АЛЕКСАНДР СЕНЬКО

РАБОТА С BIGDATA В ОБЛАКАХ

ОБРАБОТКА И ХРАНЕНИЕ ДАННЫХ
С ПРИМЕРАМИ ИЗ MICROSOFT AZURE

ББК 32.973.233-018
УДК 004.62
С31

Сенько А.

С31 Работа с BigData в облаках. Обработка и хранение данных с примерами из Microsoft Azure. — СПб.: Питер, 2019. — 448 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-0578-6

Перед вами — первая исходно русскоязычная книга, в которой на реальных примерах рассматриваются секреты обработки больших данных (Big Data) в облаках.

Основное внимание уделено решениям Microsoft Azure и AWS. Рассматриваются все этапы работы — получение данных, подготовленных для обработки в облаке, использование облачных хранилищ, облачных инструментов анализа данных.

Особое внимание уделено службам SAAS, продемонстрированы преимущества облачных технологий по сравнению с решениями, развернутыми на выделенных серверах или в виртуальных машинах.

Книга рассчитана на широкую аудиторию и послужит превосходным ресурсом для освоения Azure, Docker и других незаменимых технологий, без которых немыслим современный энтерпрайз.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.233-018
УДК 004.62

ISBN 978-5-4461-0578-6

Оглавление

Часть IV. Анализ BigData в облаке

Благодарности

Я хотел бы посвятить эту книгу моей семье: обожаемой жене Серафиме и любимому сыну Демьяну. Выражаю огромную признательность и благодарность своей неповторимой, замечательной и горячо любимой супруге. Ты очень поддерживала, помогала и вдохновляла меня. Спасибо тебе, Любимая!

Также выражаю благодарность всем тем, кто поддерживал меня и помогал мне: замечательной теще Инессе Витальевне, маме, папе, моему начальнику Мартину Ларссону и всем коллегам по работе. Выражаю отдельную благодарность специалисту проектной группы Олегу Сивченко и ведущему редактору Надежде Гринчик за терпение и неоценимую помощь.

Введение

...Мы просто расхаживаем по библиотеке, заполненной книгами на непонятном языке, и глазеем на цветные корешки... Вот и все!

Станислав Лем. Солярис

Человек, что бы он ни делал, почти никогда не знает, что именно он делает, во всяком случае, не знает до конца.

Станислав Лем. Сумма технологии

Книги, популяризующие нынешнее состояние знаний — скажем, знаний в области физики, — причем популяризующие хорошо, представляют дело так, будто существуют две четко отделенные друг от друга области: область того, что наукой уже раз и навсегда установлено, и того, что еще до конца не выяснено. Это похоже на посещение прекрасного, снизу доверху великолепно обставленного здания, его отдельных покоев, где то тут, то там лежат на столах нерешенные головоломки. Мы покидаем сей храм с уверенностью, что эти загадки рано или поздно будут решены, в чем убеждает нас великолепие всей постройки. У нас даже не мелькнет и мысли, что решение этих головоломок может привести к разрушению половины здания.

Станислав Лем. Сумма технологии

Любое время хорошо для пополнения знаний...

Жюль Верн. Дети капитана Гранта

Здравствуй, дорогой читатель!

В твоих руках книга о двух очень интересных направлениях современных Computer Science: облачных технологиях и больших данных. В последнее время оба этих направления получили широчайшее распространение благодаря новым интересным возможностям, которые предоставляются вместе с традиционными информационными системами, а также выгодам, получаемым конечным пользователем. Такое положение дел достигается за счет того, что технологии больших данных позволяют работать с огромными массивами неструктурированных и слабоструктурированных данных или с потоками данных, анализируя их и находя в них скрытые закономерности.

В качестве источников больших данных часто выступают события, совершаемые массово и фиксируемые в базе данных (БД) или в файлах. Например, это могут быть файлы логов высоконагруженного веб-сервера, игровые действия пользователя в массовой игре, хранящиеся в нереляционной базе, или данные, относящиеся к бизнес-процессам, хранящиеся в реляционном хранилище данных.

Прежде чем попасть в базу данных, все эти события на пути от источников к приемнику образуют поток сообщений. Если источников событий много и количество сообщений, отправляемых одиночным источником в единицу времени, велико, то возникает задача концентрации этих сообщений, то есть предоставления им общей входной точки («воронки» или «хаба») для последующего сохранения в той или иной базе данных. Согласование сообщений источников и хранилища, помимо их концентрации, может потребовать предварительной фильтрации (когда сохраняются только те сообщения, которые отвечают определенным критериям), маршрутизации (перенаправление сообщений различного типа в разные источники) и трансформации (например, выборка определенных полей сообщения, агрегация, арифметические преобразования и пр.). Выполнение всех этих действий относится к *потоковому анализу* больших данных. Помимо пассивного сохранения сообщений, очень часто может требоваться выполнение каких-либо действий в ответ на появление определенного сообщения или тренда в потоке. Пример такой задачи — обработка потока транзакций от банкомата в банковский процессинговый центр для установления факта мошеннических действий. Концептуально похожий пример — анализ потока сообщений в системе логирования в целях выявления хакерских атак (например, веб-приложений) или мошеннических действий (в онлайн-играх).

После того как в системе накопилось много данных за определенный промежуток времени, может потребоваться их анализ специалистом по анализу данных. Например, данные системы мониторинга потребления электроэнергии представляют интерес с точки зрения выявления как общей структуры энергопотребления, так и частных аномалий в виде поиска наиболее расточительных потребителей, узких мест. Подобный анализ должен производиться при непосредственном взаимодействии с исследователем данных (data scientist) и потому называется *интерактивным*.

Когда закономерности в данных нельзя обнаружить с помощью традиционных методов анализа, то есть путем выполнения стандартных действий (фильтрация, агрегирование, объединение, пересечение, сортировка), используются алгоритмы глубокого изучения данных, часто называемые *машинным обучением*. Суть его состоит в том, что для представления закономерностей, скрытых в данных, задействуют различные математические модели, и собственно «обучение» состоит в подборе изменяемых параметров этих моделей так, чтобы обеспечить наибольшее согласование между реальными результатами и результатами, полученными моделью.

Как только характер закономерностей в больших данных установлен на этапе интерактивного анализа, может возникнуть задача построения системы, выполняющей периодический анализ накопившихся данных — *пакетный анализ*. Самый простой пример — система мониторинга электроэнергии, в которой ежедневно формируются отчеты о суммарном потреблении каждым потребителем, распределяется потребление электроэнергии по часам, для каждого потребителя в отдельности, по потребителям и др. Результаты, полученные на этапе интерактивного анализа, могут быть использованы для выявления паттернов нерационального потребления электроэнергии и выдачи рекомендаций.

Помимо собственно данных, выдаваемых техническими устройствами того или иного вида, к большим данным можно отнести данные, генерируемые сложными социальными и техническими системами. В качестве примера можно привести систему общественного транспорта с регистрацией как прямых событий (включая данные мониторинга транспортных средств, оплаты пассажирами проезда), так и связанных с ними (например, событий, связанных с поломками подвижного состава и инфраструктурных элементов). Накопление и централизация данных в этом случае позволяет глобально оптимизировать транспортные потоки. Подобного результата можно достичь, например анализируя корреляции в загрузке транспорта и оперативно перераспределяя его между участками с неодновременными пиками.

Кроме того, анализ потоков поломок и установление корреляции его с внешними событиями (загруженность маршрута, износ транспортного средства, погодные условия, дорожная обстановка, стиль вождения и пр.) позволяет применить так называемое упреждающее обслуживание (predictive maintenance). Суть его заключается в том, чтобы на основе анализа исторических данных, касающихся отказов устройств и систем, выявить паттерны наступления отказов (допустим, перегрузка электросети при планируемом резком похолодании вечером в пятницу) и предотвратить их, заблаговременно приняв соответствующие меры. То есть в данном случае прогнозируется наступление отказа на основе анализа большого объема исторических данных и текущего состояния системы. Очевидно, что при правильном прогнозе своевременная диагностика и обслуживание позволят транспортным компаниям снизить математическое ожидание потерь, не прибегая при этом к существенному повышению стоимости обслуживания, поскольку его периодичность, по сути, не меняется, за исключением внепланового обслуживания, требующегося по результатам предсказания. И здесь снова возникает проблема сбора, хранения

и анализа большого количества событий (в данном случае это отказ, информация о местоположении транспортного средства в настоящий момент и т. п.).

В настоящей книге будет уделяться пристальное внимание облачным сервисам потоковой, интерактивной и пакетной обработки, а также сервисам хранения и копирования этих данных между различными источниками. Машинное обучение (возможно, несправедливо) исключено из материала книги ввиду его специфичности и необходимости дополнительного описания теории. Эта интереснейшая тема заслуживает отдельной книги.

При построении систем, оперирующих большими данными, возникает много технических проблем, связанных с хранением данных и их обработкой, которые сводятся к построению больших кластеров серверов, объединенных высокоскоростными и высокопроизводительными сетями передачи данных. С ростом масштаба больших данных (объемы хранения, ежесекундный поток данных и др.) требуются все более мощные вычислительные ресурсы. Как следствие, каждой организации, отвечающей за подобные системы, становится необходимо иметь свой центр обработки данных (ЦОД, датацентр), что влечет определенные трудности: нужно помещение ЦОД с системой поддержания микроклимата, системой электроснабжения, вентиляции, кондиционирования и т. д. Кроме того, требуется штат как высококвалифицированных системных администраторов разного профиля для обслуживания центра, так и энергетиков, специалистов по кондиционированию, вентиляции и пр.

А как быть с переменной нагрузкой на все имеющиеся серверы, например очень большой в одних ситуациях (возможно, 10 % времени), и совсем маленькой в других (остальные 90 %)? Масштабирование системы в подобных случаях сопряжено с необходимостью закупки серверов, расширения площадей хранения, расширения полосы пропускания сети, подвода новых мощностей от электросетей и т. д. И это решение совершенно негибкое, то есть слабо нагруженные серверы все равно будут включены и задействованы с минимальной нагрузкой или просто будут занимать место в стойках ЦОДа. Приведу другой пример: допустим, данные нужно обработать разово, но быстро, что требует использования больших вычислительных ресурсов. Например, необходимо проанализировать логи веб-сервера, чтобы определить посещаемость за большой промежуток времени. Если лог-файл огромен, а анализ надо произвести разово, то как быть с вычислительными мощностями? Закупить, а потом продать?

В подобных ситуациях на помощь приходят облака. По сути, это сети дата-центров, по требованию предоставляющие их пользователям вычислительные ресурсы в аренду. Масштабы ресурсов могут значительно разниться: от «маленьких» виртуальных машин с оперативной памятью 0,25 Гбайт и одним низкопроизводительным ядром до кластеров из многих сотен виртуальных машин. Кроме того, при использовании облака легко решается проблема масштабирования ресурсов — у облачных провайдеров выделение и освобождение ресурсов происходит динамически и занимает минуты, так что можно создать автоматически масштабируемые архитектуры. Однако виртуальные машины, пусть и размещенные в облаках, все

равно требуют штата системных администраторов, ведь необходимо обслуживать их операционные системы, собирать метрики, обеспечивать безопасность, надежность, доступность...

Чтобы помочь справиться со всеми этими задачами, облачные провайдеры предоставляют целый арсенал сервисов, упрощающих хранение, анализ и визуализацию больших данных. Итак, обе технологии — большие данные и облачные среды — дополняют и обогащают друг друга, создавая симбиотическую среду для анализа и обработки огромных массивов информации.

Из приведенного описания может сложиться впечатление, что большие данные, да еще и в виде размещенной в облаках системы, имеют отношение только к «большим» проектам, но это далеко не так. Задачи, подобные анализу файлов логов веб-сервера, онлайн-обработке событий из мобильных приложений или сети технических устройств, тоже можно решить с помощью систем больших данных. Однако на сей раз решающий фактор — высокая скорость обработки относительно небольших объемов данных, быстрота и простота построения системы анализа. Это возможно благодаря тому, что облачные среды предоставляют сервисы, значительно упрощающие работу с большими данными.

Настоящая книга описывает существующие облачные сервисы и облачные архитектуры, предназначенные для обработки данных, на примере Microsoft Azure и Amazon Web Services (AWS). Другие популярные и интересные облачные платформы (IBM, Google Cloud и др.) не рассмотрены ввиду ограниченности объема книги и конечности сроков ее написания. Кроме того, не затрагиваются вопросы машинного обучения и визуализации данных.

Вы можете недоуменно спросить, что же есть в этой книге? Отвечаю: описание облачных сервисов хранения данных разных типов, сервисов онлайн- и пакетного анализа данных, концентраторов сообщений и коннекторов для доставки данных в облако. Помимо того, представлены Hadoop as a Service (aaS), средства копирования и трансформации данных, архитектуры систем, описано применение подхода Event Driven Design. Примером практического использования этого подхода в книге служит построение сервиса онлайн-покера. Обе технологии, описываемые в книге (облака и большие данные) развиваются очень быстро, и потому крайне тяжело написать о них книгу, которая будет актуальной через год, два, три... Как следствие, я постарался сделать упор на универсальные принципы, идеи и концепции, которые не скоро утратят актуальность. Но книга отнюдь не теоретическая, в ней достаточно кода, примеров из моей практики и рекомендаций по применению той или иной технологии. Особенностью книги является то, что основной упор в ней сделан на архитектуре и возможностях систем, построенных на основе современных сервисов, представленных в облачных средах. Очень широко и всесторонне рассматриваются сервисы и концепции, относящиеся к концепции Event Driven Design. Собственно наука анализа данных затронута лишь в том объеме, который необходим для понимания того, как работает тот или иной сервис, поскольку это тема отдельной книги. Кроме того, разнообразные алгоритмы из арсенала науки анализа данных мало влияют на вид облачной архитектуры конечной системы.

Я надеюсь, что после прочтения книги вы будете ясно понимать принципы работы общих архитектур и существующих облачных сервисов, получите четкое представление о том, как создавать свое приложение на их основе. Знакомство с разнообразием сервисов и технологий доставило мне огромное удовольствие, и я выражаю надежду на то, что вы испытаете это чувство хотя бы частично. Кроме того, еще раз обращаю внимание: как большие данные, так и облачные среды сейчас являются бурно развивающимися областями, и новые сервисы появляются очень быстро. Например, пока я писал книгу, и у Azure, и у AWS появилось порядка полудюжины новых сервисов хранения и анализа данных, а у части существующих поменялся пользовательский интерфейс. (Возьмем сервис Azure Data Factory. Когда я только обсуждал концепцию книги с издательством, интерфейс был убран. А за пару недель до крайнего срока сдачи книги он опять появился! Это повлекло срочный пересмотр и переписывание соответствующей главы, что потребовало от меня много усилий и нервов.) Однако не стоит думать, что книга неактуальна или скоро перестанет быть таковой. Все описанные в ней сервисы будут существовать еще долго, а концепции и архитектуры помогут вам выполнить самостоятельный анализ и принять решение о создании систем в других облачных средах или на собственном физическом оборудовании.

Часть I
Общие вопросы и понятия

1 Что такое облако

Хотите —
буду от мяса бешеный
— и, как небо, меняя тона —
хотите —
буду безукоризненно нежный,
не мужчина, а — облако в штанах!

Владимир Маяковский.
Облако в штанах

1.1. Общие сведения

Облачные технологии появились совсем недавно: в 2006 году один из крупнейших американских интернет-магазинов Amazon предоставил свои неиспользуемые вычислительные ресурсы (а к тому времени их объем стал огромным) совершенно новым образом. Традиционно для аренды ресурсов в дата-центрах необходимо было составить договор и внести плату за определенный срок. Линейка типоразмеров серверов (объем оперативной памяти, количество ядер, размер дискового пространства и др.) достаточно обширна и выбирается заранее, до подписания договора. Можно арендовать много серверов, связать их высокопроизводительной сетью, подключить балансировщик нагрузки и получить систему, обрабатывающую большую нагрузку. У подобной модели использования ресурсов есть существенные неудобства. При создании приложений зачастую неизвестно, какая потребуется нагрузка, на какой срок арендовать серверы приложения. Или такой пример: создается стартап, арендуются серверы и до завершения срока аренды этот стартап «умирает». Что делать со ставшими ненужными арендованными серверами? Еще сложнее дело обстоит с покупкой физических серверов. Ведь их, помимо администрирования операционной системы и установленных приложений, необходимо обслуживать физически. Сюда входит подбор помещения, электропитания, системы охлаждения, вентиляции... Все эти проблемы можно решить с помощью эластичных вычислительных ресурсов, предоставляемых облачными провайдера-

ми. (Платформа Amazon Web Services называет эти ресурсы EC2 — Elastic Cloud Computers.) Ключевые преимущества облачной модели таковы:

- ресурсы предоставляются по требованию и таким же образом освобождаются;
- плата начисляется за фактическое время использования ресурсов;
- предоставление и освобождение ресурсов производится самим потребителем ресурсов через веб-портал, без всякой бумажной волокиты с договорами.

Помимо виртуальных машин, в облачных средах предоставляются различные сервисы, позволяющие строить различные архитектуры: сервисы виртуальных сетей, подсетей, балансировщики нагрузки, списки контроля доступа (Access Control Lists, ACL)), выделенные IP-адреса и др. Эти сервисы составляют основу инфраструктуры как сервиса (Infrastructure as a Service, IaaS). Конечно, *прямая* стоимость годовой аренды физического сервера может быть меньше, чем стоимость аренды облачного сервера с почасовой оплатой с такими же характеристиками, но многие облачные провайдеры (например, AWS) предоставляют возможность долгосрочной аренды виртуальных машин по ценам существенно меньшим, чем при почасовой оплате. Если же серверы требуются на небольшое время и заранее не известно, какого размера должна быть виртуальная машина, то эластичные виртуальные машины могут стать единственным приемлемым выбором. В случае же прямой покупки физических серверов задача выбора, приобретения, настройки и обслуживания, а также их продажи после применения становится весьма непростой. Чтобы обеспечить возможность выделения пользователям ресурсов, облачные провайдеры имеют крупные, географически разнесенные дата-центры, веб-порталы для получения доступа к их ресурсам, а также API для программного доступа. Это позволяет сделать то, что нельзя выполнить с помощью любой другой традиционной технологии: код программы может сам себе выделять столько ресурсов, сколько ему нужно. Или же программы могут создавать инфраструктуру, на которой они будут выполняться.

Помимо «голой» инфраструктуры, облачные провайдеры предоставляют наиболее типовые приложения в виде веб-сервисов. В качестве примера можно привести облачное хранилище данных (cloud storage), сервис предоставления учетных записей (identity provider), сервис хостинга веб-приложений, базу данных как сервис, брокер сообщений, концентратор сообщений и др. Все эти сервисы, кажущиеся на первый взгляд разрозненным набором, предоставляются как общая платформа. Доступ к ним унифицируется в виде единообразных API, SDK, возможны их «соединение» между собой, общий мониторинг логов и событий и пр. Это иной уровень применения ресурсов облака: платформа как сервис (Platform as a Service). PaaS позволяет пользователям создавать не просто программные продукты в рамках одной операционной системы, веб-платформы и др., но целые информационные системы, компонентами которых будут экземпляры облачных сервисов. Подобно IaaS, сервисы PaaS обычно допускают масштабирование (как ручное, путем выбора соответствующего их размера, так и автоматическое, с помощью различных метрик и событий). Как правило, сервисы PaaS предоставляют

гораздо меньшие права для доступа к вычислительным ресурсам инфраструктуры, лежащей в их основе. Например, сервисы хостинга веб-приложений не позволяют установить специфические программы, COM-компоненты, поменять библиотеку DLL в GAC, изменить запись в реестре и др., поскольку отсутствует root-доступ. Но взамен они предоставляют удобные порталы администрирования, интеграцию с другими сервисами, встроенные средства логирования и мониторинга, доступность 99,99 % времени и др.

В настоящее время крупнейшими облачными провайдерами являются Amazon Web Services (AWS), Microsoft Azure, Google Cloud, IBM Bluemix, Oracle. В книге приведены описания сервисов двух облачных провайдеров: AWS и Microsoft Azure. AWS — первый в истории облачный провайдер, а Microsoft Azure — облачный провайдер от корпорации Microsoft, обеспечивающий интеграцию практически со всеми сервисами Microsoft.

1.2. Способы создания ресурсов в облаке

> В каюте первого класса Остап, лежа с башмаками на кожаном диване и задумчиво глядя на пробочный пояс, обтянутый зеленой парусиной, допрашивал Ипполита Матвеевича:
>
> — Вы умеете рисовать? Очень жалко. Я, к сожалению, тоже не умею.
>
> Он подумал и продолжал:
>
> — А буквы вы умеете рисовать? Тоже не умеете? Совсем нехорошо! Ведь мы-то попали сюда как художники. Ну, дня два можно будет мотать, а потом выкинут.
>
> *Ильф и Петров. Двенадцать стульев*

Прежде чем начать описывать способы создания ресурсов, поясню, что это такое. Как отмечалось выше, облачные провайдеры имеют в основе своих сервисов огромные дата-центры, чьи вычислительные ресурсы с помощью системы виртуализации разделяются на небольшие части: голые виртуальные машины различных размеров с установленной операционной системой (IaaS) и группы виртуальных машин с установленным софтом, предоставляющим доступ только к своим возможностям (PaaS). Так вот, создать облачный ресурс — значит отправить запрос контроллеру ресурсов, размещенному в облачном ЦОДе, на выделение требуемых вычислительных ресурсов из пула доступных. То есть, по сути, ресурс не создается из ничего, а только выделяется по требованию. И тут

возможна ситуация (редко, но бывает), когда пользователь запросил ресурсы у контроллера, а они не появились. Это случается из-за того, что физические ресурсы, на которых размещаются виртуальные, уже задействованы другими пользователями. Задачу оптимального распределения доступных ресурсов между пользователями целиком решает контроллер. И если пользователь при создании ресурсов столкнулся с проблемой, он должен повторить попытку, прибегнув к различным вариациям (повторить через некоторое время, сменить регион и повторить, сменить аккаунт и повторить и пр.).

С точки зрения пользователя, выделение ресурсов выглядит как создание ресурсов: он выполнил ряд действий на веб-портале, и в последнем появились ресурсы. На самом деле, конечно же, они были выделены, и об этом не стоит забывать. Однако для простоты и наглядности я буду применять термин «создание».

Существует четыре способа управления облачной инфраструктурой (рис. 1.1). Первый, самый простой и очевидный — задействовать веб-портал. При этом пользователь должен иметь соответствующие права на создание ресурсов. Ручной способ очень прост: у всех облачных провайдеров есть удобные порталы, обширная документация, видеоинструкции и др. Не нужны никакие дополнительные сервисы, SDK и пр.

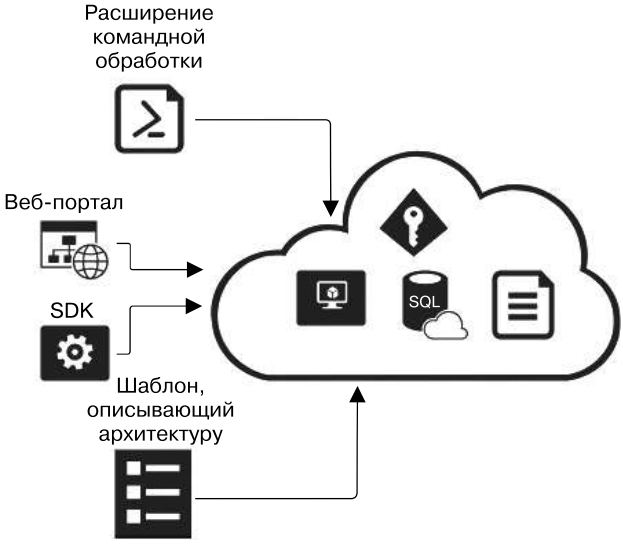


Рис. 1.1. Способы управления облачной инфраструктурой

Однако у данного способа есть недостатки:

- длительное время создания инфраструктуры;
- недостаточная надежность (в случае проблем с ресурсами их придется пересоздавать вручную, со всеми ручными настройками, конфигурированием и пр.);

- трудность переноса инфраструктуры в новый регион или аккаунт — ее понадобится вручную клонировать или копировать (данный недостаток частично сглаживается тем, что облачные провайдеры позволяют копировать или клонировать ресурсы, но эта процедура все равно требует ручного инициирования);
- процесс создания ресурсов в этом случае невозможно автоматизировать.

Второй способ — применить программные библиотеки (Software Development Kit, SDK), обеспечивающие доступ к ресурсам облака из кода пользовательских программ. Как правило, SDK представляет собой набор классов и методов, облегчающих программные операции с ресурсами облака. Чтобы обеспечить доступ к таким ресурсам, программа с облачным SDK должна содержать ключи учетной записи, которая будет иметь доступ к облаку. В облаке эти ключи зарегистрированы в виде пользователя в активном каталоге облачного аккаунта, обладающего правами выполнять программное манипулирование ресурсами облака (такой пользователь называется принципалом — service principal). И управление этими учетными записями происходит таким же образом, как и учетными записями пользователей облачного веб-портала. В числе достоинства такого подхода — возможность создания программ, которые сами себе создают облачные ресурсы, а также автоматизированного управления облачным аккаунтом.

К третьему способу создания облачных ресурсов относят специализированные расширения для языков командной строки — shell, CMD и др. (например Azure PowerShell, AWS CLI и пр.), работающие в ней напрямую. Для подключения этих расширений к облачным ресурсам необходимо импортировать ключи или выполнить вход в аккаунт через форму ввода логина/пароля. Как и в случае SDK для сценарных языков программирования, SDK для командной оболочки позволяет описывать облачную инфраструктуру в виде набора команд, каждая из которых создает или конфигурирует соответствующий облачный сервис.

И SDK, и команды оболочки оперируют в конечном итоге с API облачного провайдера (как правило, REST API), доступ к которым также позволит манипулировать ресурсами облака.

Четвертый способ создания облачных ресурсов — применить шаблоны. В этом случае все требуемые ресурсы и связи между ними описываются с помощью текстового файла в формате YAML или JSON. Такой шаблон может быть загружен в соответствующий облачный сервис напрямую через веб-портал или через CLI-команды.

Описание инфраструктуры через шаблон — очень мощный механизм, широко применяемый для конфигурирования различных инфраструктур (например, в системах Ansible, Chef, Puppet и др.). Как уже указывалось, шаблоны представляют собой текстовые файлы, которые могут храниться в репозитории шаблонов или чаще всего в репозитории GitHub. Для облака AWS сервис создания ресурсов с помощью шаблонов называется CloudFormation (поддерживает YAML и JSON), у Azure это ARM Template (в настоящее время поддерживает только JSON).

На веб-портале AWS имеется специальный редактор, упрощающий создание и конфигурирование шаблона. Последний может быть загружен в файловое хранилище S3, репозиторий CodeCommit или любое другое место, доступное для сервиса CloudFormation. Этот сервис создает стек — набор ресурсов, управляемых совместно (создание, удаление и обновление).

Сервис CloudFormation очень удобен в применении со сторонними сервисами конфигурирования — например, Ansible. Это широко используемое приложение, задействующее YAML для создания конфигурационных шаблонов, которые служат для администрирования группы серверов (преимущественно Linux, но есть расширения и для Windows), не требуя инсталляции на этих серверах «агентов». Для работы Ansible необходимы только ключи доступа к ресурсам (SSH-ключи для Linux-хостов, сертификат для PowerShell-доступа к Windows-хостам или ключи доступа к AWS). Шаблон CloudFormation для Ansible представлен в виде JINJA, допускающего передачу параметров через переменные Ansible.

1.3. Безопасность облачных ресурсов

> В мире существует нежелательный парадокс: чем больше власти, тем меньше ответственности.
>
> *Валентин Пикуль. Битва железных канцлеров*

> Воруют так, что печку раскаленную нельзя без присмотра оставить.
>
> Отвернись только — и печку голыми руками вынесут...
>
> *Валентин Пикуль. На задворках великой империи*

Наряду с неоспоримыми преимуществами, хранение и обработка данных в облачных средах потенциально может доставить ряд проблем, которых нет (или, вернее, они проявляются не так отчетливо) в случае размещения и обработки данных в собственных дата-центрах. Это обусловлено рядом причин. Во-первых, облачные среды сами по себе публично доступны и все сервисы, если явно не сконфигурировано иное, доступны для всех в Интернете. Во-вторых, защита данных и инфраструктуры от непреднамеренных действий пользователей лежит вне компетенции облачного провайдера. Кроме того, облачные инфраструктуры, работающие с большими данными, часто содержат в своем составе большие кластеры виртуальных машин, что требует применения специальных мер для обеспечения надежной работы всей системы. Помимо этого, информация физически будет передаваться

по незащищенным каналам и существует угроза ее перехвата. Рассмотрим подробнее все перечисленные и некоторые другие аспекты безопасности облачных сред.

Наиболее распространенный способ защиты конечных точек облачных сервисов — ограничение доступа к ним с помощью механизмов аутентификации и создания списков разрешенных IP-адресов, с которых можно получить доступ к точкам. Рассмотрим прежде всего различные способы обеспечения доступа из заданного адресного пространства.

