

Earlier in this chapter, we discussed the virtues of immutability and the pitfalls of mutability. However, even though unguarded mutability is fundamentally unsafe, it also has very good single-threaded performance. Now, what if there was a way to restrict the mutable operation in a local context in order to provide safety guarantees? That would be equivalent to combining the performance advantage and local safety guarantees. This can be done with the abstraction called **transients**, which is provided by Clojure.

First, let us verify that it is safe:

```
user=> (let [t (transient [:a])]
  @(future (conj! t :b)))
IllegalAccessError Transient used by non-owner thread  clojure.lang.
PersistentVector$TransientVector.ensureEditable (PersistentVector.
java:463)
```

As we can see, a transient created in one thread cannot be accessed by another:

```
user=> (let [t (transient [:a])] (seq t))

IllegalArgumentException Don't know how to create ISeq from: clojure.
lang.PersistentVector$TransientVector  clojure.lang.RT.seqFrom (RT.
java:505)
```

So, transients cannot be converted to seqs. Hence, they cannot participate in the birthing of new persistent data structures and leak out of the scope of execution. Consider the following code:

```
(let [t (transient [])]
  (conj! t :a)
  (persistent! t)
  (conj! t :b))
IllegalAccessError Transient used after persistent! call
clojure.lang.PersistentVector$TransientVector.ensureEditable
(PersistentVector.java:464)
```

The `persistent!` function permanently converts a transient into an equivalent persistent data structure. Effectively, transients are for one-time use only.

Conversion between persistent and transient data structures (functions `transient` and `persistent!`) is constant time, that is, an O(1) operation. Transients can be created from unsorted maps, vectors, and sets only. The functions that mutate transients are: `conj!`, `disj!`, `pop!`, `assoc!`, and `dissoc!`. Read-only operations such as `get`, `nth`, `count`, and many more work as usual on transients, but functions such as `contains?` and those that imply seqs, such as `first`, `rest`, and `next`, do not.

Fast repetition

The function `clojure.core/repeatedly` lets us execute a function many times and produce a lazy sequence of results. Peter Taoussanis, in his open source serialization library **Nippy** (`https://github.com/ptaoussanis/nippy`), wrote a transient-aware variant that performs significantly better. It is reproduced, as shown, with his permission (note that the arity of the function is not the same as `repeatedly`):

```
(defn repeatedly*
  "Like `repeatedly` but faster and returns given collection type."
  [coll n f]
  (if-not (instance? clojure.lang.IEditableCollection coll)
    (loop [v coll idx 0]
      (if (>= idx n)
        v
        (recur (conj v (f)) (inc idx))))
    (loop [v (transient coll) idx 0]
      (if (>= idx n)
        (persistent! v)
        (recur (conj! v (f)) (inc idx))))))
```

Performance miscellanea

Besides the major abstractions we saw earlier in the chapter, there are other smaller, but nevertheless very performance-critical, parts of Clojure that we will see in this section.

Disabling assertions in production

Assertions are very useful to catch logical errors in the code during development, but they impose a runtime overhead that you may like to avoid in production environment. Since `clojure.core/*assert*` is a compile time var, the assertions can be silenced either by binding `*assert*` to false or by using `alter-var-root` before the code is loaded. Unfortunately, both the techniques are cumbersome to use. Paul Stadig's library called **assertions** (`https://github.com/pjstadig/assertions`) helps with this exact use case by enabling or disabling assertions via command-line argument `-ea` to the Java Runtime. You must include it in your Leiningen `project.clj` file as a dependency to use it:

```
:dependencies [;; other dependencies…
               [pjstadig/assertions "0.1.0"]]
```

You must use this library's `assert` macro instead of Clojure's own, so each `ns` block in the application should look something like this:

```
(ns example.core
  (:refer-clojure :exclude [assert])
  (:require [pjstadig.assertions :refer [assert]]))
```

When running the application, you should include the `-ea` argument to the JRE to enable assertions, whereas its exclusion implies no assertion at runtime:

```
$ JVM_OPTS=-ea lein run -m example.core
$ java -ea -jar example.jar
```

Note that this usage will not automatically avoid assertions in the other code and dependency libraries that use `clojure.core/assert`.

Destructuring

Destructuring is one of Clojure's built-in mini languages and, arguably, a top productivity booster during development. This feature leads to the parsing of values to match the left-hand side of the binding forms. The more complicated the binding form is, the more work needs to be done. Not surprisingly, this has a little bit of performance overhead.

It is easy to avoid this overhead by using explicit functions to unravel data in the tight loops and other performance-critical code. After all, it all boils down to making the program work less and do more.

Recursion and tail-call optimization (TCO)

Functional languages have this concept of tail-call optimization related to recursion. So, the idea is that when a recursive call is at the tail position, it does not take up space on the stack for recursion. Clojure supports a form of user-assisted recursive call to make sure the recursive calls do not blow the stack. This is kind of an imperative looping, but it is extremely fast.

When carrying out computations, it may make a lot of sense to use `loop-recur` in the tight loops instead of iterating over synthetic numbers. For example, let's say we want to add all odd integers from zero through 1,000,000. Let's compare the code:

```
(defn oddsum-1 [n]  ; using iteration
  (->> (range (inc n))
    (filter odd?)
    (reduce +)))
(defn oddsum-2 [n]  ; using loop-recur
  (loop [i 1 s 0]
    (if (> i n)
      s
      (recur (+ i 2) (+ s i)))))
```

When we run the code we get interesting results:

```
user=> (time (oddsum-1 1000000))
"Elapsed time: 109.314908 msecs"

250000000000
user=> (time (oddsum-2 1000000))
"Elapsed time: 42.18116 msecs"

250000000000
```

The `time` macro is far from perfect as the performance benchmarking tool, but the relative numbers indicate a trend—in the subsequent chapters, we will look at the Criterium library for more scientific benchmarking. Here, we use `loop-recur` not only to iterate faster, but we are also able to change the algorithm itself by iterating only about half as many times as we did in the other example.

Premature end in reduce

When accumulating over a collection, in some cases, we may want to end it prematurely. Prior to Clojure 1.5, `loop-recur` was the only way to do it. When using `reduce`, we can do just that using the `reduced` function introduced in Clojure 1.5 as shown:

```
;; let coll be a collection of numbers
(reduce (fn ([x] x)
            ([x y] (if (or (zero? x) (zero? y))
                     (reduced 0)
                     (* x y))))
        coll)
```

Here, we multiply all numbers in a collection and, upon finding any of the numbers as zero, immediately return the result zero instead of continuing till the last element.

Multimethods versus protocols

Multimethods are a fantastic expressive abstraction for a polymorphic dispatch on a dispatch function's return value. The dispatch functions associated with a multimethod are maintained at runtime and looked up whenever a multimethod call is invoked. While multimethods provide a lot of flexibility in determining the dispatch, the performance overhead is simply too high compared to that of protocol implementations.

Protocols (`defprotocol`) are implemented using `reify`, records (`defrecord`), and types (`deftype`, `extend-type`) in Clojure. This is a big discussion topic—since we are discussing the performance characteristics, it should suffice to say that protocol implementations dispatch on polymorphic types and are significantly faster than multimethods. Protocols and types are generally the implementation detail of an API, so they are usually fronted by functions.

Due to the multimethods' flexibility, they still have a place. However, in performance critical code, it is advisable to use protocols, records, and types instead.

Inlining

It is well known that macros are expanded inline at the call site and avoid a function call. As a consequence, there is a small performance benefit. There is also a `definline` macro that lets you write a function just like a normal macro. It creates an actual function that gets inlined at the call site:

```
(def PI Math/PI)
(definline circumference [radius]
  `(* 2 PI ~radius))
```

> Note that the JVM also analyzes the code it runs and does its own inlining of code at runtime. While you may choose to inline the hot functions, this technique is known to give only a modest performance boost.

When we define a `var` object, its value is looked up each time it is used. When we define a `var` object using a `:const` meta pointing to a `long` or `double` value, it is inlined from wherever it is called.

```
(def ^:const PI Math/PI)
```

This is known to give a decent performance boost when applicable. See the following example:

```
user=> (def a 10)
user=> (def ^:const b 10)
user=> (def ^:dynamic c 10)
user=> (time (dotimes [_ 100000000] (inc a)))
"Elapsed time: 1023.745014 msecs"
nil
user=> (time (dotimes [_ 100000000] (inc b)))
"Elapsed time: 226.732942 msecs"
nil
user=> (time (dotimes [_ 100000000] (inc c)))
"Elapsed time: 1094.527193 msecs"
nil
```

Summary

Performance is one of the cornerstones of Clojure's design. Abstractions in Clojure are designed for simplicity, power, and safety with performance firmly in mind. We saw the performance characteristics of various abstractions and also how to make decisions about abstractions depending on performance use cases.

In the next chapter, we will see how Clojure interoperates with Java and how we can extract Java's power to derive optimum performance.

3
Leaning on Java

Being hosted on the **Java Virtual Machine** (**JVM**), there are several aspects of Clojure in which it really helps to know about the Java language and platform. The need is not only due to interoperability with Java or understanding its implementation, but also for performance reasons. In certain cases, Clojure may not generate optimized JVM bytecode by default; in some other cases, you may want to go beyond the performance Clojure data structures offer — you can use the Java alternatives via Clojure to get better performance. This chapter discusses those aspects of Clojure. In this chapter, we will discuss:

- Inspecting Java generated from Clojure source
- Numerics and primitives
- Working with arrays
- Reflection and type hinting

Inspect the equivalent Java source for Clojure code

Inspecting the equivalent Java source for a given Clojure code provides a great insight into how that might impact its performance. However, Clojure generates only Java bytecodes at runtime unless we compile a namespace out to disk. When developing with Leiningen, only selected namespaces under the `:aot` vector in the `project.clj` file are output as the compiled `.class` files containing bytecodes. Fortunately, an easy and quick way to know the equivalent Java source for Clojure code is to ahead-of-time (AOT) compile namespaces and then decompile the bytecodes into equivalent Java sources using a Java bytecode decompiler.

There are several commercial and open source Java bytecode decompilers available. One of the open source decompilers we will discuss here is **JD-GUI**, which you can download from its website (`http://jd.benow.ca/#jd-gui`). Use a version suitable for your operating system.

Create a new project

Let us see how exactly to arrive at the equivalent Java source code from Clojure. Create a new project using Leiningen: `lein new foo`. Then edit the `src/foo/core.clj` file with a `mul` function to find out the product of two numbers:

```
(ns foo.core)

(defn mul [x y]
  (* x y))
```

Compile Clojure sources into Java bytecode

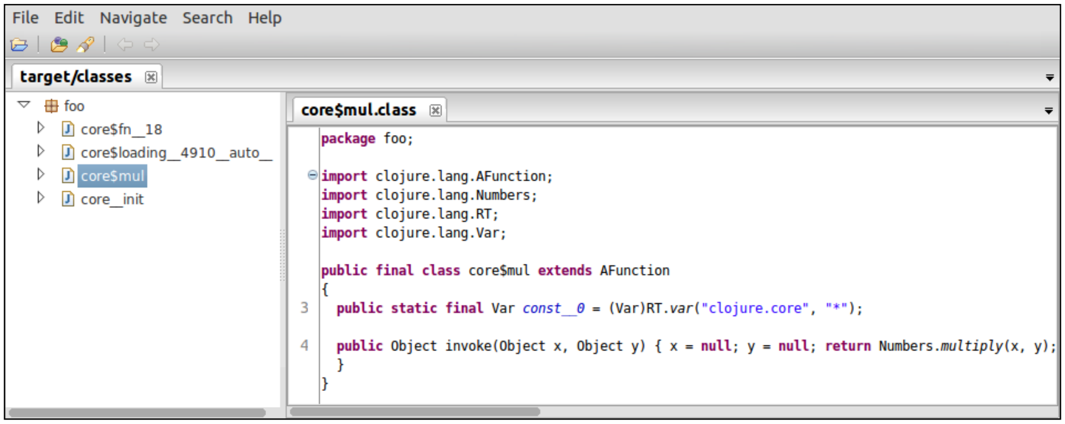
To compile Clojure sources into bytecodes and output them as `.class` files, run the `lein compile :all` command. This will create the `.class` files in the `target/classes` directory of the project as follows:

```
target/classes/
`-- foo
    |-- core$fn__18.class
    |-- core__init.class
    |-- core$loading__4910__auto__.class
    `-- core$mul.class
```

You can see that the `foo.core` namespace has been compiled into three `.class` files.

Decompile the .class files into Java source

Assuming that you have already installed JD-GUI, decompiling the `.class` files is as simple as opening them using the JD-GUI application. Open the JD-GUI application and then open a compiled class file using the menu option:



```
package foo;

import clojure.lang.AFunction;
import clojure.lang.Numbers;
import clojure.lang.RT;
import clojure.lang.Var;

public final class core$mul extends AFunction
{
  public static final Var const__0 = (Var)RT.var("clojure.core", "*");

  public Object invoke(Object x, Object y) { x = null; y = null; return Numbers.multiply(x, y);
  }
}
```

On inspection, the code for the `foo.core/mul` function looks as follows:

```
package foo;

import clojure.lang.AFunction;
import clojure.lang.Numbers;
import clojure.lang.RT;
import clojure.lang.Var;

public final class core$mul extends AFunction
{
  public static final Var const__0 = (Var)RT.var("clojure.core", "*");

  public Object invoke(Object x, Object y) { x = null; y = null;
return Numbers.multiply(x, y);
  }
}
```

It is easy to understand from the decompiled Java source that the `foo.core/mul` function is an instance of the `core$mul` class in the `foo` package extending the `clojure.lang.AFunction` class. We can also see that the argument types are of the `Object` type, which implies the numbers will be boxed. In a similar fashion, you can decompile class files of any Clojure code to inspect the equivalent Java code. If you can combine this with knowledge about Java types and potential reflection and boxing, you can find the suboptimal spots in code and focus on what to improve upon.

Numerics, boxing, and primitives

Numerics are scalars. The discussion on numerics was deferred until this chapter for the sole reason that the numerics implementation in Clojure has strong Java underpinnings. Since Version 1.3, Clojure has settled with 64-bit numerics as the default. Now, `long` and `double` are idiomatic and are the default numeric types. Note that these are primitive Java types, not objects. Primitives in Java lead to high performance and have several optimizations associated with them at compiler and runtime levels. A local primitive is created on the stack (hence does not contribute to heap allocation and GC) and can be accessed directly without any kind of dereferencing. In Java, there also exist object equivalents of the numeric primitives, known as boxed numerics — these are regular objects that are allocated on the heap. The boxed numerics are also immutable objects, which mean not only does the JVM need to dereference the stored value when reading it, but also needs to create a new boxed object when a new value needs to be created.

It should be obvious that boxed numerics are slower than their primitive equivalents. The Oracle HotSpot JVM, when started with the `-server` option, aggressively inlines those functions (on frequent invocation) that contain a call to primitive operations. Clojure automatically uses primitive numerics at several levels. In the `let` blocks, `loop` blocks, arrays, and arithmetic operations (`+`, `-`, `*`, `/`, `inc`, `dec`, `<`, `<=`, `>`, `>=`), primitive numerics are detected and retained. The following table describes the primitive numerics with their boxed equivalents:

Primitive numeric type	**Boxed equivalent**
byte (1 byte)	`java.lang.Byte`
short (2 bytes)	`java.lang.Short`
int (4 bytes)	`java.lang.Integer`
float (4 bytes)	`java.lang.Float`
long (8 bytes)	`java.lang.Long`
double (8 bytes)	`java.lang.Double`

In Clojure, sometimes you may find that numerics are passed or returned as boxed objects to or from functions due to the lack of type information at runtime. Even if you have no control over such functions, you can coerce the values to be treated as primitives. The `byte`, `short`, `int`, `float`, `long`, and `double` functions create primitive equivalents from given boxed numeric values.

One of the Lisp traditions is to provide correct (http://en.wikipedia.org/wiki/Numerical_tower) arithmetic implementation. A lower type should not truncate values when overflow or underflow happens, but rather should be promoted to construct a higher type to maintain correctness. Clojure follows this constraint and provides autopromotion via prime (http://en.wikipedia.org/wiki/Prime_(symbol)) functions: +', -', *', inc', and dec'. Autopromotion provides correctness at the cost of some performance.

There are also arbitrary length or precision numeric types in Clojure that let us store unbounded numbers but have poorer performance compared to primitives. The bigint and bigdec functions let us create numbers of arbitrary length and precision.

If we try to carry out any operations with primitive numerics that may result in a number beyond its maximum capacity, the operation maintains correctness by throwing an exception. On the other hand, when we use the prime functions, they autopromote to provide correctness. There is another set of operations called unchecked operations which do not check for overflow or underflow and can potentially return incorrect results. In some cases, they may be faster than regular and prime functions. Such functions are unchecked-add, unchecked-subtract, unchecked-multiply, unchecked-divide, unchecked-inc, unchecked-dec, unchecked-negate, and unchecked-remainder. We can also enable unchecked math behavior for regular arithmetic functions using the *unchecked-math* var; simply include the following in your source code file:

```
(set! *unchecked-math* true)
```

One of the common needs in arithmetic is division that is used to find out the quotient and remainder after a natural number division. Clojure's / function provides a rational number division yielding a ratio and the mod function provides a true modular arithmetic division. These functions are slower than the quot and rem functions that compute the division quotient and the remainder respectively.

Arrays

Beside objects and primitives, Java has a special type of collection storage structure called arrays. Once created, arrays cannot be grown or shrunk without copying data and creating another array to hold the result. Array elements are always homogeneous in type. Array elements are like places that you can mutate to hold new values. Unlike collections such as list and vector, arrays can contain primitive elements, which make them a very fast storage mechanism without **Garbage Collection** (**GC**) overhead.

Arrays often form a basis for mutable data structures. For example, Java's `java.lang.ArrayList` implementation uses arrays internally. In Clojure, arrays can be used for fast numeric storage and processing, efficient algorithms, and so on. Unlike collections, arrays can have one or more dimensions. So, you could lay out data in an array such as a matrix or cube. Let us see Clojure's support for arrays:

Description	Example	Notes
Create array	`(make-array Integer 20)`	Array of type (boxed) integer
	`(make-array Integer/TYPE 20)`	Array of primitive type integer
	`(make-array Long/TYPE 20 10)`	Two-dimensional array of primitive long
Create array of primitives	`(int-array 20)`	Array of primitive integer of size 20
	`(int-array [10 20 30 40])`	Array of primitive integer created from a vector
Create array from coll	`(to-array [10 20 30 40])`	Array from sequable
	`(to-array-2d [[10 20 30][40 50 60]])`	Two-dimensional array from collection
Clone an array	`(aclone (to-array [:a:b:c]))`	
Get array element	`(aget array-object 0 3)`	Get element at index [0][3] in a 2-D array
Mutate array element	`(aset array-object 0 3:foo)`	Set obj :foo at index [0][3] in a 2-D array
Mutate primitive array element	`(aset-int int-array-object 2 6 89)`	Set value 89 at index [2][6] in 2-D array
Find length of array	`(alength array-object)`	`alength` is significantly faster than `count`

Description	Example	Notes

Map over an array	`(def a (int-array [10 20 30 40 50 60]))`
	`(seq`
	`  (amap a idx ret`
	`    (do (println idx (seq ret))`
	`      (inc (aget a idx)))))`
		Unlike map, `amap` returns a non-lazy array, which is significantly faster over array elements. Note that `amap` is faster only when properly type hinted. See next section for more on type hinting.

Reduce over an array	`(def a (int-array [10 20 30 40 50 60]))`
	`(areduce a idx ret 0`
	`  (do (println idx ret)`
	`    (+ ret idx)))`
		Unlike `reduce`, `areduce` is significantly faster over array elements. Note that `reduce` is faster only when properly type hinted. See next section for more on type hinting.

Cast to primitive arrays	`(ints int-array-object)`	Used with type hinting (see next section)

Like `int-array` and `ints`, there are functions for other types as well:

Array construction function	**Primitive-array casting function**	**Type hinting (does not work for vars)**	**Generic array type hinting**
boolean-array	booleans	`^booleans`	`^"[Z"`
byte-array	bytes	`^bytes`	`^"[B"`
short-array	shorts	`^shorts`	`^"[S"`
char-array	chars	`^chars`	`^"[C"`
int-array	ints	`^ints`	`^"[I"`
long-array	longs	`^longs`	`^"[J"`
float-array	floats	`^floats`	`^"[F"`
double-array	doubles	`^doubles`	`^"[D"`
object-array	--	`^objects`	`^"[Ljava.lang.Object"`

Arrays are favored over other data structures mainly due to performance and sometimes due to interop. Take extreme care to type hint the arrays and use the appropriate functions to work with them.