Сервисы, относящиеся к IaaS, а также в ряде случаев к PaaS, требуют для своего создания сконфигурированной облачной виртуальной частной сети (VNet, VPC), разбитой на подсети. Доступ к конечным точкам сервисов, расположенным в этих подсетях, можно регулировать с помощью конфигурирования сетевых групп безопасности (Network Security Group, NSG) (рис. 1.2), которые представляют собой списки контроля доступа, ACL.

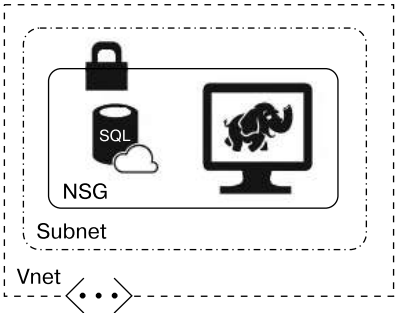


Рис. 1.2. Ограничение доступа к конечным точкам облачных сервисов с помощью сетевых групп безопасности

Итак, виртуальная часть сети — один из базовых сервисов IaaS. Он представляет собой облачный аналог локальной сети и служит для предоставления диапазона IP-адресов для размещения в них ресурсов. Виртуальную частную сеть можно разделить на подсети (subnet), а между ними — установить правила маршрутизации IP-пакетов. Кроме того, на подсети можно установить списки контроля доступа, которые именуются сетевыми группами безопасности. Это позволяет логически разделять архитектуры информационных систем на различные уровни (например, уровень данных, бизнес-логики, фронтенд) путем размещения каждого уровня в своей подсети и установления правил маршрутизации.

NSG представляет собой список доступа, содержащий набор записей. Каждая запись состоит из таких элементов, как:

- название;
- число, определяющее приоритет просмотра списка записей;
- диапазон IP-адресов (для одного конкретного адреса это /32);
- номер порта;

- действие — ALLOW или DENY («Позволить» или «Отклонить») по отношению к запросу, поступившему с данного адреса.

Кроме того, указывается протокол, к которому применимо действие ALLOW или DENY (TCP, UDP, ICMP и пр.). Безопасность конечных точек в данном случае обеспечивается ограничением к ним доступа извне. Помимо NSG, ряд облачных сервисов, не требующих виртуальной частной сети (например, Azure SQL), имеют фаерволы — списки «разрешенных» и «запрещенных» диапазонов. Хорошей практикой является повсеместное использование NSG и фаерволов. При этом необходимо, чтобы все порты, относящиеся к удаленному доступу/управлению (например, 22 для SSH, 3388 для RDP) или непосредственно к сервису (скажем, 1433 для MS SQL), были недоступны из Интернета вне диапазона адресов виртуальной частной сети. Для получения же доступа к сервисам из «разрешенной» локальной сети или с разрешенного компьютера следует установить VPN-шлюз из локальной сети или с компьютера в виртуальную частную сеть или непосредственно к экземпляру сервиса. Помимо шлюза, при соединении локальной сети с виртуальной частной сетью необходимо применить промежуточный хост, прокси-хост (рис. 1.3), который будет транслировать запросы и разрешать частные адреса облачных ресурсов из локальной сети. Прокси-хост в различных реализациях можно разместить как в облачной сети, так и в локальной. В последней может располагаться контроллер домена, сервер БД с данными, которые не могут быть размещены в облаке, а также прочие серверы, которые могут быть размещены только в локальной сети.

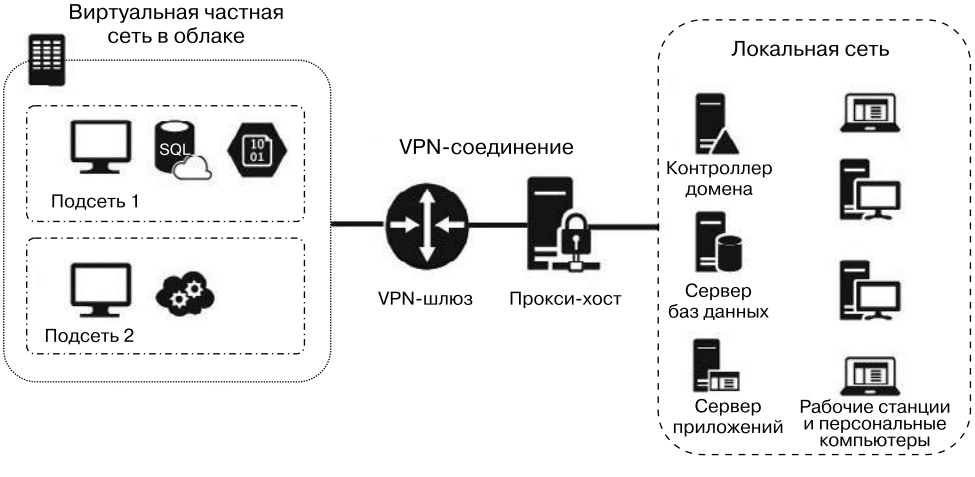


Рис. 1.3. Обеспечение защищенного доступа к облачным ресурсам

Некоторые конечные точки (скажем, API управления самого облачного аккаунта) не оборудованы ни фаерволом, ни NSG. Для их защиты используется как шифрование трафика (HTTPS), так и специальные ключи доступа к ресурсам. Ключи могут генерироваться непосредственно защищаемым сервисом и применяться

в заголовках REST-запросов. Например, сервис хранилища Azure Storage задействует аутентификацию на основе HMAC — Hash-based Message Authentication Code, который должен содержать подписанный алгоритмом SHA-256 токен как заголовок аутентификации. Следующий стандартный механизм аутентификации запросов — применение протокола OAuth 2.0, при котором пользовательские учетные данные хранятся в сервисе хранения учетных данных (в случае Azure это Azure Active Directory, а AWS — Cognito). В общем случае облачный сервис хранения учетных данных включает в себя пользователей, роли и API-ресурсы, защищаемые этим протоколом. Любой запрос к REST API ресурсов, зарегистрированным в этом сервисе (а это, по сути, все REST API конечных точек управления облачными ресурсами), должен в своем заголовке содержать токен, который можно получить, если выполнить процедуру ввода учетных данных в каталог.

Еще один «эшелон» защиты данных в облачных ресурсах — это их шифрование. Распространенный подход в данном случае — так называемое прозрачное шифрование данных (transparent data encryption, TDE). Суть его состоит в том, что все шифрование и дешифрование данных происходит «за кулисами», без участия пользователя, с помощью ключей шифрования, которые генерируются и хранятся самим облачным аккаунтом. В случае Azure эти ключи хранятся в Azure KeyVault, а в случае AWS — в AWS KMS.

Следующее звено защиты — сервисы, отвечающие за мониторинг запросов, поступающих к ресурсам и осуществляющие аудит пользовательской активности. Например, Azure Security Center, который может выполнять мониторинг очень многих ресурсов, выдает рекомендации по их защите и следит за входящим трафиком. На практике в реальной рабочей системе подобный сервис позволил обнаружить и успешно отразить несколько крупных хакерских атак на систему, за которую я отвечал, в том числе с использованием распределенной сети зараженных серверов (ботнет). Поэтому я настоятельно рекомендую применять подобные сервисы для обязательной защиты облачных ресурсов.

И последняя линия защиты данных, встроенная в облачные ресурсы, включает в себя сервисы анализа пользовательской активности, например Audit & Threat Detection для Azure SQL. Этот сервис обеспечивает внутренний аудит всех запросов в базе данных Azure SQL и анализ их на предмет подозрительных действий. У данного вида защиты есть один недостаток: в результате использования в реальной системе при включении максимального уровня аудита и защиты существенно снижается отзывчивость и общая производительность.

1.4. Резюме

В данной главе мы рассмотрели пути создания облачных ресурсов и уровни обеспечения безопасности. Эти ресурсы можно создавать с помощью веб-портала, через SDK (или напрямую с использованием REST API), расширения для интерфейса командной строки и, наконец, применив специальный шаблон, описываю-

щий требуемое состояние облачных ресурсов. Шаблоны автоматизации наиболее удобны при построении больших и сложных систем, поскольку обеспечивают полный контроль над ресурсами и не допускают неконтролируемого разрастания неиспользованных ресурсов и, соответственно, увеличения месячного счета за использованные ресурсы.

Вопросы безопасности информации в облаках являются комплексными. Пользователь облачных ресурсов сам отвечает за защиту своих ресурсов от несанкционированного доступа и должен самостоятельно выбирать для этого стратегию.

2 Что такое BigData

Поэзия —
та же добыча радия,
в грамм добыча,
в годы труды.
Изводишь
единого слова ради
Тысячи тонн
словесной руды.

Владимир Маяковский

Следует избегать бесполезных знаний.

Станислав Лем. Магелланово облако

Современное состояние Computer Science характеризуется тем, что, помимо естественных данных — результатов научных наблюдений, метеорологических данных, социологических и др., — появляется огромное количество данных, связанных с работой информационных систем. Эти новые данные существенно отличаются от тех, что анализировались на заре компьютерной эры. Те старые данные (их можно условно назвать естественно-научными) в основном требовали математической обработки.

В отличие от них данные современных информационных систем (большие данные) не могут быть представлены простыми математическими моделями, чьи параметры следует определить. Кроме того, эти данные отличаются существенной неоднородностью, разнообразной и непредсказуемой структурой, и зачастую непонятно, как их обрабатывать и нужно ли это вообще? Можно ли в них найти что-либо полезное? Этих данных настолько много, что их анализ за разумное время требует вычислительных ресурсов, существенно превышающих вычислительные

ресурсы самой информационной системы. Это значит, что данные часто лежат мертвым грузом, несмотря на скрытые в них закономерности, составляющие полезную информацию, которую требуется найти. Поиск таких закономерностей называется Data Mining — добывание данных из груды пустых данных (по аналогии с пустой породой). Что значит «обрабатывать» данные, и как их добывать? Для ответа на все приведенные вопросы необходимо сначала выяснить, откуда берутся эти большие данные. Их источниками могут быть:

- социальные сети — посты, комментарии, сообщения между пользователями и пр.;
- события, связанные с действиями пользователей в веб- или мобильных приложениях;
- логи приложений;
- телеметрия сети устройств из мира «Интернета вещей» (Internet of Things, IoT);
- потоки событий крупных веб-приложений;
- потоки транзакций банковских платежей с метаданными (время, место платежа и т. д.).

Все эти данные должны быть обработаны в режиме реального времени или же постфактум. В обоих случаях они могут размещаться в различных хранилищах (как общего назначения, так и специализированных) и в разных форматах: CVS, XML, JSON, таблицы в реляционных БД, базах данных NoSQL и пр.

Для пакетной обработки исторических данных различных форматов, расположенных в разных хранилищах, необходим единый подход, обеспечивающий выполнение запросов к данным, хранящимся в указанных выше форматах. В настоящее время распространены следующие подходы.

1. Преобразование данных из различных форматов в общий, допускающий выполнение запросов к единообразным данным. Это можно сделать с помощью облачных сервисов трансформации и копирования, таких как Azure Data Factory и AWS Glue, которые консолидируют данные из разных источников в один. Такое хранилище традиционно называется Data Warehouse (DWH, «склад данных»), а преобразование данных — ETL (Extract Transform Load — «извлечение, преобразование, загрузка»). Данный подход достаточно распространен в традиционных системах, в которых DWH строится на основе кластера SQL-серверов. Подход позволяет использовать все элементы синтаксиса SQL.
2. Складирование данных в единое хранилище без изменения формата. При этом форматы данных остаются прежними (JSON, XML, CSV и т. д). Такое хранилище, в котором данные размещаются в виде несвязанного набора данных, называется Data Lake («озеро данных»). Файловая система, лежащая в основе подобных хранилищ, совместима с HDFS — распределенной файловой системой,

которая, в свою очередь, совместима с Hadoop (Azure Data Lake, AWS EMRFS). Такое хранилище позволяет задействовать сервисы из экосистемы Hadoop (например, Hive, Apache Spark и др.) и применять иной подход к операциям с данными: ELT (Extract Load Transform — «извлечение, загрузка, преобразование»), когда данные можно трансформировать после загрузки. При этом используется сервер аналитики или кластер серверов, содержащий процессор специализированного языка запросов, в котором все разнородные источники данных представляются как внешние источники данных, к которым применим SQL-подобный синтаксис. Для подобного хранилища также может использоваться подход MapReduce (будет более подробно описан далее) и обработка данных в оперативной памяти (in memory processing).

3. Кроме того, для обработки потоковых данных существуют специализированные сервисы, допускающие обработку потока сообщений с помощью SQL-подобного синтаксиса (например, сервисы Azure Stream Analytics, AWS Kinesis Analytics) или программных структур (Apache Spark Streaming), а также сервисы для приема и концентрации этих сообщений (например, Azure Event Hub, Kafka, Azure Spark и пр.).

Итак, что же такое большие данные? Прежде всего, это огромные массивы данных или потоки, которые содержат подлежащую извлечению информацию, или же умеренно большие объемы данных, требующие быстрой интерактивной обработки с целью исследования, проверки гипотез, тренировки алгоритмов машинного обучения. Для выполнения этой обработки необходим высокий уровень параллелизма, большой объем оперативной памяти (для in memory processing), что достигается применением кластеров виртуальных машин. Вот тут-то и проявляются все преимущества облачных сред: модель IaaS позволяет создавать кластеры и удалять их по требованию с минимальными затратами. Виртуальные серверы создаются и задействуются только в течение того промежутка времени, когда они нужны, и, соответственно, плата за них взимается только во время прямого использования. Но просто создание кластера для обработки больших данных с последующей установкой требуемых программ и их настройкой — весьма трудоемкое занятие. Кроме того, облачные провайдеры предоставляют отдельные сервисы PaaS больших данных.

2.1. Обработка больших данных

А что значит «обработать большие данные»? Поясню на трех возможных видах анализа: пакетном, интерактивном и потоковом.

Большие данные могут быть обработаны в *пакетном* режиме, когда они уже присутствуют в хранилище. Чаще всего это необходимо для агрегирования данных и построения аналитических отчетов на их основе.

Рассмотрим в качестве примера систему мониторинга электроэнергии сети зданий. В этой системе замеры потребляемой мощности передаются с малой периодичностью как сообщения от каждого измерительного модуля. Чтобы получить величину дневного потребления электроэнергии, необходимо сложить все замеры с измерительного модуля каждого пользователя в отдельности. Если итоговый результат нужно, к примеру, формировать в виде ежедневного (еженедельного, ежемесячного и пр.) отчета, то наиболее просто реализовать такую систему следующим образом (рис. 2.1).

Рис. 2.1. Архитектура системы учета электроэнергии, построенная на основе разделения хранилищ сырых и агрегированных данных — так называемая лямбда-архитектура

Все сообщения от измерительных устройств можно хранить в нереляционном хранилище табличного типа (этот вопрос подробнее рассматривается в части II), например HBase или Cassandra. Каждая строка таблицы будет содержать временну́ю метку (то есть время поступления или отправления сообщения), идентификатор устройства, его отправившего, и собственно величину замера.

Для построения периодического отчета с помощью системы бизнес-аналитики (business intelligence, BI) (например, PowerBI или Microsoft SSRS) или отображения этой величины в браузере необходимо, чтобы данные в агрегированном виде были размещены в БД SQL. А почему нельзя сразу размещать их непосредственно в этой базе? Посчитаем. Предположим, что измерительное устройство отсылает сообщения каждые пять минут. Это значит — 12 отсчетов в час, или около 105 тыс. в год. Теперь предположим, что система мониторинга собирает данные энергопотребления с каждого устройства заказчика, которых могут быть десятки, а самих заказчиков — тысячи. В итоге таблица, хранящая события в реляционной БД, будет содержать многие миллионы строк: допустим, при наличии 100 заказчиков

со средним числом подключенных приборов 20 за год такая таблица пополнится 200 миллионами строк. Вот они, большие данные!

Простое применение запроса на выборку данных к подобной таблице может занять очень много время. А если добавить еще необходимость постоянных запросов на обновление таблицы поступающими от устройств сообщениями, а также постоянные запросы от веб-портала или от пользователей на получение отчетов, то станут очевидны будущие проблемы такой архитектуры с одной базой данных. Пакетная же обработка с группировкой и суммированием результатов может быть выполнена, например, с использованием Hadoop MapReduce или Apache Spark. Сами алгоритмы группировки в данном случае можно реализовать с помощью программ на Java (для MapReduce, Spark) или Python (Spark) и запустить в кластере. В итоге размеры таблицы с агрегированными данными в базе данных SQL будут существенно меньше, чем в таблице с сырыми данными. Так, если хранится часовое агрегирование, то в SQL-таблице в 12 раз меньше строк, чем в NoSQL, а если суточное — то в 288 раз. При этом для клиентов запрос на получение данных будет очень простой и высокопроизводительный: выбрать из таблицы агрегатов строки, отфильтрованные по идентификатору заказчика и по требуемому временно́му интервалу без каких бы то ни было группировок в самом запросе.

По поводу подобной архитектуры следует сделать целый ряд замечаний.

1. Обеспечить высокую точность позволяет большое количество замеров, следующих с малым временны́м интервалом. Если интервал постоянный, то можно упустить включение/выключение прибора, произошедшее между замерами. Эта проблема решается путем передачи не периодических замеров, а замеров, приуроченных к событию: включению или изменению величины проходящей мощности более чем на заданную величину. Такое решение, во-первых, разгрузит сеть от слишком частых передач отсчетов (но не полностью — необходимо оставить периодические сообщения от измерителя о его работоспособности), и во-вторых, существенно уменьшит объем сырой таблицы.
2. В случае проблем с сетью и для обеспечения высокой точности вместе с требованием разгрузки сети необходимо физическое разделение отсчетов замера мощности на уровне АЦП измерительного прибора (они могут быть выполнены с высокой частотой дискретизации) и отсылка результатов суммирования измерений в виде сообщения в систему. Чтобы обеспечить это разделение, измерительное устройство должно быть достаточно «интеллектуальным»: обладать памятью и иметь возможность синхронизировать часы, чтобы установить временну́ю метку. Может показаться, что достаточно иметь момент времени *приема* сообщения, но это неверно. Для получения высокой точности и обеспечения надежности важно каждое сообщение. Они могут быть потеряны как из-за отказа измерительного устройства, так и из-за проблем с телекоммуникационной сетью. Чтобы устранить проблемы с сетью, устройство может запоминать сообщения и пересылать их, когда сеть восстановит работоспособность. И вот тут-то очень важно, чтобы каждому сообщению была присвоена метка

времени: когда оно *сгенерировано*. Это позволит в итоге правильно подсчитать суммарное энергопотребление в заданный временной интервал.

Теперь рассмотрим еще один вопрос: а зачем вообще нужно промежуточное хранилище такого большого объема (рис. 2.2)?

Рис. 2.2. Архитектура системы учета электроэнергии, построенная на основе единого реляционного хранилища данных

Ведь можно периодически очищать сырую таблицу в SQL-хранилище после заполнения данными агрегированной таблицы и хранить только результаты агрегирования. Действительно, при наличии данных энергопотребления в течение каждого дня недели/месяца/года можно легко подсчитать суммарное энергопотребление за бóльшие периоды времени. Против такого подхода есть ряд возражений. Например, возможна ситуация, когда из-за проблем с сетью не все устройства отослали свои данные вовремя. И если сырая таблица очистится перед тем, как восстановится работоспособность сети и устройства сбросят свои данные, то последние останутся неучтенными и не будет никакого смысла дополнительно усложнять измерители в виде внутреннего буфера сообщений и синхронизируемых часов. В то же время реализация дополнительной логики, обеспечивающей пересчет агрегированных данных в *пакетном режиме* при приеме недостающих сообщений, позволит произвести правильный и надежный подсчет потребления даже при ненадежной сети передачи данных.

Тут мы сталкиваемся еще с одним аспектом анализа больших данных: *потоковой обработкой*. В данном случае необходимо из всего потока сообщений обнаруживать те, чья временна́я метка меньше, чем текущее время минус самая большая

временная задержка, и при их обнаружении запускать внепланово или планировать дополнительно запустить в определенное время (если построение агрегированных таблиц происходит в конкретное время, скажем по ночам) задачу по повторному пересчету. Для этой цели может служить, например, Apache Storm, Apache Spark Streaming или же Apache Pig.

Следующая причина, побуждающая оставить-таки сырую таблицу без удаления данных, — возможность провести *интерактивный* анализ хранящихся в ней данных. То есть применить к ним запросы на специальном языке, позволяющем комбинировать, фильтровать, группировать, проводить арифметические операции в режиме реального времени. Это позволяет находить скрытые закономерности в данных, например определять пики энергопотребления, устанавливать корреляцию их с внешними событиями (скажем, с погодой), определять профили пользователей, характеризующиеся оптимальным энергопотреблением. Или для одного пользователя строить типовой профиль применения энергоресурсов и обнаруживать отклонения от него (допустим, утечку электроэнергии, несвоевременное выключение освещения и пр.). Подобные задачи могут решаться с помощью системы интерактивного анализа данных, таких как Spark SQL или Apache Hive, а расширенный интеллектуальный анализ — благодаря применению библиотеки машинного обучения Spark MLib.

Архитектура, содержащая в своем составе хранилища как сырых данных, так и агрегированных, является очень гибкой и позволит создать не просто очередную систему учета электроэнергии, но и интеллектуальную, которая может «подсказать», как уменьшить потребление энергии, где есть узкие места, а это уже принципиально новый уровень по сравнению с простой телеметрией. Такое развитие возможно благодаря тому, что данные в ней хранятся в том виде, в котором они наиболее удобны для выполнения анализа различными сервисами: для обычного сервиса построения отчетов об энергопотреблении — в агрегированном виде, а для целей Data Mining — в сыром.

Ключевой момент всех технологий, работающих с большими данными, — возможность распараллеливания выполняемых задач, областей хранения, памяти и т. д. между группой компьютеров (кластер). Кроме того, эти технологии должны:

- обладать возможностью линейного масштабирования и наращивания производительности путем добавления новых серверов в кластер. Линейность масштабирования означает пропорциональность производительности/объема хранения количеству компьютеров в кластере;
- иметь специальную файловую систему для надежного хранения и доступа к данным, позволяющую оперировать очень большими объемами данных и допускать их репликацию в целях повышения надежности и производительности;
- позволять выполнять запросы к файлам с помощью специального языка запросов или программного интерфейса;

- иметь планировщик, позволяющий распределять эти запросы среди узлов кластера для обеспечения их параллельной работы.

Всеми этими свойствами обладает наиболее популярный фреймворк Apache Hadoop и компоненты его экосистемы — MapReduce, Hive, Pig, Spark, Storm, Kafka, HBase и др. Они более подробно будут описаны ниже.

2.2. Резюме

В главе 2 мы начали знакомиться с большими данными. На примере системы мониторинга электроэнергии я показал, как в системе, оперирующей достаточно малым количеством источников, могут возникнуть большие данные. Отмечалось, что чаще всего сырые данные малопригодны для построения информационных систем и требуется их предварительная обработка, чаще всего агрегация. Однако при этом неизбежно теряется часть информации, которую потенциально можно использовать для глубокого анализа (именуемого еще интеллектуальным, Data Mining). Чтобы разрешить данное противоречие, следует разделять хранение сырых и агрегированных данных. Последнее обеспечивается тем, что источники данных посылают их непосредственно (или через специальные сервисы-концентраторы) в хранилище сырых данных. Далее с помощью сервисов трансформации и копирования информация поступает в хранилище агрегированных данных и становится доступна для построения отчетов.

Мы рассмотрели применение трех видов анализа данных: пакетного, потокового и интерактивного — в контексте этой системы мониторинга. В последующих главах каждый составной элемент подобной системы будет подробно описан.

3 Архитектура облачных систем, оперирующих BigData

Пожалуй, самым трудным и вместе с тем обязательным в архитектуре является простота. Простота форм обязывает придавать им прекрасные пропорции и соотношения, которые сообщали бы необходимую гармонию.

Алексей Викторович Щусев

3.1. Общие сведения

Итак, в предыдущих главах мы рассмотрели облачные среды и небольшой пример архитектуры системы, оперирующей с большими данными. Теперь обобщим все эти сведения и кратко рассмотрим сервисы, предоставляемые облачными провайдерами (рис. 3.1).

Итак, облачные провайдеры позволяют подключать различные устройства, программные продукты, сервисы (игровые устройства, стационарные устройства IoT, подключенные автомобили, мобильные приложения, веб-серверы и серверы приложений, медиасервисы и пр.) через специальные сервисы концентраторов сообщений и шлюзы устройств «Интернета вещей» (IoT Gateway). Концентраторы (AWS Kinesis Stream, Azure Event Hub) обеспечивают однонаправленный прием сообщений извне в облако, а IoT-шлюз (Azure IoT Hub) — двунаправленную коммуникацию с устройствами, то есть возможность обратной отсылки команд устройствам из облака.

Этот поток может быть обработан сервисами потоковой обработки и анализа. Я разделил оба вида терминологически и на рис. 3.1. К сервисам *потокового анализа* относятся сервисы, которые допускают интерактивный анализ потока данных и позволяют создавать аналитические запросы на специальном языке (чаще всего с SQL-подобным синтаксисом), интерактивно их применять и отображать результаты (например, Azure Stream Analytics). К сервисам *потоковой обработки* относятся те, которые задействуют модули, написанные на компилируемых языках для построения потоковых задач анализа (таких как Apache Storm в HDInsight или AWS EMR) и не допускающие интерактивного использования.

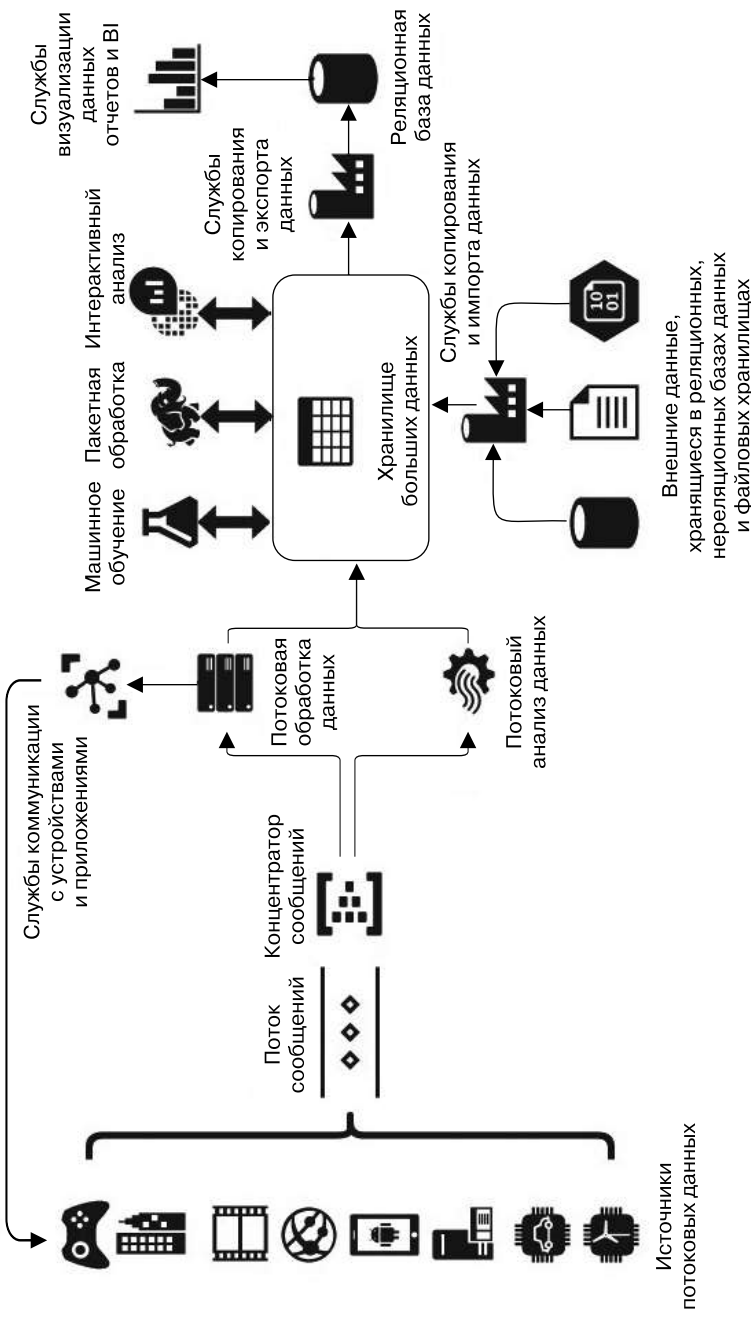


Рис. 3.1. Облачные сервисы, относящиеся к большим данным

Сообщения, полученные от концентраторов или IoT-шлюзов, могут быть направлены по своим назначениям, в зависимости от внутренних признаков (такая возможность обеспечивается системой маршрутизации сообщений: Azure Event Grid или аналогичной системой в IoT-шлюзе), или целиком направлены в облачное хранилище. Это может быть либо хранилище общего назначения (типа Azure BLOB Storage или AWS S3), либо HDFS-совместимое (Azure Data Lake).

Кроме того, данные в такое хранилище могут доставляться сервисами копирования и трансформации данных (AWS Glue или Azure DataFactoy) или специализированными сторонними программами через предоставляемые этими сервисами API. Источниками данных могут быть внешние реляционные и нереляционные базы данных и файлы.