Reflection and type hints

Sometimes, as Clojure is dynamically typed, the Clojure compiler is unable to figure out the type of object to invoke a certain method. In such cases, Clojure uses reflection, which is considerably slower than direct method dispatch. Clojure's solution to this is something called type hints. Type hints are a way to annotate arguments and objects with static types so that the Clojure compiler can emit bytecodes for efficient dispatch.

The easiest way to know where to put type hints is to turn on reflection warning in the code. Consider this code that determines the length of a string:

```
user=> (set! *warn-on-reflection* true)
true
user=> (def s "Hello, there")
#'user/s
user=> (.length s)
Reflection warning, NO_SOURCE_PATH:1 - reference to field length can't
be resolved.
12
user=> (defn str-len [^String s] (.length s))
#'user/str-len
user=> (str-len s)
12
user=> (.length ^String s)  ; type hint when passing argument
12
user=> (def ^String s "Hello, there")  ; type hint at var level
#'user/s
user=> (.length s)  ; no more reflection warning
12
```

When working on a project, you may want reflection warning to be turned on for all files. You can do this easily in Leiningen. Just put the following entry in your `project.clj` file:

```
:profiles {:dev {:global-vars {*warn-on-reflection* true}}}
```

This will automatically turn on reflection warning every time you begin any kind of invocation via Leiningen in the dev workflow such as REPL and test.

Array of primitives

Recall the examples on `amap` and `areduce` from the previous section. If we run them with reflection warning on, we'd be warned that it uses reflection. Let's type hint them:

```
(def a (int-array [10 20 30 40 50 60]))
;; amap example
(seq
  (amap ^ints a idx ret
    (do (println idx (seq ret))
      (inc (aget ^ints a idx)))))
;; areduce example
(areduce ^ints a idx ret 0
  (do (println idx ret)
    (+ ret idx)))
```

Note that the primitive array hint `^ints` does not work at the var level. So, it would not work if you defined the var `a` like the following:

```
(def ^ints a (int-array [10 20 30 40 50 60]))  ; wrong, will complain
later
(def ^"[I" a (int-array [10 20 30 40 50 60]))  ; correct
```

This notation is for an array of integers. Other primitive array types have similar type hints. Refer to the previous section for type hinting for various primitive array types.

Primitives

Type hinting of primitive locals is neither required nor allowed. However, you can type hint function arguments as primitives. Clojure allows up to four arguments in functions to be type hinted:

```
(defn do-something
  [^long a ^long b ^long c ^long d]
  ..)
```

> Boxing may result in something not always being a primitive. In those cases, you can coerce those using respective primitive types.

Macros and metadata

In macros, type hinting does not work the way it does in other parts of the code. Since macros are about transforming the **Abstract Syntax Tree** (**AST**) , we need to have a mental map of the transformation and we should add type hints as metadata in the code. For example, if `str-len` is a macro to find the length of a string, we make use of the following code:

```
(defmacro str-len
  [s]
  `(.length ~(with-meta s {:tag String})))
```

In the preceding code, we alter the metadata of the symbol `s` by tagging it with the type `String`, which happens to be the `java.lang.String` class in this case. For array types, we can use `"[Ljava.lang.String"` for an array of string objects and similarly for others.

Type hinting via metadata also works with functions, albeit in a different notation:

```
(defn foo [] "Hello")
(defn ^String foo [] "Hello")
(defn ^{:tag String} foo [] "Hello")
```

Except for the first example in the preceding snippet, they are type hinted to return the `java.lang.String` type.

Miscellaneous

In a type (as in `deftype`), the mutable instance variables can be optionally annotated as `^:volatile-mutable` or `^:unsynchronized-mutable` for concurrent behavior, covered in *Chapter 5*, *Concurrency*. For example:

```
(deftype Counter [^:volatile-mutable ^long now]
  ..)
```

Unlike `defprotocol`, the `definterface` macro lets us provide a return type hint for methods:

```
(definterface Foo
  (^long doSomething [^long a ^double b]))
```

The proxy-super macro (which is used inside the proxy macro) is a special case where you cannot directly apply a type hint. The reason being that it relies on the implicit this object that is automatically created by the proxy macro. In this case, you must explicitly bind this to a type:

```
(proxy [Object][]
  (equals [other]
    (let [^Object this this]
      (proxy-super equals other))))
```

Type hinting is quite important for performance in Clojure. Fortunately, we need to type hint only when required, and it's easy to find out when. In many cases, a gain from type hinting overshadows the gains from code inlining.

Using array/numeric libraries for efficiency

You may have noticed in the previous sections that, when working with numerics, performance depends a lot on whether the data is based on arrays and primitives. It may take a lot of meticulousness on the programmer's part to correctly coerce data into primitives and arrays at all stages of the computation in order to achieve optimum efficiency. Fortunately, the high performance enthusiasts from the Clojure community realized this issue early on and created some dedicated open source libraries to mitigate the problem.

HipHip

HipHip is a Clojure library built to work with arrays of primitive types. It provides a safety net; that is, it strictly accepts only primitive array arguments to work with. As a result, passing silently boxed primitive arrays as arguments always results in an exception. HipHip macros and functions rarely need the programmer to type hint anything during the operations. It supports arrays of primitive types such as int, long, float, and double.

The HipHip project is available at https://github.com/Prismatic/hiphip.

As of the time of writing, HipHip's most recent version is 0.1.0. This version supports Clojure 1.5.x and is tagged as an Alpha release. There is a standard set of operations provided by HipHip for arrays of all of the four primitive types: integer array operations are in the namespace `hiphip.int`, double precision array operations in `hiphip.double`, and so on. The operations are all type hinted for the respective types. All of the operations for `int`, `long`, `float`, and `double` in respective namespaces are essentially the same except for the array type:

Category	Function/macro	Description
Core functions	`aclone`	Like `clojure.core/aclone` for primitives
	`alength`	Like `clojure.core/alength` for primitives
	`aget`	Like `clojure.core/aget` for primitives
	`aset`	Like `clojure.core/aset` for primitives
	`ainc`	Increments an array element by specified value
Equiv `hiphip.array` operations	`amake`	Makes a new array and fills values computed by expression
	`areduce`	Like `clojure.core/areduce` with HipHip array bindings
	`doarr`	Like `clojure.core/doseq` with HipHip array bindings
	`amap`	Like `clojure.core/for` but it creates a new array
	`afill!`	Like preceding `amap` but it overwrites an array argument
Mathy operations	`asum`	Compute sum of array elements using expression
	`aproduct`	Compute product of array elements using expression
	`amean`	Compute mean over the array elements
	`dot-product`	Compute dot product of two arrays
Finding minimum/maximum, Sorting	`amax-index`	Find maximum value in array and return the index
	`amax`	Find maximum value in an array and return it
	`amin-index`	Find minimum value in an array and return the index
	`amin`	Find minimum value in an array and return it
	`apartition!`	Three-way partition of array: less, equal, greater than pivot
	`aselect!`	Gather smallest `k` elements at the beginning of an array

Category	Function/macro	Description
	`asort!`	Sort array in-place using Java's built-in implementation
	`asort-max!`	Sort array in-place gathering top `k` elements to the end
	`asort-min!`	Sort array in-place gathering top `k` elements to the top
	`apartition-indices!`	Like `apartition!` but mutates index-array instead of values
	`aselect-indices!`	Like `aselect!` but mutates index-array instead of values
	`asort-indices!`	Like `asort!` but mutates index-array instead of values
	`amax-indices`	Get index-array; last `k` indices pointing to max `k` values
	`amin-indices`	Get index-array; first `k` indices pointing to min `k` values

To include HipHip as a dependency in your Leiningen project, specify it in `project.clj`:

```
:dependencies [;; other dependencies
               [prismatic/hiphip "0.1.0"]]
```

As an example of how to use HipHip, let us see how to compute normalized values of an array:

```
(require '[hiphip.double :as hd])

(def xs (double-array [12.3 23.4 34.5 45.6 56.7 67.8]))

(let [s (hd/asum xs)] (hd/amap [x xs] (/ x s)))
```

Unless we make sure that `xs` is an array of primitive doubles, HipHip will throw `ClassCastException` when the type is incorrect and `IllegalArgumentException` in other cases. I recommend exploring the HipHip project for more insight into using it effectively.

primitive-math

We can set `*warn-on-reflection*` to `true` to let Clojure warn us when reflection is used at invocation boundaries. However, when Clojure has to implicitly use reflection to perform math, the only resort is to either use a profiler or compile the Clojure source down to bytecode and analyze boxing and reflection with a decompiler. This is where the `primitive-math` library helps by producing extra warnings and throwing exceptions.

The `primitive-math` library is available at `https://github.com/ztellman/primitive-math`.

As of the time of writing, `primitive-math` is at Version 0.1.3; you can include it as a dependency in your Leiningen project by editing `project.clj` as follows:

```
:dependencies [;; other dependencies
                [primitive-math "0.1.3"]]
```

The following code is how it can be used (recall the example from the *Decompile the .class files into Java source* section):

```
;; enable reflection warnings for extra warnings from primitive-math
(set! *warn-on-reflection* true)
(require '[primitive-math :as pm])
(defn mul [x y] (pm/* x y))  ; produces reflection warning
(mul 10.3 2)                        ; throws exception
(defn mul [^long x ^long y] (pm/* x y))  ; no warning after type hint
(mul 10.3 2)  ; returns 20
```

> The math operations in `primitive-math` (like HipHip) are implemented via macros. Therefore, they cannot be used as higher order functions and as a consequence, may not compose well with other code. I recommend exploring the project to see what suits your program use case.

Resorting to Java and native code

In a handful of cases, where the lack of imperative, stack-based, mutable variables in Clojure may make the code not perform as well as Java, we may need to evaluate alternatives to make it faster. I would advise that you consider writing such code directly in Java for better performance.

Another consideration is to use native OS capabilities, such as memory-mapped buffers (`http://docs.oracle.com/javase/7/docs/api/java/nio/MappedByteBuffer.html`) or files and unsafe operations (`http://highlyscalable.wordpress.com/2012/02/02/direct-memory-access-in-java/`). Note that unsafe operations are potentially hazardous and are not recommended in general. Such times are also an opportunity to consider writing performance-critical pieces of code in C or C++ and accessing them via **Java Native Interface** (**JNI**).

Proteus – mutable locals in Clojure

Proteus is an open source Clojure library that lets you treat a local like a local *variable*, thereby allowing its unsynchronized mutation within the local scope only. Note that this library depends on the internal implementation structure of Clojure as of Clojure 1.5.1. The Proteus project is available at `https://github.com/ztellman/proteus`.

You can include Proteus as a dependency in the Leiningen project by editing `project.clj`:

```
:dependencies [;;other dependencies
                [proteus "0.1.4"]]
```

Using Proteus in code is straightforward, as shown in the following code snippet:

```
(require '[proteus :as p])
(p/let-mutable [a 10]
  (println a)
  (set! a 20)
  (println 20))
;; Output below:
;; 10
;; 20
```

Since Proteus allows mutation only in the local scope, the following throws an exception:

```
(p/let-mutable [a 10 add2! (fn [x] (set! x (+ 2 x)))]
  (add2! a)
  (println a))
```

The mutable locals are very fast and may be quite useful in tight loops. Proteus is unconventional by Clojure idioms, but it may give the required performance boost without having to write Java code.

Summary

Clojure has strong Java interoperability and underpinning, and due to which, programmers can leverage the performance benefits nearing Java. For performance-critical code, sometimes it is necessary to know how Clojure interacts with Java and how to turn the right knobs. Numerics is a key area where Java interoperability is required to get optimum performance. Type hinting is another important performance trick that is frequently useful. There are several open source Clojure libraries that make such activities easier for the programmer.

In the next chapter, we will dig deeper below Java and see how the hardware and the JVM stack play a key role to offer the performance we get, what their constraints are, and how to use the understanding to get better performance.

4
Host Performance

In the previous chapters, we noted how Clojure interoperates with Java. In this chapter we will go a bit deeper to understand the internals better. We will touch upon several layers of the entire stack, but our major focus will be the **Java Virtual Machine** (**JVM**), in particular the Oracle HotSpot JVM, though there are several JVM vendors to choose from (`http://en.wikipedia.org/wiki/List_of_Java_virtual_machines`). At the time of writing, Oracle JDK 1.7 is the latest stable release, and early OpenJDK 1.8 milestones are available. In this chapter we will discuss:

- How the hardware subsystems function from a performance viewpoint
- Organization of the JVM internals and how that is related to performance
- How to measure the amount of space occupied by various objects in the heap
- How to profile Clojure code for latency using Criterium

The hardware

There are various hardware components that may impact the performance of software in different ways. The processors, caches, memory subsystem, I/O subsystems, and so on, all have varying degrees of performance impact depending upon the use cases. In the following sections we will look into each of those aspects.

Processors

Since about the late 1980s, microprocessors have been employing pipelining and instruction-level parallelism to speed up their performance. Processing an instruction at the CPU level consists of typically four cycles: **fetch**, **decode**, **execute**, and **writeback**. Modern processors optimize the cycles by running them in parallel—while one instruction is executed, the next instruction is being decoded and the one after that is being fetched, and so on. This style is called **instruction pipelining**.

In practice, in order to speed up execution even further, the stages are subdivided into many shorter stages, thus leading to deeper super-pipeline architecture. The length of the longest stage in the pipeline limits the clock speed of the CPU. By splitting stages into substages, the processor can be run at a higher clock speed where more cycles are required for each instruction, but the processor still completes one instruction per cycle. Since there are more cycles per second now, we get better performance in terms of throughput per second even though the latency of each instruction is now higher.

Branch prediction

The processor must fetch and decode instructions in advance even when it encounters instructions of the conditional `if-then` form. Consider an equivalent of the `(if (test a) (foo a) (bar a))` Clojure expression. The processor must choose a branch to fetch and decode; the question is, should it fetch the `if` branch or the `else` branch? Here, the processor makes a guess as to which instruction to fetch/decode. If the guess turns out to be correct, it is a performance gain as usual; otherwise, the processor has to throw away the result of the fetch/decode process and start on the other branch afresh.

Processors deal with branch prediction using an on-chip *branch prediction table*. It contains recent code branches and two bits per branch indicating whether or not the branch was taken, while also accommodating one-off not-taken occurrences.

Today, branch prediction is extremely important in processors for performance, so modern processors dedicate hardware resources and special *predication* instructions to improve the prediction accuracy and lower the cost of *mispredict penalties*.

Instruction scheduling

High-latency instructions and branching usually lead to empty cycles in the instruction pipeline known as **stalls** or **bubbles**. These cycles are often used to do other work by the means of *instruction reordering*. Instruction reordering is implemented at the hardware level via **out of order execution** and at the compiler level via **compile time instruction scheduling** (also called **static instruction scheduling**).

The processor needs to remember the dependencies between instructions when carrying out the out-of-order execution. This cost is somewhat mitigated by using renamed registers, wherein register values are stored into / loaded from memory locations, potentially on different physical registers, so that they can be executed in parallel. This necessitates that out-of-order processors always maintain a mapping of instructions and corresponding registers they use, which makes their design complex and power hungry. With a few exceptions, almost all high-performance CPUs today have out-of-order designs.

Good compilers are usually extremely aware of processors, and they are capable of optimizing the code by rearranging processor instructions in a way that there are fewer bubbles in the processor instruction pipeline. A few high-performance CPUs still rely on only static instruction reordering instead of out-of-order instruction reordering and in turn save chip area due to simpler design—the saved area is used to accommodate extra cache or CPU cores. Low-power processors, such as those from the ARM and Atom family, use in-order design. Unlike most CPUs, modern GPUs use in-order design with deep pipelines that are compensated by very fast context switching. This leads to high latency and high throughput on GPUs.

Threads and cores

Concurrency and parallelism via context switches, hardware threads, and cores are very common today, and we have accepted them as a norm to implement in our programs. However, we should know why we needed such a design in the first place. Most of the real-world code we write today does not have more than a modest scope for instruction-level parallelism. Even with hardware-based, out-of-order execution and static instruction reordering, no more than two instructions per cycle are truly parallel. Hence, another potential source of instructions that can be pipelined and executed in parallel are the programs other than the currently running one.

The empty cycles in a pipeline can be dedicated to other running programs which assume there are other currently running programs that need the processor's attention. **Simultaneous multithreading** (**SMT**) is a hardware design that enables such kinds of parallelism. Intel implements SMT named **HyperThreading** in some of its processors. While SMT presents a single physical processor as two or more logical processors, a true multiprocessor system executes one thread per processor, thus achieving simultaneous execution. A multicore processor includes two or more processors per chip, but has the properties of a multiprocessor system.

In general, multicore processors significantly outperform SMT processors. Performance on SMT processors can vary by the use case. It peaks in those cases where code is highly variable or where threads do not compete for the same hardware resources, and dips when the threads are cache-bound on the same processor. What is also important is that some programs are simply not inherently parallel. In such cases it may be hard to make them go faster without explicit use of threads in the program.

Memory systems

It is important to understand the memory performance characteristics to know the likely impact on the programs we write. Data-intensive programs that are also inherently parallel, such as audio/video processing and scientific computation, are largely limited by memory bandwidth, not by the processor. Adding processors would not make them faster unless the memory bandwidth is also increased. Consider another class of programs, such as 3D graphics rendering or database systems, that are limited mainly by memory latency but not the memory bandwidth. SMT can be highly-suitable for such programs where threads do not compete for the same hardware resources.

Memory access roughly constitutes a quarter of all instructions executed by a processor. A code block typically begins with memory load instructions and the remaining portion depends on the loaded data. This stalls the instructions and prevents large-scale, instruction-level parallelism. As if that was not bad enough, even superscalar processors (which can issue more than one instruction per clock cycle) can issue at most two memory instructions per cycle. Building fast memory systems is limited by natural factors such as the speed of light. This impacts the signal round trip to the RAM. This is a natural hard limit and any optimization can only work around it.

Data transfer between the processor and motherboard chipset is one of the factors that induce memory latency. This is countered using a faster **front-side bus** (**FSB**). Nowadays, most modern processors fix this problem better by integrating the memory controller directly at the chip level. The significant difference between the processor versus memory latencies is known as **memory wall**. This has plateaued in recent times due to processor clock speeds hitting power and heat limits, but still this memory latency continues to be a significant problem.

Unlike CPUs, the GPUs typically realize a sustained high memory bandwidth. Due to latency hiding, they utilize the bandwidth even during a high number crunching workload.

Cache

To overcome the memory latency, modern processors employ a special type of very fast memory placed onto the processor chip or close to the chip. The purpose of the cache is to store the most recently used data from the memory. Caches are of different levels: the L1 cache is located on the processor chip, while the L2 cache is bigger and located farther away from the processor compared to L1. There is often an L3 cache, which is even bigger and located farther from the processor than L2. In Intel's Haswell processor, as of October 2013, the L1 cache is generally 64 kilobytes (32 KB instruction plus 32 KB data) in size, L2 is 256 KB per core, and L3 is 8 MB.

In cases where memory latency is bad, fortunately caches seem to work very well. The L1 cache is many times faster than accessing the main memory. The reported cache hit rates in real-world programs is 90 percent, which makes a strong case for caches. A cache works like a dictionary of memory address to a block of data values. Since the value is a block of memory, caching of adjacent memory locations has mostly no additional overhead. Note that L2 is slower and bigger than L1, and L3 is slower and bigger than L2. On Intel Sandybridge processors, register lookup is instantaneous; the L1 cache lookup takes three clock cycles, L2 takes nine, L3 takes 21, and main memory access takes 150 to 400 clock cycles.

Interconnect

A processor communicates with the memory and other processors via an interconnect, which are generally of two types of architecture: **symmetric multiprocessing** (**SMP**) and **non-uniform memory access** (**NUMA**). In SMP, a bus interconnects processors and memory with the help of bus controllers. The bus acts as a broadcast device for the end points. The bus often becomes a bottleneck with a large number of processors and memory banks. SMP systems are cheaper to build and harder to scale to a large number of cores compared to NUMA. In a NUMA system, collections of processors and memory are connected point to point to other such groups of processors and memory. Every such group is called a **node**. Local memory of a node is accessible by other nodes and vice versa. Intel's **HyperTransport** and **QuickPath** interconnect technologies support NUMA.

Storage and networking

Storage and networking are the most commonly-used hardware components besides the processor, cache, and memory. Many real-world applications are more often I/O-bound than execution-bound. Such I/O technologies are continuously advancing, and there is a wide variety of components available on the market. The consideration of such devices should be based on the exact performance and reliability characteristics for the use case. Another important criterion is to know how well they are supported by the target operating system drivers. Current day storage technologies mostly build upon hard disks and solid state drives. Applicability of network devices and protocols vary widely as per the business use case. A detailed discussion of I/O hardware is out of the scope of this book.

The Java Virtual Machine

The Java Virtual Machine is a bytecode-oriented, garbage-collected virtual machine that specifies its own instruction set. The instructions have equivalent bytecodes that are interpreted and compiled to the underlying OS and hardware by the **Java Runtime Environment** (**JRE**). Objects are referred to using symbolic references. The data types in the JVM are fully standardized as a single spec across all JVM implementations on all platforms and architectures. The JVM also follows the network byte order, which means communication between Java programs on different architectures can happen using the big-endian byte order. **Jvmtop** (`https://code.google.com/p/jvmtop/`) is a handy JVM monitoring tool like the `top` command in Unix-like systems.

The just-in-time (JIT) compiler

The JIT compiler is part of the JVM. When the JVM starts up, the JIT compiler knows hardly anything about the running code, so it simply interprets the JVM bytecodes. As the program keeps running, the JIT compiler starts profiling the code by collecting statistics and analyzing the call and bytecode patterns. When a method call count exceeds a certain threshold, the JIT compiler applies a number of optimizations to the code. The most common optimizations are inlining and native code generation. The `final` and `static` methods and classes are great candidates for inlining. JIT compilation does not come without a cost; it occupies memory to store the profiled code and sometimes it has to revert wrong speculative optimization. However, JIT compilation is almost always amortized over the long duration of code execution. In rare cases, turning off JIT compilation may be useful if either the code is too large or there are no hotspots in the code due to infrequent execution.

A JRE has typically two kinds of JIT compilers: client and server. Which JIT compiler is used by default depends on the type of hardware and platform. The client JIT compiler is meant for client programs such as command-line and desktop applications. We can start the JRE with the `-server` option to invoke the server JIT compiler, which is really meant for long running programs on a server. The threshold for JIT compilation is higher in the server than the client. The difference in the two kinds of JIT compilers is that the client targets upfront, visible lower latency and the server is assumed to be running on a high-resource hardware and tries to optimize for throughput.

The JIT compiler in Oracle HotSpot JVM observes the code execution to determine the most frequently invoked methods, which are hotspots. Such hotspots are usually just a fraction of the entire code that can be cheap to focus on and optimize. The **HotSpot JIT** compiler is lazy and adaptive. Lazy, because it compiles only those methods to native code that have crossed a certain threshold, and not all the code that it encounters. Compiling to native code is a time consuming process and compiling all code would be wasteful. It is adaptive at gradually increasing the aggressiveness of its compilation on frequently called code, which implies that the code is not optimized only once but many times over as the code gets executed repeatedly. After a method call crosses the first JIT compiler threshold, it is optimized and the counter is reset to zero. At the same time, the optimization count for the code is set to one. When the call exceeds the threshold yet again, the counter is reset to zero and the optimization count is incremented; this time, a more aggressive optimization is applied. This cycle continues until the code cannot be optimized anymore.

The HotSpot JIT compiler does a whole bunch of optimizations. Some of the most prominent ones are as follows:

- **Inlining**: Inlining of methods—very small methods, the static and final methods, methods in final classes, and small methods involving only primitive numerics are prime candidates for inlining.
- **Lock elimination**: Locking is a performance overhead. Fortunately, if the lock object monitor is not reachable from other threads, the lock is eliminated.
- **Virtual call elimination**: Often, there is only one implementation for an interface in a program. The JIT compiler eliminates the virtual call and replaces that with a direct method call on the class implementation object.
- **Non-volatile memory write elimination**: The nonvolatile data members and references in an object are not guaranteed to be visible by the threads other than the current thread. This criterion is utilized not to update such references in memory, but rather to use hardware registers or the stack via native code.

- **Native code generation**: The JIT compiler generates native code for frequently invoked methods together with the arguments. The generated native code is stored in the code cache.
- **Control flow and local optimizations**: The JIT compiler frequently reorders and splits the code for better performance. It also analyzes the branching of control and optimizes code based on that.

There should rarely be any reason to disable JIT compilation, but it can be done by passing the `-Djava.compiler=NONE` parameter when starting the JRE. The default compile threshold can be changed by passing `-XX:CompileThreshold=9800` to the JRE executable where 9800 is the example threshold. The `XX:+PrintCompilation` and `-XX:-CITime` options make the JIT compiler print the JIT statistics and time spent on JIT.

Memory organization

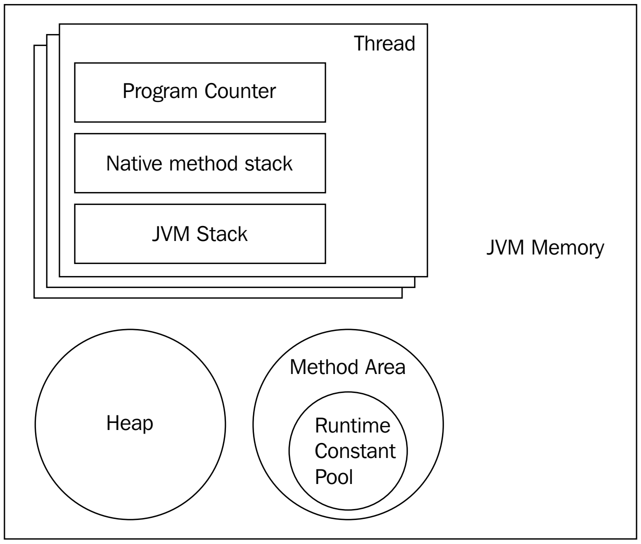
The memory used by the JVM is divided into several segments. JVM being a stack-based execution model, one of the memory segments is the stack area. Every thread is given a stack where the stack frames are stored in **Last-in-First-out** (**LIFO**) order. The stack includes a **program counter** (**PC**) that points to the instruction in the JVM memory currently being executed. When a method is called, a new stack frame is created containing the local variable array and the operand stack. Contrary to conventional stacks, the operand stack holds instructions to load local variable/field values and computation results—a mechanism that is also used to prepare method parameters before a call and to store the return value. The stack frame itself may be allocated on the heap. The easiest way to inspect the order of stack frames in the current thread is to execute the following code:

```
(require 'clojure.repl)
(clojure.repl/pst (Throwable.))
```

When a thread requires more stack space than what the JVM can provide, `StackOverflowError` is thrown.

The heap is the main memory area where the object and array allocations are done. It is shared across all JVM threads. The heap may be of fixed size or expanding depending on the arguments passed to the JRE on startup. Trying to allocate more heap space than what the JVM can make room for results in `OutOfMemoryError` being thrown. The allocations in the heap are subject to garbage collection. When an object is no more reachable via any reference it is garbage collected, with the notable exception of weak, soft, and phantom references. Objects pointed to by nonstrong references take longer to GC.

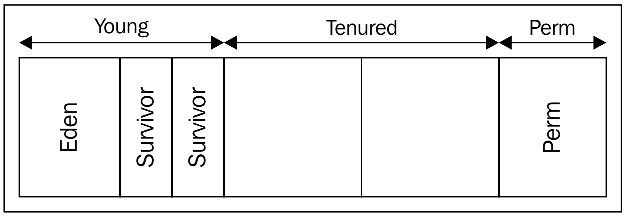
The method area is logically a part of the heap memory and contains per-class structures such as the field and method information, the runtime constant pool, the code for methods, and constructor bodies. It is shared across all JVM threads. In the Oracle HotSpot JVM (up to Version 7), the method area is found in a memory area called the **permanent generation**. In HotSpot Java 8, the permanent generation is replaced by a native memory area called **Metaspace**.



The JVM contains the native code and the Java bytecode to be provided to the Java API implementation and the JVM implementation. The native code call stack is maintained separately for each thread stack. The JVM stack contains the Java method calls. Please note that the JVM spec for Java SE 7 does not imply a native method stack, but it does for Java SE 5 and 6.

HotSpot heap and garbage collection

The Oracle HotSpot JVM uses a generational heap. The three main generations are *young*, *tenured* (old), and *permanent* (till HotSpot JDK 1.7 only) generations. As objects survive garbage collection, they move from Eden to Survivor and from Survivor to Tenured spaces. The new instances are allocated in the Eden segment, which is a very cheap operation (as cheap as a pointer bump, faster than a C `malloc` call) if it already has sufficient free space. When the Eden area does not have enough free space, a minor GC is triggered. This copies the live objects from Eden into the Survivor space. In the same operation, live objects are checked in Survivor-1 and copied over to Survivor-2, thus keeping the live objects only in Survivor-2. This scheme keeps Eden and Survivor-1 empty and unfragmented to make new allocations, and this is known as **copy collection**.



After a certain survival threshold in the young generation, the objects are moved to the tenured/old generation. If it is not possible to do a minor GC, a major GC is attempted. The major GC does not use copying, but rather relies on mark-and-sweep algorithms. We can use throughput collectors (`Serial`, `Parallel`, and `ParallelOld`) or low-pause collectors (`Concurrent` and `G1`) for the old generation.

Sometimes due to running full GC multiple times, the Tenured space may have become so fragmented that it may not be feasible to move objects from Survivor to Tenured space. In those cases, a full GC with compaction is triggered. During this period, the application may appear unresponsive due to the full GC in action.

Measuring memory (heap/stack) usage

One of the prime reasons for performance hit in the JVM is garbage collection. It certainly helps to know how heap memory is used by the objects we create and how to reduce the impact on GC by means of lower footprint. Let us inspect how the representation of an object may lead to heap space.

Every (uncompressed) object or array reference on a 64-bit JVM is 16 bytes long. On a 32-bit JVM, every reference is 8 bytes long. As the 64-bit architecture is becoming more commonplace now, the 64-bit JVM is more likely to be used on servers. Fortunately, for a heap size of up to 32 GB, the JVM can use compressed pointers (default behavior) that are only 4 bytes in size.

	Uncompressed	**Compressed**	**32-bit**
Reference (pointer)	8	4	4
Object header	16	12	8
Array header	24	16	12
Superclass padding	8	4	4

This table illustrated pointer sizes in different modes (reproduced with permission from Attila Szegedi: `http://www.slideshare.net/aszegedi/everything-i-ever-learned-about-jvm-performance-tuning-twitter/20`).

We saw in the previous chapter how many bytes each primitive type takes. Let us see how the memory consumption of the composite types looks with compressed pointers (a common case) on a 64-bit JVM with heap size smaller than 32 GB:

Java Expression	**64-bit memory usage**	**Description (b = bytes, padding toward memory word size in approximate multiples of 8)**
`new Object()`	16 bytes	12 b header + 4 b padding
`new byte[0]`	16 bytes	12 b `obj` header + 4 b `int` length = 16 b array header
`new String("foo")`	40 bytes (interned for literals)	12 b header + (12 b array header + 6 b char-array content + 4 b length + 2 b padding = 24 b) + 4 b hash
`new Integer(3)`	16 bytes (boxed integer)	12 b header + 4 b `int` value
`new Long(4)`	24 bytes (boxed long)	12 b header + 8 b `long` value + 4 b padding

`class A { byte x; }`
`new A();`
	16 bytes	12 b header + 1 b value + 3 b padding

`class B extends A {byte y;}`
`new B();`
	24 bytes (subclass padding)	12 b reference + (1 b value + 7 b padding = 8 b) for A + 1 b for value of `y` + 3 b padding

Java Expression	64-bit memory usage	Description (b = bytes, padding toward memory word size in approximate multiples of 8)

`clojure.lang.Symbol.intern("foo")`
`// clojure 'foo`
	104 bytes (40 bytes interned)	12 b header + 12 b ns reference + (12 b name reference + 40 b interned chars) + 4 b `int` hash + 12 b meta reference + (12 b `_str` reference + 40 b interned chars) – 40 b interned `str`

`clojure.lang.Keyword.intern("foo")`
`// clojure :foo`
	184 bytes (fully interned by factory method)	12 b reference + (12 b symbol reference + 104 b interned value) + 4 b `int` hash + (12 b `_str` reference + 40 b interned `char`)

A comparison of the space taken up by a symbol and a keyword created from the same given string demonstrates that even though a keyword has slight overhead over a symbol, the keyword is fully interned and would provide better guard against memory consumption and thus GC over time. Moreover, the keyword is interned as a weak reference, which ensures that it is garbage collected when no keyword in memory is pointing to the interned value anymore.

Measuring latency with Criterium

Clojure has a neat little macro called `time` that evaluates the body of code passed to it and then prints out the time it took and simply returns the value. However, we can note that often the time taken to execute the code varies quite a bit across various runs.

```
user=> (time (reduce + (range 100000)))
"Elapsed time: 112.480752 msecs"
4999950000
user=> (time (reduce + (range 1000000)))
"Elapsed time: 387.974799 msecs"
499999500000
```

There are several reasons associated to this variance in behavior. When cold started, the JVM has its heap segments empty and is unaware of the code path. As the JVM keeps running, the heap fills up and the GC patterns start becoming noticeable. The JIT compiler gets a chance to profile the different code paths and optimize them. Only after quite some GC and JIT compilation rounds does the JVM performance get less unpredictable.

Criterium (`https://github.com/hugoduncan/criterium`) is a Clojure library to scientifically measure the latency of Clojure expressions on a machine. A summary of how it works can be found at the Criterium project page. The easiest way to use Criterium is to use it with Leiningen. If you want Criterium to be available only in the REPL and not as a project dependency, add the following entry to the `~/.lein/profiles.clj` file:

```
{:user {:plugins [[criterium "0.3.1"]]}}
```

Another way is to include Criterium in your project in the `project.clj` file:

```
:dependencies [[org.clojure/clojure "1.5.1"]
               [criterium "0.3.1"]]
```

Once done with the editing of the file, launch REPL using `lein repl`:

```
user=> (require '[criterium.core :as c])
nil
user=> (c/bench (reduce + (range 100000)))
Evaluation count : 1980 in 60 samples of 33 calls.
             Execution time mean : 31.627742 ms
    Execution time std-deviation : 431.917981 us
   Execution time lower quantile : 30.884211 ms ( 2.5%)
   Execution time upper quantile : 32.129534 ms (97.5%)
nil
```

Now we can see that on average, the expression took 31.6 ms on a certain test machine.

Criterium and Leiningen

By default, Leiningen starts the JVM in a low-tiered compilation mode, which causes it to start up faster, but impacts the optimizations that the JRE can perform at runtime. To get best effects when running tests with Criterium and Leiningen for a server-side use case, be sure to override the defaults in `project.clj` as follows:

```
:jvm-opts ^:replace ["-server"]
```

The `^:replace` hint causes Leiningen to replace its own defaults with what is provided under the `:jvm-opts` key. You may like to add more parameters as needed, such as minimum and maximum heap size to run the tests.

Summary

The performance of a software system is directly impacted by its hardware components, so understanding how the hardware works is crucial. The processor, caches, memory, and I/O subsystems have different performance behaviors. With Clojure being a hosted language, understanding the performance properties of the host, that is, the JVM, is equally important. The Criterium library is useful to measure the latency of the Clojure code—we will discuss Criterium again in *Chapter 6*, *Optimizing Performance*. In the next chapter we will look at the concurrency primitives in Clojure and their performance characteristics.

5
Concurrency

Concurrency was one of the chief design goals of Clojure. Considering the concurrent programming model in Java, it is not only too low level but also so tricky to get right that without strictly following patterns, you are more likely to shoot yourself in the foot. Locks, synchronization, and unguarded mutation—these are recipes for concurrency pitfalls unless exercised with extreme caution. Clojure's design choices deeply influence the way concurrency patterns can be achieved in a safe and functional manner. In this chapter we will discuss:

- Low-level concurrency support at the hardware and JVM levels
- The concurrency primitives of Clojure—atoms, agents, refs, and vars
- The built-in concurrency features in Java that are safe and useful for use with Clojure
- Parallelization with Clojure features and reducers

Low-level concurrency

Non-cooperative concurrency and parallelism cannot be achieved without explicit hardware support. We discussed SMT and multicore processors in *Chapter 4, Host Performance*. Recall that every processor core has its own L1 cache and several cores share the L2 cache. The shared L2 cache provides a fast mechanism to the processor cores to coordinate their cache access, eliminating the comparatively expensive memory access. Additionally, a processor buffers the writes to memory into something known as a **dirty write-buffer**. This helps the processor issue a batch of memory update requests, reorders the instructions, and then determines the final value to write to memory, known as **write absorption**.

Hardware memory barrier instructions

Memory access reordering is great for a sequential (single-threaded) program performance, but it is hazardous for concurrent programs where the order of memory access in one thread may disrupt the expectations in another thread. The processor needs a means of synchronizing the access. This should be such that memory reordering is either compartmentalized in code segments where it does not matter or is prevented where it might have undesirable consequences. The hardware supports such a safety measure in terms of a **memory barrier**, also known as a **fence**.

There are several kinds of memory barrier instructions found on different architectures with potentially different performance characteristics. The compiler, or the JIT compiler in the case of the JVM, usually knows about the fence instructions on the architectures it runs on. The common fence instructions are read barrier, write barrier, acquire barrier, and release barrier. The barriers do not guarantee latest data; rather, they only control relative ordering of memory access. Barriers cause the write-buffer to be flushed after all writes are issued, before the barrier is visible to the processor that issued it.

Read and write barriers control the order of reads and writes respectively. Writes happen via a write-buffer, but reads may happen out of order or from the write-buffer. To guarantee correct ordering, acquire and release blocks/barriers are used. Acquire and release are considered as *half barriers*; both of them together form a *full barrier*. A full barrier is more expensive than a half barrier.

Java support and its Clojure equivalent

In Java, the memory barrier instructions are inserted by the higher-level coordination primitives. Even though fence instructions are expensive to run (taking hundreds of CPU cycles), they provide a safety net that makes accessing shared variables safe within critical sections. In Java, the `synchronized` keyword marks a *critical section* which can be executed by only one thread at a time, thus making a tool for "mutual exclusion". In Clojure, the equivalent of Java's `synchronized` is the `locking` macro:

```
// Java example
synchronized (someObject) {
    // do something
}
;; Clojure example
(locking some-object
  ;; do something
  )
```

The `locking` macro builds upon two special forms: `monitor-enter` and `monitor-exit`. Note that the `locking` macro is a low-level and imperative solution, just like Java's `synchronized`. Their use is not considered idiomatic in Clojure. The special forms `monitor-enter` and `monitor-exit` respectively enter and exit the lock object's monitor. They are even lower level and not recommended for direct use.

Someone measuring the performance of code that uses such locking should be aware of its single-threaded versus multithreaded latencies. Locking in a single thread is cheap; however, the performance penalty starts kicking in when there are two or more threads contending for a lock on the same object monitor. A lock is acquired on the monitor of an object, called **intrinsic** or **monitor** lock. Object equivalence (that is, when the `=` function returns true) is never used for the purpose of locking—make sure the object references are the same (that is, when `identical?` returns true) when locking from different threads.

Acquiring a monitor lock by a thread entails a read barrier, which invalidates the thread-local cached data and corresponding processor registers and cache lines. This forces a re-read from memory. On the other hand, releasing a monitor lock results in a write barrier, which flushes all changes to memory. These are expensive operations that impact parallelism, but they ensure consistency of data for all threads.

Java supports a `volatile` keyword for data members in a class, which guarantees that read and write to the attribute outside of a `synchronized` block would not be reordered. It is interesting to note that unless an attribute is declared `volatile`, it is not guaranteed visibility in all the threads accessing it. The Clojure equivalent of Java's `volatile` is the metadata `^:volatile-mutable` that we discussed in *Chapter 2, Clojure Abstractions*. An example of `volatile` in Java and Clojure is as follows:

```
// Java example
public class Person {
    volatile long age;
}
;; Clojure example
(deftype Person [^:volatile-mutable ^long age])
```

Reading and writing a `volatile` data requires read-acquire or write-release fence respectively, which means we need only a half barrier to individually read or write the value. Note that due to the half barrier, read-followed-by-write operations are not guaranteed to be atomic. For example, the expression `age++` first reads the value, then increments and sets it. This makes it two memory operations, which means it's not a half barrier any more.

Atomic updates and state

It is a common use case to read a data element, execute some logic, and update with a new value. For single-threaded programs, it bears no consequences, but for concurrent scenarios, the entire operation must be carried out in a lockstep as an atomic operation. This case is so common that many processors support this at the hardware level using a special **Compare-and-swap** (**CAS**) instruction, which is much cheaper than locking. On x86/x64 architectures, the instruction is called **CompareExchange** (`CMPXCHG`).

Unfortunately, it is possible that another thread updates the variable with the same value that the thread working on the atomic update is going to compare the old value against. This is known as the **ABA** problem. The set of instructions **load-linked** (**LL**) and **store-conditional** (**SC**), which are found in some other architectures, provide an alternative to CAS without the ABA problem. After the LL instruction reads the value from an address, the SC instruction to update the address with a new value goes through only if the address was not updated after the LL instruction was successful.