После размещения в облачном хранилище информацию можно обработать в пакетном режиме (например, с помощью Hadoop MapRerduce в Azure HDInsight или AWS EMR), интерактивном режиме (Azure Data Lake Analytics, AWS Athena) или с применением машинного обучения. Результаты обработки могут быть размещены в реляционном хранилище данных и доступны для средств BI (Microsoft PowerBI или AWS QuickSight).

Опишу подробнее облачные сервисы. Любой облачный провайдер предоставляет сервисы IaaS, позволяющие создать ряд виртуальных машин, которые можно объединить в кластер. В этом случае пользователю придется самому создавать и конфигурировать нужный BigData-фреймворк. Ситуация в какой-то мере облегчается тем, что имеются готовые образы виртуальных машин с предустановленными утилитами и компонентами. Но реализация IaaS-решения требует достаточно высокой квалификации у пользователей облачных сред. Ведь, помимо навыков инсталляции и конфигурирования образцов виртуальных машин, необходимо оперировать сервисами IaaS, чтобы построить нужную инфраструктуру, что требует больших затрат времени и обширных знаний, не относящихся к области BigData. Частично задачу упрощает тот факт, что облачные сервисы можно создавать с помощью шаблонов: AWS CloudFormation или Azure ARM Template. Но остаются сложности интеграции IaaS-решения с другими сервисами, сервисами логирования и мониторинга.

В то же время для каждого описанного компонента общей архитектуры, приведенной на рис. 3.1, у облачных провайдеров Microsoft Azure и AWS существует свой PaaS-сервис. Но у обоих провайдеров для ряда компонентов имеются два вида сервисов.

К первому виду относятся сервисы, которые целиком применяют существующие фреймворки больших данных из экосистемы Hadoop. Задействуются стандартные средства взаимодействия с пользователем (например, Jupiter Notebook для написания запросов и отображения результатов). При этом создание, конфигурирование и масштабирование кластера полностью автоматизировано. К таким сервисам относятся Azure HDInsight и AWS EMR. Они позволяют разворачивать кластеры Hadoop, Spark, Storm, HBase, Kafka и R-Server (только Azure). Кроме того, компоненты облачных сервисов могут быть размещены на кластерах AWS ECS / EKS или Azure Kontainer Services в Docker-контейнерах.

Ко второму виду относятся облачные сервисы, целиком и полностью (нативно) предоставляемые облачными провайдерами. Такие сервисы конфигурируются и управляются только из облачного портала или через его REST API и не используют сторонние средства и сервисы.

3.2. Архитектуры традиционных информационных систем

> Традиции всех мертвых поколений тяготеют, как кошмар, над умами живых.
>
> *Карл Маркс*

Для получения представления о том, что такое большие данные, как они возникают и обрабатываются, следует подробнее разобраться с тем, что собой представляют современные информационные системы. Системы обработки больших данных подразумевают возможность, позволяющую реализовать не только систему обработки данных, но и приложение, которое решает конкретную бизнес-задачу. Эта идея — асинхронная передача сообщений (событий) между компонентами системы и применение отдельных сервисов, обеспечивающих надежную передачу сообщений. Казалось бы, практически любая информационная система имеет дело с обменом сообщений, представленном в том или ином виде, так что тут нового? Разберемся по порядку.

Многие годы информационные системы строились в виде одного крупного монолита с единообразной кодовой базой, которая разворачивается на сервере как единое целое: вместе со всеми подключаемыми модулями, библиотеками и т. д. Такой монолит (рис. 3.2) отвечал за все (или почти): за взаимодействие с базами данных, обеспечение системы контроля учетных данных, работу периодических сервисов синхронизации, логирования и пр.

Ниже представлены проблемы, свойственные такой архитектуре.

- Любое изменение в любом компоненте монолита, даже самое незначительное, требует компилирования и разворачивания всей системы, что при большом размере монолита задача весьма небыстрая.
- Монолит очень плохо масштабируется и подвержен существенным проблемам с использованием ресурсов. Действительно, как правило, подобные архитектуры разворачиваются на одном сервере или в виде полных копий на группе серверов. Поэтому любой компонент (или компоненты), который задействует ресурсы сервера (скажем, оперативную память, процессор и т. д.), в значительной степени может затруднить работу всего монолита и потребует увеличения производительности всего сервера, на котором расположена данная архитектура. Компоненты монолита невозможно масштабировать независимо.

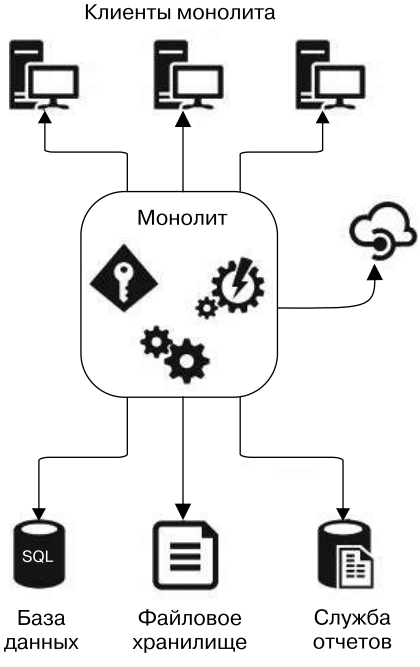


Рис. 3.2. Монолитная архитектура

- Монолит жестко диктует стек технологий программирования, и обновить его, а также обновить саму архитектуру — значит, по сути, переписать весь код заново. Сюда же можно отнести проблемы с быстрым «старением» монолита, склонностью к обрастанию спагетти-кодом, а также «тупиковой внутренней архитектурой» — это когда невозможно исправить баг или улучшить что-либо, не вызвав появления нового бага или ухудшения системы.

Сопровождение монолита может причинить множество неудобств программистам, системным администраторам, тестировщикам...

Решить указанные проблемы была призвана многослойная архитектура (рис. 3.3). В ней за специфическую задачу отвечает отдельный слой: пользовательского интерфейса (user interface layer, UI layer), бизнес-логики (business logic layer, BL layer) и доступа к данным (data access layer, DAL).

Рассмотрим эту архитектуру подробнее. Каждый слой отвечает только за определенную группу задач. UI-слой содержит только веб-серверы, являющиеся источником веб-страниц, или размещает сервисы (REST, SOAP или др.) для взаимодействия с клиентскими приложениями. Кроме того, данный слой принимает запросы от клиентов и транслирует их слою бизнес-логики. Поскольку с клиентом напрямую взаимодействует только UI-слой, то на одном уровне с ним в тесной интеграции находится сервис аутентификации клиентов. Слой BL содержит серверы приложений и отвечает, собственно, за бизнес-логику и интеграцию со сто-

Рис. 3.3. Многослойная архитектура

ронними сервисами. Слой DAL обеспечивает программный доступ слоя BL к базе данных и файловому хранилищу.

Подобная архитектура по сравнению с монолитной имеет следующие преимущества.

- Разделение на слои позволяет реализовать в каждом слое наиболее подходящий для него стек технологий. Можно независимо обновлять фреймворки на каждом слое.

- Логическое разделение на слои существенно упрощает процесс разработки и сопровождение всей системы. Распределение команд программистов по слоям и специализация разработчиков (фронтенд, бэкенд), уменьшение объема кода на каждом слое, а также независимый деплой (развертывание) значительно улучшают качество всей системы, делая ее более гибкой и пригодной для сопровождения.
- Возможно независимое масштабирование каждого слоя.

Наиболее часто подобная архитектура реализуется в виде серверов, расположенных в локальной сети, разбитой на подсети с настроенными фаерволами. Адреса серверов (IP или URL) разных слоев и учетные данные для доступа к ним прописаны в конфигурационных файлах других серверов. В этой архитектуре уже необходимо централизованно хранить учетные данные, иметь серверы DNS, сервис хранения логов и мониторинга, а также сервер для администрирования. В монолитной архитектуре все это могло быть расположено на одном большом общем сервере. Логирование также было весьма простым, поскольку вся система строилась в рамках одного стека технологий. В многослойной архитектуре каждый слой может генерировать логи, существенно отличающиеся от логов другого слоя. Кроме того, увеличение количества серверов тоже ведет к увеличению количества и видов логов.

Многослойная архитектура информационных систем в настоящее время чрезвычайно распространена, она более гибкая и удобная по сравнению с монолитной, однако не может решить ряд проблем. В частности, при разрастании проекта до такого масштаба, что слой BL становится сопоставимым с монолитом, появляются аналогичные проблемы с масштабированием, производительностью, сопровождением и т. д.

Следующая разновидность многослойной архитектуры — SOA: сервис-ориентированная архитектура. Ее квинтэссенцией является архитектура микросервисов (рис. 3.4), суть которой заключается в том, что каждое приложение разбивается на наименьшие самостоятельные модули, и каждый из них отвечает только за один аспект: отсылку писем, загрузку и выгрузку файлов и пр. Каждый такой сервис можно совершенно независимо развернуть и обновить в любой момент, не затрагивая остальные сервисы. Код подобного сервиса может быть совсем небольшим, и для его сопровождения нужна маленькая команда. Более того, все микросервисы можно писать на разных языках, лучше всего подходящих для решения текущей задачи. И наконец, каждый микросервис может быть развернут на своем сервере или в кластере серверов, которые могут быть совершенно независимо масштабируемы. Каждый сервис доступен по URL конечной точки и из них, как из элементов конструктора, можно собрать новые системы, а каждый сервис использовать в различных системах.

Однако работа с микросервисной архитектурой предполагает ряд трудностей.

- Большое количество сервисов, размещенных на многих серверах, означает большое количество разнообразных логов. По сути, вот они, большие данные! Анализ этих данных, определение метрик производительности, узких мест,

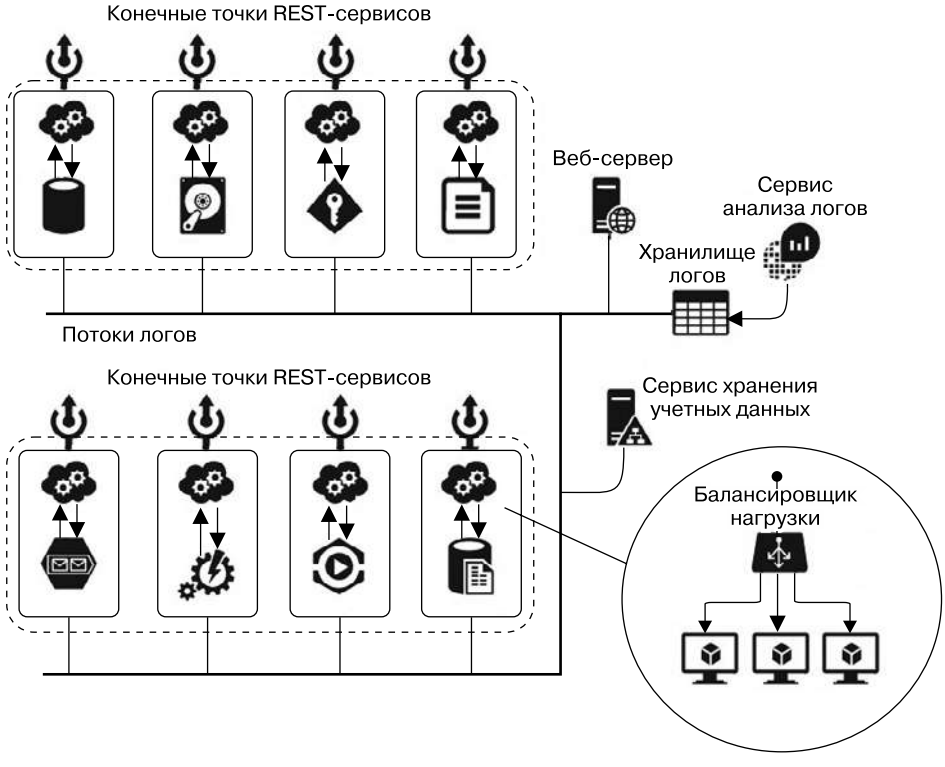


Рис. 3.4. Архитектура микросервисов

поиск логов и отображение результатов в виде графиков в режиме, близкому к режиму реального времени, требует не просто сервиса, но целой подсистемы, сопоставимой с основной системой. В качестве примера такой подсистемы можно привести ELK-стек — Elasticsearch (хранение логов и поиск), Logstash (агенты, обеспечивающие доставку логов с серверов в кластер Elasticsearch) и Kibana (интерактивные диаграммы, графики и др.), Splunk.

- Взаимодействие между сервисами происходит преимущественно с помощью REST. А потому каждый сервис должен знать URL всех сервисов, с которыми он может потенциально взаимодействовать.
- Если по какой-то причине запрос не был обработан (например, сервис был недоступен), то для его повторения необходимо организовать логику повтора в рамках самого кода.

Синхронизировать работу микросервисов позволит отдельный сервис, обеспечивающий надежную доставку сообщений, а также предоставляющий малое количество конечных точек. И вот тут мы постепенно подходим к Event Driven Design — архитектуре, основанной на обмене сообщениями. Базовые элементы

сервисов обмена сообщениями, лежащие в основе этой архитектуры, будут рассмотрены в части III данной книги.

Архитектуры, построенные на базе микросервисов, каждый из которых можно разместить в кластере серверов, дополненные сервисами обмена сообщениями (брокерами сообщений — message brokers), могут быть чрезвычайно масштабируемыми. Для этого нужно, чтобы сами сервисы обмена сообщениями были масштабируемыми. Брокеры, как правило, имеют архитектуру с одним или несколькими головными узлами, отвечающими за управление кластером, и исполнительными узлами, на которых выполняются необходимые вычисления и хранятся данные (сообщения) (рис. 3.5). Чтобы обеспечить надежность кластера, данные могут быть реплицированы между исполнительными узлами.

Рис. 3.5. Общая структура масштабируемой системы, построенной на основе кластера

Как вы уже поняли, подобные структуры весьма сложны: кластеры серверов, балансировщики нагрузки, системы управления конфигурацией, логирования и пр. И сейчас самое время напомнить преимущества облачных платформ: все подобные системы представляются как сервисы — AWS ECS и Azure Service Fabric. Подробное описание этих сервисов и архитектур на их основе выходит за рамки книги, укажем только, что данные сервисы отвечают за масштабирование, мониторинг, развертывание приложений или контейнеров на них.

В облачных средах, наряду с сервисами PaaS, отвечающими за обработку больших данных, Azure и AWS предоставляют сервисы, занимающие промежуточное положение между IaaS и PaaS. Эти сервисы позволяют применять наиболее популярные фреймворки, работающие с большими данными, — Apache Spark, Storm, Kafka, HBase, Storm и ряд других. К таким сервисам относятся AWS EMR и Azure HDInsight. Они берут на себя все трудности, связанные с настройкой и конфигурированием сервисов в кластерах, а конечному пользователю предоставляют готовый и настроенный сервис. Это очень большое преимущество подобных сервисов по сравнению с ручным конфигурированием кластера из виртуальных машин. Практически все сервисы Apache Hadoop являются продуктами с открытым исходным кодом, имеют подробную документацию и поддержку обширного сообщества, однако их установка, настройка и конфигурирование на практике очень трудны. Я помню случай, когда заказчик, обладающий

очень крупной системой мониторинга подключенных IoT-устройств, отказался от Apache Kafka в пользу AWS Kinesis Stream только потому, что для заданных потоков данных кластер виртуальных машин не мог быть настроен в сжатые установленные временные сроки для достижения требуемых надежности и доступности (uptime). В то же время сервис AWS Kinesis настраивается и конфигурируется в течение нескольких минут.

Итак, сервисы AWS EMR и Azure HDInsight занимают промежуточное положение между IaaS и PaaS, сочетая все возможности сервисов Apache с простотой PaaS. Эти сервисы представляют собой надстройки над набором виртуальных машин. Последние создаются, запускаются, останавливаются и удаляются только на время выполнения задания в сервисе или по мере надобности (как в случае Apache Kafka, например). Сервисы AWS EMR и Azure HDInsight очень хорошо интегрируются с сервисами копирования и трансформации данных (скажем, AWS Data Pipeline), составляя, по сути, вычислительные ресурсы для заданий ETL. При создании кластеров с портала для HDInsight и AWS EMR указываются размеры виртуальных машин и сценарий, который должен выполняться при запуске сервиса. Создание и удаление управляемых кластеров полностью автоматизируемо, что и обусловливает удобство их использования в случае построения систем пакетного анализа, копирования и трансформации данных.

3.3. Бессерверные архитектуры

> За огромным письменным столом с массивной чернильницей сидел пустой костюм и не обмакнутым в чернила сухим пером водил по бумаге. Костюм был при галстуке, из кармашка костюма торчало самопишущее перо, но над воротником не было ни шеи, ни головы, равно как из манжет не выглядывали кисти рук. Костюм был погружен в работу и совершенно не замечал той кутерьмы, что царила кругом. Услыхав, что кто-то вошел, костюм откинулся в кресле, и над воротником прозвучал хорошо знакомый бухгалтеру голос Прохора Петровича.
>
> *Михаил Булгаков. Мастер и Маргарита*

Архитектура, основанная на обмене сообщениями, характеризуется тем, что все компоненты системы взаимодействуют через малые порции сообщений, требующие вычислительных ресурсов только непосредственно в момент обработки сообщений. Все остальное время сервисы заняты только тем, что прослушивают конечные точки сервисов обмена сообщениями. Во время прослушивания серверы или

виртуальные машины не загружены. Но в облачных средах принимается плата за включенную виртуальную машину вне зависимости от того, насколько интенсивно она используется. Таким образом, если поток сообщений не слишком сильный, то сервисы работают вхолостую. Но зачастую необходимо выполнить программу однократно или в непредсказуемые моменты времени в ответ на поступившее сообщение, для чего использовать виртуальную машину нецелесообразно. Для такого случая облачные провайдеры обеспечивают возможность выполнения кода по требованию без предоставления серверов — бессерверные сервисы (serverless). Подобный сервис от Microsoft называется Microsoft Azure Function, а от AWS — AWS Lambda. Последний является классическим вариантом. Этот сервис позволяет размещать исполняемый код и запускать его внешним событием, которое может прийти от сервисов SNS, SQS, Kinesis Stream, S3, API Gateway и др. Сам сервис имеет различные уровни производительности процессора и памяти, которые используются лишь при исполнении кода, и плата начисляется только в этот момент. Безусловно, внутри сервиса AWS Lambda содержится цикл, ожидающий прихода сообщения, но вычислительные ресурсы задействуются (и, соответственно, за их применение начисляются деньги) только для выполнения кода пользователя.

Концепция serverless очень удобна для построения приложений с малым, средним и умеренно большим потоком сообщений.

Главные преимущества serverless таковы:

- более дешевая среда исполнения кода, чем виртуальные машины, при небольших, средних и умеренно больших потоках;
- более высокая надежность сервиса, чем в случае одиночной виртуальной машины, — SLA на уровне 99,9 %;
- простота интеграции с другими облачными сервисами, не требующая внесения изменения в код;
- простота развертывания кода и конфигурирования — необходимо только добавить код напрямую или через ZIP-файл и выбрать уровень производительности.

Однако подобным сервисам присущи и недостатки:

- ограниченная масштабируемость;
- ограниченное время выполнения кода;

Но вместе с тем serverless-сервисы крайне удобны для построения архитектур, основанных на событиях (event driven design). AWS Lambda наиболее удобно использовать для инициирования выполнения задач копирования/трансформации данных, которые должны последовать после наступления какого-либо события. Рассмотрим такую задачу: требуется обработать файлы логов, которые первоначально копируются в хранилище AWS S3 (оно будет подробнее рассмотрено в следующей части). Добавление файла в S3 инициирует помещение сообщения в сервис SQS (сервис очередей) или SNS (сервис передачи сообщений многим

подписчикам). Далее можно создать AWS Lambda, которая будет инициировать сообщение из SQS/SNS. Код в AWS Lambda запускает задачу копирования обработки логов, скажем, на основе AWS EMR. Под обработкой понимаются различные задачи: прямое копирование данных из файла в реляционное хранилище, поиск ошибок и пересылка строк с ошибками в реляционное хранилище или SNS, агрегирование данных (подсчет успешных обращений к серверу за определенный временно́й промежуток или подсчет количества попыток выполнения логина за заданный временно́й интервал и пр.). В подобных случаях AWS Lambda — идеальный инструмент управления и синхронизации более мощных ресурсов (например, кластера EMR).

3.4. Описание примера

— Фукс, какой же вы к дьяволу матрос?!

— А я не матрос...

— Но позвольте, Лом уверял меня, что вы способный и умеете разбираться в картах.

— О-о! Это сколько угодно! Карты — это мой хлеб. Только не морские, а игральные карты.

Андрей Некрасов. Приключения капитана Врунгеля

Рассмотрим конкретный пример системы, построенной на основе как обработки и анализа большого потока сообщений в режиме реального времени, так и анализа исторических данных. Описанный пример будет опираться на все основные облачные сервисы, оперирующие большими данными. Предположим, требуется создать масштабируемую систему онлайн-покера, нагрузка для которой может быть как очень большой, так и минимальной. В самом общем виде архитектура подобной системы состоит из двух частей: мобильных и десктопных клиентов и бэкенда. Кроме того, возможна реализация браузерной версии, притом примеры кода могут относиться к бэкенду сервера, но для наглядности будем говорить о клиентах. Клиенты — это программы, которые применяются пользователями для игры, или сервисы, запускаемые при действиях пользователя в браузере. Каждое действие пользователя, относящееся к игровому процессу, программно преобразуется в сообщение заданного формата, пересылаемое в бэкенд. Все пользователи сервиса онлайн-покера логически объединены в группы пользователей, играющих за одним столом (румы). Действие одного пользователя должно приводить к изменению отображения в пользовательских интерфейсах во всех остальных клиентах.

Таким образом, помимо потока сообщений в сторону бэкенда, должен присутствовать поток сообщений от бэкенда в сторону клиентов. Эти сообщения должны содержать в сериализованном виде информацию, необходимую для обновления пользовательского интерфейса каждого отдельного клиента. Таким образом, пользовательским клиентам следует содержать графический интерфейс и обработчики действий пользователей. Графический интерфейс должен обновляться при изменении игровой ситуации — при выполнении действий пользователями. Причем это должно происходить синхронно для всех игроков стола. Чтобы это обеспечить, коду клиента надлежит, с одной стороны, посылать сообщения в ответ на действие данного пользователя, а с другой — принимать их в ответ на действия других. Кроме того, клиенты должны иметь возможность регистрироваться в системе, выполнять вход через ввод учетных данных (имя пользователя, пароль) и выход из системы, создавать покер-румы или подключаться к существующим, смотреть игровую статистику, а также взаимодействовать с сервисами платежных систем/рекламы.

Бэкенд должен:

- содержать сервисы, обеспечивающие прием сообщений от клиентов;
- вмещать сервисы обработки сообщений и отправки результатов обработки клиентам;
- хранить отображение игровой ситуации, историю ходов, логи приложений и прочие данные;
- анализировать ходы пользователей на предмет мошенничества, подсчитывать статистику;
- хранить учетные данные пользователей.

Рассмотрим более подробно возможные архитектуры нашей системы онлайн-покера. Традиционная клиент-серверная архитектура игровых сервисов состоит из сервера и клиентов, обменивающихся сообщениями напрямую с помощью различных протоколов. Например, клиенты могут установить TCP-соединение с сервером и через TCP-сокеты обмениваться сообщениями. Но такое построение системы требует учитывать в разработке много низкоуровневых деталей: обработку обрывов, новые попытки соединения, шифрование и др. Существуют более удобные протоколы: SOAP, RPC и пр. Но реализация масштабируемых и надежных систем на их основе тоже сопряжена с большими трудностями.

Облачные провайдеры позволяют создать такую инфраструктуру с помощью облачных сервисов и SDK для программного доступа к ним. Рассмотрим, как это сделать. Для захвата потоков сообщений от клиентов используются сервисы концентраторов сообщений: Azure Event Hub, AWS Kinesis. Эти сервисы перенаправляют сообщения в несколько других сервисов: облачное хранилище (AWS S3, Azure BLOB Storage), сервис потокового анализа (Azure Stream Analytics, AWS Kinesis Analytics), а также в игровые сервисы, которые могут быть реализованы напрямую на базе виртуальных машин или же в Docker-контейнерах, разворачиваемых в кластерах. Эти игровые программы содержат код, взаимодействующий

с SDK концентраторов сообщений, SDK хранилища игровой ситуации и собственно игрового алгоритма. И все. Создателям программы не нужно заботиться ни об аутентификации, ни о масштабировании, ни об обработке ситуаций обрыва связи... Основные усилия разработчика при этом сосредотачиваются на игровом алгоритме — все сложности коммуникации берет на себя облако. Поскольку работа игрового алгоритма состоит из одиночных актов реакции на входные сообщения, то можно применить еще один архитектурный паттерн — serverless, обратившись к сервисам AWS Lambda или Azure Function. События, которые запускают эти сервисы, — приход сообщений в их концентратор, поступление сообщений в шину интеграции, загрузка объекта в облачное хранилище, запрос REST API и др.

Итак, игровой алгоритм обрабатывает входной поток от пользовательских клиентов и генерирует выходной поток в сторону пользователей на основе текущей игровой обстановки (раскладка карт у игроков, сделанные ставки и объем наличных денег). Игровая обстановка должна храниться в базе данных с быстрым доступом, простой структурой и отличным масштабированием. Хорошо подойдут NoSQL-базы типа Azure Table Storage или AWS DynamoDB. Ключом в этой базе будет ID игрового рума. Поля данных содержат раскладки игроков, текущие ставки и ситуацию на общем столе. Это могут быть поля таблицы или одно поле, в котором данные хранятся в сериализованном виде.

При поступлении сообщения игровой сервис извлекает из сообщения ID игрока и измененную ситуацию с точки зрения данного игрока. Далее он извлекает предыдущую игровую ситуацию из NoSQL и обновляет ее в соответствии с поступившим сообщением. Затем нужно обновить ситуацию и у всех остальных игроков — они должны получить сообщения на перерисовку пользовательского интерфейса. Для этого игровой алгоритм рассылает им всем соответствующие сообщения. Каждый ход каждого пользователя должен сохраняться в хранилище, допускающем обработку истории ходов игроков с целью анализа статистики. Это могут быть как NoSQL-базы, так и специальные хранилища данных: Azure Data Lake, Azure SQL DWH или Amazon RedShift. Анализ этих данных позволит отслеживать, кто из игроков играет лучше всего, наличие самых отстающих, потенциальных шулеров и пр. Конечно, все хранилище можно реализовать с помощью одного SQL-сервера, но это будет означать, что он будет принимать на себя всю нагрузку без возможности масштабирования. Разделение данных на DWH-данные, NoSQL и данные в хранилище позволит независимо масштабировать эти сервисы и выполнять к ним запросы. За аутентификацию пользователей, регистрацию их в системе и выход из нее отвечает облачный сервис аутентификации: Azure AD, AWS Directory Service.

Как видим, в случае реализации нашей системы онлайн-покера на облачной платформе большая часть сервисов, обеспечивающих работоспособность системы, уже доступна «из коробки», и сложность разработки такой системы значительно уменьшается. Я не стал описывать вспомогательные сервисы (например, вебпортал администратора системы, систему онлайн-распознавания мошеннических действий и др.), которые тоже можно реализовать в облаках.

3.5. Резюме

В этой главе были описаны типовые архитектуры информационных систем: как традиционные (монолитная, трехслойная), так и ориентированные на работу с сообщениями — Event Driven, в том числе микросервисы, бессервисные приложения и кластеры. Кроме того, описан пример, который будет разобран на части, — игровой сервис онлайн-покера.

Обзор этих архитектур крайне важен для понимания особенностей облачных сервисов и места больших данных среди них. В дальнейшем я не буду акцентировать внимание на традиционных архитектурах, а остальные подходы — Event Driven, микросервисы, кластерные технологии — рассмотрим весьма подробно.

Часть II
Хранение BigData в облаке

> Объем человеческих знаний удваивается примерно каждые пять лет, причем время этого удвоения постоянно уменьшается. На переломе XIX–XX веков период этот составлял около пятидесяти лет. Ежедневно в мире публикуется 7 тысяч статей, печатается более 300 миллионов газет, а книг — 250 тысяч, радиоприемников и телевизоров эксплуатируется уже около 640 миллионов. Поскольку эти данные четырехлетней давности, они наверняка являются заниженными, особенно из-за стремительного роста знаний благодаря спутниковому телевидению.
>
> *Станислав Лем. Молох*

Любые данные, как большие, так и маленькие, хранятся в файлах. Это могут быть двоичные файлы или текстовые файлы того или иного формата (CSV, TXT, JSON и пр.). Любая база данных любого типа в конечном итоге хранит данные в виде специальных файлов на дисках, но терминологически и фактически базы данных (database) и хранилища данных (data storage) предоставляют различные концепции хранения информации и доступа к ней.

Базы данных (БД) должны обеспечивать хранение информации, доступ к выборке произвольной части информации и ее обновление. В этом случае важна сама полученная информация, которая находится как на жестком диске сервера базы данных, так и в оперативной памяти этого сервера. Важнейшим элементом базы является система управления базой данных (СУБД) — специальное программное обеспечение, служащее для обработки запросов к БД и выполнения различных административных операций (мониторинг, резервное копирование, управление доступом и др.). СУБД отделяет физическое хранение данных от их логического представления.

Традиционно все базы делят на два крупных вида: реляционные (иначе называются SQL-базы) и нереляционные (NoSQL). В реляционных логически информация представляется в виде набора таблиц, связанных между собой, то есть находящихся в определенных отношениях (отсюда и название, содержащее корень relatio — «отношения»). Нереляционные базы (типа «ключ — значение», граф, семейство колонок, документоориентированные) очень разнообразны и объединены лишь тем, что модель представления данных в них отличается от реляционной. Для программного доступа к СУБД необходимо знать адрес сервера, на котором она размещена, учетные данные пользователя, имеющего соответствующие права, и программную библиотеку, поддерживающую протокол обмена данными управляющими сигналами с СУБД.

В отличие от БД *хранилище данных* содержит файлы и позволяет получать к ним доступ для загрузки, обновления и удаления файлов, в которых хранится

информация в физически и логически неотделимом от файла виде. Хранилища — универсальное место хранения для файлов любых типов: текстовых и бинарных. Классически хранилища данных в виде сетей хранилищ данных (SAN), которые представляли собой группу серверов, имеющих высокоскоростные диски (SSD, зачастую объединенные в RAID-массивы), объединенных в сеть. Для доступа к файлам извне необходима поддержка протоколов сетевых файловых систем NFS, CIFS, SMB и др., а также прямая ссылка, чтобы файлы можно было скачивать. В отличие от СУБД, где физическое хранение информации отделено от ее логического представления, при хранении данных в файловом хранилище физическая структура файла полностью повторяет их логическую структуру. Например, в файлах логов информация представлена в виде таблицы, каждая строка которой состоит из отдельных столбцов. Чтобы получить доступ к ней, файл можно открыть с помощью любого текстового редактора или стандартных программных библиотек ввода-вывода. В этой части книги представлены различные сервисы хранения больших данных в облаках. Сервисы для обработки данных будут описаны в последующих главах.

4 Хранилища общего назначения

— Толстенные книги получатся, если записать все, что люди знают.

— А еще толще книги можно написать о том, чего люди до сих пор не знают.

Жюль Верн. Таинственный остров

4.1. Общие сведения

Облачные хранилища файлов — это сервисы, которые обеспечивают хранение файлов, предоставляют доступ к ним с помощью REST API, позволяют скачивание по прямой ссылке (постоянной или с конечным сроком действия) и в некоторых случаях предоставляют доступ к файловой системе по протоколу SMB.

Хранить файлы, содержащие большие данные, в облачном хранилище очень дешево и в ряде случаев удобнее, чем в других хранилищах. Наиболее типовой пример размещения данных — это хранение файлов логов приложения, которые копируются туда периодически из сервера-источника или создаются и заполняются специальной программой-клиентом, размещенной на сервере-источнике логов (рис. 4.1).

Кроме того, в облачных файловых хранилищах могут размещаться виртуальные жесткие диски (virtual hard drive, VHD) облачных виртуальных машин, на которых, в свою очередь, тоже можно размещать файлы. Но в этом случае ответственность за доступность информации ложится на владельца виртуальных машин. Рассмотрим подробнее различные форматы хранения больших данных.

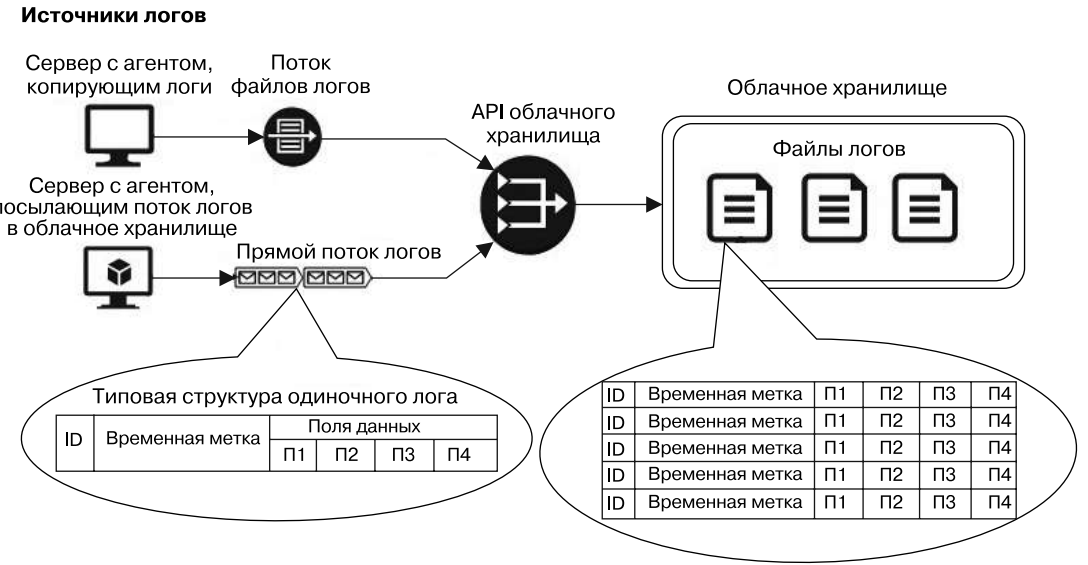


Рис. 4.1. Структура потока логов серверов в облачное хранилище

4.2. Форматы хранения данных

Как уже отмечалось, наиболее типовой случай использования текстовых файлов для хранения больших данных — хранение логов приложений в том или ином текстовом формате. Такие файлы могут иметь расширения `.log`, `.txt`, `.csv` и др. Общее у этих форматов то, что логи в них, по сути, хранятся в виде таблицы (отсюда и название — *табличный формат*), каждая строка которой — запись конкретного события. Строка состоит из набора столбцов, разделенных некими символами. Это могут быть пробелы, символы табуляции, двоеточие, точка с запятой и т. д. Подобные файлы можно открыть с помощью любого текстового редактора (если они не очень велики), и для их анализа (например, поиска запросов с 500 ошибками) подойдут стандартные утилиты командной оболочки (grep, awk и пр.) или же специализированные сервисы аналитики (к этому вопросу мы еще вернемся). Кроме того, табличная структура подобного файла очень удобна для импорта в реляционную или нереляционную СУБД табличного типа. Особенность таких файлов — линейная структура со строго одинаковым количеством столбцов в каждой записи и в общем случае линейное время доступа к записям. Строго говоря, возможны ситуации, когда разные строки могут содержать различное количество столбцов. Например, запись в лог-файле, отражающая ошибку приложения и трассировку стека, может включать гораздо меньше строк, чем запись, отражающая какое-либо событие, характеризующее нормальную работу приложения. В общем же случае записи логов содержат как минимум временную метку. Остальные поля (уникальный идентификатор, URL, сообщение ошибки и пр.) могут и отсутствовать. Возникает вопрос: полезна ли запись с одним полем (временной меткой)? Да. Скажем, необходимо проанализировать посещаемость сайта во времени. Просуммировать запросы для заданного временного интервала (например, 15 минут) поможет именно временная метка.

Табличные файлы имеют и ряд неудобств в использовании. Во-первых, для доступа к элементу информации необходимо знать номер строки и номер столбца. Нет универсального стандарта или языка запросов (за исключением SQL-подобного языка специализированных сервисов аналитики, но об этом позже), что весьма затрудняет анализ логов. Во-вторых, в таких форматах очень просто добавить новую запись в конец файла, но крайне трудно и затратно вставить, удалить или изменить произвольную строку. Как правило, все программные библиотеки ввода-вывода позволяют добавлять строку в конец файла с помощью стандартных средств, но для произвольной манипуляции данными программисту нужно будет создать отдельную логику в коде, и эта логика весьма непроста.

Помимо текстового файла с табличной структурой, широко распространено хранение информации в более структурированных форматах JSON, XML. Их преимущества в том, что они очень удобны для сериализации/десериализации информации в объектно-ориентированном виде в коде программы. Кроме того, для обоих

форматов существуют стандарты выполнения запросов на выборку данных (для XML — XPath, XQuery, для JSON — JSONPath, JSONQuery). Кратко рассмотрим эти форматы.

JSON представляет собой формат, при котором данные хранятся в текстовом виде как объект JavaScript (аббревиатура образована от JavaScript Object Notation). JSON-файл выглядит так (рис. 4.2).

```
{
  "PlayerId" : "1", // partition key
  "TableId": "1", // partition key
  "StepId" : "guid", // row key
  "Pot": "700", // money on table
  "Players" :
  [
    {
        "PlayerId" : "1",
        "Action" : "", // bet, all-in, call, raise, fold, check
        "Rate": "10",
        "Amount": "500", // bankroll for current game
        "DoneAction": "true",
          "PlayerCards":
          [
            "JH","KC"
          ]
    },
    {
        "PlayerId" : "2",
        "Action" : "", // bet, all-in, call, raise, fold, check
        "Rate": "10",
        "Amount": "500", // bankroll for current game
        "DoneAction": "false",
          "PlayerCards":
          [
            "JC","KH"
          ]
    },
    {
        ...
    },
    {
        ...
    }
  ],

  "OpenCards":[
    "7D","10S","3C","","9D"
  ]

}
```

Рис. 4.2. Пример организации информации с помощью формата JSON

На данном рисунке приведен вариант организации информации в формате JSON для нашего сквозного примера. В основе этого документа (как и любого документа JSON) лежит *объект*, который состоит из набора «ключ — значение». В качестве ключа выступают текстовые величины (`PlayerId`, `Players`, `StepId` и др.). В качестве значений допускаются следующие типы.

- *Атомарный тип* (для ключей `PlayerId`, `StepId`) — величина, состоящая из одного конкретного значения: строки, числа, временно́й метки.
- *Массив* (например `Players`) — группы величин одного типа. (Это не совсем точное определение, в общем случае все элементы массива могут быть совершенно разного типа, но, как правило, коллекции объектов, сериализуемые в JSON в реальных программах, будут иметь одинаковый тип.) В качестве типов элементов массива могут выступать атомарные типы, другие массивы или другие объекты. Массивы обозначаются прямоугольными скобками — `[ ]`, а элементы в массиве разделяются запятыми, допустим: `[ "a", "c", "d" ]`. Доступ к элементу происходит по числовому индексу, например `[0]`.
- *Объект* — коллекция «ключ — значение». Обозначается фигурными скобками `{ }` и имеет синтаксис `{"key": value}`, где `value` может иметь атомарный тип, тип массива и объекта. По сути, эта коллекция является ассоциативным массивом, в котором для доступа к значению необходимо использовать строковый ключ.

Стоит заметить, что документ JSON может содержать в качестве корневого элемента не объект, а массив, например: `[ { "key": 1 }, { "key": 2 }, { "key": 3 } ]`. Это допустимо. Но в примерах, приводимых в книге, вы будете встречаться с JSON-документами с корневыми элементами типа «объект».

Преимущество этого формата текстового файла — возможность описания структур произвольной глубины вложенности (объект внутри объекта, коллекция внутри объекта и пр.), что невозможно в случае табличного представления. Кроме того, существенно упрощается сериализация и десериализация объектов, особенно из JavaScript-приложения. Обработка JSON-файлов в специализированных СУБД (DocumentDB) описана в книге ниже, а некоторые реляционные базы данных (например, PostgresSQL) широко поддерживают этот формат. Очень важным преимуществом формата JSON является то, что он позволяет строить информационные системы с помощью одной и той же технологии на всех уровнях: начиная с документоориентированной БД, хранящей данные в формате JSON (Azure DocumentDB, MongoDB), бэкенда на основе диалекта JavaScript (например, Node.js) и заканчивая фронтендом на основе того или иного фреймворка JavaScript (например, ReactJS, Angular). Это уникальная возможность, реализованная сейчас только в рамках JavaScript/JSON (например, популярен фреймворк MEAN).

К недостаткам JSON можно отнести его «многосимвольность»: для группировки данных используются символы (запятые, скобки, кавычки), которые сами по себе не несут информации. Кроме того, присутствуют имена полей. Перечисленные недостатки проявляются наиболее ярко в случае файлов крупного размера, относящихся к большим данным. Но это неизбежная плата за возможность сериализации/десериализации сложных структур. Избавиться от части избыточных элементов синтаксиса поможет формат YAML, но пока что он используется преимущественно в качестве шаблона конфигурационных файлов (в системах Ansible, CloudFormation и др.), а не для хранения данных. Кроме того, сериализация и десериализация YAML не так широко поддерживается программными библиотеками ввода-вывода. Выборка значений из документа JSON происходит с первоначальной десериализацией его в объект в программном коде, что наиболее удобно и естественно происходит в языке на основе JavaScript.

Следующий популярный формат хранения данных в текстовых файлах — *XML* (eXtensible Markup Language — расширяемый язык разметки). Традиционно он использовался для разметки (markup), то есть структурирования текстовых документов. Термин «язык» (language) говорит о том, что XML содержит строгий набор синтаксических правил, на основании которых можно построить конкретное расширение (extension) этого языка. Рассмотрим, что это значит.

В общем случае у XML есть два основных компонента: теги и атрибуты (см. выше рис. 4.1 — пример лога). *Тег* — синтаксический элемент, ограничивающий конкретную порцию информации: `<ИмяТега>Информация в текстовом виде<ИмяТега/>`. Любой тег начинается открывающим (`<`) и закрывающим (`>`) символами. Информация, представленная в текстовом виде, расположена между тегами, последний из которых состоит из двух символов (`/>`). Возможны и вложенные структуры тегов и коллекции последних. *Атрибуты* относятся к конкретному тегу и представляют собой коллекцию «ключ — значение». На рис. 4.3 приведена структура XML-документа, представляющего информацию, показанную на рис. 4.2.

Согласно стандарту XML для тега с конкретным именем существует строго определенный набор атрибутов, причем каждый тег *должен* содержать конкретный набор атрибутов или не включать их вовсе. Возможно также построение коллекций — см., например, теги `<OpenCards>` или `<OtherPlayers>` на рис. 4.3.

Итак, язык XML состоит из общих правил построения синтаксиса, а расширение данного языка представляет собой конкретный набор вложенных тегов и соответствующих им атрибутов, который обеспечивает упорядоченное представление конкретной структурированной информации. Существует стандартизированный способ преобразования информации из одного XML в другой с помощью XSLT — eXtensible Stylesheet Language Transformation. Это тоже XML, элементами которого является не информация, а правила, по которым она из одного типа XML преобразуется в другой. Кроме того, есть XSD (XML Schema

```
<GameEvent>
    <CurrentPlayer>
    <Player "ID" = "1" "TableID" = "1" "stepId"="guid" "Pot"="700"></Player>
    </CurrentPlayer>
    <OtherPlayers>
        <Player "ID" = "1" "TableID" = "1" "stepId"="guid" "Pot"="700" "DoneAction"="true" >
            <PersonalCards>
                <Card>7D</Card>
                <Card>10S</Card>
            </PersonalCards>
        </Player>
        <Player "ID" = "2" "TableID" = "1" "stepId"="guid" "Pot"="500" "DoneAction"="false" > ...  </Player>
        <Player "ID" = "3" "TableID" = "1" "stepId"="guid" "Pot"="700"  "DoneAction"="false" > ... </Player>
        <Player "ID" = "1" "TableID" = "1" "stepId"="guid" "Pot"="700"  "DoneAction"="false" > ... </Player>

    </OtherPlayers>

    <OpenCards>
        <Card>7D</Card>
        <Card>10S</Card>
        <Card>3C</Card>
        <Card></Card>
        <Card>9D</Card>
    </OpenCards>

</GameEvent>
```

Рис. 4.3. Представление информации в формате XML

Definition) — синтаксис, описывающий схему, то есть структуру документа. Безусловно, стандарт XML содержит еще ряд других элементов, я описал наиболее употребительные для случаев хранения данных.

В отличие от JSON XML гораздо более строг и не допускает произвольной вложенности и комбинирования элементов. Например, для одного тега не допускаются разные наборы элементов. Не допускается хранение XML в виде значения атрибута (конечно, так можно сделать, ведь атрибут является текстовым значением, но это очень плохой стиль), в случае коллекции всем ее элементам необходимо иметь совершенно одинаковую схему, то есть набор тегов и атрибутов должен быть одинаков для каждого элемента. В отличие от JSON практически все основные БД поддерживают работу с XML как с самостоятельным типом данных, поскольку он достаточно строг и однозначен. Поддержка в этом случае состоит в предоставлении встроенных средств сериализации таблиц в XML и обратно. Кроме того, XML широко применяется для создания языков описания разметки гипертекстовых документов, при построении пользовательского интерфейса (HTML, XAML, RDLC и пр.).

Ниже представлены достоинства XML как средства хранения информации:

- строгость синтаксиса существенно упрощает построение систем сериализации/десериализации;
- этот формат широко поддерживается различными БД как встроенный тип данных;
- можно выполнить простое прямое преобразование в другие языки, в том числе в языки разметки графических элементов (например, *HTML*);

- существуют специальные расширения языка, описывающие различного рода преобразования и трансформации *(XSLT)*;
- имеется стандартный способ построения запроса к элементу или выборки элементов (*XPath*, *XQuery*).

К недостатком данного формата можно отнести то, что он гораздо более «многословен», чем JSON, поскольку содержит больше чисто синтаксических символов.

4.3. Облачное хранилище Microsoft Azure Storage

Как молодой повеса ждет свиданья
С какой-нибудь развратницей лукавой
Иль дурой, им обманутой, так я
Весь день минуты ждал, когда сойду
В подвал мой тайный, к верным сундукам.

Александр Сергеевич Пушкин. Скупой рыцарь

Теперь рассмотрим, как реализовано облачное хранилище, на примере Microsoft Azure Storage. Оно состоит из четырех сервисов: BLOB Storage, Queue Storage, Table Storage и File Storage (рис. 4.4).

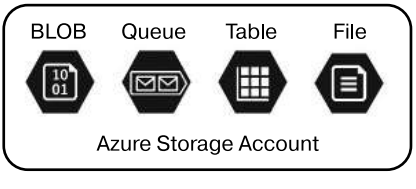


Рис. 4.4. Виды сервисов Azure Storage Account

Непосредственно хранение информации осуществляется в сервисах BLOB, Table и File Storage. Queue Storage — это облачный сервис обмена сообщениями и синхронизации распределенных приложений. Сервис Table Storage — база данных NoSQL типа «ключ — значение», которая будет подробно описана далее в книге. Пока же сосредоточимся на Azure File и BLOB Storage.

Рассмотрим подробнее, как создавать Azure Storage Account и управлять с вебпортала. Прежде всего необходимо нажать ссылку добавления новых ресурсов Azure, расположенную в левом верхнем углу, — + New. Затем в открывшемся окне (рис. 4.5) выбрать последовательно Storage — Storage account.

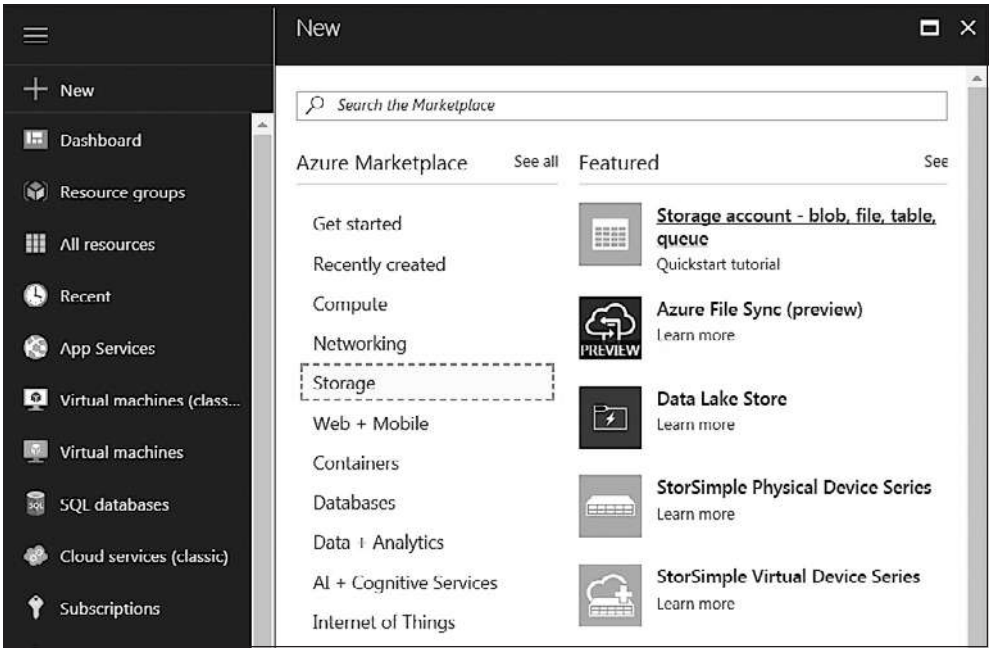


Рис. 4.5. Добавление нового Storage account через веб-портал

После нажатия ссылки Storage account откроется форма настройки Storage Account (рис. 4.6). В ней доступны следующие конфигурации.

- Имя аккаунта (поле Name) будет частью URL аккаунта `<Name>.core.windows.net`, а потому правила наименования ресурсов точно такие же, как и в случае именования ресурсов, доступных по URL.
- Тип модели развертывания ресурсов (deployment model) — выбираем Resource Manager, чтобы иметь возможность добавить аккаунт к общей ресурсной группе PockerRumExample.
- Тип Storage (Account kind) — Storage (general purpose v1). Помимо этого, доступны типы Storage (general purpose v2) и BLOB.
- Выбираем уровень производительности (Performance) — Standard. В зависимости от выбранного уровня производительность операций чтения-записи будет отличаться. Наиболее высоким уровнем будет обладать комбинация Account kind = BLOB, Performance = Premium. Для premium-уровня в качестве физических устройств хранения выступают SSD-диски.
- В качестве режима репликации (Replication) выбираем Locally-Redundant storage (LRS). Эта опция означает, что информация, хранящаяся в Storage Account, физически реплицируется три раза, но в пределах одного дата-центра. Помимо

LRS, доступны различные режимы географической репликации в разных датацентрах в пределах одной зоны (ZRS) или среди нескольких различных зон (GRS и ReadOnly-GRS — репликация по различным географическим зонам с репликой, доступной только для чтения). Все эти режимы различаются по стоимости и уровню надежности и доступности, что позволяет реализовать облачные хранилища, отвечающие различным наборам требований.

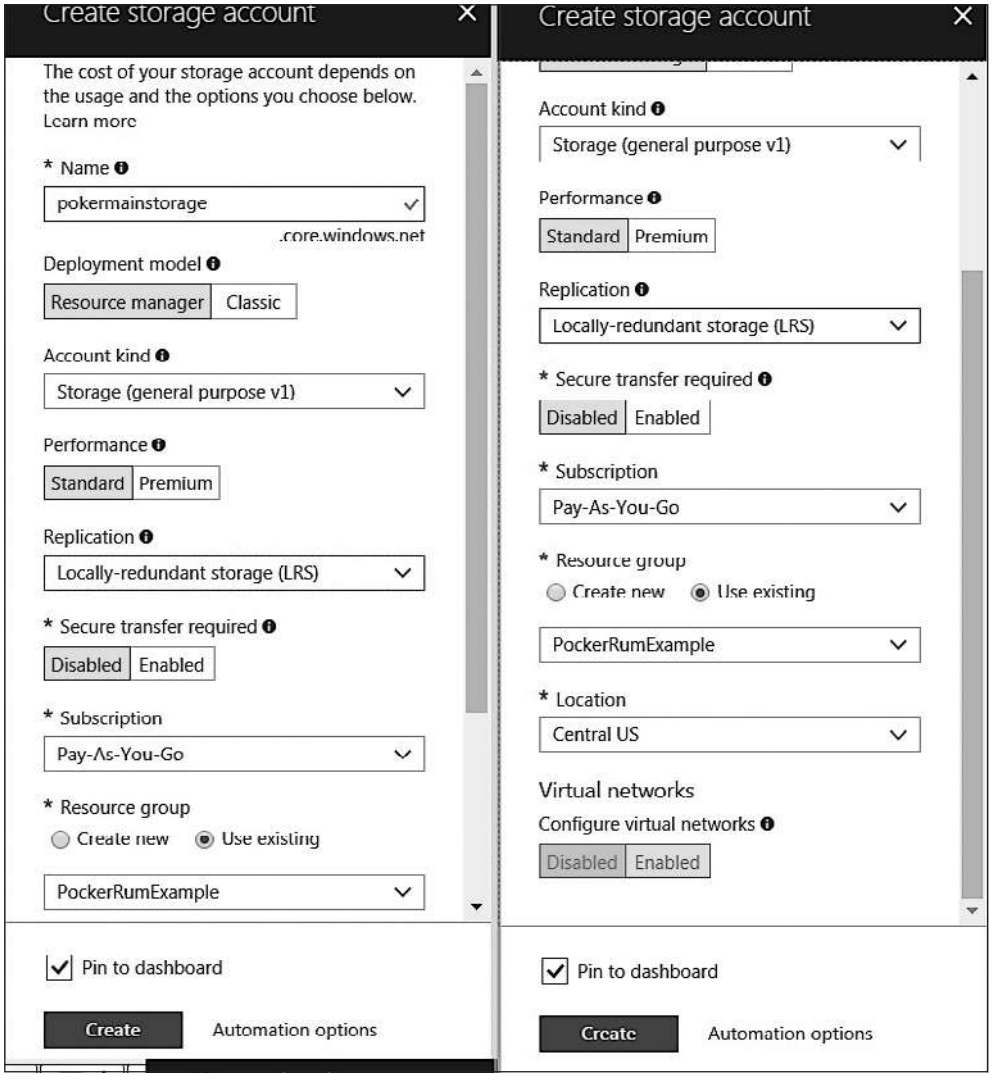


Рис. 4.6. Форма настройки Storage Account (показана со сдвигом ползунка)

После создания Storage Account доступна следующая панель мониторинга и управления (рис. 4.7).

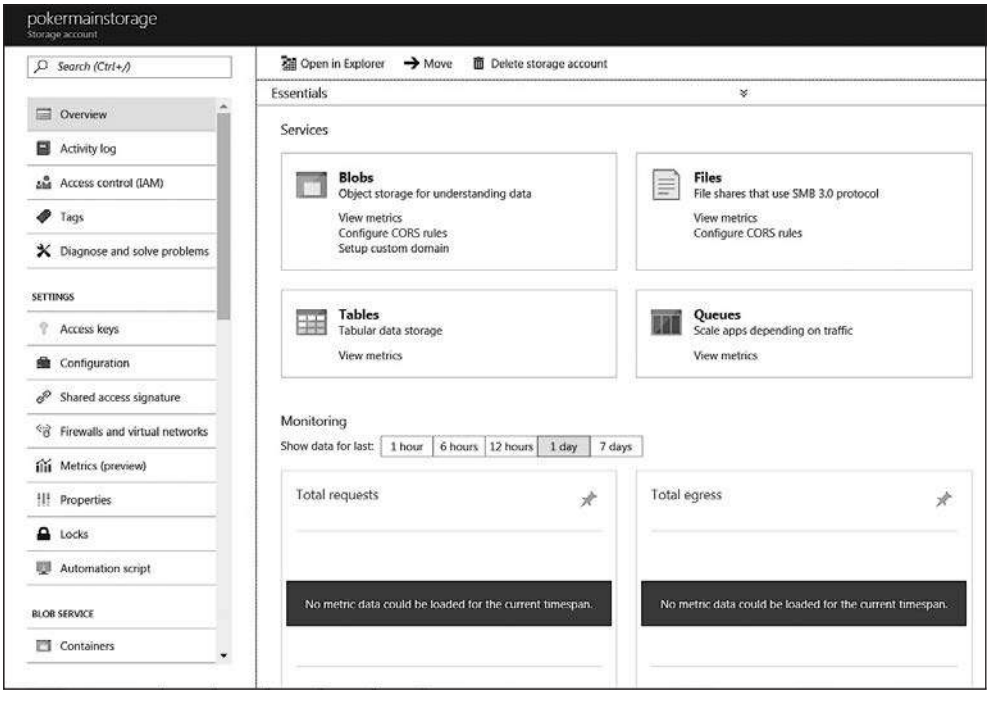


Рис. 4.7. Общая панель мониторинга и управления Storage Account

На этой панели доступны все четыре сервиса, входящие в состав Storage Account: Blob, File, Table и Queue. Далее в главе рассмотрим Blob Storage и File Storage. Сервис Table Storage изучим в главе 6, посвященной нереляционным базам данных. Queue не будет рассматриваться, поскольку он предназначен для синхронизации сервисов «потребитель — производитель» и фрагментарно будет рассмотрен в части III.

Кратко рассмотрим возможности конфигурирования Storage Account в целом.

Прежде всего, доступен File Explorer — бесплатная программа от Microsoft, позволяющая просматривать содержимое всех сервисов всех аккаунтов хранения. Я не буду представлять эту программу подробно, приведу только ее внешний вид (рис. 4.8). На этом рисунке можно увидеть результат выборки программы из хранилища Table Storage (таблица events), которая содержит тестовые данные телеметрии метеостанции, полученные путем приема сообщений через сервисы концентратора EventHub и потоковой аналитики Stream Analytics Job (подробно они будут описаны в части III).

Чтобы обеспечить доступ к аккаунту извне, необходимо знать ключи доступа и URL, которые доступны на вкладке Access Keys (рис. 4.9).