Atomic updates in Java

Java has a bunch of built-in, lock free, atomic, thread-safe, compare-and-swap abstractions for state management. They live in the `java.util.concurrent.atomic` package. For primitive types such as boolean, integer, and long, there are `AtomicBoolean`, `AtomicInteger`, and `AtomicLong` classes respectively. The latter two classes support additional atomic add/subtract operations. For atomic reference updates, there are the `AtomicReference`, `AtomicMarkableReference`, and `AtomicStampedReference` classes for arbitrary objects. There is also support for arrays where the array elements can be updated atomically—`AtomicIntegerArray`, `AtomicLongArray`, and `AtomicReferenceArray`. They are easy to use:

```
(def ^AtomicReference x (AtomicReference. "foo"))
(.compareAndSet x "foo" "bar")
(def ^AtomicInteger y (AtomicInteger. 10))
(.getAndAdd y 5)
```

However, where and how to use them is entirely subjective to the update points and logic in the code. The atomic updates are not guaranteed to be nonblocking. Atomic updates are not a substitute for locking in Java, but rather a convenience only when the scope is limited to a compare-and-swap operation for one mutable state and you need to squeeze in more cycles in concurrent programming.

Clojure's support for atomic updates

Clojure's atomic update abstraction is called **atom**. It uses `AtomicReference` under the hood. An operation on `AtomicInteger` or `AtomicLong` may be slightly faster than on the Clojure `atom` because the former uses primitives, but neither of them is too cheap due to the compare-and-swap instruction they use in the CPU. The speed really depends on how frequently the mutation happens and how the JIT compiler optimizes the code. The benefit of speed may not be seen until the code is run several hundred thousand times, and having an atom mutated very frequently will increase the latency due to retries. Measuring the latency under actual (or similar to actual) load can explain this better. An example of using an atom is given as follows:

```
user=> (def a (atom 0))
#'user/a
user=> (swap! a inc)
1
user=> @a
1
user=> (compare-and-set! a 1 5)
true
user=> (reset! a 20)
20
```

The `swap!` function provides a notably different style of carrying out atomic updates than the `compareAndSwap(oldval, newval)` method. While `compareAndSwap()` compares and sets the value returning true on success and false on failure, `swap!` keeps on trying to update in an endless loop until it succeeds. This style is a popular pattern that is even followed by Java developers. However, there is also a potential pitfall associated with the update-in-loop style. As the concurrency of updaters gets higher, the performance of an update may gradually degrade. Then again, high concurrency on atomic updates raises a question as to whether uncoordinated updates were a good idea at all for the use case. `compare-and-set!` and `reset!` are pretty straightforward.

The function passed to `swap!` is required to be pure (as in side-effect free) because it is retried several times in a loop during contention. If the function is not pure, the side effect may happen as many times as the retries.

It is noteworthy that atoms are not coordinated. This means that when an atom is used concurrently by different threads, we cannot predict the order in which the operations work on it, and as a consequence, we cannot guarantee the end result. The code we write around atoms should be designed with this constraint in mind. In many scenarios, atoms may not be a good fit due to lack of coordination—watch out for this during program design. Atoms support metadata and basic validation mechanisms via extra arguments. The following examples illustrate those features:

```
user=> (def a (atom 0 :meta {:foo :bar}))
user=> (meta a)
{:foo :bar}
user=> (def age (atom 0 :validator (fn [x] (<= x 200))))
user=> (swap! age 200)
200
user=> (swap! age inc)
IllegalStateException Invalid reference state  clojure.lang.ARef.
validate (ARef.java:33)
```

The second important thing that atoms support is adding and removing watches on them. We will discuss watches later in this chapter.

Asynchronous agents and state

While atoms are synchronous, agents are the asynchronous mechanism in Clojure that affect any change in state. Every agent is associated with a mutable state. We pass a function (known as **action**) to an agent with optional additional arguments—this function gets queued for processing in another thread by the agent. All agents share two common thread pools: one for low-latency (potentially CPU-bound, cache-bound, or memory-bound) jobs and one for blocking (potentially I/O-related or lengthy processing) jobs. Clojure provides the `send` function for low-latency actions, `send-off` for blocking actions, and `send-via` to have the action executed on the user-specified thread pool instead of either of the preconfigured thread pools. All of `send`, `send-off`, and `send-via` return immediately. The following is how we can use them:

```
(def a (agent 0))
(send a inc)  ; invokes (inc 0) in another thread, sets a to result
@a  ; returns 1 (only if the `inc` action is done ; also see `await)
(send a + 2 3)  ; invokes (+ 1 2 3) in another thread, sets a = result
@a  ; returns 6
(def key nil)
(send-off event poll-network "DXUHCGE663GU")
```

```
(shutdown-agents)  ; shuts down the thread-pools
(send a inc)  ; does not execute action anymore, so no result update
@a  ; returns 6
```

On inspection of the Clojure 1.5.1 source code, we can find that the thread pool for low-latency actions is named `pooledExecutor` (a bounded thread pool initialized to a maximum of "2 + number of hardware processors" threads) and the thread pool for high-latency actions is named `soloExecutor` (an unbounded thread pool). The premise of this default configuration is that the CPU-, cache-, or memory-bound actions run most optimally on a bounded thread pool with the default number of threads. The I/O-bound tasks do not consume CPU resources. Hence, a relatively larger number of such tasks can execute at the same time without significantly affecting the performance of CPU-, cache-, or memory-bound jobs. The following is how you can access and override the thread pools:

```
(import  'clojure.lang.Agent)
Agent/pooledExecutor ; thread pool for low-latency actions
Agent/soloExecutor ; thread pool for I/O actions
(import 'java.util.Executors)
(def  a-pool (Executors/newFixedThreadPool 10)) ; 10 threads
(def  b-pool (Executors/newFixedThreadPool 100)) ; 100 threads
(def  a (agent  0))
(send-via  a-pool a inc) ; use a-pool for the action
(set-agent-send-executor! a-pool ; set default pool for send
(set-agent-send-off-executor! b-pool ; set default for send off
```

If a program carries out a large number of I/O or blocking operations through agents, it probably makes sense to limit the number of threads dedicated for such actions. Overriding the `send-off` thread pool using `set-agent-send-off-executor!` is the easiest way to limit the thread pool size. A more granular way to isolate and limit the I/O actions on agents is to use `send-via` with thread pools of appropriate sizes for various kinds of I/O and blocking operations.

Asynchrony, queuing, and error handling

Sending an action to an agent returns immediately without blocking. If the agent is not already busy executing any action, it reacts by submitting the action to the respective thread pool. If the agent is busy executing another action, the new action is simply en-queued. Once an action is executed from the action queue, the queue is checked for more entries and triggers the next action if entries are found. This whole *reactive* mechanism of triggering actions obviates the need for polling the queue. This is possible only because the entry points to an agent's queue are controlled.

Actions are executed asynchronously on agents, which raises the question of how errors are handled. The error cases need to be handled with explicit predefined functions. When using the default agent construction, such as `(agent :foo)`, the agent is created without any error handler and gets suspended on any exception. It caches the exception and refuses to accept any more actions—it throws the cached exception upon sending any action until the agent is restarted. A suspended agent can be reset using the `restart-agent` function. The objective of such a suspension is safety and supervision. When asynchronous actions are executed on an agent and suddenly an error occurs, it requires attention.

```
(def g (agent 0))
(send g (partial / 10)) ; ArithmeticException due to divide by zero
@g ; returns 0, because the error did not change the old state
(send g inc) ; throws the cached ArithmeticException
(agent-error g) ; returns (doesn't throw) the exception object
(restart-agent g @g) ; clears the suspension of the agent
(agent-error g) ; returns nil
(send g inc) ; works now because we cleared the cached error
@g ; returns 1
(dotimes [_ 1000] (send-off g long-task))
(await-for 100 g) ; block for 100ms or until all actions over
(whichever earlier)
(await g) ; block until all actions dispatched till now are over
```

There are two optional parameters `:error-handler` and `:error-mode` that we can configure on an agent to have finer control over error handling and suspension.

```
;; incorrect arity for error handler function below
(def g (agent 0 :error-handler (fn [x] (println "Found:" x))))
;; no error will be encountered because error-handler arity is wrong
(send g (partial / 10))
;; correct arity below
(def g (agent 0 :error-handler (fn [ag x] (println "Found:" x))))
```

```
(send g (partial / 10)) ; prints the message
;; we can set error-handler after constructing an agent
(set error-handler! g (fn [ag x] (println "Found:" x)))
;; we can define agents to ignore errors and continue
(def h (agent 0 :error-mode :continue))
(send h (partial / 10)) ; error encountered, but agent not suspended
(send h inc)
@h ; returns 1
;; we can set the error-mode later, other possible value :fail
(set-error-mode! h : continue)
```

Advantages of agents

Just as the atom implementation uses only compare-and-swap instead of locking, the underlying agent-specific implementation uses mostly compare-and-swap operations. The agent implementation uses locks only when dispatching an action in a transaction (discussed in the next section) or when restarting an agent. All actions are queued and dispatched serially in the agents, regardless of the concurrency level. Their serial nature makes it possible to execute actions in an independent and contention-free manner. On one agent, there can never be more than one action being executed. Since there is no locking, reads (`deref` or `@`) on agents are never blocked due to writes. However, all actions are independent of each other—there is no overlap in their execution.

The implementation goes so far as to ensure that the execution of an action blocks the other actions behind it in the queue. Even though the actions are executed in a thread pool, actions for the same agent are never executed concurrently. This is an excellent ordering guarantee that also extends a natural coordination mechanism due to its serial nature. However, note that this ordering coordination is limited to only a single agent. If an agent action sends actions to two other agents, they are not automatically coordinated. You may want to use transactions (discussed in the next section) for such a situation.

Since agents distinguish between low-latency and blocking jobs, the jobs are executed in appropriate thread pools. Actions on different agents may execute concurrently, thereby making optimum use of threading resources. Unlike atoms, the performance of agents is not impeded by high contention. In fact, in many cases, agents make a lot of sense due to the serial buffering of actions. In general, agents are great for high volume I/O tasks or where the ordering of operations provides a win in high-contention scenarios.

Nesting

When an agent action sends another action to the same agent, that is a case of **nesting**. This would have been nothing special if agents didn't participate in STM transactions (covered in the next section). However, agents do participate in STM transactions and that places certain constraints on agent implementation, which warrants a second-layer buffering of actions. For now, it should suffice to say that the nested sends are queued in a thread-local queue instead of the regular queue in the agent. The thread-local queue is visible to only the thread in which the action is executed. Upon executing an action, unless there was an error, the agent implicitly calls the equivalent of the `release-pending-sends` function, which transfers the actions from a second-level, thread-local queue to the normal action queue. Note that nesting is simply an implementation detail of agents and has no other impact.

Coordinated transactional ref and state

We saw in an earlier section that an atom provides an atomic read-and-update operation. What if we need to perform an atomic read-and-update operation across two or even more atoms? This clearly poses a coordination problem. Some entity has to be watching over the process of reading and updating so that the values are not corrupted. This is what a ref provides—a system based on **software transactional memory** (**STM**). This takes care of concurrent atomic read-and-update operations across multiple refs such that either all updates go through or, in the case of failure, none do. Like atoms, on failure, refs retry the whole operation from scratch with new values.

Clojure's STM implementation is coarse grained—it works on the application level's objects and aggregates (references to aggregates), which are scoped to just all the refs in a program constituting *Ref world*. Any update to a ref can only happen synchronously in a transaction in a `dosync` block of code within the same thread—it cannot span beyond the current thread. The implementation detail reveals that a thread-local transaction context is maintained during the lifetime of a transaction. The same context is no longer available the moment the control reaches another thread.

Like the other reference types in Clojure, reads on a ref are never blocked by the updates and vice versa. However, unlike the other reference types, the implementation of ref does not depend on lock-free spinning (that is, retrying in a loop until success); rather, it uses locks, low-level wait/notify, deadlock detection, and age-based barging (that is, arbitration between concurrent older and younger transactions) internally.

The alter function is used to read and update the value of a ref, and ref-set is used to reset the value. Roughly, alter and ref-set for refs are analogous to swap! and reset! for atoms. Just like swap!, alter accepts a function (and arguments) with no side effects and may be retried several times during contention. However, unlike atoms, not only alter, but also ref-set and a simple deref may cause a transaction to be retried during contention. The following is a very simple example of how we may use a transaction:

```
(def r1 (ref [:a :b :c]))
(def r2 (ref [1 2 3]))
(alter r1 conj :d)  ; IllegalStateException No transaction running...
(dosync (let [v (last @r1)] (alter r1 pop) (alter r2 conj v)))
@r1  ; returns [:a :b]
@r2  ; returns [1 2 3 :c]
(dosync (ref-set r1 (conj @r1 (last @r2))) (ref-set r2 (pop @r2)))
@r1  ; returns [:a :b :c]
@r2  ; returns [1 2 3]
```

Ref characteristics

Clojure maintains **atomicity**, **consistency**, and **isolation** (**ACI**) characteristics in a transaction. This overlaps with A, C, and I of the **Atomicity**, **Consistency**, **Isolation**, and **Durability** (**ACID**) guarantee that many databases provide. Atomicity implies that either all of the updates in a transaction are completed successfully or none of them are completed. Consistency means that the transaction must maintain general correctness and should honor constraints set by validation—any exception or validation error should rollback the transaction. Unless a shared state is guarded, concurrent updates on it may lead to a multistep transaction seeing different values at different steps. Isolation implies that all steps in a transaction will see the same value no matter how concurrent the updates are.

Clojure refs use something called **Multiversion concurrency control** (**MVCC**) to provide *Snapshot Isolation* to transactions. In MVCC, instead of locking (which could block transactions), queues are maintained so that each transaction can occur using its own snapshot copy taken at its *read point*, independent of other transactions. The main benefit of this approach is that the read-only, out-of-transaction operations can go through without any contention. Transactions without ref contention go through concurrently. In a rough comparison to the database systems, the Clojure ref isolation level is *Read-Committed* for reading a ref outside a transaction and *Repeatable-Read* by default when inside the transaction.

Ref history and intransaction deref operations

We discussed earlier that both read and update operations on a ref may cause a transaction to be retried. The reads in a transaction can be configured to use ref history such that the snapshot isolation instances are stored in history queues and are used by the read operations in transactions. The default uses a small history that conserves heap space.

Using ref history reduces the likelihood of transaction retries caused by read contention, thereby providing weak consistency. Therefore, it is a tool for performance optimization at the cost of consistency. In many scenarios, programs do not need strong consistency—we can choose appropriately if we know the trade-off and what we need. The snapshot isolation mechanism in Clojure's ref implementation is backed by adaptive history queues. The history queues grow dynamically to meet the read requests and do not overshoot the maximum limit set for the ref. By default, the `min-history` and `max-history` values are set to `0` and `10` respectively. The following is an example of how to use history:

```
(def r (ref 0 :min-history 5 :max-history 10))
(ref-history-count r) ; returns 0, no snapshot instances queued yet
(ref-min-history r) ; returns 5
(ref-max-history r) ; returns 10
(future (dosync (println "Sleeping 20 sec") (Thread/sleep 20000)
                (ref-set r 10)))
(dosync (alter r inc)) ; execute within few seconds after last expr
;; message "Sleeping 20 sec" appears twice due to transaction retry
(ref-history-count r) ; returns 2, count of snapsho  history entries
(.trimHistory ^clojure.lang.Ref r)
(ref-history-count r) ; returns 0 because we wiped the history
(ref-min-history r 10) ; reset the min history
(ref-max-history r 20) ; reset the max history count
```

Minimum/maximum history limits are proportional to the length of the staleness window of data. They also depend on the relative latency difference of update and read operations to see what range of `min-history` and `max-history` works well on a given host system. It may take some amount of trial and error to get the range right. As a ballpark figure, read operations need only as many `min-history` elements to avoid transaction retries as many updates can go through during one read operation. The `max-history` elements can be a multiple of `min-history` to cover for any history overrun or underrun. If the relative latency difference is unpredictable, we have to either plan `min-history` for the worst case scenario or consider other approaches.

Transaction retries and barging

A transaction can internally be in one of the five distinct states—*running*, *committing*, *retry*, *killed*, and *committed*. A transaction can be killed for various reasons. Exceptions are a common reason for killing a transaction. But let's consider the corner case where a transaction is retried many times but it does not appear to commit successfully—what is the resolution? Clojure supports age-based barging, wherein an older transaction automatically tries to abort a younger transaction so that the younger transaction is retried later. If barging still doesn't work, as a last resort, the transaction is killed after a hard limit of 10,000 retry attempts and an exception is thrown.

Upping transaction consistency with ensure

Clojure's transactional consistency is a good balance between performance and safety. However, at times, we may need **serializable** consistency in order to preserve the correctness of a transaction. Concretely, in the face of transaction retries, when a transaction's correctness depends on the state of a ref in the transaction wherein the ref is updated simultaneously in another transaction, we have a condition called **write skew**.

For example, let us say a ref `process-order` points to a flag that when true implies that order processing can take place. Transaction `t1` carries out the order processing steps after reading `process-order` while simultaneously another transaction `t2` checks stock and wants to update `process-order–f`. How can we isolate `t1` and `t2` to stop them from overlapping each other? The logical inconsistency following from overlaps is `write skew`. Imagine a scenario where the transactions touch several refs to do their job—how can we keep the ref world in a logically consistent state?

write skew can be solved using the `ensure` function that essentially prevents a ref from modification by other transactions. It is like a locking operation, but in practice it provides better concurrency than explicit read-and-update operations and does not cause a deadlock. Using `ensure` is quite simple: `(ensure ref-object)`. It holds locks during the transaction but may still offer better performance as it avoids abort and retry. In the example from the previous paragraph, each of `t1` should call `(ensure process-order)` in order to avoid `write skew`.

Fewer transaction retries with commutative operations

Commutative operations are independent of the order in which they are applied. For example, incrementing a counter ref `c1` from transactions `t1` and `t2` would have the same effect irrespective of the order in which `t1` and `t2` commit their changes. Refs have a special optimization for change functions that are commutative for transactions—the `commute` function, which is like `alter` (same syntax) but with different semantics. Like `alter`, `commute` functions are applied atomically during a transaction commit. However, unlike `alter`, `commute` does not cause a transaction retry on contention and there is no guarantee about the order in which `commute` functions are applied. This effectively makes `commute` nearly useless for returning a meaningful value as a result of the operation. All `commute` functions in a transaction are reapplied with the final in-transaction ref values during a transaction commit.

As we can see, `commute` reduces contention thereby optimizing the performance of overall transaction throughput. Once we know that an operation is commutative and we are not going to use its return value in a meaningful way, there is hardly any trade-off deciding on whether to use commute—just go ahead and use it. In fact, a program design with respect to ref transactions with `commute` in mind is not a bad idea.

Agents can participate in transactions

In the previous section on agents, we discussed how agents work with queued change functions. Agents can also participate in ref transactions, thereby making it possible to combine the use of refs and agents in transactions. However, agents are not included in the *Ref world*; hence, a transaction scope is not extended till the execution of the change function in an agent. Rather, transactions only make sure that changes sent to agents are queued until a transaction commit happens.

The Nesting subsection in the earlier section on agents discusses a second-layer thread-local queue. That thread-local queue is used during a transaction to hold the sent changes to an agent until commit. The thread-local queue does not block the other changes being sent to an agent. The out-of-transaction changes are never buffered in the thread-local queue; rather, they are added to the regular queue in the agent.

Participation of agents in transactions provides an interesting angle of design, where coordinated and independent/sequential operations can be pipelined as a workflow for better throughput and performance.

Nested transactions

Clojure transactions are nesting-aware and they compose well. But, why would you need a nested transaction? Often independent units of code may have their own low-granularity transactions that a higher-level code can make use of. When the higher-level caller itself needs to wrap actions in a transaction, nested transactions occur. Nested transactions do not have their own life cycle and run-state; rather, they are absorbed into the outer transaction without introducing deadlocks or inconsistencies. However, an outer transaction can abort an inner transaction on detection of failure.

The `ref world` snapshot ensures and commutes are shared among all (that is, outer and inner) levels of a nested transaction. Due to this, the inner transaction is treated as any other ref change operation (such as `alter`, `ref-set`, and many more) within an outer transaction. The watches and internal lock implementation are handled at the respective nesting level. Detection of contention in the inner transactions causes a restart of not only the inner, but also the outer transaction. Commits at all levels are carried out together finally when the outermost transaction commits. The watches, even though tracked at each individual transaction level, are finally enforced during the commit. A closer look at a nested transaction implementation shows that nesting has little or no impact on the performance of transactions.