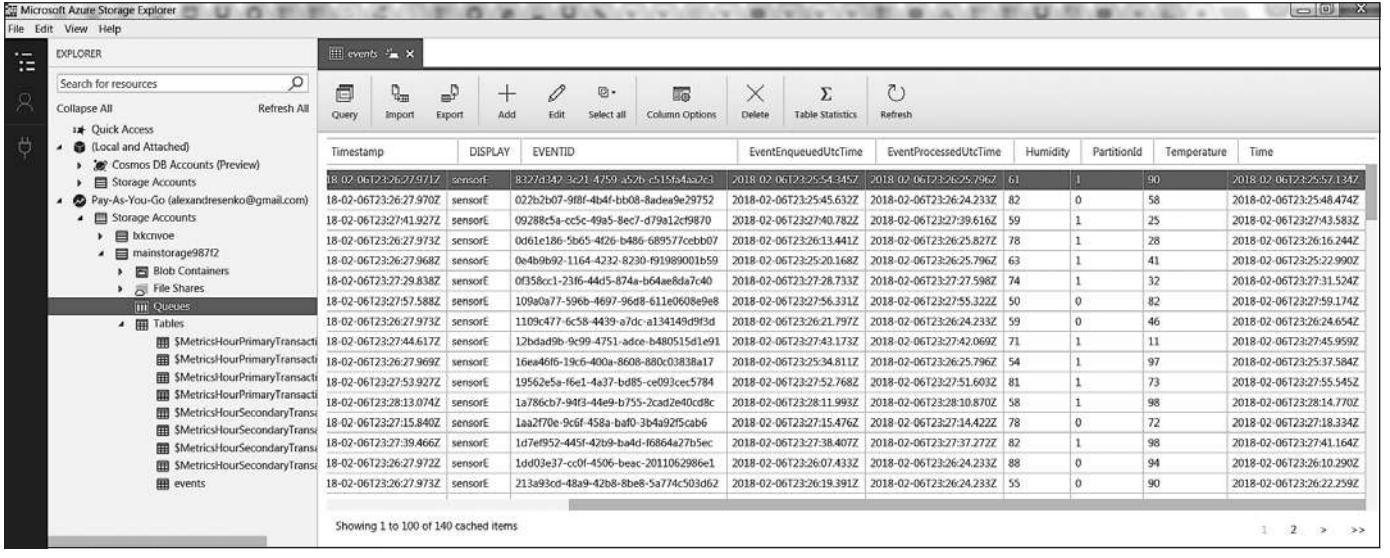


Рис. 4.8. Внешний вид программы File Explorer от Microsoft

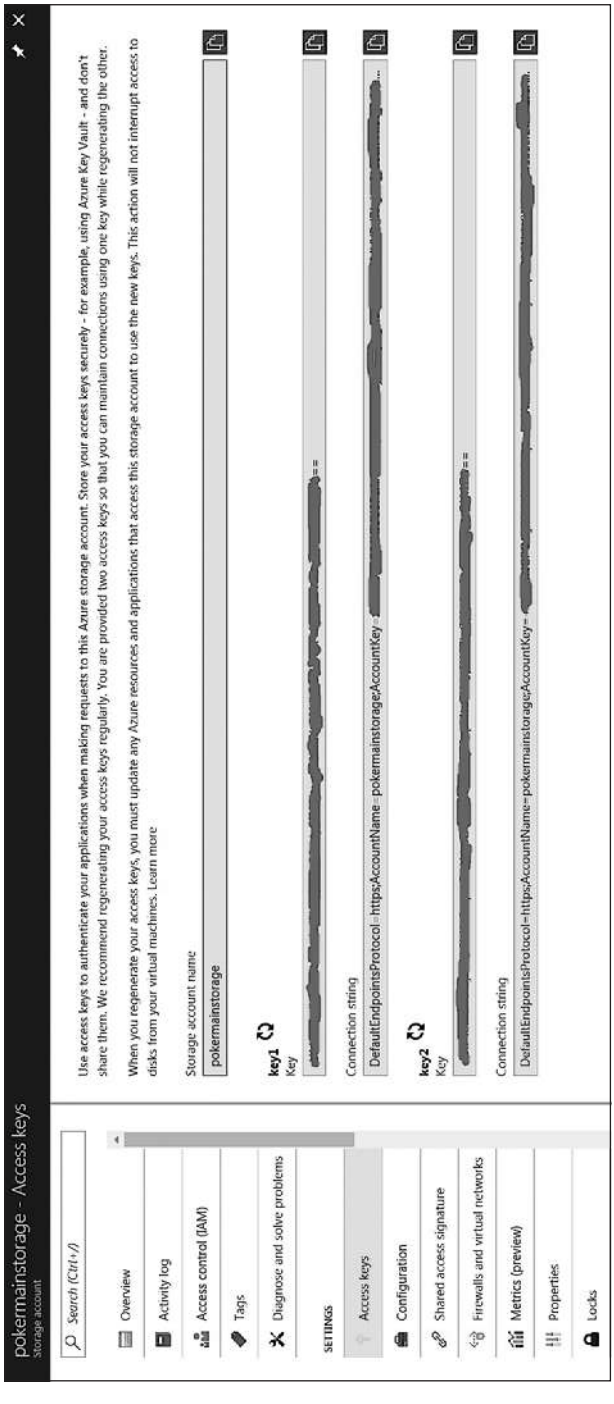


Рис. 4.9. Ключи доступа и строки подключения к Storage Account

Две пары ключей или строк подключения нужны для обеспечения «бесшовного» обновления ключей. Для этого первоначально используется ключ key1, затем клиенты переключаются на ключ key2, а ключ key1 обновляется. После этого клиенты переключаются на новый ключ key1, а ключ key2 обновляется.

Страница конфигурирования аккаунта в целом показана на рис. 4.10. Она позволяет менять тип аккаунта (Account kind), уровень производительности (Performance). К подобным манипуляциям следует отнестись внимательно, поскольку каждый уровень имеет различную стоимость, а также переключение между ними может занять время, если в аккаунте много данных.

Следующий важный конфигурируемый параметр — Shared access signature (SAS) (рис. 4.11). Концепция SAS состоит в том, что к объектам, расположенным в Storage Account, можно предоставить прямой доступ для скачивания — с помощью URL, который содержит ряд ограничивающих параметров, а именно: время жизни ссылки и ограничения IP-адресов, с которых доступен ресурс. Эти параметры подписываются ключом аккаунта, и данная подпись добавляется в конец URL, по которому данный ресурс может быть доступен.

Теперь подробнее познакомимся с отдельными сервисами хранения файлов Storage Account.

Сервис *Azure BLOB Storage* предназначен для хранения различных файлов, потому и называется хранилищем больших двоичных объектов (Binary Large OBjects Storage, BLOB). Двоичные объекты могут храниться в нем как непосредственно, так и будучи размещенными в контейнерах (не путайте с Docker-контейнерами, речь идет о контейнерах BLOB Storage) или на виртуальных дисках, тоже расположенных в BLOB Storage Account. Чтобы прояснить ситуацию, рассмотрим рис. 4.12.

Аккаунт Azure может содержать один или несколько Storage-аккаунтов. Каждый такой аккаунт способен непосредственно хранить файлы, виртуальные жесткие диски и контейнеры BLOB. Последние, в свою очередь, тоже могут включать файлы, виртуальные жесткие диски и другие контейнеры. Таким образом, с помощью контейнеров BLOB реализована иерархическая структура организации файлов.

Тут необходимо сделать пояснения. На самом деле в BLOB Storage могут храниться любые файлы: текстовые, двоичные и др., но для разных типов хранимых объектов требуется разный тип BLOB. Всего есть три типа BLOB: Page BLOB, Block BLOB и Append BLOB.

Page BLOB — бинарный объект со страничной организацией памяти. Этот тип используется только для размещения виртуальных жестких дисков виртуальных машин.

Block BLOB — BLOB с блочной организацией памяти, служащий для хранения всех видов файлов (кроме VHD), включая контейнеры BLOB. Это основной тип хранения файлов в BLOB Storage обычных файлов.

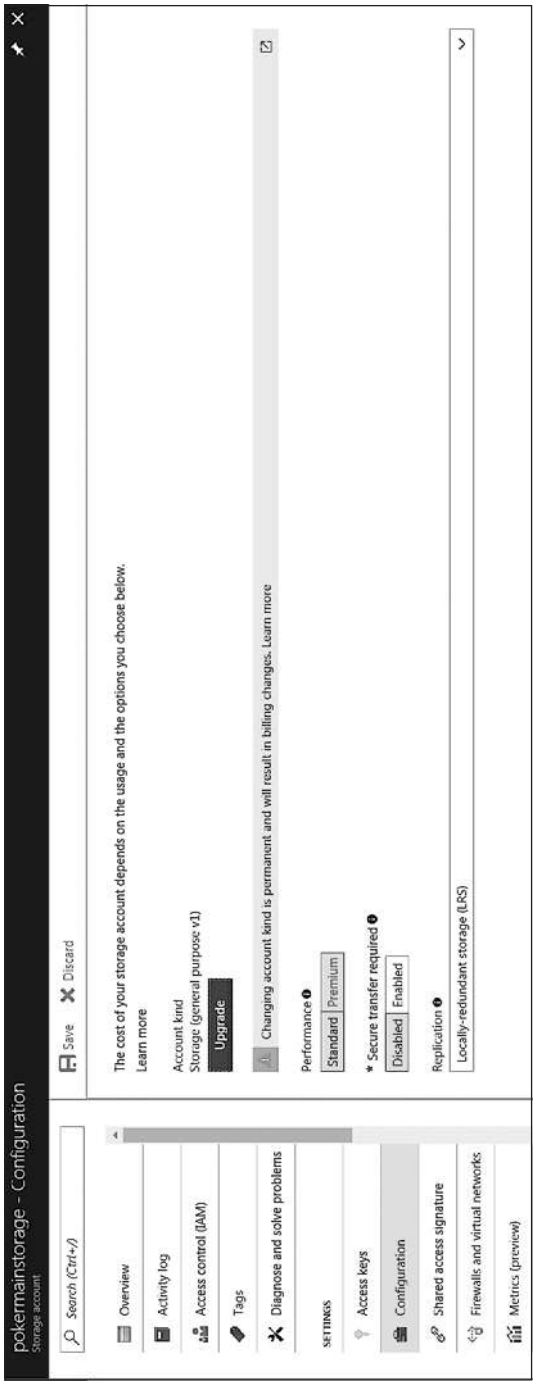


Рис. 4.10. Вкладка конфигурирования типа аккаунта, уровня производительности и репликации

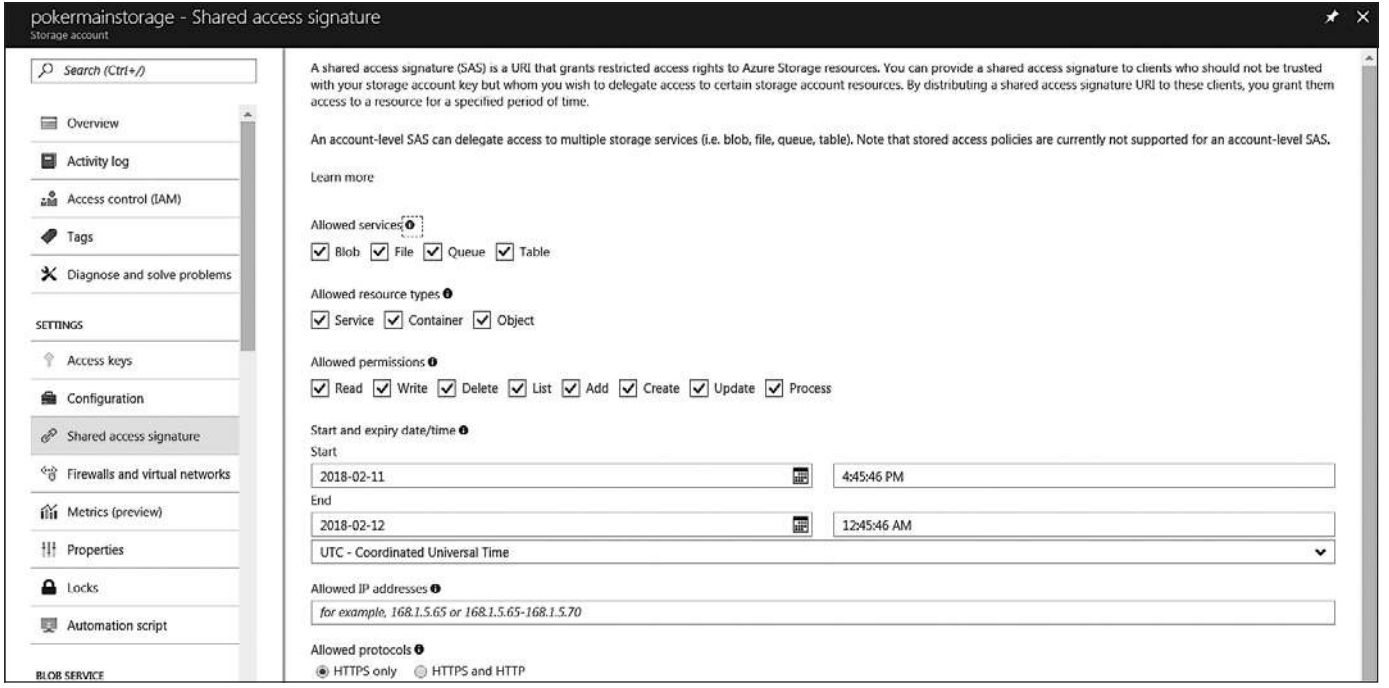


Рис. 4.11. Вкладка настройки параметров Shared Access Signature

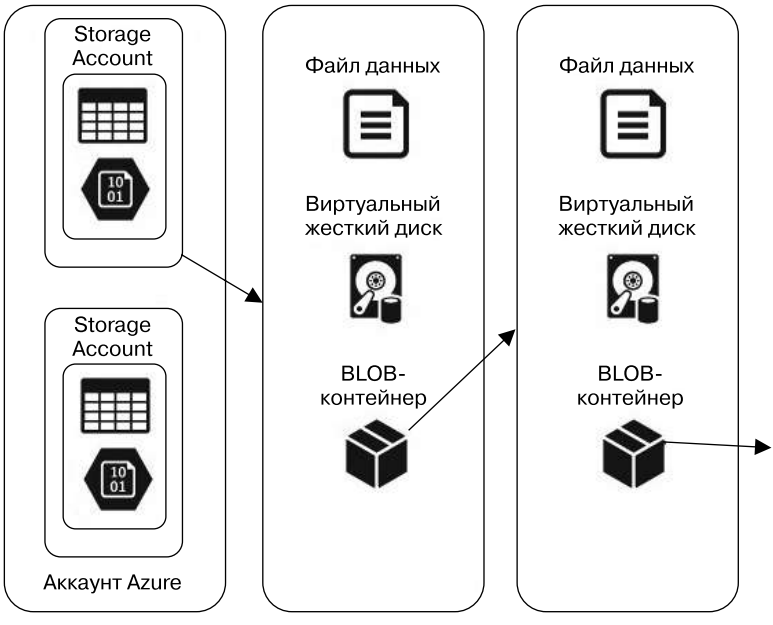


Рис. 4.12. Структура вложенности объектов в хранилище BLOB

Append Block — BLOB с блочной организацией памяти, представляющий собой текстовый файл, размещенный в Azure Storage и допускающий добавление новой записи в конец файла. В остальных типах BLOB файлы нередактируемые, то есть, чтобы отредактировать файл, его необходимо скачать, открыть, отредактировать и закачать обратно. Понятно, что эти действия сопряжены с большими трудностями при работе с файлами логов. В то же время Append Block как раз оптимизирован для сценариев прямой записи логов.

Все объекты, расположенные в BLOB Storage, могут быть доступны через веб-потрал, с помощью SDK, команд расширения командной оболочки, а также по прямой ссылке.

Доступ к Storage-аккаунту с помощью веб-портала позволяет создавать файлы, контейнеры, просматривать список файлов, добавлять, удалять и загружать их, менять области видимости файлов (они могут быть общедоступными по ссылке, закрытыми для всех, кроме сервисов Azure). Интерфейс веб-портала удобен для работы с небольшим количеством файлов. Добавлять в облачное хранилище через веб-портал можно файлы не слишком большого размера. А вот файлы, загружаемые из облака, могут быть любого размера, допустимого в хранилище. Кроме того, для файлов можно открыть общий доступ, и они станут доступными для скачивания по ссылке (эта ситуация отражена на рис. 4.13). Доступ может быть открыт для файлов как в корневом каталоге, так и в контейнерах внутри хранилища.

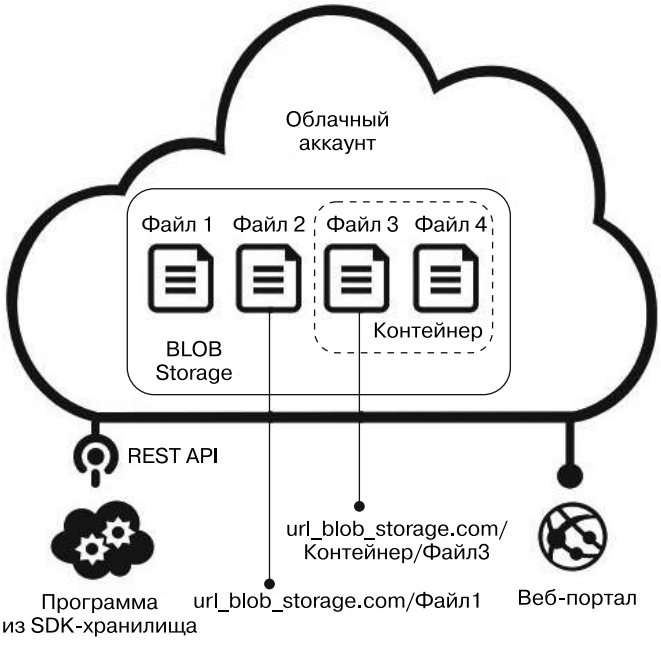


Рис. 4.13. Способы доступа к файлам в облачном хранилище BLOB

Для программного доступа облачный аккаунт содержит REST API, который, в свою очередь, через SDK предоставляет гораздо бо́льшие возможности: синхронную и асинхронную загрузку и выгрузку, удаление, создание, добавление в конец Append BLOB и пр. Кроме того, через SDK можно создать временную ссылку на файл, то есть ссылку, становящуюся нерабочей через определенный промежуток времени.

Рассмотрим подробнее, как работать с хранилищем BLOB с помощью веб-портала. Чтобы перейти к хранилищу BLOB, необходимо нажать ссылку Blobs на общей панели аккаунта (см. рис. 4.7). В результате откроется вкладка, показанная на рис. 4.14.

Далее требуется добавить контейнеры. Для этого необходимо нажать ссылку + Container (рис. 4.15).

В данной форме указано имя (свойство Name) — `pokercomm`; уровень доступа (Public access level) — Private (no anonymous access). Этот тип доступа подразумевает доступ не по прямой ссылке, а только с помощью ключей аккаунта с использованием SDK. Другие варианты типа доступа: Blob (anonymous read access for blobs only) — разрешает анонимный доступ только к файлам (BLOB); Container (anonymous read access for containers and blobs) — разрешен анонимный доступ к контейнерам и файлам. Вкладка созданного контейнера выглядит следующим образом (рис. 4.16).

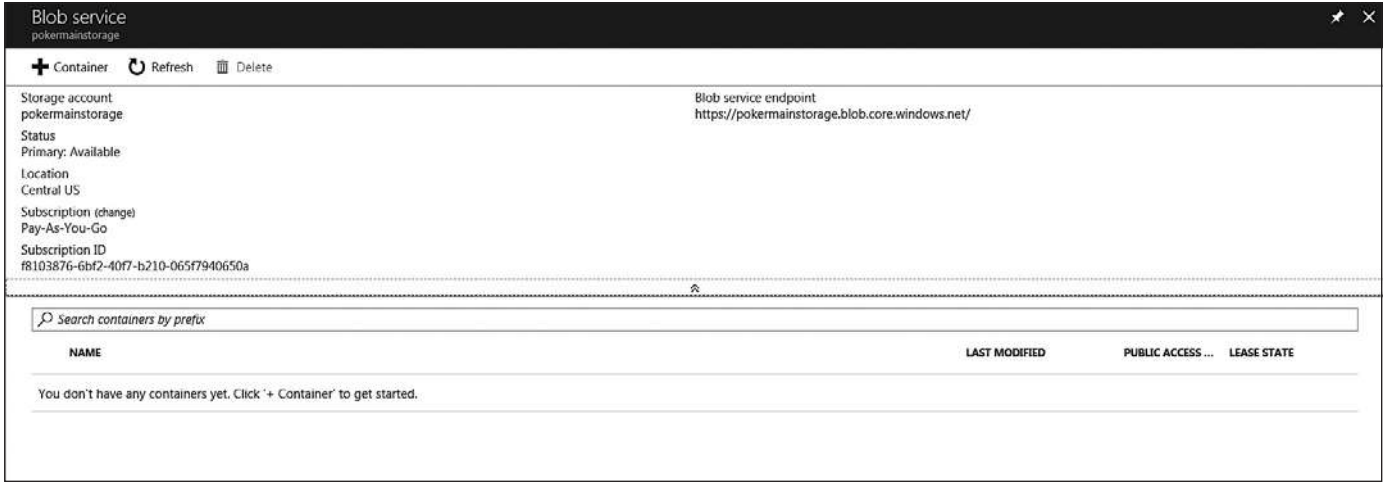


Рис. 4.14. Общая панель сервиса BLOB Storage

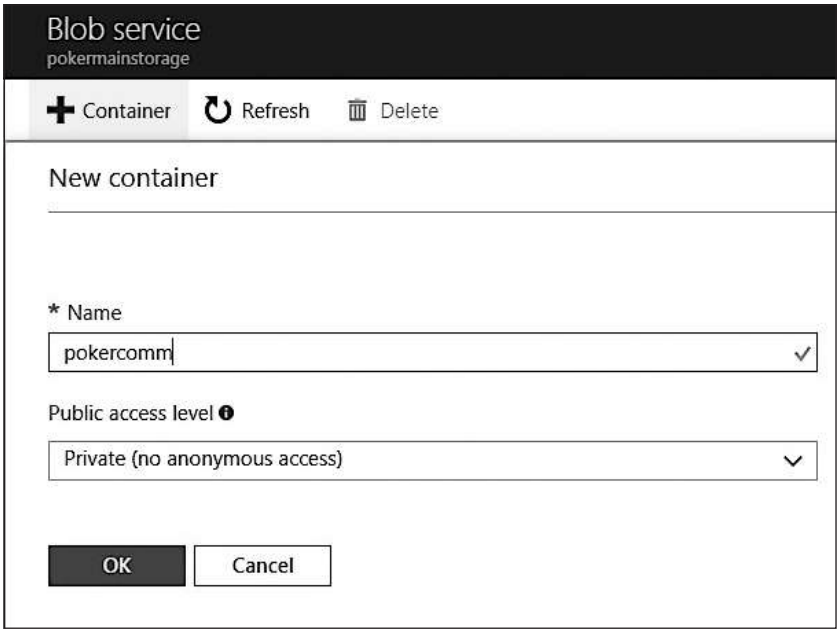


Рис. 4.15. Форма создания нового контейнера

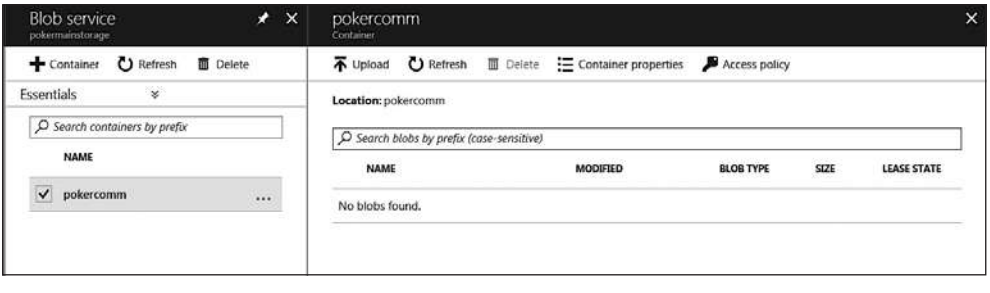


Рис. 4.16. Вкладка созданного контейнера

Загрузить файл в контейнер можно с помощью ссылки Upload. Свойства контейнера доступны по ссылке Container properties и показаны на рис. 4.17. Они включают в себя имя, адрес, статус, количество и суммарный размер BLOB-объектов, статус.

Для контейнера доступна настройка политики доступа через вкладку Access policy (рис. 4.18). Эта настройка позволяет организовать различный уровень доступа к различным контейнерам и объектам BLOB.

Помимо упомянутых настроек, для BLOB доступен ряд других. Опишу их кратко (рис. 4.19).

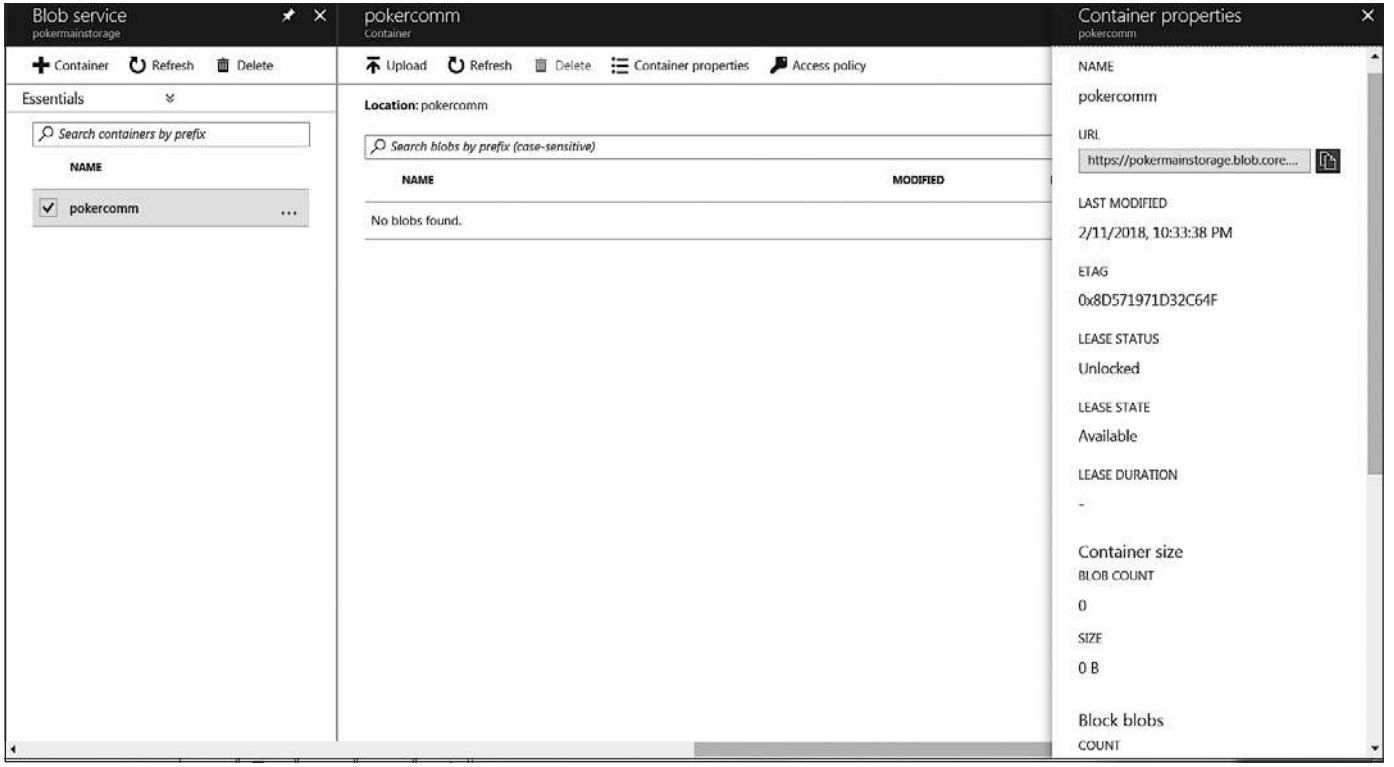


Рис. 4.17. Вкладка свойств контейнера

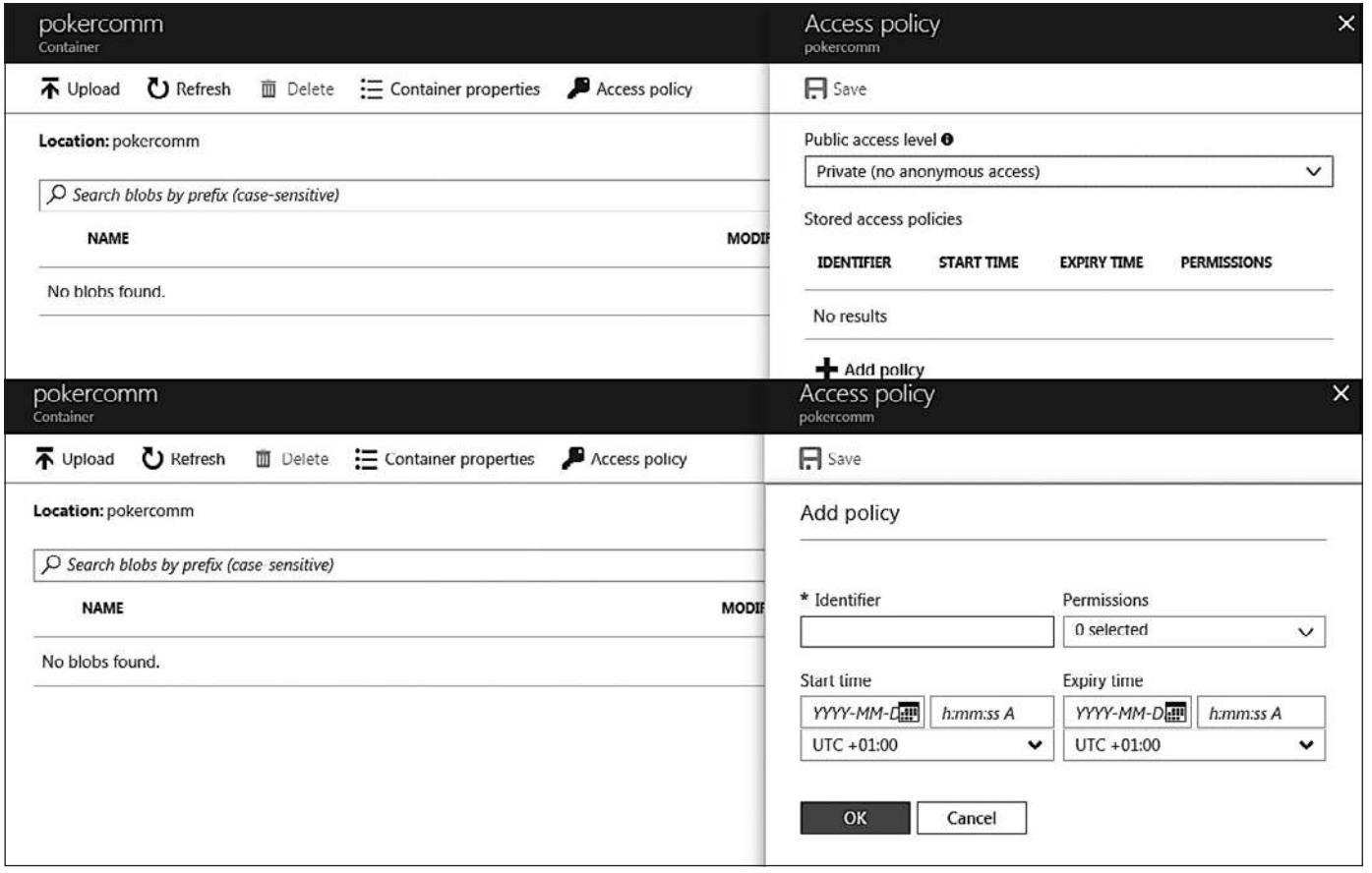


Рис. 4.18. Вкладка настройки политики доступа

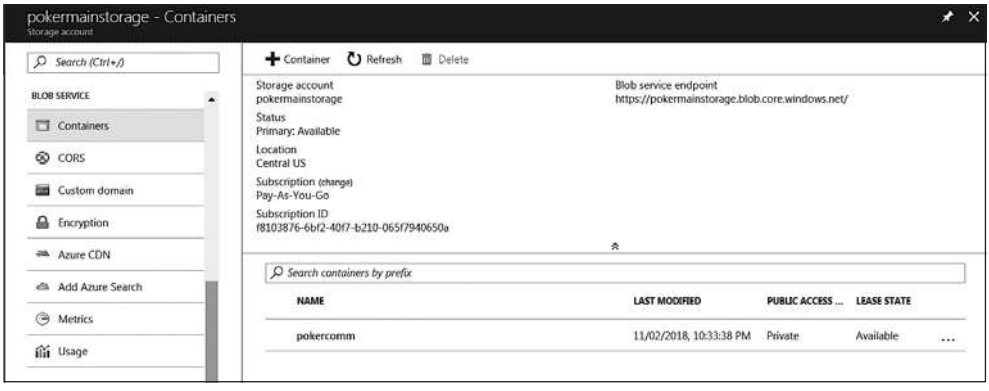


Рис. 4.19. Различные настраиваемые сервисы Blob Storage

Самые важные параметры:

- CORS (cross-origin resource sharing — совместное использование ресурсов между разными источниками) — свойство, обеспечивающее доступ к ресурсам BLOB из другого домена;
- Custom domain — конфигурирование DNS-записей CNAME в целях указания домена пользователя, в дополнение к домену аккаунту Azure Storage;
- Encryption — шифрование объектов в хранилище;
- Azure CDN — конфигурирование Azure Content Delivery Network, которая служит для хранения часто используемого контента из хранилища BLOB с анонимным доступом.

Как уже было сказано, доступ к объектам в BLOB Storage возможен по прямой ссылке URL или с помощью интерфейса REST API (напрямую либо через SDK). Данный способ хранения обеспечивает самый быстрый доступ (минимальное время загрузки/выгрузки), но требует применения специальных программных клиентов, взаимодействующих с этими API. Последнее условие может помешать существующим приложениям большого масштаба мигрировать в облако. Кроме того, в ряде случаев нужно создать облачное хранилище, которое должно быть доступно из виртуальной машины без всяких «самописных» программных клиентов. Для этого в хранилище BLOB можно разместить виртуальный жесткий диск (или набор дисков для RAID-массива) и примонтировать его к виртуальной машине. Преимущество такого решения состоит в том, что данные из виртуальной машины могут быть доступны точно таким же образом, как и с физического диска, подключенного к ней. Недостаток такого способа заключается в его низкой масштабируемости (верхний предел размера ограничен на уровне, определяемом операционной системой виртуальной машины и типом ее устройства ввода-вы-

вода), дороговизне, а также в недоступности файлов извне по прямой ссылке. Скорость доступа к файлам тоже определяется конфигурацией топологии соединения дисков в массив RAID и в ряде ситуаций существенно ниже, чем в случае BLOB. Кроме того, один VHD может быть примонтирован строго к одной виртуальной машине. Невозможный или затрудненный доступ к файлам извне, а также сложности при конфигурировании совместного доступа к файлам из нескольких виртуальных машин значительно ограничивают возможности такого хранилища в облачных архитектурах для оперирования больших данных. Чтобы преодолеть эти ограничения и одновременно обеспечить работу с файлами стандартными средствами операционных систем и программных библиотек ввода-вывода, следует использовать облачное файловое хранилище Azure File Storage (рис. 4.20).

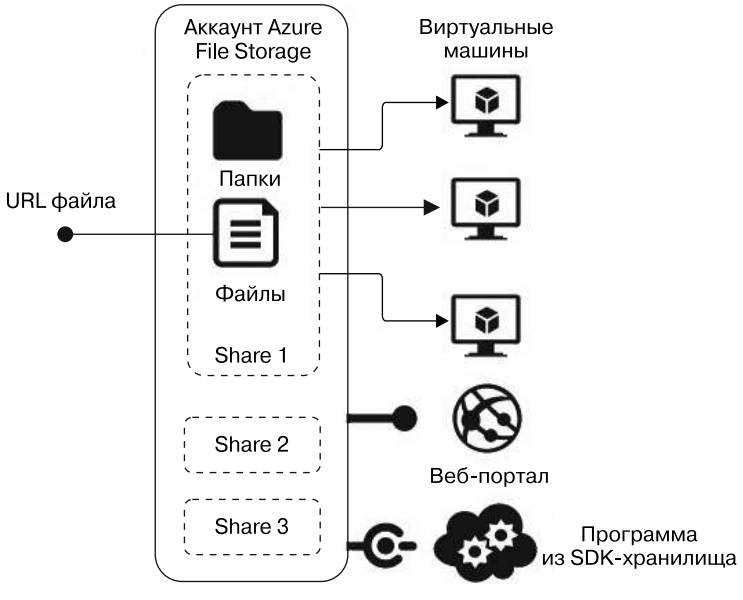


Рис. 4.20. Способы доступа к облачному файловому хранилищу Azure File Storage

Итак, сервис *Azure File Storage* составляет часть Azure Storage Account. Каждый Storage Account может включать в себя одну или несколько шар (share). Чтобы создать новую шару, необходимо с общей панели (см. рис. 4.7) добавить вкладку, нажав + File share. В появившейся форме (рис. 4.21) указывается имя шары (Name) — в нашем примере это `pokerfileshare` — и ее размер (Quota), в данном случае 10 Гбайт. Размер каждой шары (квота) может изменяться (с помощью веб-портала, SDK-управления облачными ресурсами или Azure CLI) и способна достигать 5 Тбайт.

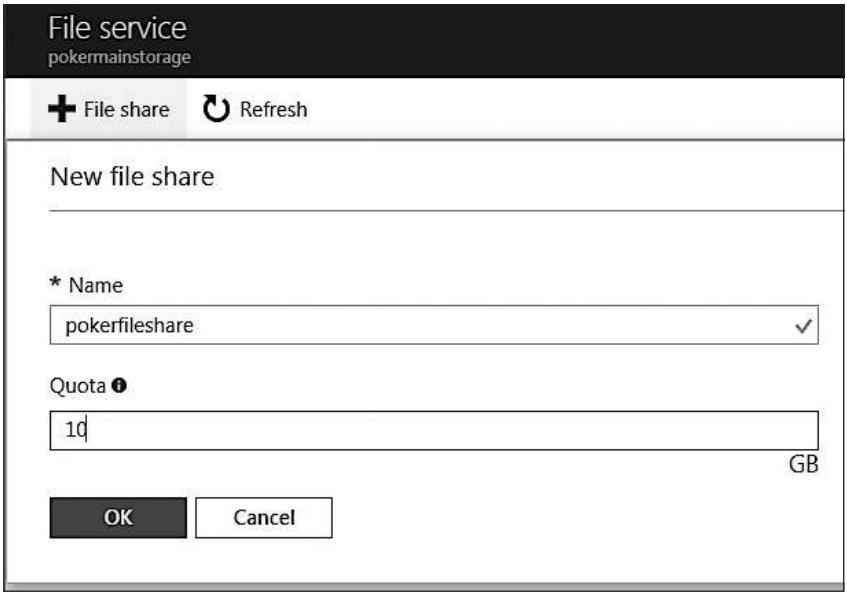


Рис. 4.21. Форма добавления новой файловой шары

Ключевым отличием шары от контейнера является то, что для нее устанавливается квота — верхний предел размера. Эта шара представляет собой корневой каталог, который доступен по протоколу SMB 3.0 и может быть примонтирован к виртуальным машинам в том же аккаунте Azure и в том же регионе, что и Azure File Storage.

Наиболее важными опциями конфигурирования является возможность подключения (Connect) к виртуальной машине и создание мгновенного снимка (View snapshot) (рис. 4.22).

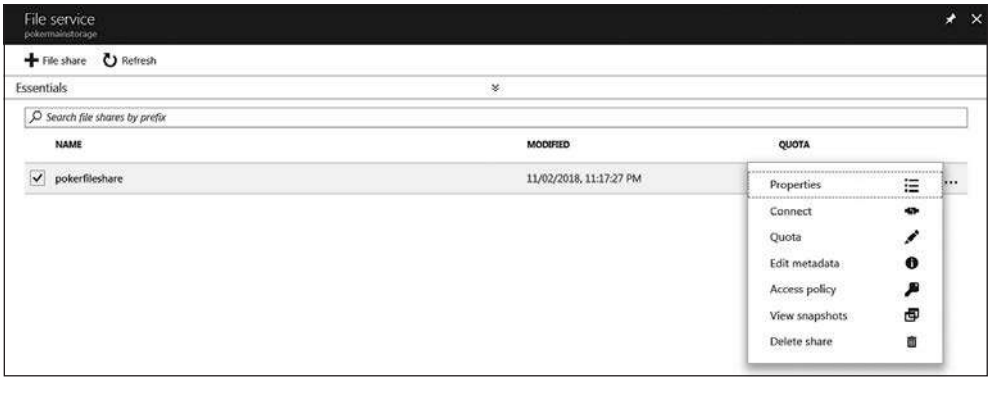


Рис. 4.22. Доступные опции конфигурирования файловой шары

Чтобы подключить шару File Storage к виртуальной машине, в оболочке виртуальной машины нужно выполнить команду монтирования сетевого диска. Эта команда может быть получена напрямую на веб-портале после щелчка на ссылке Connect (рис. 4.23).

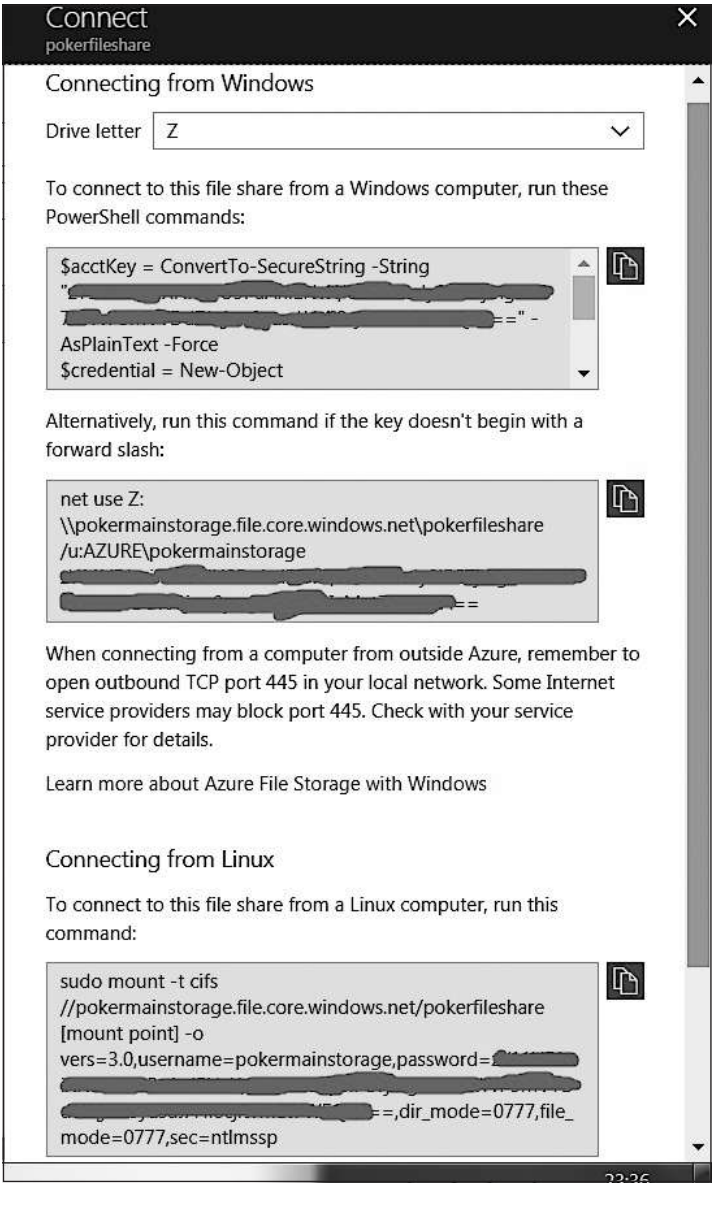


Рис. 4.23. Вкладка Connect Azure File Storage

Достоинства файлового хранилища представлены ниже.

- Возможность прямой миграции файловой системы из локального хранилища в облачное. Будет полностью сохранена иерархия этой системы (вероятные ограничения на символьную длину пути и глубину вложенности каталогов см. в документации).
- Простота взаимодействия из всех программных продуктов, реализующих стандартные интерфейсы ввода-вывода. Не требуется никаких модификаций кода программ, они могут быть устаревшими и все равно будут работать с Azure File Storage, поскольку это хранилище монтируется к основной файловой системе виртуальной машины на уровне операционной системы.
- Возможен просмотр списка файлов с помощью стандартных средств Windows или Linux.

Недостатки файлового хранилища:

- ограниченный размер файловой шары (5 Тбайт) и одного файла (1 Тбайт);
- более высокая по сравнению с BLOB Storage цена за гигабайт;
- меньшая по сравнению с BLOB Storage производительность операций чтения-записи.

4.4. Облачные хранилища AWS

> — Доллары в вентиляции, — задумчиво сказал первый и спросил Никанора Ивановича мягко и вежливо: — Ваш пакетик?
>
> — Нет! — ответил Никанор Иванович страшным голосом. — Подбросили враги!
>
> *Михаил Булгаков. Мастер и Маргарита*

Теперь рассмотрим облачные хранилища общего назначения, предоставляемые облачным провайдером Amazon Web Service. В отличие от Azure у AWS хранилища не объединяются логически в Storage Account, а представляют собой набор отдельных сервисов. Хранение двоичных объектов обеспечивается сервисом *AWS S3 — Amazon Simple Storage Service*. Экземпляр сервиса AWS S3 может быть логически разбит на *бакеты* (buckets, дословный перевод — «ведра»), которые будут определять видимость объектов в них, шифрование, жизненный цикл и пр.

Каждый объект (будь то бакет или хранящийся в нем файл) содержит ассоциированный с ним *класс хранения* (storage class). Ниже перечислены существующие классы хранения.

- *Стандартный* (STANDARD) — класс, задаваемый по умолчанию. Предназначен для случая, когда частота доступа к файлам высокая, в связи с чем требуется высокая производительность операций чтения и записи.
- *Стандартный с нечастым доступом* (STANDARD_IA, Standard Infrequent Access) — предназначен для долговременного хранения файлов, доступ к которым будет производиться нечасто (например, бэкапы), но тем не менее высокая производительность операций чтения-записи все еще важна. Оба класса — STANDARD и STANDARD_IA — обеспечивают доступ к файлу в режиме, близком к реальному времени, то есть с минимальной задержкой между запросом и получением данных.
- *Замороженный*, или, точнее, *ледник* (GLACIER), — предназначен для файлов большого размера, доступ к которым будет запрашиваться очень редко (архивы, бэкапы и др.). Принципиальное его отличие от двух предыдущих классов хранения — невозможность прямого доступа к файлам, которые перед выгрузкой должны быть сначала восстановлены в другие классы.
- *С уменьшенной избыточностью* (REDUCED_REDUNDANCY) — предназначен для хранения некритических файлов с той же скоростью доступа, что и у STANDARD, но за счет уменьшенной избыточности допускается потеря 0,01 % объектов.

Принципиальное различие между всеми классами — их стоимость. Очевидно, STANDARD обладает наивысшей стоимостью, а GLACIER и REDUCED_REDUNDANCY — наименьшей. В ряде сценариев перемещение файлов из одного класса в другой может быть широко используемой операцией. Например, это происходит на фотохостинге, размещающем картинки, доступные для скачивания по ссылке. Широко известен интернет-феномен мемов, когда контент внезапно становится очень популярным и в течение нескольких месяцев количество его скачиваний может резко возрасти, а затем значительно упасть. Если у этого фотохостинга большой объем хранимого контента, то становится актуальной проблема оптимизации стоимости хранения. Решить ее призван специальный встроенный механизм AWS S3: *управление жизненным циклом объектов* (object lifecycle management).

Данный сервис автоматизирует перемещение объектов между классами хранения по цепочке STANDARD ▸ STANDARD_IA ▸ GLACIER, что избавляет пользователя от необходимости строить специальные сервисы, ответственные за это перемещение. Следующий сценарий применения этого сервиса — жизненный цикл бэкапа. Это необходимо для оптимизации системы хранения бэкапа с точки зрения стоимости/доступности. Дело в том, что при необходимости восстановить информацию в системе с бóльшей долей вероятности понадобится последний по времени бэкап. Но, с другой стороны, все бэкапы следует хранить достаточно долго. Содержать их все в одном классе хранения довольно затратно, и перемещение более старых версий в более дешевые классы позволит снизить стоимость системы.

Рассмотрим, как создавать бакеты и манипулировать объектами в них. Начальная страница сервиса AWS S3 выглядит следующим образом (рис. 4.24).

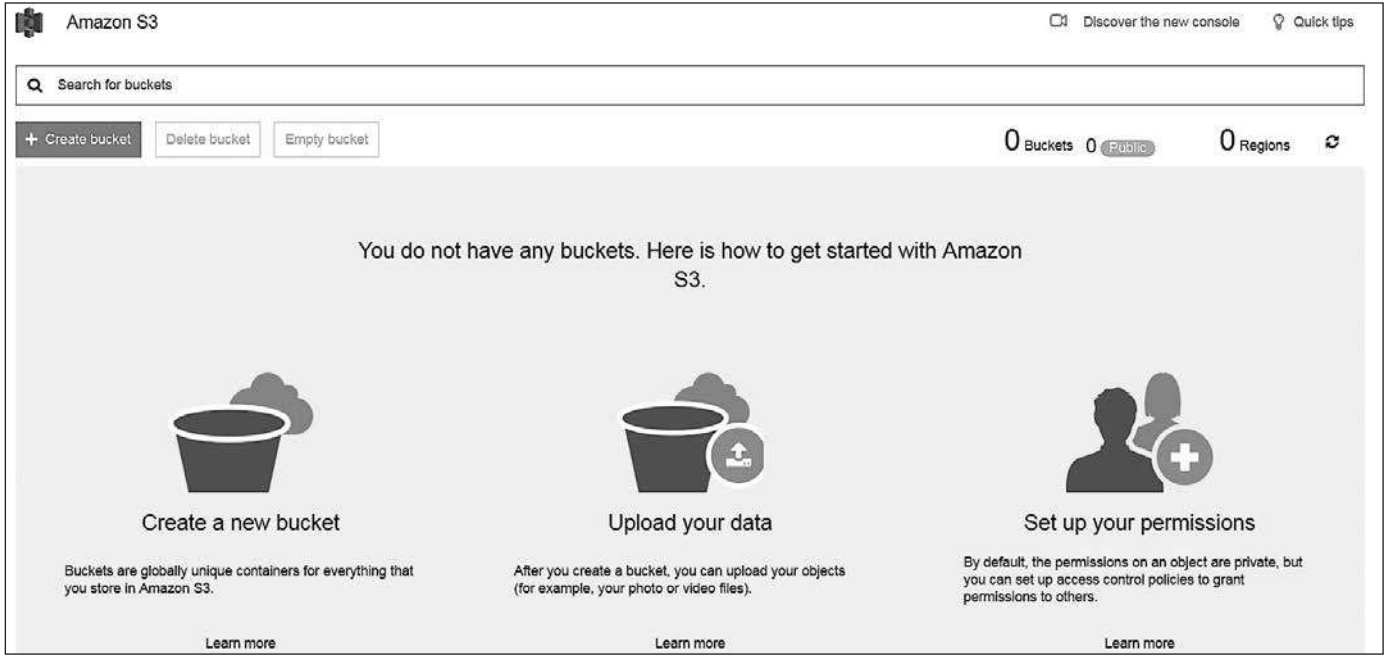


Рис. 4.24. Стартовая страница сервиса AWS S3

Чтобы создать бакет, надо щелкнуть на ссылке + Create bucket в левом верхнем углу. В результате откроется следующая форма (правая половинка доступна после успешного заполнения всех полей первой формы — вкладки Name and region (Имя и регион)) (рис. 4.25).

Остановимся подробнее на вкладке Set properties (Настройка свойств). Она содержит следующие вкладки:

- настройки возможности хранения нескольких версий одного файла (Versioning);
- логирование на уровне объекта (Object-level logging) с помощью сервиса AWS CloudTrail;
- логирование доступа на уровне сервера (Server access logging), используемое для детального анализа статистики запросов на получение объектов (широко применяется для интеграции AWS S3 с платежными сервисами);
- теги (Tags) — добавление тегов в виде набора «ключ — значение»;
- шифрование (Encryption) не показано на снимке экрана. Обеспечивает следующие возможности: отсутствие шифрования (по умолчанию), AES-256 (шифрование объектов с помощью ключей, предоставляемых сервисом Amazon S3), AWS KMS (шифрование объектов с применением ключей, предоставляемых сервисом AWS KMS). В последнем случае возможно выборочное участие пользователя в управлении ключами.

Следующие шаги (рис. 4.26) включают в себя настройку прав доступа к бакету (3 Set permissions) и финальный обзор/редактирование свойств бакета (4 Review). Рассмотрим подробнее управление правами. По умолчанию тип доступа к бакету — Private, то есть бакет доступен только для создателя, который вошел в консоль AWS. Секция Manage Users позволяет управлять привилегиями доступа для различных пользователей аккаунта в рамках этого бакета. Кроме того, можно добавлять иные аккаунты AWS, которые также могут иметь доступ к бакетам текущего аккаунта. Здесь же можно установить права для публичного доступа к объектам. На шаге 4 производится финальный обзор бакета.

После создания бакет имеет следующие свойства (рис. 4.27).

Как видите, в области Permissions можно настроить CORS для бакета.

Понять, как работать с бакетом в части управления объектами, поможет рис. 4.28.

Помимо этого, сервис S3 позволяет строить поверх себя HDFS-совместимые хранилища (будут описаны далее), например EMRFS. В дополнение к основной функции хранения объектов, AWS S3 также обеспечивает ряд интересных возможностей:

- интеграцию с Amazon DevPay — биллинговой системой Amazon, позволяющей монетизировать контент, размещенный в AWS S3 и доступный для скачивания;
- встроенную поддержку peer-to-peer протокола BitTorrent.