Performance considerations

Clojure ref is perhaps the most complex reference type implemented yet. Due to its characteristics, especially due to its transaction retry mechanism, it may not be immediately apparent how such a system would have good performance during high-contention scenarios. Understanding its nuances and the best ways of using it should help:

- We do not use changes with side effects in a transaction, except for sending I/O changes to agents where the changes are buffered until commit. So, by definition, we do not carry out any expensive I/O work in a transaction. Hence, a retry of that work would be cheap as well.
- A change function for a transaction should be as small as possible. This lowers the latency and hence the retries will also be cheaper.
- Any ref that is not updated along with at least one more ref simultaneously need not be a ref—an atom would do just fine in that case. Now that refs make sense only in a group, their contention is directly proportional to the group size. Small groups of refs used in transactions lead to low contention, lower latency, and higher throughput.
- Commutative functions provide a good opportunity to enhance transaction throughput without any penalty. Identifying such cases and designing with `commute` in mind can help performance significantly.
- Refs are very coarse grained—they work at the application-aggregate level. Often a program may need to have finer grained control over transaction resources. This can be enabled by ref striping (in the same vein as **lock striping**) refer to `http://cljme.cgrand.net/2011/10/06/aworldinaref/` for more details.
- In high-contention scenarios, where the ref group size in a transaction cannot be small, consider using agents as they have no contention due to their serial nature. Agents may not be a replacement for transactions, but rather you can employ a pipeline consisting of atoms, refs, and agents to ease out the contention versus latency concerns.

Refs and transactions have an intricate implementation. Fortunately, we can inspect the source code and browse through available online and offline resources.

Dynamic var binding and state

The fourth kind of Clojure reference type is the dynamic var. Since Clojure 1.3, all vars are static by default. A var must be explicitly declared in order to be dynamic. Once declared, a dynamic var can be bound to new values on a per-thread basis.

Bindings on different threads do not block each other. An example is as follows:

```
(def ^:dynamic *foo* "bar")
(println *foo*)  ; prints bar
(binding [*foo* "baz"] (println *foo*))  ; prints baz
(binding [*foo* "bar"] (set! *foo* "quux") (println *foo*))  ; prints quux
```

As dynamic binding is thread-local, it may be tricky to use in multithreaded scenarios. Dynamic vars have been long abused by libraries and applications as a means to pass in a common argument to be used by several functions. However, that style is acknowledged to be an antipattern and is discouraged. Typically, in antipattern, dynamic vars are wrapped by a macro to contain the dynamic thread-local binding in the lexical scope. This causes problems with multithreading and lazy sequences.

So, how can dynamic vars be used effectively? A dynamic var lookup is more expensive than looking up a static var. Even passing a function argument is performance-wise much cheaper than looking up a dynamic var. Binding a dynamic var incurs additional cost. Clearly, in performance-sensitive code, dynamic vars are best not used at all. However, dynamic vars may prove to be useful to hold temporary thread-local state in a complex or recursive call-graph scenario where performance does not matter significantly, without being advertised or leaked into the public API. Dynamic var bindings can nest and unwind like a stack, which makes them attractive and suitable for such tasks.

Validating and watching the reference types

Vars (both static and dynamic), atoms, refs, and agents provide a way to validate the value being set as state—a `validator` function that accepts a new value as argument and returns a logical true on success, or throws exception/returns a logical false (`false` and `nil` values) on error. They all honor what the validator function returns. On success, the update goes through and on encountering an error, an exception is thrown instead. The following is the syntax of how validators can be declared and associated with the reference types:

```
(def t (atom 1 :validator pos?))
(def g (agent 1 :validator pos?))
(def r (ref 1 :validator pos?))
(swap! t inc) ; ; goes through, value after increment (2) is positive
(swap! t (constantly -3))  ; throws exception
(def v 10)
(set-validator! (var v) pos?)
(set-validator! t #(>= % 10))
(set-validator! g #(>= % 10))
(set-validator! r #(>= % 10))
```

Validators cause actual failure within a reference type while updating them. For vars and atoms, they simply prevent the update by throwing an exception. In an agent, a validation failure causes agent failure and needs a restart of the agent. Inside a ref, validation failure causes the transaction to rollback and throws the exception once more.

Another mechanism to observe the changes to reference types is a **watcher**. Unlike validators, a watcher is passive—it is notified of the update after the fact. Hence, a watcher cannot prevent updates from going through because it is only a notification mechanism. For transactions, a watcher is invoked only after the transaction commit. While only one validator can be set on a reference type, it is possible to associate multiple watchers to a reference type. Secondly, when adding a watch, we can specify a key, so that notifications can be identified by the key and dealt with accordingly by the watcher. The following is how to use watchers:

```
(def t (atom 1))
(defn w [key iref oldv newv] (println "Key:" key "Old:" oldv "New:"
newv))
(add-watch t :foo w)
(swap! t inc)  ; prints "Key: :foo Old: 1 New: 2"
```

Like validators, watchers are executed synchronously in the thread of the reference type. For atoms and refs, this may be fine, since while the notification to watchers goes on, other threads may proceed with their updates. However, in agents, the notification happens in the same thread where the update happens—this makes the update latency higher and the throughput potentially lower.

Java concurrent data structures

Java has a number of mutable data structures that are meant for concurrency and thread-safety, which implies multiple callers can safely access these data structures at the same time without blocking each other. When we need only highly concurrent access without state management, these data structures may be a very good fit. (several of these employ lock-free algorithms). We discussed Java atomic state classes in the *Atomic updates and state* section, so we will not repeat them here. Rather, we will only discuss the concurrent queues and other collections. All these data structures live in the `java.util.concurrent` package. These concurrent data structures are tailored to leverage the JSR 133 *Java Memory Model and Thread Specification Revision* implementation that first appeared in Java 5.

Concurrent maps

Java has a mutable concurrent hash-map, `java.util.concurrent.ConcurrentHashMap` (**CHM** for short). The concurrency level can be optionally specified when instantiating the class, which is 16 by default. The CHM implementation internally partitions the map entries into hash buckets and uses multiple locks to reduce contention on each bucket. Reads are never blocked by writes; therefore, they may be stale or inconsistent. This is countered by a built-in detection of such situations and by issuing a lock in order to read the data again in a synchronized fashion. This is an optimization for scenarios where reads significantly outnumber writes. In CHM, all individual operations are near constant time unless stuck in a retry loop due to lock contention.

In contrast with Clojure's persistent map, CHM cannot accept `null` (`nil`) as a key or value. Clojure's immutable scalars and collections are automatically well suited for use with CHM. An important thing to note is that only the individual operations in CHM are atomic and exhibit strong consistency. As CHM operations are concurrent, the aggregate operations provide rather weak consistency than true operation-level consistency. The following code shows how we can use CHM. The individual operations in CHM that provide better consistency are safe to use, and aggregate operations should be reserved for when we know its consistency characteristics and the related trade-off:

```
(import 'java.util.concurrent.ConcurrentHashMap)
(def ^ConcurrentHashMap m (ConcurrentHashMap.))
(.put m :english "hi") ; individual operation
(.get m :english) ; individual operation
(.putIfAbsent m :spanish "alo") ; individual operation
(.replace m :spanish "hola") ; individual operation
(.replace m :english "hi" "hello") ; individual CAS atomic operation
(.remove m :english) ; individua l operation
(.clear  m) ; aggregate operation
(.size  m) ; aggregate operation
(count  m ) ; internally uses the .size() method
(.putAll {:french "bonjour" :italian "buon giorno"}) ; aggregate op
(.keySet m) ; aggregate operation
(keys  m) ; -> CHM.entrySet(), eachpair -> java.util.Map.Entry.getKey()
(vals m) ; -> CHM.entrySet(), pair -> java.util.Map.Entry.getValue()
```

The `java.util.concurrent.ConcurrentSkipListMap` class (**CSLM** for short) is another concurrent mutable map data structure in Java. The difference between CHM and CSLM is that CSLM offers a sorted view of the map at all times with O(log N) time complexity. The sorted view has the natural order of keys by default, which can be overridden by specifying a comparator implementation when instantiating CSLM. The implementation of CSLM is based on the skip list and provides navigation operations.

The `java.util.concurrent.ConcurrentSkipListSet` class (**CSLS** for short) is a concurrent mutable set based on the CSLM implementation. While CSLM offers the map API, CSLS behaves as a set data structure while borrowing features of CSLM.

Concurrent queues

Java has built-in implementation of several kinds of mutable and concurrent in-memory queues. The queue data structure is a useful tool for buffering, producer-consumer style implementation, and for pipelining such units together to form high-performance workflows. We should not confuse them with durable queues that are used for similar purposes in batch jobs for high throughput. Java's in-memory queues are not transactional, but they provide atomicity and strong consistency guarantee for the individual queue operations only. Aggregate operations offer weaker consistency.

The `java.util.concurrent.ConcurrentLinkedQueue` (**CLQ**) class is a lock-free, wait-free unbounded **First-In-First-Out** (**FIFO**) queue. FIFO implies that the order of queue elements will not change once added to the queue. CLQ's `size()` method is not a constant time operation; it depends on the concurrency level. A few examples of using CLQ are as follows:

```
(import 'java.util.concurrent.ConcurrentLinkedQueue)
(def ^ConcurrentLinkedQueue q (ConcurrentLinkedQueue.))
(.add q :foo)
(.add q :bar)
(.poll q)  ; returns :foo
(.poll q)  ; returns :bar
```

A summary of concurrent queues is listed in the following table:

Queue	Blocking?	Bounded?	FIFO?	Fairness?	Notes
CLQ	No	No	Yes	No	Wait-free, but `size()` is not constant-time
ABQ	Yes	Yes	Yes	Optional	Capacity is fixed at instantiation
DQ	Yes	No	No	No	Elements implement the `Delayed` interface
LBQ	Yes	Optional	Yes	No	Capacity flexible, but no fairness option
PBQ	Yes	No	No	No	Elements are consumed in priority order
SQ	Yes	–	–	Optional	No capacity; serves as a channel

In the `java.util.concurrent` package, `ArrayBlockingQueue` (**ABQ**), `DelayQueue` (**DQ**), `LinkedBlockingQueue` (**LBQ**), `PriorityBlockingQueue` (**PBQ**), and `SynchronousQueue` (**SQ**) implement the `BlockingQueue` (**BQ**) interface—its Javadoc describes the characteristics of its method calls. ABQ is a fixed-capacity, FIFO queue backed by an array. LBQ is also a FIFO queue backed by linked nodes, and it is optionally bounded (default `Integer.MAX_VALUE`). ABQ and LBQ generate *back pressure* by blocking the enqueue operations on full capacity. ABQ supports optional fairness (with performance overhead) in the order of threads that access it.

DQ is an unbounded queue that accepts elements associated with the delay. The queue elements cannot be null and must implement the `java.util.concurrent.Delayed` interface. Elements are available for removal from the queue only after the delay has expired. DQ can be very useful for scheduling the processing of elements at different times.

PBQ is unbounded and blocking while letting elements to be consumed from the queue as per priority. Elements have natural ordering by default which can be overridden by specifying a comparator implementation when instantiating the queue.

SQ is not really a queue at all. Rather, it's just a barrier for a producer or a consumer thread. The producer forms a block until a consumer removes the element and vice versa. SQ does not have a capacity. However, SQ supports optional fairness (with performance overhead) in the order in which threads access it.

There are some new concurrent queue types introduced after Java 5. Since JDK 1.6, in the `java.util.concurrent` package Java has `BlockingDeque` (**BD**) with `LinkedBlockingDeque` (**LBD**) as the only available implementation. BD builds on BQ by adding `Deque` (**double-ended queue**) operations, that is, the ability to add elements to and to consume elements from both ends of the queue. LBD can be instantiated with optional capacity (bounded) to block on overflow. JDK 1.7 introduced `TransferQueue` (**TQ**) with `LinkedTransferQueue` (**LTQ**) as the only implementation. TQ extends the concept of SQ such that producers and consumers block on a queue of elements—this would help utilize producer and consumer threads better by keeping them busy. LTQ is an unbounded implementation of TQ, where the `size()` method is not a constant time operation.

Clojure support for concurrent queues

We discussed persistent queues in *Chapter 2*, *Clojure Abstractions*. Clojure has a built-in `seque` function that builds over a BQ implementation (LBQ by default) to expose a write-ahead sequence. The sequence is potentially lazy and the write-ahead buffer throttles how many elements can be realized. As opposed to chunked sequences (of chunk size 32), the size of the write-ahead buffer is controllable and potentially populated at all times until the source sequence is exhausted. Unlike chunked sequences, the realization doesn't happen suddenly for a chunk of 32 elements. It does so gradually and smoothly.

Clojure's `seque` function uses an agent under the hood to backfill data into the write-ahead buffer. In the arity-2 variant of `seque`, the first argument should either be a positive integer or an instance of BQ (ABQ, LBQ, and so on) that is preferably bounded.

Concurrency with threads

On the JVM, threads are the de-facto, fundamental instrument of concurrency. Multiple threads live in the same JVM; they share the heap space and compete for resources.

JVM support for threads

JVM threads are the operating system threads. Java wraps an underlying OS thread as an instance of the `java.lang.Thread` class and builds up an API around it to work with threads. A thread on the JVM has a number of states: `New`, `Runnable`, `Blocked`, `Waiting`, `Timed_Waiting`, and `Terminated`. A thread is instantiated by overriding the `run()` method of the `Thread` class or by passing an instance of the `java.lang.Runnable` interface to the constructor of the `Thread` class. Invoking the `start()` method of a `Thread` instance starts its execution in a new thread. Even when just a single thread is running in the JVM, the JVM would not shut down. Calling the `setDaemon(boolean)` method of a thread with the argument `true` tags the thread as a daemon that can be automatically shut down if no other non-daemon thread is running.

All Clojure functions implement the `java.lang.Runnable` interface. Therefore, invoking a function in a new thread is very easy:

```
(defn foo [] (dotimes [_ 5] (println "Foo")))
(defn bar [n] (dotimes [_ n] (println "Bar")))
(.start (Thread. foo)) ; prints "Foo" 5 times
(.start (Thread. (partial bar 3))) ; prints "Bar" 3 times
```

The `run()` method does not accept any argument. We can work around that by creating a higher-order function that needs no arguments but internally applies the argument `3`.

Thread pools in the JVM

Creating threads leads to operating system API calls, which is not always a cheap operation. The general practice is to create a pool of threads that can be recycled for different tasks. Java has a built-in support for threads pools. The interface `java.util.concurrent.ExecutorService` represents the API for a thread pool. The most common way to create a thread pool is to use a factory method in the `java.util.concurrent.Executors` class:

```
((import 'java.util.concurrent.Executors)
(import 'java.util.concurrent.ExecutorService)
(def ^ExecutorService a (Executors/newSingleThreadExecutor)) ;bounded
(def ^ExecutorService b (Executors/newCachedThreadPool)) ; unbounded
(def ^ExecutorService c (Executors/newFixedThreadPool 5)) ; bounded
(.execute b #(dotimes [_ 5] (println "Foo"))) ; prints "Foo" 5 times
```

The preceding example is equivalent to the examples with raw threads, which we saw in the previous subsection. Thread pools are also capable of tracking the completion and return value of a function executed in a new thread. An `ExecutorService` element accepts an instance of the `java.util.concurrent.Callable` instance as argument to several methods that launch a task and return a `java.util.concurrent.Future` instance to track the final result. All Clojure functions also implement the `Callable` interface, so we can use them as follows:

```
(import 'java.util.concurrent.Callable)
(import 'java.util.concurrent.Future)
(def ^ExecutorService e (Executors/newSingleThreadExecutor))
(def ^Future f (.submit e (cast Callable #(reduce + (range
10000000)))))
(.get f)  ; blocks until result is processed, then returns it
```

The thread pools described here are the same as the ones we saw briefly in the agents section earlier. Thread pools need to be shut down by calling the `shutdown()` method when they are no longer required.

Clojure concurrency support

Clojure has some nifty built-in features to deal with concurrency. We already discussed agents and how they use the thread pools in an earlier section. There are some more concurrency features in Clojure to deal with various use cases.

Asynchronous execution with Futures

We saw earlier in this chapter how we can use the Java API to launch a new thread to execute a function and also how to get the result back. Clojure has built-in support, called **Futures**, to do those things in a much smoother and integrated manner. The basis of Futures are the `future-call` function (take a no-arg function as argument) and the `future` macro (take the body of code as argument) which builds on the former. Both of them immediately start a thread to execute the supplied code. The following snippet illustrates the functions that work with Futures and how to use them:

```
;; runs body in new thread
(def f (future (println "Calculating") (reduce + 1e7)))
;; takes no-arg fn
(def g (futurecall # (do (println "Calculating") (reduce + 1e7))))
(future? f) ; returns true
;; cancels execution unless already over (can stop mid-way)
(future-cancel g)
```

```
;; returns true if canceled due to request, or due to exception
(future-cancelled? g)
;; returns true if terminated successfully, or canceled
(future-done? f)
;; same as future-done? for futures
(realized? f)
;; blocks if computation not yet over (use deref for timeout)
```

One of the interesting aspects of `future-cancel` is that it can sometimes not only cancel tasks that haven't started yet, but may abort those that are half way through execution:

```
(let [f (future (println "[f] Before sleep")
                (Thread/sleep 2000)
                (println "[f] After sleep")
               2000)]
  (Thread/sleep 1000)
  (future-cancel f)
  (future-cancelled? f))
;; [f] Before sleep printed message (second message is never printed)
;; true    returned value (due to future-cancelled?)
```

The preceding scenario happens because Clojure's `future-cancel` function cancels a Future in such a way that if the execution has already started, it may be interrupted causing `InterruptedException`, which if not explicitly caught would simply abort the block of code. In order for an executing Future task to be canceled cleanly via `future-cancel`, it must support catching of `InterruptedException` and logically cancel the task if `Object.wait()` or `Thread.join()` or `Thread.sleep()` are invoked; also, it must check `Thread/interrupted` for any cancel requests. Beware of exceptions arising from the code executed in a Future, because they are not verbosely reported by default! Clojure Futures use the *solo* thread pool (used to execute potentially blocking actions) that we discussed earlier with respect to agents.

Anticipated asynchronous execution result with promises

A **promise** is a placeholder for the result of a computation that may or may not have occurred. A promise is not directly associated with any computation. By definition, a promise does not imply when the computation might occur.

Typically, a promise originates from one place in the code and is realized by some other portion of the code that knows when and how to realize the promise. Very often, this happens in multithreaded code. If a promise is not realized yet, any attempt to read the value blocks all callers. If a promise is realized, all callers can read the value without being blocked. As with Futures, a promise can be read with a timeout using `deref`. The following is a very simple example showing how to use promises:

```
(def p (promise))
(realized? p) ; returns false
@p ; will block until another thread delivers the promise
(deliver p :foo)
@p ; returns :foo (for timeout use deref)
```

A promise is a very powerful tool that can be passed around as function arguments, stored in a reference type, or simply used for high-level coordination.

Clojure parallelization and the JVM

We observed in *Chapter 1*, *Performance by Design*, that parallelism is a function of the hardware, whereas concurrency is a function of the software assisted by hardware support. Except for algorithms that are purely sequential by nature, concurrency is the favored means to facilitate parallelism and achieve better performance. Immutable and stateless data is a catalyst to concurrency as there is no contention between threads due to the absence of mutable data.

Moore's law

In 1965, Intel's co-founder *Gordon Moore* made an observation that the number of transistors per square inch on integrated circuits doubles every 24 months. He also predicted that the trend would continue for 10 years, but in practice, it has continued till now, marking almost half a century. More transistors resulted in more computing power. With a larger number of transistors in the same area, we need higher clock speed to transmit signals to all of the transistors. Secondly, transistors need to get smaller to fit in. Around 2006 to 2007, the clock speed that the circuitry could work with topped out at about 2.8 GHz due to heating issues and laws of physics. Then multicore processors were born.

Amdahl's law

Multicore processors naturally require splitting up computation in order to achieve parallelization. Here begins a conflict—a program that was made to be run sequentially cannot make use of the parallelization features of multicore processors. The program must be altered to find the opportunity to split up computation at every step while keeping the cost of coordination in mind. This results in a limitation that a program can be no faster than its longest sequential part and the coordination overhead. This characteristic is described by Amdahl's law.

Clojure support for parallelization

A program that relies on mutation cannot parallelize its parts without creating contention on the mutable state. It requires coordination overhead, which makes the situation worse. Clojure's immutable nature is better suited to parallelize parts of a program. Clojure also has some constructs that are suited for parallelism by the virtue of Clojure's consideration of available hardware resources. The result is that the operations executed are optimized for certain use case scenarios.

pmap

The function `pmap` (like `map`) accepts as arguments a function and one or more collections of data elements. The function is applied to each of the data elements in such a way that some of the elements are processed by the function in parallel. The parallelism factor is chosen at runtime by the `pmap` implementation as two greater than the total number of available processors. It still processes the elements lazily, but the realization factor is the same as the parallelism factor:

```
(pmap (partial reduce +)
        [(range 1000000)
         (range 1000001 2000000)
         (range 2000001 3000000)])
```

To use `pmap` effectively, it is imperative that we understand what it is meant for. As the documentation says, it is meant for computationally-intensive functions. It is optimized for CPU-bound and cache-bound jobs. High-latency, low-CPU tasks such as blocking I/O is a gross misfit for `pmap`. Another pitfall to be aware of is whether the function used in `pmap` performs a lot of memory operations. Since the same function will be applied across all threads, all processors (or cores) may compete for the memory interconnect and subsystem bandwidth. If parallel memory access becomes a bottleneck, `pmap` cannot make the operation truly parallel due to the contention on memory access.