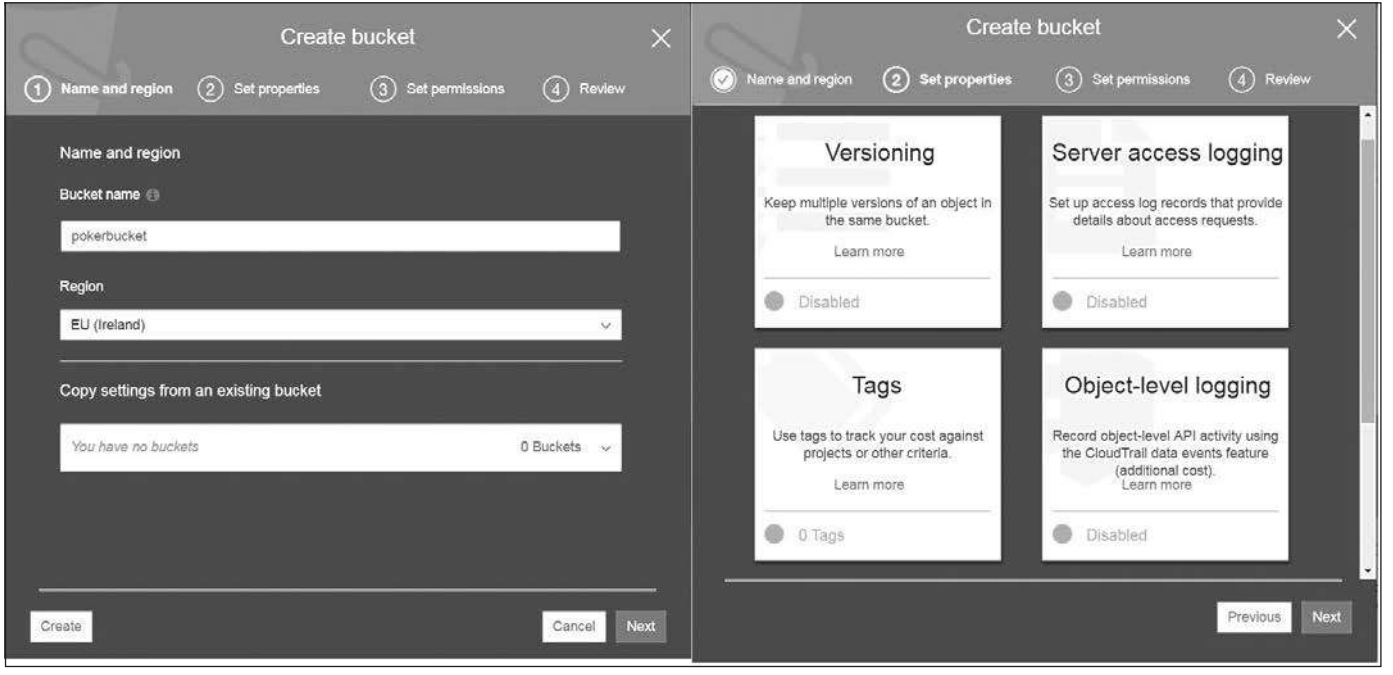


Рис. 4.25. Последовательные шаги создания бакета: вкладки Name and region (Имя и регион) и Set properties (Настройка свойств)

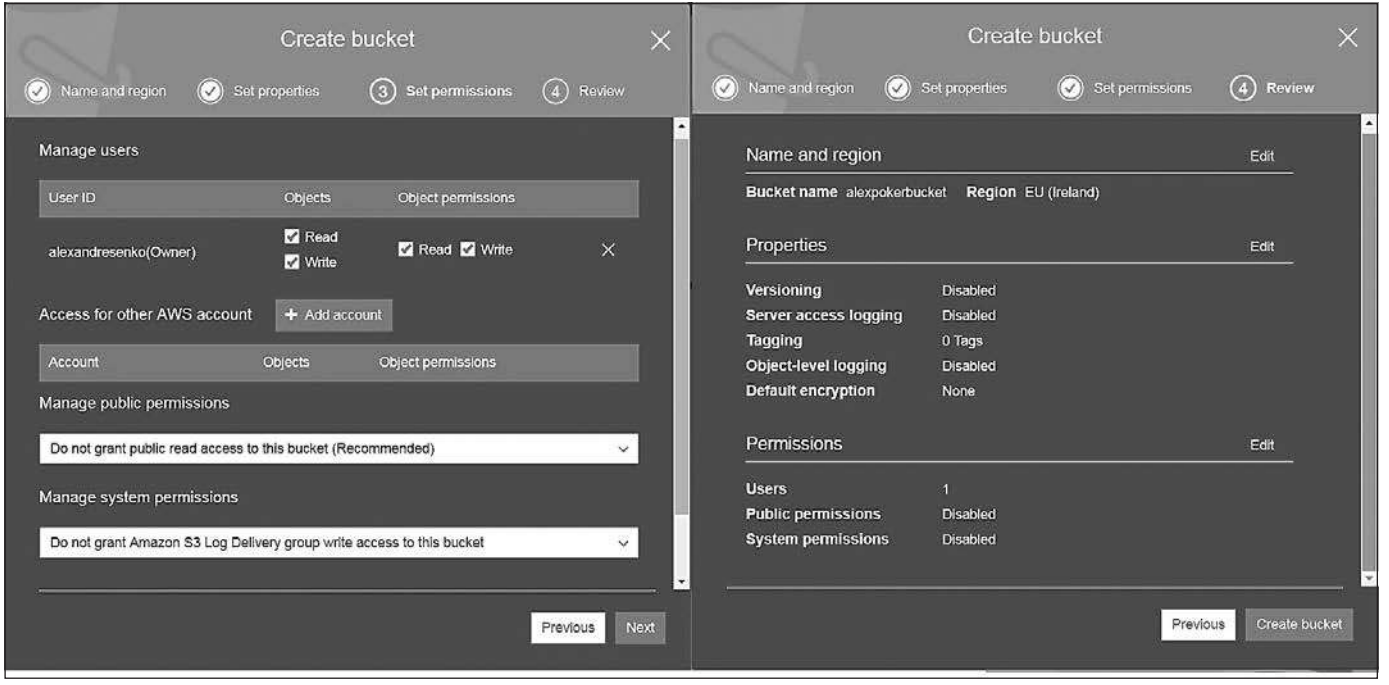


Рис. 4.26. Дальнейшие шаги создания бакета: Set permissions (Настройка прав доступа) и Review (Финальный обзор)

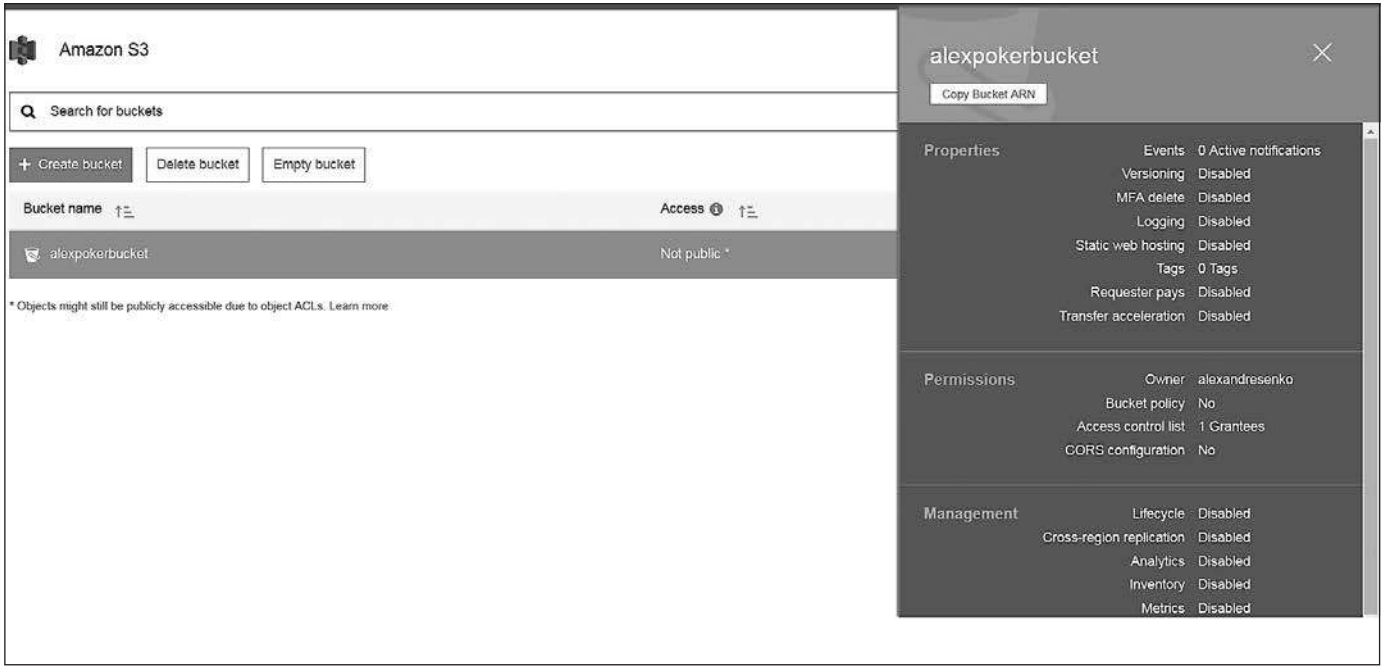


Рис. 4.27. Созданный бакет с раскрытой формой свойств

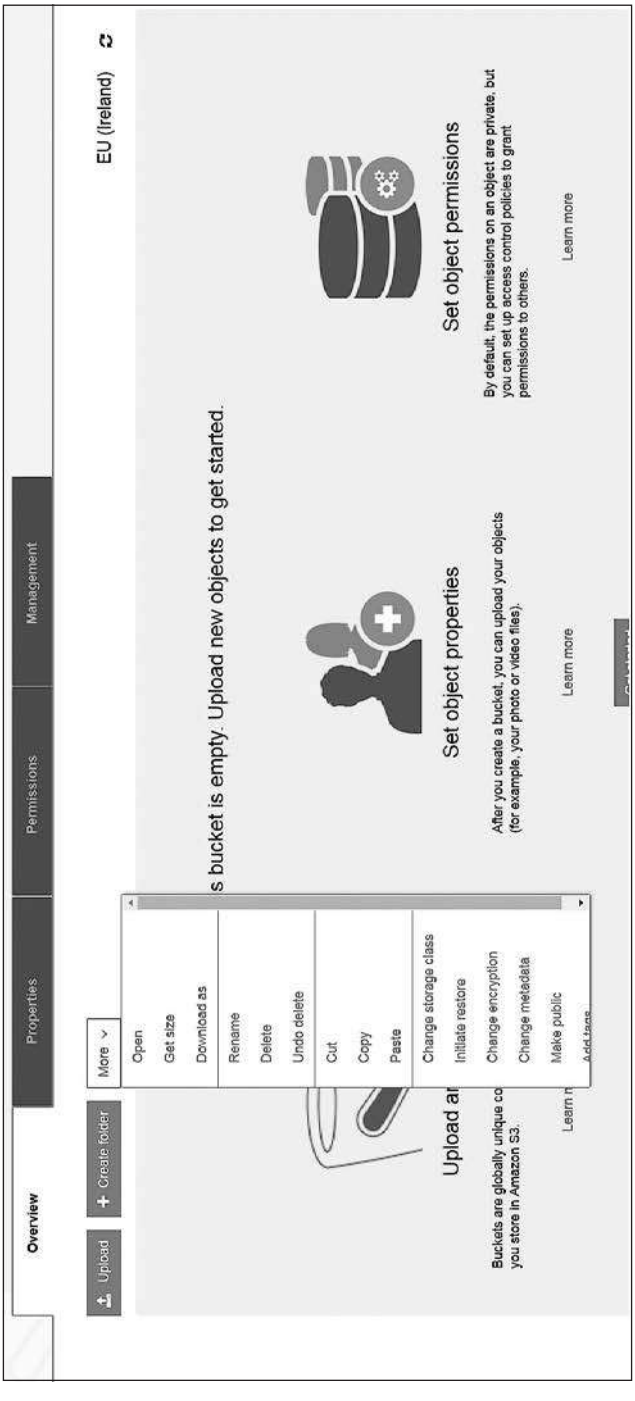


Рис. 4.28. Вкладка бакета, позволяющая управлять объектами

Следующее хранилище файлов, предоставляемое AWS, — *Amazon Elastic File Service* (EFS). Этот сервис является облачной реализацей сетевого хранилища с поддержкой протокола NFSv4. Он изначально предназначен для доступа с виртуальных машин AWS EC2. Файлы и метаданные хранятся в виде нескольких копий в разных зонах доступности (availability zone) в пределах одного региона. Масштабирование хранилища происходит вплоть до петабайтных размеров. Поскольку этот сервис тесно связан с AWS EC2, то для его конфигурирования требуется настраивать сетевые IaaS-сервисы — VPC, подсети, availability zone, сетевые группы безопасности (рис. 4.29).

На следующей вкладке указываем теги, уровень производительности (General purpose или Max I/O) и шифрование (Enable encryption) (рис. 4.30).

Далее выполняем обзор параметров создаваемого сервиса и в итоге имеем готовый сервис, который можно подключать к виртуальной машине AWS EC2 (рис. 4.31).

Чтобы получить инструкции по подключению файловой системы к виртуальной машине, следует нажать на ссылку Amazon EC2 mount instructions или AWS Direct Connect mount instructions.

Интересная возможность AWS EFS — функция EFS FileSync, позволяющая синхронизировать файлы в EFS и файлы в локальной файловой системе как EC2, так и физического компьютера вне AWS. В последнем случае нужно установить специальный агент на тот компьютер, к которому будут подключаться диски, требующие синхронизации.

В отличие от Azure, где для хранения дисков виртуальных машин используется обычный аккаунт BLOB Storage, в котором хранятся BLOB-объекты с блочной организацией, в Amazon применяется специальный сервис — AWS EBS (Elastic Block Storage). Этот сервис отвечает за создание, размещение, шифрование, подключение к виртуальным машинам, создание бэкапов в виде мгновенных снимков (snapshots). Чаще всего пользователь не будет взаимодействовать с ним напрямую, этот сервис доступен в «связке» с другими: AWS EC2, AWS ECS и пр. То есть при создании виртуальной машины тип и количество виртуальных дисков, их точки монтирования или метки указывается прямо в момент создания. Однако нужно помнить, что бывают сценарии, когда эти виртуальные диски *не* удаляются при удалении виртуальной машины. Например, пользователь забыл установить соответствующий флажок или чаще такое происходит при операциях с кластером ECS. Экземпляры уходят, а диски остаются, и пользователь за них продолжает платить. Помните об этом!

Диски EBS можно разделить на две большие группы: HDD (Hard Disk Drive) (классический магнитный диск) и SSD (Solid State Drive) (твердотельный диск). Обе группы имеют следующие разновидности.

- *Холодный HDD* (Cold HDD) — самый дешевый тип диска EBS. Сценарии его использования включают хранение больших объемов данных, которые редко запрашиваются извне в случаях, когда стоимость хранения имеет значение. Применение этого типа сопряжено с рядом ограничений: загрузочный (boot) диск его не поддерживает. Кроме того, существуют разного рода ограничения для использования холодного HDD вместе с ECS-оптимизированными виртуальными машинами.

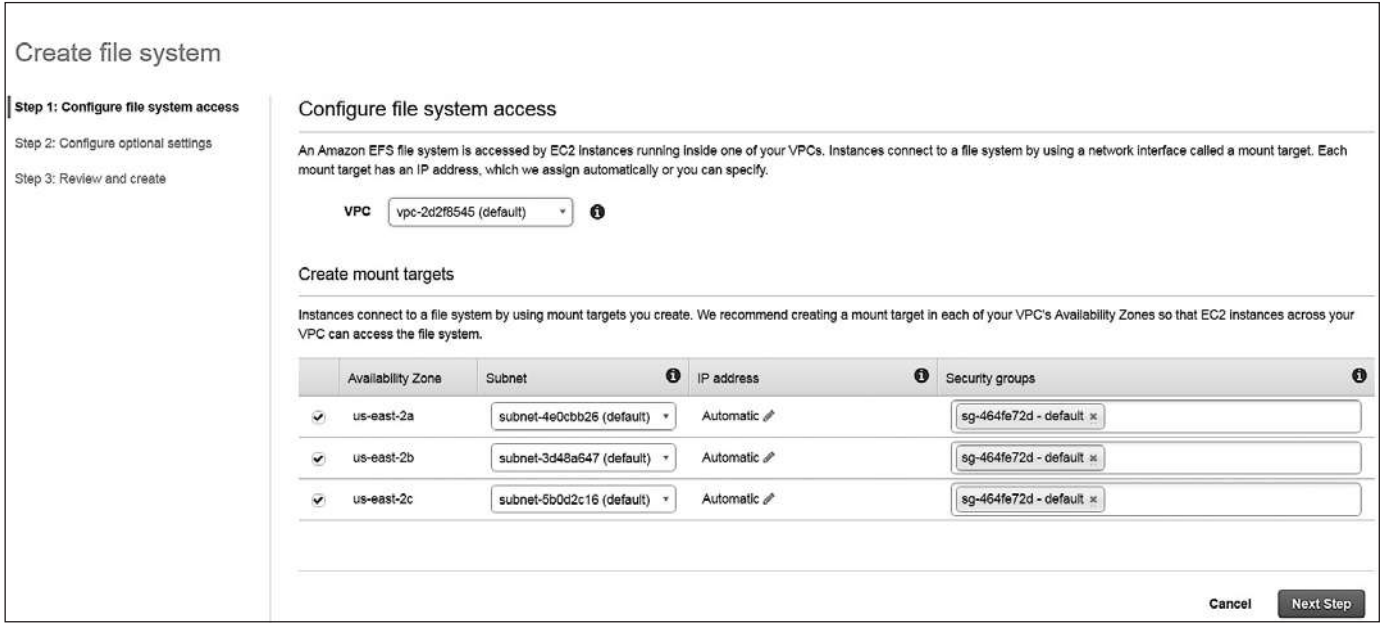


Рис. 4.29. Начальная страница конфигурирования сервиса AWS EFS

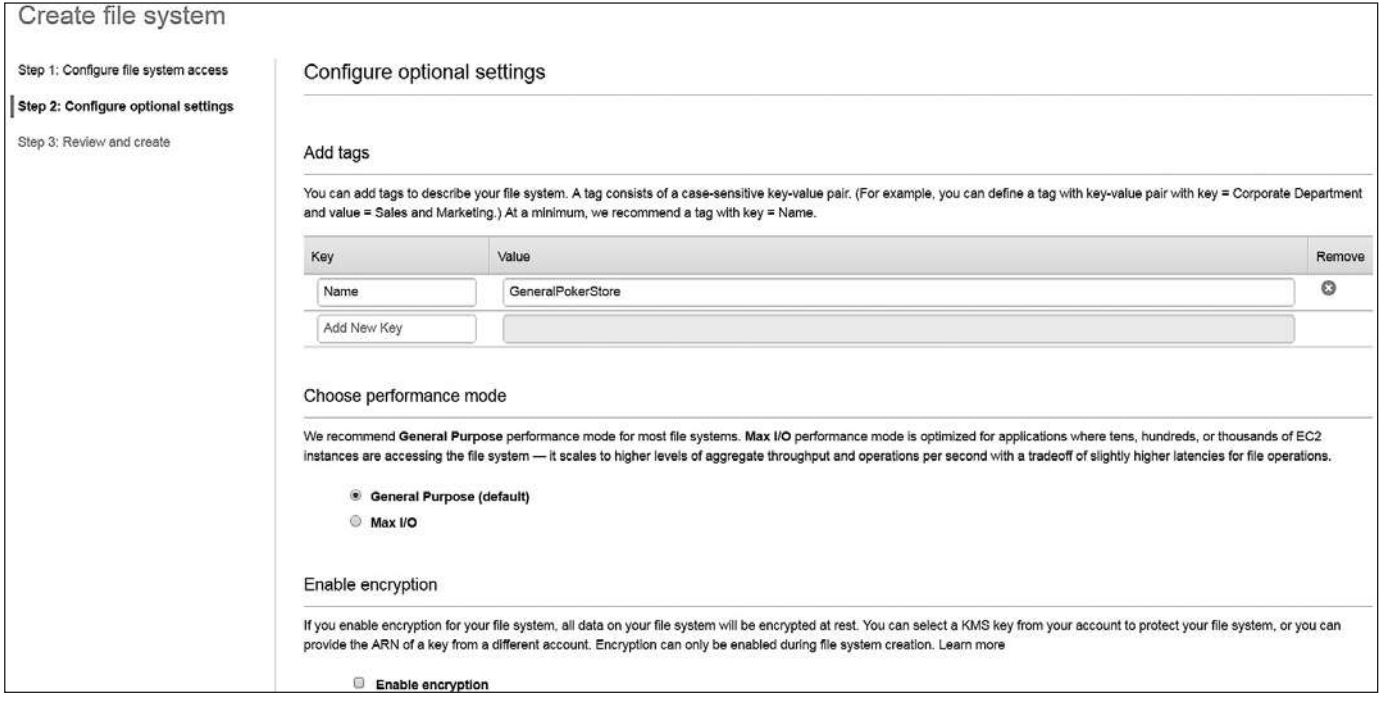


Рис. 4.30. Дальнейшее конфигурирование AWS EFS

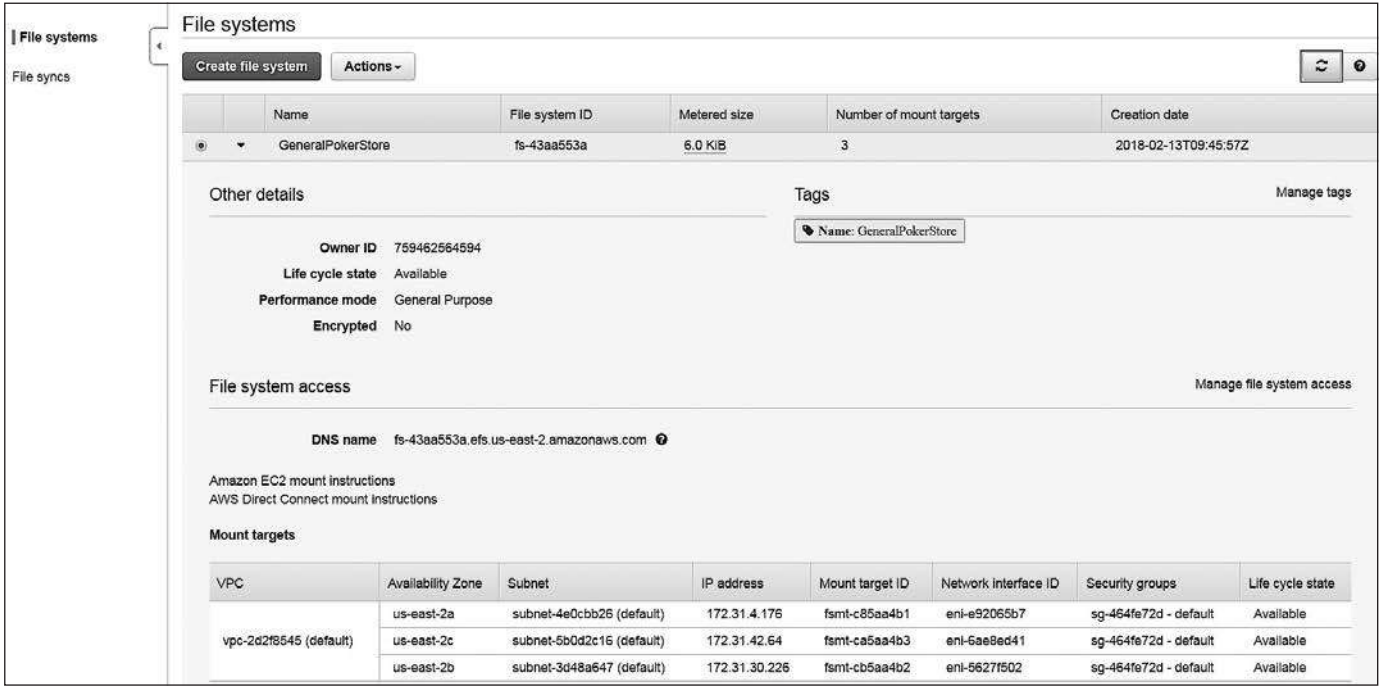


Рис. 4.31. Созданная файловая система AWS EFS, готовая к монтированию

- *Горячий HDD* (throughput oriented HDD, дословно «HDD с оптимизированной производительностью») — магнитный диск малой стоимости, оптимизированный для сценариев хранения больших объемов данных, частота запросов которых высока и для которых требуется высокая производительность при минимальной цене. В документации AWS EBS указаны следующие сценарии применения этого типа: потоковые чтение-запись с высоким и постоянным уровнем производительности при минимальной цене; хранение и оперирование большими данными, в том числе складирование данных, построение DataWarehouse и обработка логов. Так же как и Cold HDD, горячий HDD не может быть загрузочным (boot).
- *SSD общего назначения* (general purpose SSD) — тип твердотельного диска, подходящий для широкого класса сценариев, включая диск для операционной системы. Его производительность, выраженная в величинах IOPS (input output per second — операции ввода/вывода в секунду), в два раза превышает таковую у горячего HDD. Этот тип диска рекомендуется для большинства случаев использования.
- *SSD с выделенной полосой пропускания* (provisioned IOPS SSD) — тип твердотельного диска, обладающий наибольшей производительностью чтения-записи (и как результат, наибольшей стоимостью), предназначенный для построения критически важных компонентов с жесткими требованиями к задержкам и производительности. В качестве типовых сценариев применения этого типа можно указать высоконагруженные диски баз данных (включая MongoDB, Cassandra, Microsoft SQL Server, MySQL, PostgreSQL, Oracle и др.).

По поводу всех представленных типов нужно сделать ряд замечаний. Прежде всего, максимальная производительность диска достигается не всегда, а только для определенного высокопроизводительного экземпляра EC2. Само же выделение квоты IOPS на том диска и на виртуальную машину у AWS — весьма запутанный процесс, который требует отдельного описания. Помимо этого, производительность дисковых операций можно существенно повысить с помощью соединения дисков в массивы RAID. Однако максимальное количество дисков в массиве и суммарный их объем сложным образом зависит от типа экземпляра.

4.5. Резюме

В этой главе были рассмотрены облачные сервисы хранения файлов общего назначения от Azure и AWS, а также сервисы сетевых файловых систем хранилища виртуальных жестких дисков. Их можно сравнивать между собой в двух плоскостях: сервисы разного типа и однотипные сервисы у различных облачных провайдеров. Сначала сравним между собой разнотипные сервисы.

Сервисы хранения файлов общего назначения наиболее удобны для построения дешевых сервисов хранения файлов большого объема или большого количества файлов. Способность загрузки файлов извне по прямой ссылке, в том числе с ре-

гулируемым временем жизни, позволяет строить системы типа файлообменника. Кроме того, файловое хранилище общего назначения очень удобно для хранения логов от всех облачных сервисов. Верхние пределы количества файлов в таком хранилище и его суммарный объем весьма высоки (речь идет о многих терабайтах и петабайтах), что вкупе с низкой стоимостью делает возможным хранение огромных объемов информации с высокой доступностью и приемлемой стоимостью.

В то же время при необходимости миграции в облако существующей информационной системы для организации файлового хранилища на основе облачного нужно переписывать программный код приложений этой системы. Это обусловлено тем, что программный доступ к такому типу хранилищ возможен только с помощью соответствующего SDK (Azure и AWS). Иногда получить его трудно или невозможно из-за «распыления» логики по многим программным компонентам (часть из которых могут быть, например, сценариями командной оболочки). Для этого случая и предусмотрены сервисы облачных файловых систем. Их можно примонтировать к основной файловой системе и использовать из любой программы, прибегнув к стандартным библиотекам программного ввода-вывода уровня операционной системы. Хранить файлы в таких системах значительно дешевле, чем задействовать виртуальные жесткие диски. Кроме того, например, для Azure File Storage имеется возможность предоставить доступ к файлам через SDK и даже открыть им доступ извне по прямой ссылке. Следующее преимущество облачных файловых систем — их динамическая масштабируемость в сторону увеличения и уменьшения. Для виртуальных жестких дисков эта операция сопряжена с рядом трудностей (порой необходимо их отключить (detach), изменить размер и потом заново присоединить).

Но иногда без использования виртуальных жестких дисков для хранения данных не обойтись: например, при наличии жестких требований к производительности файловой системы и времени доступа. Скажем, в случае установленной на виртуальной машине базы данных единственно верный выбор для их хранения — это виртуальные жесткие диски. Выбор типа жесткого диска (для AWS) или ценового уровня хранилища (для Azure) позволяет обеспечить заданное соотношение «производительность/цена». При этом не стоит забывать, что высокая производительность обеспечивается не дисками самими по себе, но комбинацией с виртуальной машиной. Эта виртуальная машина должна иметь соответствующий тип (ценовой уровень). Если имеющейся комбинации параметров не удается достигнуть заданного отношения «производительность/цена», то следует использовать конфигурации из дисков в виде массивов RAID.

Существенный недостаток хранения информации на виртуальном жестком диске — к ней невозможен прямой доступ. Необходимо использовать программы, работающие на виртуальной машине, к которой подключен такой диск. Например, на виртуальной машине может быть установлен сервер баз данных, чьи данные хранятся на виртуальном диске. В случае же файлового хранилища сервер должен быть сконфигурирован как файловый вручную. Следующий значительный недостаток такой системы хранения — меньшая надежность и доступность, поскольку

за согласованность и доступность в данном случае отвечают программы пользователя, размещенные на его виртуальной машине. А для поддержания заданного уровня надежности и доступности необходимо выполнить дублирование, троирование и т. д., что потенциально ведет к сложностям в хранении и обеспечении согласованности данных и требует больших первоначальных затрат на настройку и конфигурирование такой системы.

Таким образом, если нашей целью является широкодоступное дешевое хранилище очень больших данных, легко сопрягаемое с другими облачными средами, то выбор однозначно падает на облачные хранилища общего назначения Azure Blob Storage и AWS S3. Оба хранилища поддерживают встроенную интеграцию с сервисами копирования и трансформации данных (Azure DataFactory, AWS EMR/Data Pipeline), сервисами анализа (AWS Athena, Azure Stream Analytics Blob triggering) и облачными сервисами — обертками для компонентов экосистемы Apache Hadoop (AWS EMR и Azure HDInsight), причем последние зачастую используют технологию «HDFS поверх S3/Blobs». При необходимости хранить информацию в базах данных, размещенных на виртуальных машинах, следует выбирать виртуальные диски. Облачные файловые системы можно задействовать в промежуточном случае — для хранения большого количества файлов (именно отдельных файлов, а не единиц хранения в БД) при невозможности применять хранилища общего назначения, но при этом теряется существенное преимущество интеграции с облачными сервисами и одновременно возрастает цена.

Теперь попробуем бегло сравнить однотипные сервисы у различных провайдеров.

Azure Blob Storage предоставляет в основном те же возможности, что и AWS S3. Концептуально Blob Container полностью соответствует AWS S3 Bucket. Различаются они контролем уровня производительности: для Azure он настраивается для всего Storage Account в целом, а для AWS S3 класс хранения может быть различным для разных файлов в пределах одного бакета. Это, в частности, позволяет настраивать перемещение файлов между разными классами хранения в автоматическом режиме. Для Azure такая процедура будет состоять в перемещении файлов между различными Storage Account с помощью сторонней программы. AWS S3 предоставляет ряд возможностей, которых нет у Azure Blob (например, поддержку BitTorrent и версионирование), но у Azure гораздо более удобная система мониторинга. Кроме того, для AWS несколько проще настроить доступ к бакетам из другого аккаунта (это можно сделать прямо при создании бакетов). В случае же Azure требуется синхронизировать активные каталоги в разных аккаунтах и предоставить необходимые права соответствующим пользователям на уровне AD в целом, что для среднестатистического пользователя облачного аккаунта может быть непросто даже после прочтения документации.

Сервисы облачного файлового хранилища у обоих облачных провайдеров значительно различаются. Если Azure File Storage позволяет создавать шару файловой системы с указанием верхнего размера (квоты), то у AWS EFS просто создается файловая система без лимита. Монтирование обоих файловых систем сводит-

ся к выполнению команд на целевых виртуальных машинах, но в случае Azure File Storage неявно подразумевается, что и файловая система, и виртуальная машина должны находиться в одном и том же регионе и в одном и том же аккаунте. Для AWS EFS необходимо явное конфигурирование виртуальной сети, подсетей, групп безопасности и зон доступности.

Отличается и внешний доступ к файлам. Для Azure FileStorage можно получить доступ к файлам с помощью SDK и прямой внешней ссылки. Это значит, что возможно построение гибридного хранилища: с одной стороны, оно может быть использовано как сетевое, подключенное к виртуальной машине, а с другой — предоставлять извне доступ к файлам точно таким же образом, как это делает Azure Blob. Для сценария миграции информационных систем в облако описанный путь является чрезвычайно удобным и плодотворным, поскольку в подобном случае имеется возможность, с одной стороны, не вносить никаких изменений в устаревший код, оставив только стандартные операторы работы с файлами на уровне операционной системы, а с другой — предоставить внешним анонимным пользователям функциональность облачного хранилища общего назначения. Всех этих особенностей лишен сервис AWS EFS, который представляет собой просто облачный сервис сетевого хранилища, без каких бы то ни было возможностей доступа к файлам, кроме как из примонтированной виртуальной машины.