Another concern is what happens when several `pmap` operations are running concurrently? Clojure does not attempt to detect multiple `pmap`s running concurrently. The developer is responsible to ensure the performance characteristics and response time of the program resulting from concurrent `pmap` executions. Usually, when latency reasons are paramount, it is advisable to limit the concurrent instances of `pmap` running in the program.

pcalls

The `pcalls` function is built using `pmap`, so it borrows properties from the latter. However, the `pcalls` function accepts zero or more functions as arguments and executes them in parallel, returning the result values of the calls as a list.

pvalues

The `pvalues` macro is built using `pcalls`, so it transitively shares the properties of `pmap`. Its behavior is like `pcalls`, but instead of functions, it accepts zero or more S-expressions that are evaluated in parallel using `pmap`.

Java 7's fork/join framework

Java 7 introduced a new framework for parallelism called **fork/join** based on divide-and-conquer and work-stealing scheduler algorithms. The basic idea of how to use the fork/join framework is fairly simple: if the work is small enough, do it directly in the same thread; otherwise, split the work in to two pieces and invoke them in a fork/join thread pool and wait for the results to combine. This way, the job gets recursively split into smaller parts like an inverted tree until the smallest part can be carried out in just a single thread. When the leaf/subtree jobs return, the parent combines the result of all children and returns the results.

The fork/join framework is implemented in Java 7 in terms of a special kind of thread pool; see `java.util.concurrent.ForkJoinPool`. The specialty of this thread pool is that it accepts jobs of the type `java.util.concurrent.ForkJoinTask`, and whenever these jobs block waiting for the child jobs to finish, the threads used by the waiting jobs are allocated to the child jobs. When the child finishes its work, the thread is allocated back to the blocked parent jobs in order to continue. This style of dynamic thread allocation is described as **work-stealing**. The fork/join framework can be used from within Clojure. The interface `ForkJoinTask` has two implementations—`RecursiveAction` and `RecursiveTask` in the `java.util.concurrent` package. Concretely, `RecursiveTask` may be more useful with Clojure as `RecursiveAction` is designed to work with mutable data and does not return any value from its operation.

Using the fork/join framework entails choosing the batch size to split a job into, which is a crucial factor in parallelizing a long job. Too large a batch size may not utilize all CPU cores enough. On the other hand, a small batch size may lead to longer overhead coordinating across parent-child batches. As we would see in the next section, Clojure integrates with the fork/join framework to parallelize the reducer's implementation.

Parallelism with reducers

Reducers are a new abstraction introduced in Clojure 1.5 and are likely to have a wider impact on the rest of the Clojure implementation in future versions. They depict a different way of thinking about processing collections in Clojure—the key concept is to break down the notion that collections can be processed only sequentially, or only lazily, or only producing a seq, and so on. Moving away from such behavior guarantee raises the potential for eager and parallel operations on one hand while incurring constraints on the other hand. Reducers are compatible with the existing collections.

For an example, an observation of the regular `map` function reveals that its classic definition is tied to the mechanism (recursion), order (sequential), laziness (often), and representation (list/seq/other) aspects of producing the result. Most of this actually defines "how" the operation is performed rather than "what" needs to be done. In the case of `map`, the "what" is all about applying a function to each element of its collection arguments; but since collection types can be of various types (tree-structured, sequence, iterator, and so on), the operating function would not know how to navigate the collection. Reducers decouple the "what" and "how" parts of the operation.

Reducible, reducer function, reduction transformation

Collections are of various kinds, and hence only a collection knows how to navigate itself. In the reducer's model at a fundamental level, an internal "reduce" operation in each collection type has access to its properties and behavior, and access to what it returns. This makes all collection types essentially "reducible". All operations that work with collections can be modeled in terms of the internal "reduce" operation—the new modeled form of such operations is a **reducing function**, which is typically a function of two arguments; the first argument being the accumulator and the second being the new input.

How does it work when we need to layer several functions one upon another over elements of a collection? For example, let us say first we need to *filter*, then *map*, and then *reduce*. In such cases, a "transformation function" is used to model a reducer function (for example, for *filter*) as another reducer function (for *map*) such that it adds the functionality during transformation. This is called **reduction transformation**.

Realizing reducible collections

While reducer functions retain the purity of the abstraction, they are not useful all by themselves. Reducer operations in the namespace `clojure.core.reducers`, such as `map and filter`, return a reducible collection that embeds the reducer functions within themselves. A reducible collection is not realized, not even lazily realized—it is rather just a recipe ready to be realized. In order to realize a reducible collection, we must use one of the `reduce` or `fold` operations.

The `reduce` operation that realizes a reducible collection is strictly sequential, albeit with performance gains compared to `clojure.core/reduce` due to reduced object allocations on the heap. The `fold` operation that realizes a reducible collection is potentially parallel and uses a "reduce-combine" approach over the fork/join framework. Unlike the traditional map-reduce style of using fork/join, the reduce-combine approach reduces at the bottom and subsequently combines by means of reduction again. This makes the `fold` implementation less wasteful and better performing.

Foldable collections and parallelism

Parallel reduction by `fold` puts certain constraints on collections and operations. The tree-based collection types (persistent map, persistent vector, and persistent set) are amenable to parallelization. At the same time, sequences may not be parallelized by `fold`. Secondly, `fold` requires that the individual reducer functions should be "associative", that is, the order of the input arguments applied to the reducer function should not matter. The reason is that `fold` can segment the elements of the collection to process in parallel and the order in which they may be combined is not known in advance.

The fold function accepts a few extra arguments, such as the combine function and the partition batch size (default being 512) for parallel processing. Choosing the optimum partition size depends on the jobs, host capabilities, and performance benchmarking. There are certain functions that are foldable (that is, parallelizable by fold), and there are others that are not, as shown below. They live in the clojure.core.reducers namespace:

- **Foldable**: map, mapcat, filter, remove, and flatten
- **Non-foldable**: take-while, take, and drop
- **Combine functions**: cat, foldcat, and monoid

A notable aspect of Reducers is that they are foldable in parallel only when the collection is a tree-type collection. That implies the entire data set must be loaded in the heap when folding over them. This has the downside of memory consumption during high load on a system. On the other hand, a lazy sequence is a perfectly reasonable solution for such scenarios. When processing a large amount of data, it may make sense to use a combination of lazy sequences and Reducers for performance.

Summary

Concurrency and parallelism are extremely important for performance in this multicore age. Effective use of concurrency requires substantial understanding of the underlying principles and details. Fortunately, Clojure provides safe and elegant ways to deal with concurrency and state. Clojure's new feature "Reducers" provides a way to achieve granular parallelism. In the coming years, we are likely to see more and more processor cores and an increasing demand to write code that takes advantage of those. Clojure places us in the right spot to meet such challenges.

In the next chapter, we will look at performance analysis and optimization, and we will briefly touch upon performance tuning.

6
Optimizing Performance

Depending on the degree of mismatch between expected and actual performance and the lack or presence of measuring systems in place, performance analysis and tuning can be a fairly elaborate process. In this chapter, we will discuss analysis of performance characteristics and the opportunities for performance optimization. The elements involved in such activities are not specific to Clojure, so it should be possible to apply these concepts to other JVM systems too. In this chapter, we will discuss the following topics:

- How to measure performance and understand the measurement results
- What performance tests to carry out for different purposes
- Monitoring performance and obtaining metrics
- Profiling Clojure code to identify performance bottlenecks
- Tuning performance measurement and statistics

Measuring performance is the stepping stone to performance analysis. Often, where we think the code underperforms is not where the problem lies. So much so that it is rarely advisable to start optimizing performance until we measure it. As we've noted earlier in this book, there are several performance parameters to measure under various scenarios. Clojure's built-in `time` macro is a tool to measure the amount of time elapsed while executing a body of code. Measuring performance factors is a much more involved process. The measured performance numbers may be linked with each other that we need to analyze. It is a common practice to use statistical concepts to establish the linkage factors. We will discuss some basic statistical concepts in this section and use them to explain how the measured data gives us the bigger picture.

A tiny statistics terminology primer

When we have a series of quantitative data, such as latency in milliseconds, for the same operation (measured over a number of executions), we can observe a number of things. First, and the most obvious, are the minimum and maximum values in the data. Let us take an example data set to analyze further:

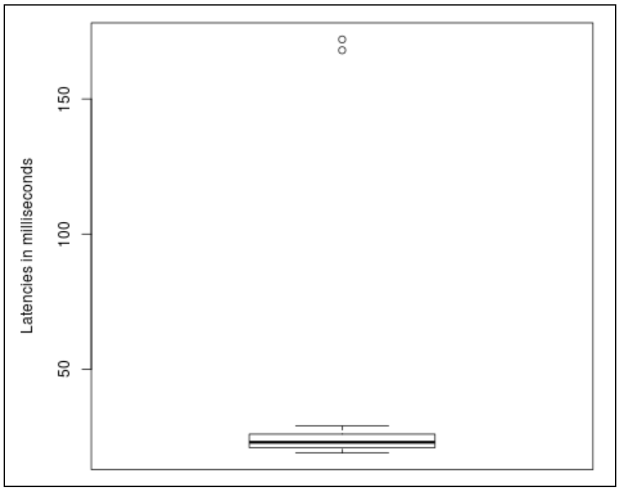
23	19	21	24	26	20	22	21	25	168	23	20	29	172	22	24	26

Median, first quartile, and third quartile

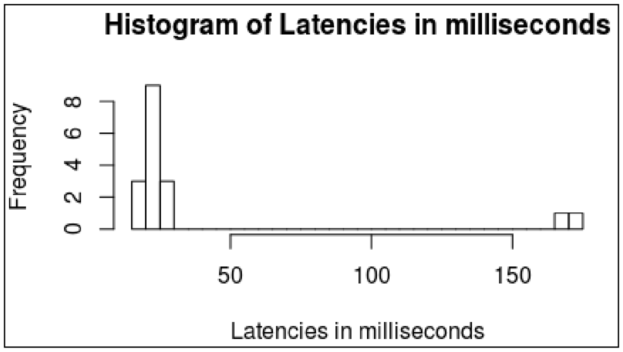
We can see that the minimum latency here is 19 ms, whereas the maximum latency is 172 ms. We can also observe that the average latency here is about 40 ms. Let us sort this data in ascending order:

19	20	20	21	21	22	22	23	23	24	24	25	26	26	29	168	172

The center element of the preceding data set, that is, the ninth element (value 23), is considered the **median** of the data set. It is noteworthy that the median is a better representative of the center of the data than the average or **mean**. The center element of the left-half of the table, that is, the fifth element (value 21), is considered the **first quartile**. Similarly, the center element of the right-half of the table, that is, the 13th element (value 26), is considered the **third quartile** of the data set. The difference between the third quartile and the first quartile is called the **Inter Quartile Range** (**IQR**), which is five in this case. This can be illustrated with a **boxplot** as follows:



A boxplot highlights the first quartile, median, and third quartile of a data set. As you can see, the two **outlying** latency numbers (168 and 172) are unusually higher than the others. The median does not represent outliers in a data set, whereas the mean does. The same data can be depicted as a histogram:



A histogram (as shown in the preceding diagram) is another way to display a data set, where we batch the data elements into **periods** and expose the **frequency** of such periods. A period contains the elements in a certain range. All periods in a histogram are generally of the same size; however, it is common to omit certain periods when there is no data.

Percentile

A **percentile** is expressed with a parameter, such as 99th percentile or 95th percentile, and so on. The percentile is the value below which all the specified percentage of data elements exist. For example, the 95th percentile indicates the value N in a data set, such that 95 percent of the elements in the data set are below N in value. As a concrete example, the 85th percentile in the data set of latency numbers we've discussed earlier in this section is 29; because, out of the 17 total elements, 14 (that is, 85 percent of 17) other elements in the data set have a value less than 29. A quartile splits a data set into chunks, each comprising 25 percent of its elements. Therefore, the first quartile is actually the 25th percentile, the median is the 50th percentile, and the third quartile is the 75th percentile.

Variance and standard deviation

The spread of the data in a data set, that is, how far the data elements are from the mean, gives us further insight into the data. Consider the **i**th deviation as the difference between the **i**th data set element value (in statistical terms, a **variable** value) and its mean; we can represent it as follows:

$$x_i - \bar{x}$$

We can express its variance and standard deviation as follows:

$$\text{Variance} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{(n-1)}$$

$$\text{Standard Deviation}(\sigma) = \sqrt{Variance} = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{(n-1)}}$$

Standard deviation is shown as the Greek letter **σ** (**sigma**) or simply **s**. Consider the following Clojure code to determine variance and standard deviation:

```
(def tdata [23 19 21 24 26 20 22 21 25 168 23 20 29 172 22 24 26])

(defn var-std-dev
  "Return variance and standard deviation in a vector"
  [data]
  (let [size (count data)
        mean (/ (reduce + data) size)
        sum (->> data
                 (map #(let [v (- % mean)] (* v v)))
                 (reduce +))
        variance (double (/ sum (dec size)))]
    [variance (Math/sqrt variance)]))

user=> (println (var-std-dev tdata))
[2390.345588235294 48.89116063497873]
```

You can use the Clojure-based platform Incanter (`http://incanter.org/`) for statistical computations. For example, you can find standard deviation using `(incanter.stats/sd tdata)` in Incanter.

The **empirical rule** states the relationship between the elements of a data set and the standard deviation (SD) in a **normal distribution**. It says that 68.3 percent of all elements in a data set lie in the range of one (positive or negative) SD from the mean, 95.5 percent of all elements lie in the range of two SDs from the mean, and 99.7 percent of all data elements lie in the range of three SDs from the mean.

Looking at the latency data set we started out with, one SD from mean is 40 ± 49 (range -9 to 89) containing 88 percent of all elements. Two SDs from the mean is 40 ± 98 (range -58 to 138) containing the same 88 percent of all elements. However, three SDs from the mean is 40 ± 147 (range -107 to 187) containing 100 percent of all elements. There is a mismatch between what the empirical rule states and the results we've found because the empirical rule applies generally to uniformly distributed data sets with a large number of elements.

Understanding criterium output

In *Chapter 4*, *Host Performance*, we introduced the Clojure library criterium to measure the latency of Clojure expressions. A sample benchmarking result is as follows:

```
user=> (bench (reduce + (range 1000)))
Evaluation count : 162600 in 60 samples of 2710 calls.
             Execution time mean : 376.756518 us
    Execution time std-deviation : 3.083305 us
   Execution time lower quantile : 373.021354 us ( 2.5%)
   Execution time upper quantile : 381.687904 us (97.5%)

Found 3 outliers in 60 samples (5.0000 %)
    low-severe   2 (3.3333 %)
    low-mild   1 (1.6667 %)
 Variance from outliers : 1.6389 % Variance is slightly inflated by
outliers
```

We can see that the result has some familiar terms we've discussed earlier in this section. A high mean and low standard deviation indicate that there is not a lot of variation in the execution times. Likewise, the lower (first) and upper (third) quartiles indicate that the values are not too far away from the mean. This result implies that the body of code is more or less stable in terms of response time. Criterium repeats the execution many times to collect the latency numbers.

However, why does criterium attempt to do a statistical analysis of the execution time? What would be amiss if we simply calculated the mean? It turns out that the response times of all executions are not always stable and there is often disparity between how the response times show up. Only upon running criterium a sufficient number of times under correctly simulated load can we get complete data and get other indicators about latency. A statistical analysis gives us an insight into whether there is something wrong with the benchmark.

Guided performance objectives

We discussed performance objectives only briefly in *Chapter 1*, *Performance by Design*. Detail discussion of performance objectives needs to refer to statistical concepts such as standard deviation, and percentile. Let us say we've identified the functional scenarios and the corresponding response times. Should the response times remain fixed? Can we constrain throughput in order to prefer a stipulated response time?

The performance objective should specify the worst-case response time, that is, the maximum latency, average response time, and maximum standard deviation. Similarly, the performance objective should also mention the worst-case throughput, maintenance window throughput, average throughput, and maximum standard deviation.

Performance testing

Testing for performance requires that we know what we are going to test, how we want to test it, and what environment to set up for the tests to execute. There are several pitfalls to be aware of, such as a lack of near-real hardware and resources of production use, similar OS and software environments, diversity of representative data for test cases, and so on. Lack of diversity in test inputs may lead to monotonic branch prediction, which introduces bias in test results. Collecting enough performance test samples is a significant criterion to obtain a statistically meaningful data set and prevent skewing.

Test environment

Concerns about the test environment begin with the hardware representative of the production environment. Traditionally, the test environment hardware has been a scaled-down version of the production environment. A performance analysis done on a non-representative hardware is almost certain to skew the results. Fortunately, in recent times, thanks to the commodity hardware and Cloud systems, providing test environment hardware that is similar to the production environment is not too difficult.

The network and storage bandwidth, OS, and software used for performance testing should, of course, be the same as those in the production environment.. What is also important is to have a load representative of the test scenarios. The load comes in different combinations, including the concurrency of requests, the throughput and standard deviation of requests, the current population level in the database or in the message queue, CPU and heap usage, and so on. It is important to simulate a representative load.

Testing often requires quite some work on the part of the piece of code that carries out the test. Be sure to keep its overhead minimal so that it does not impact the benchmark results. When possible, use a system other than the test target to generate requests.

What to test

Any implementation of a non-trivial system typically involves many hardware and software components. Performance testing a certain feature or service in the entire system needs to account for the way it interacts with the various subsystems; a break-down of the time spent across subsystems gives us quicker insight into the potential bottlenecks. For example, a web service call may touch multiple layers, such as the web server (request/response marshaling and unmarshaling), URL-based routing, service handler, application-database connector, the database layer, logger component, and so on. Testing only the service handler would be a terrible mistake because that does not depict exactly the performance that the web client will experience. The performance test should test at the perimeter of a system, for example at the HTTP layer in the case of a web service, to keep the results realistic. To test a network traffic it is preferable to have a third-party observer that incurs little overhead on the client or the server.

The performance objectives state the criteria for testing. It would be useful not to test what is not required by the objective, especially when the tests are run concurrently. Running meaningful performance tests may require a certain level of isolation.

Measuring latency

The latency obtained by executing a body of code may vary slightly on each run. This necessitates that we execute the code many times and collect samples. The latency numbers may be impacted by the JVM warm-up time, GC, and the JIT compiler having not yet kicked in. So, the test and sample collection should ensure that these conditions do not impact the results. Criterium follows such methods while capturing samples. When we test a very small piece of code this way, it is called a **Micro-benchmark**.

As the latency of some operations may vary during runs, it is useful to collect all samples and segregate them into periods and frequencies, forming a histogram. The maximum latency is an important metric when measuring latency; it indicates the worst-case latency. Besides the maximum, the 99th percentile and 95th percentile latency numbers help to put things in perspective. It's crucial to actually collect the latency numbers instead of inferring them from the standard deviation, as we've noted earlier that the empirical rule works only for normal distributions without significant outliers.

The outliers are an important data point when measuring latency. A proportionately higher count of outliers indicates the possibility of degradation of service.

Measuring throughput

Throughput is expressed per unit of time. Coarse-grained throughput, that is, the throughput number collected over a long period of time, may hide facts about instances when the throughput is actually delivered in bursts instead of a uniform distribution. Granularity of the throughput test is subject to the nature of the operation. A batch process may process bursts of data, whereas a web service may deliver uniformly distributed throughput.

Load, stress, and endurance tests

One of the characteristics of tests is that each run only represents the slice of time it is executed through. Repeated runs establish their general behavior. But, how many runs should be enough? There may be several anticipated load scenarios for an operation. So, there is a need to repeat the tests in the various load scenarios. Simple test runs may not always exhibit the long term behavior and response of the operation. Running the tests under varying high load for a longer duration allows us to observe them for any odd behavior that may not show up in a short test cycle. Often, performance issues crop up in specific scenarios and corner cases, all of which may be difficult to frame in example-based testing. For the rigorous testing of various use cases, you may like to consider simulation testing using tools such as Simulant available at `https://github.com/Datomic/simulant`.

When we test an operation at a load far beyond its anticipated latency and throughput objectives, that is called **stress testing**. The intent of a stress test is to ascertain a reasonable behavior exhibited by the operation beyond the maximum load it was developed for. Another way to observe the behavior of an operation is to see how it behaves when it's run for a very long duration, typically for several days or weeks. Such prolonged tests are called **endurance tests**. While a stress test checks the graceful behavior of the operation, an endurance test checks the consistent behavior of the operation over a long period.

There are several tools that may help with load and stress testing. Engulf (`http://engulf-project.org/`) is a distributed HTTP-based load generation tool written in Clojure. JMeter (`http://jmeter.apache.org/`) and Grinder (`http://grinder.sourceforge.net/`) are Java-based load generation tools. Grinder can be scripted using Clojure. Apache Bench (`http://httpd.apache.org/docs/2.4/programs/ab.html`) is a load testing tool for web systems. Tsung (`http://tsung.erlang-projects.org/`) is an extensible, high-performance load testing tool written in Erlang.

Performance monitoring

During prolonged testing or after the application has gone to production, we need to monitor its performance to make sure the application continues to meet the performance objectives. There may be infrastructural or operational issues impacting the performance or availability of the application, occasional spikes in latency, or dips in throughput. Generally, monitoring alleviates such risk by generating a continuous feedback stream.

Roughly, there are three kinds of components used to build a monitoring stack. A **collector** sends the numbers from each host that needs to be monitored. Usually we explicitly make calls in code to periodically send performance data or events to the collector. The collector gets host information and the performance numbers and sends them to the **aggregator**. An aggregator receives the data sent by the collector and persists them until asked by a **visualizer** on behalf of the user– the visualizer displays the data in a suitable format.

The project **metrics-clojure** (`https://github.com/sjl/metrics-clojure`) is a Clojure wrapper over the **Metrics** (`http://metrics.codahale.com/`) Java framework, which acts as a collector. **Statsd** (`https://github.com/etsy/statsd/`) is a well-known aggregator that does not persist data by itself, but passes it on to a variety of servers. One of the popular visualizer projects is **Graphite** (`http://graphite.wikidot.com/`), which stores the data as well as produces graphs for requested periods. There are several other alternatives to these, notably Riemann (`http://riemann.io/`), which is written in Clojure and Ruby. Riemann is an event-processing-based aggregator.

Introspection

Both Oracle JDK and OpenJDK provide two GUI tools called JConsole (executable name `jconsole`) and JVisualVM (executable name `jvisualvm`) that we can use to explore running JVMs for instrumentation data. There are also some command-line tools (`http://docs.oracle.com/javase/7/docs/technotes/tools/`) in the JDK to peek into the inner details of the running JVMs.

A common way to introspect a running Clojure application is to have an **nREPL** (`https://github.com/clojure/tools.nrepl`) service running so that we can connect to it later using an nREPL client. Interactive introspection over nREPL using the Emacs editor (an embedded nREPL client) is popular among some, whereas some others prefer to script an nREPL client to carry out tasks. However, leaving the nREPL server exposed to the outside world poses a severe security threat; be sure to restrict it for authorized use only.

JVM instrumentation via JMX

The JVM has a built-in mechanism to introspect managed resources via the extensible **Java Management Extensions** (**JMX**) API. It provides a way for application maintainers to expose manageable resources as **MBeans**. Clojure has an easy-to-use `contrib` library called `java.jmx` (`https://github.com/clojure/java.jmx`) to access JMX. There is a decent amount of open source tooling for visualization of JVM instrumentation data via JMX, such as jmxtrans and jmxetric which integrate with Ganglia and Graphite.

Getting quick memory stats of the JVM is pretty easy using Clojure:

```
(let [^Runtime r (Runtime/getRuntime)]
  (println “Maximum memory” (.maxMemory r))
  (println “Total memory” (.totalMemory r))
  (println “Free memory” (.freeMemory r)))
Output:
Maximum memory 704643072
Total memory 291373056
Free memory 160529752
```

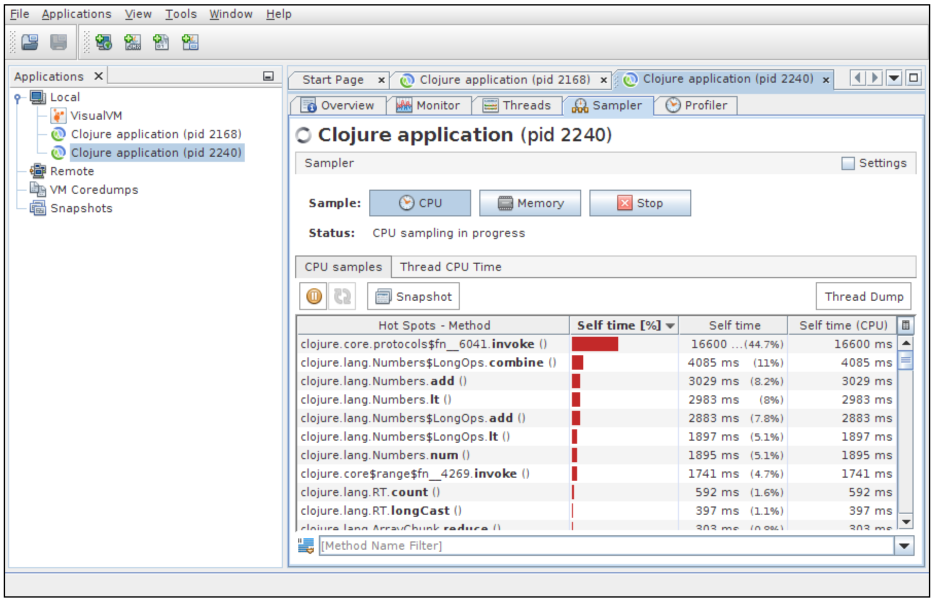
Profiling

We've briefly discussed profiler types in *Chapter 1*, *Performance by Design*. The **JVisualVM** tool we discussed with respect to introspection in the previous section is also a CPU and memory profiler that comes bundled with the JDK. Let us see them in action. Consider the following two Clojure functions that stress on the CPU and memory respectively:

```
(defn cpu-work []
  (reduce + (range 100000000)))

(defn mem-work []
  (->> (range 1000000)
       (map str)
       vec
       (map keyword)
       count))
```

Using JVisualVM is pretty easy; open the Clojure JVM process from the left-pane. It has the sampler and regular profiler styles for profiling. Start profiling for CPU or memory use when the code is running and wait for it to collect enough data to plot on the screen. The following screenshot shows CPU profiling in action:



Clojure application (pid 2240)

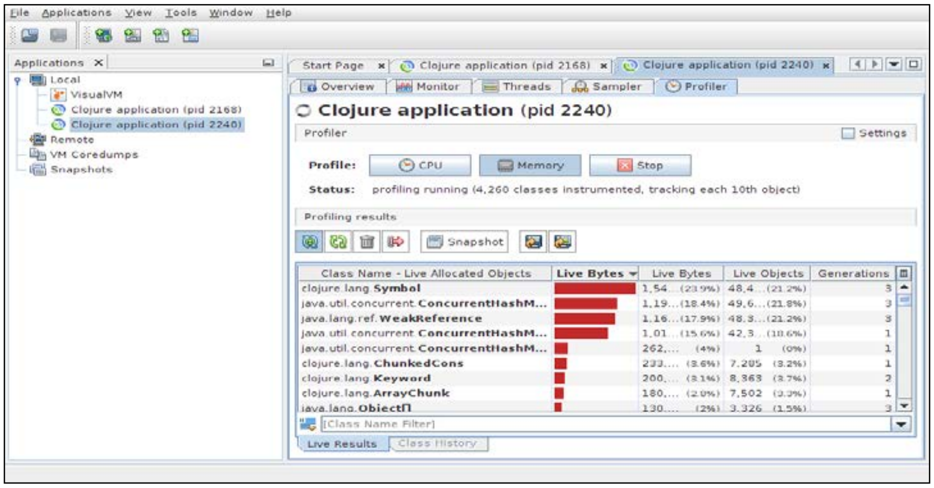
Sampler

Sample: CPU Memory Stop

Status: CPU sampling in progress

Hot Spots - Method	Self time [%]	Self time	Self time (CPU)
clojure.core.protocols$fn__6041.**invoke** ()		16600 ...(44.7%)	16600 ms
clojure.lang.Numbers$LongOps.**combine** ()		4085 ms (11%)	4085 ms
clojure.lang.Numbers.**add** ()		3029 ms (8.2%)	3029 ms
clojure.lang.Numbers.**lt** ()		2983 ms (8%)	2983 ms
clojure.lang.Numbers$LongOps.**add** ()		2883 ms (7.8%)	2883 ms
clojure.lang.Numbers$LongOps.**lt** ()		1897 ms (5.1%)	1897 ms
clojure.lang.Numbers.**num** ()		1895 ms (5.1%)	1895 ms
clojure.core$range$fn__4269.**invoke** ()		1741 ms (4.7%)	1741 ms
clojure.lang.RT.**count** ()		592 ms (1.6%)	592 ms
clojure.lang.RT.**longCast** ()		397 ms (1.1%)	397 ms
clojure.lang.ArrayChunk.**reduce** ()		303 ms (0.8%)	303 ms

The following screenshot shows memory profiling in action:



Clojure application (pid 2240)

Profiler

Profile: CPU Memory Stop

Status: profiling running (4,260 classes instrumented, tracking each 10th object)

Profiling results

Class Name - Live Allocated Objects	Live Bytes	Live Bytes	Live Objects	Generations
clojure.lang.**Symbol**		1,54...(23.9%)	48,4...(21.2%)	3
java.util.concurrent.**ConcurrentHashM...**		1,19...(18.4%)	49,6...(21.8%)	3
java.lang.ref.**WeakReference**		1,16...(17.9%)	48,3...(21.2%)	3
java.util.concurrent.**ConcurrentHashM...**		1,01...(15.6%)	42,3...(18.6%)	1
java.util.concurrent.**ConcurrentHashM...**		262,... (4%)	1 (0%)	1
clojure.lang.**ChunkedCons**		233,... (3.6%)	7,205 (3.2%)	1
clojure.lang.**Keyword**		200,... (3.1%)	8,363 (3.7%)	2
clojure.lang.**ArrayChunk**		180,... (2.0%)	7,502 (3.3%)	1
java.lang.**Object[]**		130,... (2%)	3,326 (1.5%)	3

JVisualVM is a very simple, entry-level profiler. There are several commercial JVM profilers on the market for sophisticated needs.

OS and CPU-cache-level profiling

Profiling only the JVM may not always tell the whole story. Getting down to OS- and hardware-level profiling often provides a better insight into what is going on with the application. On Unix-like operating systems, command-line tools, such as `top`, `htop`, `perf`, `iostat`, `netstat`, `vmstat`, `mpstat`, and `pidstat` can help. On Linux, MSR tools such as `cpuid`, `rdmsr`, and `wrmsr` can provide additional information. Profiling the CPU for cache misses and other information is a useful way of catching performance issues. Among open source tools for Linux, **Likwid** (`http://code.google.com/p/likwid/`) is small, yet effective for Intel and AMD processors; **i7z** (`https://code.google.com/p/i7z/`) is specifically for Intel processors. There are also dedicated commercial tools such as **Intel VTune Analyzer** for elaborate needs.

I/O profiling

Profiling I/O may require special tools too. Besides `iostat` and `blktrace`, **ioping** (`https://code.google.com/p/ioping/`) is useful to measure real-time I/O latency on Linux/Unix systems. The **vnStat** tool is useful to monitor and log network traffic on Linux. The IOPS of a storage device may not tell the whole truth unless it is accompanied by latency information for different operations and how many reads and writes can happen simultaneously.

In an I/O-bound workload, you have to look for the read and write IOPS over time and set a threshold to achieve optimum performance. The application should throttle I/O access so that the threshold is not crossed.

Performance tuning

Once we get an insight into the code via test and profiling results, we need to analyze the bottlenecks worth considering for optimization. A better approach is to find the most underperforming portion and optimize it, thereby eliminating the weakest link. We discussed performance aspects of hardware and JVM/Clojure in previous chapters. Optimization and tuning requires rethinking the design and code in light of those aspects and refactoring for performance objectives.

Once we establish the performance bottlenecks, we have to pinpoint the root cause and experiment with improvisations, one step at a time, to see what works. Tuning for performance is an iterative process backed by measurement, monitoring, and experimentation.

Identifying the nature of the performance bottleneck helps a lot in order to experiment with the right aspects of the code. The key is to determine the origin of cost and whether the cost is reasonable. As a general rule, we have to see if the type hints are applied in order to avoid reflection and boxing and whether or not we are performing unnecessary computation inside loops. If the code in question is CPU-bound, we have to see whether or not we are using the data types that can fit well in CPU registers and whether or not we can reduce branch mispredictions. For cache- and memory-bound code, we have to know whether or not there are cache misses, and the reason is that, often, the data might be too large to fit in a cache line. Knowing the memory layout (for example, in primitive arrays) can help us prefetch spatial and sequential data in adjacent cache lines. For memory-bound code, we have to care about data locality, whether the code is hitting the interconnect too often, page size versus the cost of paging, and whether or not memory representation of data can be slimmed down.

JVM tuning

Often, Clojure applications might inherit bloat from Clojure/Java libraries or frameworks, which causes poor performance. Hunting down unnecessary abstractions and layers of code may bring decent performance gains. Reasoning with the performance of dependency libraries/frameworks before their inclusion in the project is a good approach.

The JIT compiler, GC, and Safepoint (in Oracle HotSpot JVM) have significant impact on the performance of applications. We discussed the JIT compiler and GC in *Chapter 4*, *Host Performance*. When the HotSpot JVM reaches a point when it cannot carry out concurrent, incremental GC anymore, it needs to suspend the JVM safely in order to carry out a full GC. This is also called the stop-the-world GC pause, and it may run up to several minutes while the JVM appears frozen.

The Oracle and OpenJDK JVMs accept many command-line options, when invoked, to tune and monitor the way components in the JVM behave. Tuning GC is common among people who want to extract optimum performance from the JVM. On the Java 6 HotSpot JVM, the **Concurrent Mark and Sweep** (**CMS**) garbage collector is well regarded for its GC performance. On the Java 7 HotSpot JVM, the recommended way forward for GC is by using the G1 garbage collector.

> The JVM GC can be tuned for different objectives; hence, the same exact configuration for one application may not work well for another.

I/O tuning and backpressure

I/O-bound tasks could be limited by bandwidth or IOPS/latency. Any I/O bottleneck usually manifests in chatty I/O calls or unconstrained data serialization. Restricting I/O to only the minimum required data is a common opportunity to minimize serialization and reduce latency. I/O operations can often be batched for higher throughput; for example, the SpyMemcached library employs an asynchronous batched operation for high throughput.

It is not uncommon to see applications behaving poorly under load. Typically, the application server simply appears unresponsive, which is often a combined result of high resource utilization, GC pressure, more threads that lead to busier thread scheduling, cache misses, and CPU stalls. If the capacity of a system is known, the solution is to apply **backpressure** by denying services after capacity is reached. Note that backpressure cannot be applied optimally until the system is load-tested for optimum capacity. The capacity threshold that triggers backpressure may or may not be directly associated with individual services, but can be defined as load criteria.

Summary

Delivering high-performance applications requires not only care for performance, but also systematic effort to measure, test, monitor, and optimize the performance of various components and subsystems. The key is to first measure, then optimize, and subsequently repeat the process during the application life cycle. These activities often require the right skill and experience. Sometimes, performance considerations may even bring system design and architecture back to the drawing board. Early structured steps taken to achieve performance go a long way in ensuring that the performance objectives are being continuously met.

In the next chapter, we will see how to address performance concerns when building applications. Our focus will be the several common patterns that impact performance.

7
Application Performance

As opposed to performance analysis and optimization at a smaller component level, it takes a holistic approach for the same at the application level. Higher level concerns, such as serving a certain threshold of users in a day or handling an identified quantum of load through a multilayered system, require us to think about how the components fit together and how the load is designed to flow through the application. In this chapter, we will discuss such high level concerns. Like the previous chapter, by and large this chapter applies to applications written in any JVM language, but it is written with a special focus on Clojure. In this chapter, we will discuss the following topics:

- General performance techniques that apply to all layers of the code
- Data sizing
- Resource pooling
- Fetching and computing in advance
- Staging and batching
- Little's law

Data sizing

The cost of abstractions in terms of data size plays an important role. For example, whether or not a data element can fit into a processor cache line depends directly upon its size. On a Linux system, we can find out the cache line size and other parameters by inspecting the values in the files under `/sys/devices/system/cpu/cpu0/cache/`. Refer to *Chapter 4*, *Host Performance*, where we discussed how to compute the size of primitives, objects, and data elements.

Another concern we generally find with data sizing is how much data we are holding at a time in the heap. As we noted in earlier chapters, GC has direct consequences on the application's performance. While processing data, often we do not really need all the data we hold on to. Consider the example of generating a summary report of sold items for a certain period (months) of time. After the subperiod (month wise), summary data is computed. We do not need the item details anymore, hence it's better to remove the unwanted data while we add the summaries. This is shown in the following example:

```
(defn summarize [daily-data]  ; daily-data is a map
  (let [s (items-summary (:items daily-data))]
    (-> daily-data
    (select-keys [:digest :invoices])  ; we keep only the required key/val pairs
    (assoc :summary s))))

;; now inside report generation code
(-> (fetch-items period-from period-to :interval-day)
  (map summarize)
  generate-report)
```

Had we not used `select-keys` in the preceding `summarize` function, it would have returned a map with extra summary data along with all the other existing keys in the map. Now, such a thing is often combined with lazy sequences. So, for this scheme to work, it is important not to hold on to the head of the lazy sequence. Recall that in *Chapter 2*, *Clojure Abstractions*, we discussed the danger of holding on to the head of a lazy sequence.

Reduced serialization

We discussed in earlier chapters that serialization over an I/O channel is a common source of latency. The perils of over-serialization cannot be overstated. Whether we read or write data from a data source over an I/O channel, all of that data needs to be prepared, encoded, serialized, de-serialized, and parsed before being worked on. It is better for every step to have less data involved in order to lower the overhead. Where there is no I/O involved, such as in-process communication, it generally makes no sense to serialize.

A common example of over-serialization is encountered while working with SQL databases. Often, there are common SQL query functions that fetch all columns of a table or a relation—they are called by various functions that implement the business logic. Fetching data that we do not need is wasteful and detrimental to the performance for the same reason that we discussed in the preceding paragraph. While it may seem more work to write one SQL statement and one database query function for each use case, it pays off with better performance. Code that uses NoSQL databases is also subject to this anti-pattern—we have to take care to fetch only what we need even though it may lead to additional code.

There's a pitfall to be aware of when reducing serialization. Often, some information needs to be inferred in absence of the serialized data. In such cases where some of the serialization is dropped so that we can infer other information, we must compare the cost of inference versus the serialization overhead. The comparison may not be necessarily done per operation, but rather on the whole. Then, we can consider the resources we can allocate in order to achieve capacities for various parts of our systems.

Chunking to reduce memory pressure

What happens when we slurp a text file regardless of its size? The contents of the entire file will sit in the JVM heap. If the file is larger than the JVM heap capacity, the JVM will terminate by throwing `OutOfMemoryError`. If the file is large but not large enough to force the JVM into an OOM error, it leaves a relatively smaller JVM heap space for other operations in the application to continue. A similar situation takes place when we carry out any operation disregarding the JVM heap capacity. Fortunately, this can be fixed by reading data in chunks and processing them before reading further. In *Chapter 3*, *Leaning on Java*, we briefly discussed memory mapped buffers, which is another complementary solution that you may like to explore.

Sizing for file/network operations

Let us take the example of a data ingestion process where a semi-automated job uploads large **Comma Separated File** (**CSV**) files via the **File Transfer Protocol** (**FTP**) to a file server, and another automated job, which is written in Clojure, runs periodically to detect the arrival of files via the **Network File System** (**NFS**). After detecting a new file, the Clojure program processes the file, updates the result in a database, and archives the file. The program detects and processes several files concurrently. The size of the CSV files is not known in advance, but the format is predefined.

As per the preceding description, one potential problem is that since there could be multiple files being processed concurrently, how do we distribute the JVM heap among the concurrent file-processing jobs? Another issue could be that the operating system imposes a limit on how many files can be opened at a time; on Unix-like systems, you can use the `ulimit` command to extend the limit. We cannot arbitrarily slurp the CSV file contents—we must limit each job to a certain amount of memory and also limit the number of jobs that can run concurrently. At the same time, we cannot read a very small number of rows from a file at a time because this may impact performance.

```
(def ^:const K 1024)

;; create the buffered reader using custom 128K buffer-size
(-> filename
  java.io.FileInputStream
  java.io.InputStreamReader
  (java.io.BufferedReader (* K 128)))
```

Fortunately, we can specify the buffer size when reading from a file or even from a network stream so as to tune the memory usage and performance as appropriate. In the preceding code example, we explicitly set the buffer size of the reader to facilitate the same.

Sizing for JDBC query results

Java's interface standard for SQL databases, JDBC (which is technically not an acronym), supports `fetch-size` for fetching query results via JDBC drivers. The default fetch size depends on the JDBC driver. Most JDBC drivers keep a low default value so as to avoid high memory usage and attain internal performance optimization. A notable exception to this norm is the MySQL JDBC driver that completely fetches and stores all rows in memory by default.

```
(require '[clojure.java.jdbc :as jdbc])

;; using prepare-statement directly (we rarely use it directly, shown
just for demo)
(with-open
  [stmt (jdbc/prepare-statement conn sql :fetch-size 1000 max-rows
9000)
   rset (resultset-seq (.executeQuery stmt))]
  (vec rset))

;; using query
(query db [{:fetch-size 1000} "SELECT empno FROM emp WHERE country=?"
1])
```

When using the Clojure Contrib library `java.jdbc` (`https://github.com/clojure/java.jdbc` as of Version 0.3.0), the fetch size can be set while preparing a statement as shown in the preceding example.

> The fetch size does not guarantee proportional latency; however, it can be used safely for memory sizing.

We must test any performance-impacting latency changes due to fetch size at different loads and use cases for the particular database and JDBC driver. Besides `fetch-size`, we can also pass the `max-rows` argument to limit the maximum rows to be returned by a query. However, this implies that the extra rows will be truncated from the result, not that the database will internally limit the number of rows to realize.

Resource pooling

There are several types of resources on the JVM that are rather expensive to initialize. Examples are HTTP connections, execution threads, JDBC connections, and so on. The Java API recognizes such resources and has built-in support for creating a pool of some of those resources so that the consumer code borrows a resource from a pool when required and at the end of the job simply returns it to the pool. Java's thread pools (discussed in *Chapter 6*, *Optimizing Performance*) and JDBC data sources are prominent examples. The idea is to preserve the initialized objects for reuse.
Even when Java does not support pooling of a resource type directly, you can always create a pool abstraction around custom expensive resources.

> The pooling technique is common in I/O activities, but it can be equally applicable to non-I/O purposes where the initialization cost is high.

JDBC resource pooling

Java supports the obtaining of JDBC connections via the `javax.sql.DataSource` interface, which can be pooled. A JDBC connection pool implements this interface. Typically, a JDBC connection pool is implemented by third-party libraries or a JDBC driver itself. Generally, very few JDBC drivers implement a connection pool, so open source third-party JDBC resource pooling libraries such as Apache DBCP, c3p0, and BoneCP are popular. They also support validation queries for the eviction of stale connections that might result from network timeouts or firewalls and guard against connection leaks. Apache DBCP and BoneCP are accessible from Clojure via their respective Clojure wrapper libraries Clj-DBCP (`https://github.com/kumarshantanu/clj-dbcp`) and Clj-BoneCP (`https://github.com/opiskelijarekisteri-devel/clj-bonecp`), and there are Clojure examples that describe how to construct c3p0 pools.

Connections are not the only JDBC resources that need to be pooled. Every time we create a new JDBC prepared statement, depending on the JDBC driver implementation, often the entire statement template is sent to the database server in order to obtain a reference to the prepared statement. As database servers are generally deployed on separate hardware, there may be network latency involved. Hence, pooling of prepared statements is a very desirable property of JDBC resource pooling libraries. Apache DBCP, c3p0, and BoneCP support statement pooling, and the Clj-DBCP wrapper enables pooling of prepared statements out of the box for better performance.

I/O batching and throttling

It is well known that chatty I/O calls generally lead to poor performance. The solution is to batch together several messages and send them in one payload. In databases and network calls, batching is a common useful technique to improve throughput. On the other hand, large batch sizes may actually harm throughput as they tend to incur memory overhead and components may not be ready to handle a large batch at once. Hence, sizing the batches and throttling are just as important as batching. I would strongly advise conducting your own tests to determine the optimum batch size under representative load.

JDBC batch operations

JDBC has batch-update support in its API, which includes the INSERT, UPDATE, and DELETE statements. The Clojure Contrib library java.jdbc supports JDBC batch operations via its own API as shown in the following code snippet:

```
(require '[clojure.java.jdbc :as jdbc])

;; multiple SQL statements
(db-do-commands
  db true
  ["INSERT INTO emp (name, countrycode) VALUES ('John Smith', 3)"
   "UPDATE emp SET countrycode=4 WHERE empid=1379"])

;; similar statements with only different parameters
(db-do-prepared
  db true
  "UPDATE emp SET countrycode=? WHERE empid=?"
  [4 1642]
  [9 1186]
  [2 1437])
```

Besides batch updates, we can also batch JDBC queries. One of the most common techniques is to use the SQL WHERE clause to avoid the N+1 selects issue. The N+1 issue indicates the situation where we execute one query in another child table for every row in a row set from the master table. A similar technique can be used to combine several similar queries on the same table into just one query and then segregate the data in the program afterwards. Consider the following example that uses clojure.java.jdbc 0.3.0-alpha5 and a MySQL database:

```
(require '[clojure.java.jdbc :as j])

(def db {:subprotocol "mysql"
               :subname "//127.0.0.1:3306/clojure_test"
               :user "clojure_test" :password "clojure_test"})

;; the following snippet uses N+1 selects (typically characterized by
SELECT in a loop)
(def rq "select order_id from orders where status=?")
(def tq "select * from items where fk_order_id=?")
(doseq [order (j/query db [rq "pending"])]
  (let [items (j/query db [tq (:order_id order)])]
    ;; do something with items
    …))

;; the following snippet avoids N+1 selects, but requires fk_order_id
to be indexed
(def jq "select t.* from orders r, items t
  where t.fk_order_id=r.order_id and r.status=? order by t.fk_order_
id")
(let [all-items (group-by :fk_order_id (j/query db [jq "pending"]))]
  (doseq [[order-id items] all-items]
    ;; do something with items
    ...))
```

In the preceding example, there are two tables: `orders` and `items`. The first snippet reads all order IDs from the `orders` table then iterates through them to query corresponding entries in the `items` table in a loop. This is the N+1 selects performance antipattern that you should keep an eye on. The second snippet avoids the N+1 selects by issuing a single SQL query, but it may not perform very well unless the `fk_order_id` column is indexed.

Batch support at API level

When designing any service, it is very useful to provide an API for batch operations. This builds flexibility in the API so that batch sizing and throttling can be controlled in a fine-grained manner. Not surprisingly, it is also an effective recipe for building high performance services. A common overhead we encounter when implementing batch operations is the identification of each item in the batch and their correlation across requests and responses. The problem becomes more prominent when requests are asynchronous.

The solution to the item identification issue is resolved in one of the following manners:

- Assigning a canonical or global ID to each item in the request (batch)
- Assigning a unique ID to every request (batch) and an ID local to the batch for each item in the request

The choice of the exact solution usually depends on the implementation details. When requests are synchronous, you can do away with the identification of each request item. Look at the Facebook API for reference: `http://developers.facebook.com/docs/reference/api/batch/`). Here, the items in response follow the same order as in the request. However, in asynchronous requests, items may have to be tracked via status-check or callbacks. The desired tracking granularity typically guides the appropriate item identification strategy.

Throttling requests to services

As every service can handle only a certain capacity, the rate at which we send requests to a service is important. The expectations about the service behavior are generally in terms of both throughput and latency. This requires us to send requests at a specified rate, as a rate lower than this may lead to under-utilization of the service and a higher rate may overload the service or result in failure, thus leading to client side under-utilization.

Let us say a third-party service can accept 100 requests per second. However, we may not know how robustly the service is implemented. Though sometimes it is not exactly specified, sending 100 requests at once (within 20ms, let's say) during each second may lead to lower throughput than expected. Evenly distributing the requests across the one second duration, that is, sending one request every 10ms (1000ms/100 = 10ms), may increase the chance of attaining the optimum throughput.

Throttling at a very fine-grained level requires that we buffer the items so that we can maintain a uniform rate. Buffering consumes memory and often requires ordering; queues (discussed in *Chapter 5*, *Concurrency*), pipeline, and persistent storage usually serve that purpose well. Again, buffering and queuing may be subject to back pressure due to system constraints. We will discuss pipelines, back pressure, and buffering in a later section in this chapter.

Precomputing and caching

While processing data, we usually come across instances where a few common computation steps precede several kinds of subsequent steps. That is to say some amount of computation is common and the remaining is different. For high latency common computations (I/O to access the data and memory/CPU to process it), it makes a lot of sense to compute them once and then store them in a digest form. Then, the subsequent steps can simply use the digest data and proceed from that point onwards, thus resulting in reduced overall latency. This is also known as staging of semi-computed data, and it is a common technique to optimize processing of nontrivial amount of data.

Clojure has decent support for caching. The built-in `clojure.core/memoize` function perform basic caching of computed results with no flexibility in using specific caching strategies and pluggable backends. The Clojure Contrib library `core.memoize` (`https://github.com/clojure/core.memoize`) offsets the lack of flexibility in `memoize` by providing several configuration options. Interestingly, the features in `core.memoize` are also useful as a separate caching library, so the common portion is factored out as a Clojure Contrib library called `core.cache` (`https://github.com/clojure/core.cache`) on the top of which `core.memoize` is implemented.

As many applications are deployed on multiple servers for availability, scaling, and maintenance reasons, they need distributed caching that is fast and space efficient. The open source memcached project is a popular in-memory, distributed key-value /object store that can act as a caching server for web applications. It hashes the keys to identify the server to store the value on and has no out of the box replication or persistence. It is used to cache database query results, computation results, and so on. For Clojure, there is a memcached client library called SpyGlass (`https://github.com/clojurewerkz/spyglass`). Of course, memcached is not limited to just web applications and can be used for other purposes too.

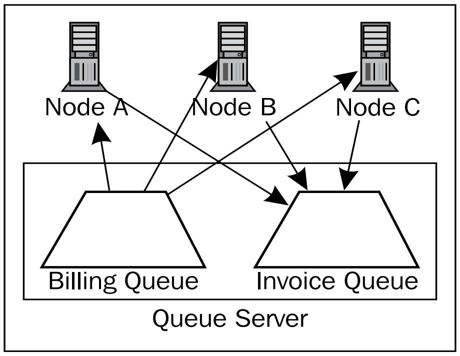
Concurrent pipelines

Imagine a situation where we have to carry out jobs at a certain throughput, such that each job includes the same sequence of a differently sized I/O task (task A), a memory-bound task (task B), and again an I/O task (task C). A naive approach would be to create a thread pool and run each job off it, but soon we realize that this is not optimum because we cannot ascertain the utilization of each I/O resource due to unpredictability of the threads being scheduled by the OS. We also observe that even though several concurrent jobs have similar I/O tasks, we are unable to batch them in our first approach.

As the next iteration, we split each job in to stages (A, B, and C) such that each stage corresponds to one task. Since the tasks are well known, we create one thread pool of appropriate size per stage and execute tasks in them. The result of task A is required by task B, and B's result is required by task C—we enable this communication via queues. Now, we can tune the thread pool size for each stage, batch the I/O tasks, and throttle them for an optimum throughput. This kind of arrangement is a concurrent pipeline. Some readers may find this feebly resembling the actor model or **Staged Event Driven Architecture** (**SEDA**) model, which are more refined models for this kind of approach. Recall that we discussed several kinds of in-process queues in *Chapter 5*, *Concurrency*.

Distributed pipelines

With this approach, it is possible to scale out the job execution to multiple hosts in a cluster using network queues, thereby offloading memory consumption, durability, and delivery to the queue infrastructure. For example, in a given scenario, there could be several nodes in a cluster, all of them running the same code and exchanging messages (requests and intermediate result data) via network queues. The following figure depicts how a simple invoice generation system might be connected to network queues:



RabbitMQ, HornetQ, ActiveMQ, Kestrel, and Kafka are some well-known open source queue systems. Once in a while, the jobs may require distributed state and coordination. The Avout (`http://avout.io/`) project implements the distributed version of Clojure's atom and ref, which can be used for this purpose. The Storm (`http://storm-project.net/`) project is a distributed, real-time stream processing system implemented partly using Clojure.

Applying back pressure

We discussed back pressure briefly in the previous chapter. Without back pressure, we cannot build a reasonable load-tolerant system with predictable stability and performance. In this section, we will look at how to apply back pressure in different scenarios in an application. At a fundamental level, we should have a threshold of the maximum number of concurrent jobs in the system, and based on that threshold, we should reject new requests above a certain arrival rate. The rejected messages may either be retried by the client or ignored if there is no control over the client. When applying back pressure to user-facing services, it may be useful to detect system load and at first deny auxiliary services in order to conserve capacity and degrade gracefully in the face of high load.

Thread pool queues

JVM thread pools are backed by a queue, which means that when we submit a job into a thread pool that already has the maximum number of jobs running, the new job lands in the queue. The queue is by default an unbounded queue, which is not suitable for applying back pressure. So, we have to create a thread pool backed by a bounded queue.

```
(import 'java.util.concurrent.LinkedBlockingDeque)
(import 'java.util.concurrent.TimeUnit)
(import 'java.util.concurrent.ThreadPoolExecutor)
(import 'java.util.concurrent.ThreadPoolExecutor$AbortPolicy)
(def tpool
  (let [q (LinkedBlockingDeque. 100)
        p (ThreadPoolExecutor$AbortPolicy.)]
    (ThreadPoolExecutor. 1 10 30 TimeUnit/SECONDS q p)))
```

Now, whenever there is an attempt on this pool to add more jobs than the capacity of the queue, it will throw an exception. The caller should treat the exception as a buffer-full condition and wait until the buffer has idle capacity again by periodically polling the `java.util.concurrent.BlockingQueue.remainingCapacity()` method.

Servlet containers like Tomcat and Jetty

In the synchronous Tomcat and Jetty versions, each HTTP request is given a dedicated thread from a common thread pool that a user can configure. The number of simultaneous requests being served is limited by the thread pool size. A common way to control the arrival rate is to set the thread pool size of the server. The Ring library uses an embedded Jetty server by default in the development mode. The embedded Jetty adapter (in Ring) can be programmatically configured with a thread pool size.

In the asynchronous (Async Servlet 3.0) versions of Tomcat and Jetty, besides the thread pool size, it is also possible to specify the timeout for processing each request. However, note that the thread pool size does not limit the number of requests in asynchronous versions in the way it does in synchronous versions. The request processing is transferred to an ExecutorService (thread pool) that may buffer requests until a thread is available. This buffering behavior is tricky because this may cause system overload—you can override the default behavior by defining your own thread pool instead of using the servlet container's thread pool to return an HTTP error at a certain threshold of waiting requests.

HTTP Kit

HTTP Kit (http://http-kit.org/) is a high performance, asynchronous web server for Clojure. It has built-in support for applying back pressure to requests over a specified queue length.

```
(require '[org.httpkit.server :as hk])

;; handler is a typical Ring handler
(hk/run-server handler {:port 3000 :queue-size 600})
```

In the preceding code snippet, the maximum queue length is specified as 600. When not specified, 20480 is the default maximum queue length for applying back pressure.

Performance and queuing theory

If we observe the performance benchmark numbers across a number of runs, even though the hardware, load, OS, and so on remain the same, the numbers are rarely exactly the same. The difference between each run may be upto as much as ±8 percent for no apparent reason. This may seem surprising, but the deep-rooted reason is that the performances of computer systems are *stochastic* by nature. There are many small factors in a computer system that make performance unpredictable at any given point of time. At best, the performance variations can be explained by a series of probabilities over random variables.

The basic premise is that each subsystem is more or less like a queue where requests await their turn to be served. The CPU has an instruction queue with unpredictable fetch/decode/branch-predict timings; the memory access again depends on the cache hit ratio and whether it needs to be dispatched via the interconnect, I/O subsystem works using interrupts that may again depend on mechanical factors of the I/O device. The OS schedules threads that wait while not executing. The software built on the top of all this basically waits in various queues to get the job done. These variations can be studied using queuing theory, something that interested readers may like to explore.

Little's Law

Little's law is a rather important theorem, which is commonly used to relate the mean number of jobs in any system with the mean time spent on each job. Little's law states the following:

$$mean\ number\ of\ jobs\ in\ a\ system = mean\ arrival\ rate \times mean\ response\ time$$

And also says that:

$$mean\ number\ of\ jobs\ in\ the\ queue = mean\ arrival\ rate \times mean\ waiting\ time$$

This is a rather important law that gives us an insight into the system capacity as it is independent of other factors.

For example, if the average time to satisfy a request is 200ms and the service rate is about 70 per second, then the mean number of requests being served is:

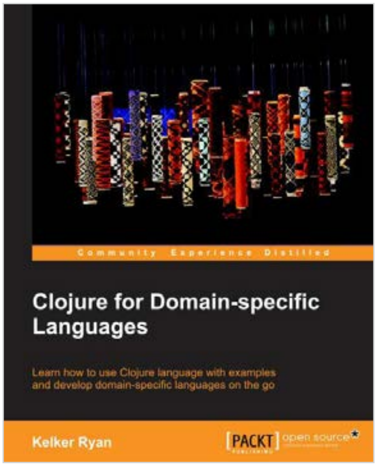
70 req/second x 0.2 second = 14 requests

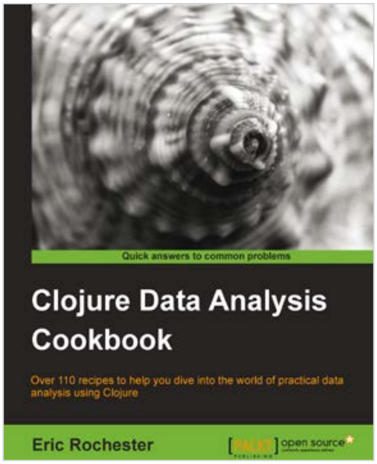
Summary

Designing an application for performance should be based on the use cases and patterns of anticipated system load and behavior. Measuring performance is extremely important to guide optimization in the process. Fortunately, there are several well-known optimization patterns to tap into, such as resource pooling, data sizing, prefetch and precompute, staging, and batching. As it turns out, application performance is not only a function of the use cases and patterns—the system as a whole is a continuous stochastic turn of events that can be assessed statistically and guided by probability.

Clojure is a fun language to do high performance programming. This book prescribes many pointers and practices for performance, but there is no mantra that can solve everything. The devil is in the details. Know the idioms and patterns, experiment to see what works for your applications, and learn which rules you can bend for performance.

Index


Community Experience Distilled
Clojure for Domain-specific Languages
Learn how to use Clojure language with examples and develop domain-specific languages on the go
Kelker Ryan
[PACKT] open source


Quick answers to common problems
Clojure Data Analysis Cookbook
Over 110 recipes to help you dive into the world of practical data analysis using Clojure
Eric Rochester
[PACKT] open source

[PACKT] open source
community experience distilled
PUBLISHING

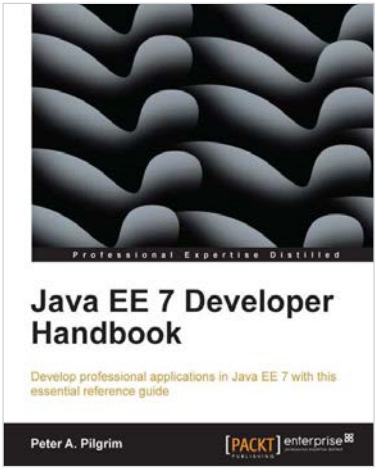
Professional Expertise Distilled
Open Text Metastorm ProVision® 6.2 Strategy Implementation
Create and implement a successful business strategy for improved performance throughout the whole enterprise
Foreword by
Ken Goerg - Director of EA Product Management at Open Text Inc
Ray Brown - Founder ClienteerHub.com
Tom Foster - Owner, Business Views LLC
Andrew Mackenzie - CEO, Shirlaws Australia
Jeremy Scrivens - Founder, TheEmotionalEconomy.com
Bill Aronson
[PACKT] enterprise

Open Text Metastorm ProVision® 6.2 Strategy Implementation

ISBN: 978-1-84968-252-7 Paperback: 260 pages

Create and implement a successful business statergy for improved performance throughout the whole enterprise

1. Fully understand the key benefits of implementing a business strategy
2. Utilize features like the integrated repository and ProVision® frameworks
3. Obtain real insights from practitioners in the field on the best strategic approaches


Professional Expertise Distilled
Java EE 7 Developer Handbook
Develop professional applications in Java EE 7 with this essential reference guide
Peter A. Pilgrim
[PACKT] enterprise

Java EE 7 Developer Handbook

ISBN: 978-1-84968-794-2 Paperback: 634 pages

Develop professional applications in Java EE 7 with this essential refrence guide

1. Learn about local and remote service endpoints, containers, architecture, synchronous and asynchronous invocations, and remote communications in a concise reference
2. Understand the architecture of the Java EE platform and then apply the new Java EE 7 enhancements to benefit your own business-critical applications

Please check **www.PacktPub.com** for information on our titles