Я хотел бы воздержаться от сравнения виртуальных жестких дисков у обоих облачных провайдеров. Эта тема требует дотошного сравнения однотипных виртуальных машин с однотипными интерфейсами (которых в общем случае нет) у обоих облачных провайдеров и в целом лишена смысла. Оба провайдера предоставляют богатый набор виртуальных машин IaaS, и пользователь в пределах своих возможностей и способностей может сконфигурировать из них то, что ему нужно. Но детальное сравнение нюансов производительности, пользовательских соглашений выходит далеко за рамки книги. Эта тема относится к информации о построении инфраструктуры в облаках, которая не может быть упрощена до уровня сравнения «на пальцах» и требует отдельной главы, а возможно, и части соответствующей книги.

5 Реляционные базы данных

Предпочитая быть романтиком
Во время тягостных решений,
Всегда завязывал я бантиком
Концы любовных отношений.

Игорь Губерман

5.1. Общие сведения о реляционных базах данных

Реляционные базы данных (РБД) на сегодняшний день самый распространенный вид баз. Они подходят для хранения данных в совершенно разных информационных системах благодаря своей универсальности и удобству. Информация в них логически представлена в виде набора таблиц, состоящих из строк и столбцов. Эти таблицы связаны друг с другом с помощью ключей — специальных ограничений целостности, соотносящих строки одной таблицы со строками другой.

Рассмотрим, в чем заключаются универсальность и удобство РБД. Универсальность состоит в том, что информацию об объектах и событиях реального мира можно представить как записи таблицы (что и было показано в предыдущей части книги). Да, собственно, любые списки любых объектов (клиентов, адресов, заказов и др.) — наиболее широко используемая абстракция в информационных системах. И точно так же, как и между объектами в реальном мире, между таблицами могут существовать различные связи и отношения. Потому-то РБД так и называются (relational — «реляционные»). Типы связей могут быть следующими: «один к одному», «один ко многим» и «многие ко многим».

«Один к одному» — одной записи в таблице соответствует строго одна запись в другой. Пример — связи между таблицами клиентов и таблицей адресов при условии, что у каждого клиента один адрес (рис. 5.1).

Рис. 5.1. Отношения между таблицами «один к одному»

«Один ко многим» — одной записи в таблице А может соответствовать одна или несколько записей в таблице Б, но каждая запись таблицы Б связана ровно с одной записью таблицы А. Пример — связь таблицы клиентов с таблицей заказов при условии, что клиенты могут делать много заказов. Но при этом каждый заказ относится строго к одному клиенту (рис. 5.2).

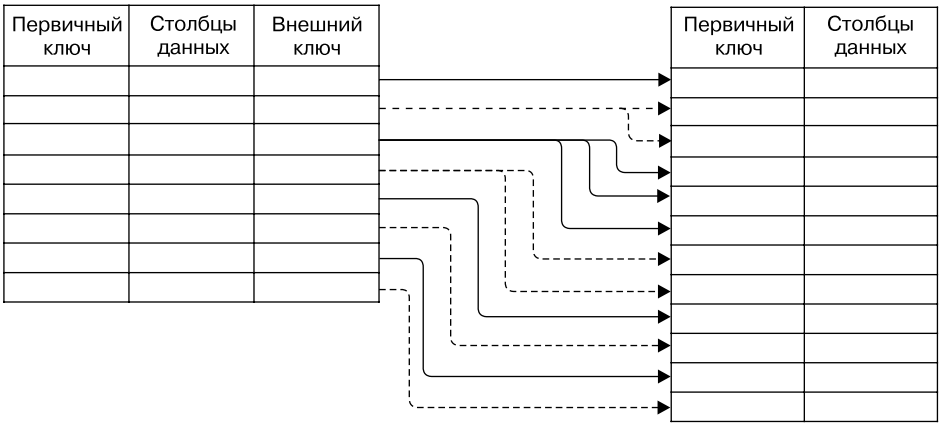


Рис. 5.2. Отношения между таблицами «один ко многим»

«Многие ко многим» — одной записи в таблице А могут соответствовать одна или несколько записей в таблице Б и наоборот. Пример — отношения, моделируемые таблицей студентов и таблицей учебных предметов. Очевидно, что на каждый предмет может быть записано несколько студентов, равно как каждый студент должен пройти обучение по нескольким предметам. Вообще, прямое использование такой структуры встречается редко, поскольку ведет к запутанной топологии и проблемам с производительностью при выполнении запросов данных. Чаще всего применяется промежуточная таблица-связка, состоящая как минимум из трех полей: первичного ключа (номера записи в таблицы связки), внешнего ключа таблицы А (номера студента) и внешнего ключа таблицы Б (номера курса).

Записей с одинаковым ключом таблицы А и записей с одинаковым ключом таблицы Б может быть много. Но невозможна и бессмысленна ситуация, когда есть несколько повторяющихся одинаковых пар «ключ А — ключ Б». Очевидно, что студент может быть только один раз записан на предмет и предмету может обучаться только один конкретный студент Иван Иванович Иванов. В этом случае можно задействовать составной ключ для таблицы-связки — то есть использовать не отдельное поле в качестве первичного ключа, а связку «ключ А — ключ Б».

Но данный подход не всегда хорош. Предположим, таблица-связка хранит полную историю всех курсов для всех выпусков. Для этого ей необходимо добавить такие поля, как дата записи на курс, дата отчисления с курса, статус, результат экзамена/зачета и пр. (или добавить внешний ключ на запись в другой таблице, которая будет содержать все необходимые сведения). В таком случае для студентов-второгодников возможно наличие дубликатов в парах «ключ А — ключ Б». В подобной ситуации, безусловно, необходим отдельный первичный ключ.

Разбирая пример моделирования отношений «многие ко многим», я столкнулся с рядом проблем, стоящих перед разработчиками баз данных. Как соотнести сущности реального мира и таблицы БД? Какие поля выбрать в качестве ключевых (использовать отдельное поле или несколько полей как составной ключ)? Какие отношения будут между таблицами? Следует ли добавлять таблицы-связки для облегчения построения связей? Помимо этого, данные нужно *нормализовать*, то есть устранить их избыточность (на самом деле нормализация — гораздо более сложное и комплексное понятие, но объем книги не позволяет описать его подробнее).

Одно из ключевых понятий РБД — схема: это, по сути, шаблон, которому должны соответствовать таблицы. Он включает в себя список столбцов таблицы с указанием имени столбца, типа данных, расширения типа (ненулевое поле, длина символов в текстовом поле и др.), ограничений (например, первичный ключ, внешний и пр.), индексов, примененных к таблице, и иных сведений. Кроме схемы таблицы, существует схема базы данных, представляющая собой еще и хранимые процедуры, триггеры, пользователей и прочие объекты. Схемы можно представить с помощью синтаксиса SQL в виде команд БД на создание, модифицирование или пересоздание объекта. Наличие строгого и однородного формата представления данных в РБД, с одной стороны, существенно упрощает взаимодействие с ними, а с другой — ставит определенные трудности при разработке информационных систем, заставляя добиваться максимальной нормализации таблиц, несмотря на хаотичность предметной области.

Чтобы обеспечить возможность выборки информации из таблиц, РБД содержат процессор языка запросов. Он компилирует команды языка запросов SQL в программные инструкции доступа к данным в таблицах. Сам синтаксис языка может отличаться у различных СУБД, которые предоставляют различные «диалекты» (например, PL/SQL, T-SQL и др.), но все эти вариации построены как расширения базового синтаксиса SQL.

РБД выполняют две основные функции. Во-первых, предоставляют хранилище данных, которые могут быть добавлены, обновлены или удалены с максимально

возможной защитой от несогласованности при одновременном доступе к ним из разных источников. Эта функция называется OLTP — оперативная транзакционная обработка данных. Во-вторых, РБД позволяют выбирать данные из различных таблиц, в том числе с целью представить результаты аналитических запросов (для целей BI, построения отчетов и пр.). Подобная функция называется OLAP — оперативная аналитическая обработка данных. В отличие от OLTP для OLAP на первое место выходит производительность запроса произвольной формы. И поскольку системы, оперирующие большими данными, хранят информацию в реляционной СУБД с целью выполнения аналитических запросов, а обновление самих данных является лишь подготовкой к этому анализу, то сосредоточимся на общих компонентах таблиц, ключевым образом влияющих на производительность запросов и их структуру, игнорируя вопросы согласованности данных.

Очень важный компонент таблицы — *первичный ключ*, обеспечивающий уникальность записи в таблице. Может быть представлен как отдельным полем, так и составным. В любом случае необходим механизм, обеспечивающий уникальность ключа. Можно задействовать сторонний сервис, который генерирует и хранит ключи для таблиц. Но в этом случае вся сложность по поддержанию целостности системы ложится на разработчика, использующего РБД. Для обеспечения уникальности ключа отлично подходит уникальный идентификатор — GUID, а также целочисленная величина. В ряде случаев подойдет временно́й штамп. Сложности поддержания целостности системы и уникальности ключа при использовании внешнего сервиса генерации ключей могут быть исключены в случае применения автоинкрементных типов данных для ключей, обеспечивающих уникальность на уровне БД.

Чтобы обеспечить связь строки одной таблицы со строкой или строками другой, необходимо иметь столбец, содержащий номер первичного ключа внешней таблицы. Такой номер называется *внешним ключом*.

Итак, мы в общих чертах разобрались, как логически представляются связанные между собой таблицы. Подобная структура позволяет создавать запросы данных как для одной таблицы, так и для группы таблиц, связанных между собой ключами. Именно наличие ключевого поля и связи между таблицами по ключам и позволяют строить высокопроизводительные запросы данных в таблицах.

Следующий очень важный момент, обеспечивающий высокую производительность запросов к одной или нескольким таблицам, — использование *индексов*. Рассмотрим, как это работает. Строки в таблице физически хранятся в произвольном порядке. Это значит, что для построения запроса на выборку данных, отвечающих какому-либо критерию фильтрации, необходимо последовательно просмотреть всю таблицу. При этом она может содержать очень большое количество строк, а скорость поиска будет линейной, то есть пропорциональной количеству строк O (N). Теперь представим ситуацию соединения таблиц в запросе. Каждой строке одной таблицы необходимо найти (помните о последовательном просмотре?) одну или несколько строк из другой таблицы, то есть количество операций просмотра здесь может быть большим, что уменьшает производительность запроса. (Конечно, в этом случае производительность будет гораздо выше O (MN), но при работе с таблицами, содержащими миллионы строк, запросы все равно будут крайне

медленными.) Индексация (создание индексов для определенных столбцов) как раз способна решить эту проблему.

Индексация — упорядочение строк таблицы и представление их в виде структуры данных, позволяющей применять быстрые алгоритмы поиска. Это могут быть те или иные разновидности сбалансированного дерева или хеш-таблиц, что в итоге дает скорость O (N Log N) и даже быстрее. Индекс, физически упорядочивающий строки таблицы, называется *кластерным* и у таблицы может быть только один. Но существуют также *некластерные* индексы, которые содержат только указатели на строки таблицы и не требуют физического упорядочения строк. В подобном случае индексы — это указатели на объекты в «куче» (строки в таблице). В отличие от кластерного индекса некластерных в таблице может быть много. Если строки таблицы в информационной системе интенсивно обновляются, удаляются и добавляются, то индексы необходимо периодически пересчитывать.

При добавлении индексов в таблицу нужно соблюдать осторожность, поскольку они увеличивают производительность в момент выполнения запросов чтения, но уменьшают скорость добавления и обновления новых строк. Кроме того, индексы занимают память в БД и при наличии объемных таблиц и большого количества проиндексированных столбцов занимают достаточно много места. В моей практике имеется случай, когда из-за того, что база данных хранила очень большое количество данных (таблицы содержали многие сотни миллионов строк) и эти данные требовались различным сервисам, таблицы базы включали множество индексов. И эти индексы со временем стали причиной достижения физических лимитов хранимого объема этой технологии базы данных. Проблему решили радикально, перейдя на иные принципы построения хранилища данных.

Концепция индексов очень удачная и используется также в других хранилищах данных, в том числе и нереляционных.

Помимо хранения данных в таблицах и выполнения запросов к ним, реляционные БД предоставляют транзакционный механизм добавления и обновления информации в них. Ключевым понятием тут является *транзакция*. В данном случае под транзакциями понимается группа операций (обновления, удаления и пр.) которые или выполняются все вместе или не выполняются вовсе. Это очень важное свойство информационной системы, обеспечивающее согласованность данных в условиях возможных сбоев. Если, например, при выполнении транзакции произошел сбой, то состояние таблиц «откатывается» к тому, которое было до начала ее выполнения. Требования к транзакционным системам можно выражать с помощью принципа *ACID*: atomicity — «атомарность», consistency — «согласованность», isolation — «изолированность» и durability — «устойчивость». Подробное описание различных уровней транзакции выходит за рамки этой книги.

Очень часто в информационных системах данные хранятся как набор множества предметно-ориентированных таблиц, каждая из которых представляет собой хранилище данных конкретной сущности в максимально нормализованном виде. Преимущество такой схемы в ее простоте, возможности простого использования совместно с объектно-реляционными библиотеками (например, EntityFramework в .NET). А ее недостатки заключаются в том, что запросы на выборку данных из

многих таблиц очень сложны, поскольку требуют соединения данных многих таблиц, да и производительность невелика, так как очень трудно подобрать набор индексов для всех таблиц запросов. Существуют схемы, более удобные для построения хранилища данных, и они будут подробнее рассмотрены в главе 7, посвященной реляционным хранилищам данных.

В облачных средах РБД представлены в трех видах: SQL as a Service — бессерверный сервис, как готовые образы для виртуальных машин и как Docker-образы, развернутые в кластерах AWS ECS или Azure Service Fabric.

Базы, размещаемые на виртуальных машинах, по сути, мало отличаются от серверов баз, размещенных на традиционных виртуальных и физических хостах. При этом выбор типа виртуальной машины, виртуальных дисков и сетевых интерфейсов значительно влияют на ее производительность (но это, как уже отмечалось выше, тема отдельной части книги). У серверов баз данных, размещенных на виртуальных машинах в облаках, есть ряд серьезных ограничений, связанных с масштабированием, доступностью и отказоустойчивостью. Например, виртуальные машины Azure имеют SLA (service level agreement — соглашение об уровне доступности) на уровне 99,95 % только в случае использования кластера минимум из двух машин, размещенных в одной общей группе доступности (availability set), при этом у них будут физически разделены сетевые интерфейсы и источники питания. Чтобы достичь уровня 99,99 %, необходимо конфигурировать кластер из нескольких виртуальных машин Always On.

Размещение БД в контейнерах Docker и размещение последних в кластерах виртуальных машин — относительно недавнее архитектурное решение. Оно очень удобно при наличии готового образа Docker, содержащего сервер СУБД и его расширения, что существенно проще в настройке, нежели в случае конфигурирования виртуальной машины. Кластеризация в подобной ситуации определяется возможностями самой базы в синхронизации данных между репликами, а потому далеко не всегда можно масштабировать БД в кластере. Кроме того, образ Docker по своей природе является stateless — не сохраняющим состояния, что требует специальных средств, сохраняющих данные при перезапуске удаленного контейнера или при передеплаивании его на новые узлы кластера. В настоящее время у облачных провайдеров отсутствует прямая поддержка этого решения и все бремя администрирования системы и сохранения информации ложится на ее создателей.

SQL as a Service обладает SLA на уровне 99,99 %, не вызывая дополнительных трудностей, связанных с конфигурированием кластеров. С другой стороны, требованиям стандартов HIPAA (см. информацию на сайте https://en.wikipedia.org/wiki/Health_Insurance_Portability_and_Accountability_Act) соответствуют SQL-серверы, размещенные на виртуальных машинах, но не сервис Azure SQL и не Docker-контейнер. В остальном же облачный сервис SQL имеет непревзойденные возможности в плане масштабирования, доступности и простоты администрирования (репликация, резервное копирование, экспорт, анализ производительности выполняемых запросов). Кроме того, SQL as a Service в ряде случаев существенно дешевле аналогичного по производительности сервера, размещаемого на виртуальной машине (я сознаю, что это очень спорное утверждение, особенно учитывая нюансы использования

ресурсов в SQL as a Service и SQL на виртуальных машинах). SQL as a Service не поддерживает всех возможностей традиционных серверов баз данных. Например, выполнение запросов информации из таблиц в одной базе в запросе внутри другой базы в случае SQL-сервиса сопряжено с рядом трудностей и ограничений. Рассмотрим примеры реализации SQL as a Service в разных облачных средах.

5.2. Azure SQL

В настоящее время платформа Microsoft Azure позволяет использовать как виртуальные машины с готовыми предустановленными серверами SQL, так и РБД-сервисы. К последним на настоящий момент (2018 год) можно отнести три: MySQL, PostgreSQL и Azure SQL. Первые два предоставляют сервис на основе баз данных MySQL и PostgreSQL и находятся в настоящее время в стадии preview, а это значит, что их наполнение и функциональность со временем может меняться.

Самый технически зрелый и долгоживущий сервис — Azure SQL, представляющий собой облачный сервис РБД на основе движка Microsoft SQL Server. В языке запросов Azure SQL реализовано подмножество функций T-SQL. Экземпляры сервисов Azure SQL, являющиеся прямыми аналогами баз MS-SQL, логически группируются в серверы Azure SQL Server. Каждый такой сервер должен располагать уникальным URL, учетными данными (имя пользователя и пароль), а также набором допустимых IP-адресов, которые могут иметь доступ к нему (этот список формируется в фаерволе сервера и регулирует правила доступа к нему).

Физически Azure SQL Server размещается в ЦОДе, расположенном в определенном географическом регионе, и в этом же регионе размещаются все экземпляры баз данных Azure SQL. Возможна также географическая репликация баз в несколько регионов по схеме Primary-Secondary или Primary — ReadOnly Replica (первичный сервер — реплика, доступная только для чтения). Это позволяет увеличить надежность и доступность баз.

Благодаря широкой поддержке T-SQL Azure SQL предоставляет возможность прямой миграции баз данных из Microsoft SQL Server в Azure SQL. Конечно, прямая миграция из одного типа базы в другую — далеко не простая и не быстрая задача, но в данном случае это принципиально возможно и напрямую поддерживается с помощью специальных программ от Microsoft, Red Gate и др. Однако следует иметь в виду: указанное программное обеспечение в ряде случаев просто «вырезает» несовместимые объекты базы, не пытаясь их адаптировать. Вероятна такая ситуация: после такой миграции БД «поднялась», но счастливые разработчики могут «внезапно» недосчитаться ряда объектов базы в Azure SQL. (Я однажды имел неудовольствие обнаружить, что при миграции базы из нее исчезли бóльшая часть триггеров и ряд хранимых процедур, поскольку содержали обращения к объектам другой БД, что в случае Azure SQL требует иного подхода.)

Каждый экземпляр баз данных имеет определенный ценовой уровень (pricing tier), который характеризуется производительностью, ограничениями по размеру, количеству точек восстановления и возможностями репликации. Все потенци-

альные ценовые уровни разделены на базовые (Basic), стандартные (Standard) и премиум (Premium). Самое главное различие ценовых уровней проявляется в разном значении DTU — обобщенного параметра, характеризующего производительность БД. Что же это за параметр? DTU (eDTU, elastic DTU) — интегральная характеристика производительности БД, включающая в себя показатели производительности центрального процессора, памяти, устройств ввода-вывода и сетевого интерфейса. DTU определяет своего рода объем, который может занимать производительность Azure SQL. Если в данный момент в базе выполняется запрос, то он потребляет определенное количество ресурсов, занимающих часть этого разрешенного объема (рис. 5.3).

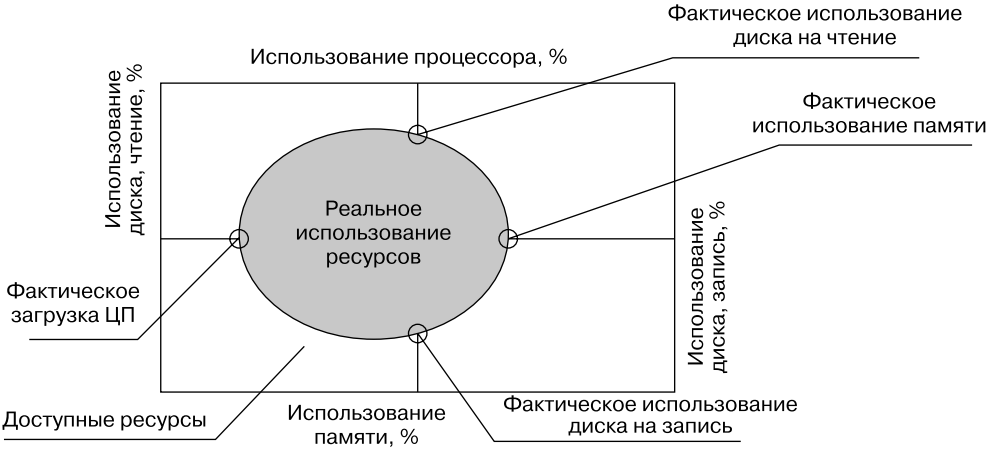


Рис. 5.3. Графическое представление DTU

При этом следует иметь в виду, что ограничивается не только объем, но и конкретные значения каждого из показателей (то есть не получится «обменять» крайне малое использование CPU на крайне большое значение памяти). Это ограничение проявляется, в частности, в том, что один «кривой» запрос может вызвать переиспользование одного из ресурсов и в итоге «подвесить» всю базу (экземпляр сервиса Azure SQL Database, но не Azure SQL Server), и точно такой же запрос будет работать нормально в традиционной БД Microsoft SQL Server. Это происходит потому, то в Azure SQL срабатывает механизм защиты ресурсов базы от чрезмерного применения, который вынуждает процесс, вызвавший запрос, отключиться по тайм-ауту, освободив тем самым ресурсы БД.

Итак, концептуально Azure SQL состоят из экземпляров Azure SQL Database, являющихся собственно реляционными хранилищами информации с определенными значениями размера и максимального DTU; экземпляра Azure SQL Server, который группирует базы Azure SQL Database, обеспечивая им общую строку соединения (connection string); правил доступа, прописанных в фаерволе; эластичного пула ресурсов (elastic database pool); секционированной базой данных (sharded

database) и реляционного хранилища данных (Azure SQL DWH). Рассмотрим эти сервисы по порядку.

Чтобы создать экземпляр Azure SQL, необходимо сначала в левом верхнем углу портала выбрать ссылку +New и затем в появившемся окне щелкнуть на ссылке SQL Database. После этого откроется страница с формой конфигурирования сервиса (рис. 5.4).

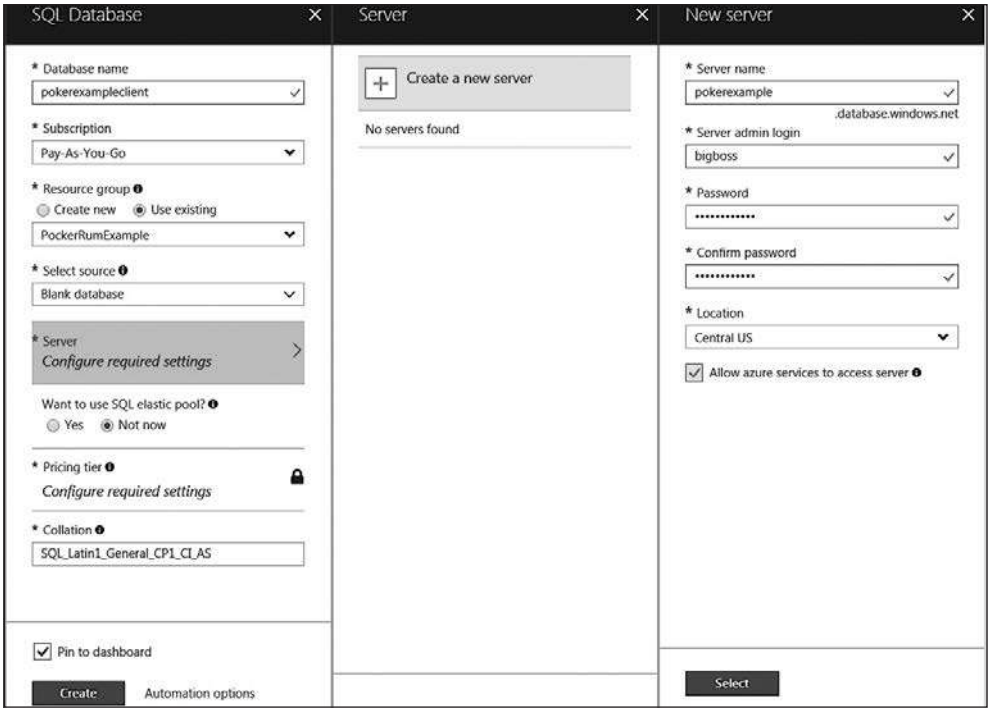


Рис. 5.4. Начальная страница конфигурирования сервиса Azure SQL

В качестве имени базы данных (Database name) указываем `pokerexampleclient`. Выбираем существующую группу ресурсов (Resource group) — PockerRumExample. В качестве источника базы (Select source) указываем Blank database. Поскольку у нас нет сервера БД, его следует создать. Для этого указываем имя сервера (Server name) — `pokerexample`, учетные данные (Server admin login и Password) пользователя-администратора и его местоположение (Location) — Central US.

Далее следует выбрать ценовой уровень (Pricing tier) (рис. 5.5). На уровне Basic можно только изменить размер базы данных. Для других уровней (Standard, Premium) доступна более гранулярная настройка размера и производительности, включая настройку подуровней.

После создания в сервисе Azure SQL доступны различные дополнительные сервисы, рассмотрим наиболее важные и полезные из них (рис. 5.6).

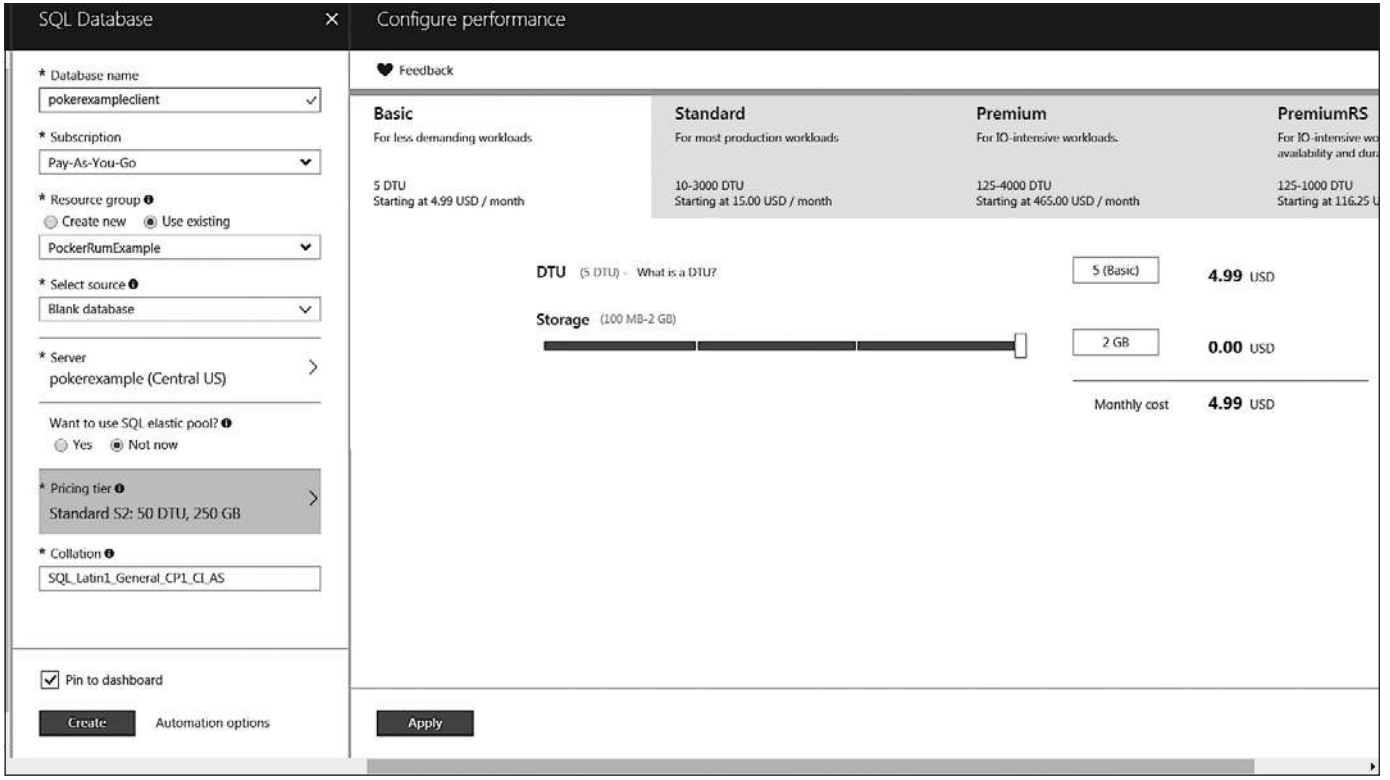


Рис. 5.5. Настройка ценового уровня базы данных

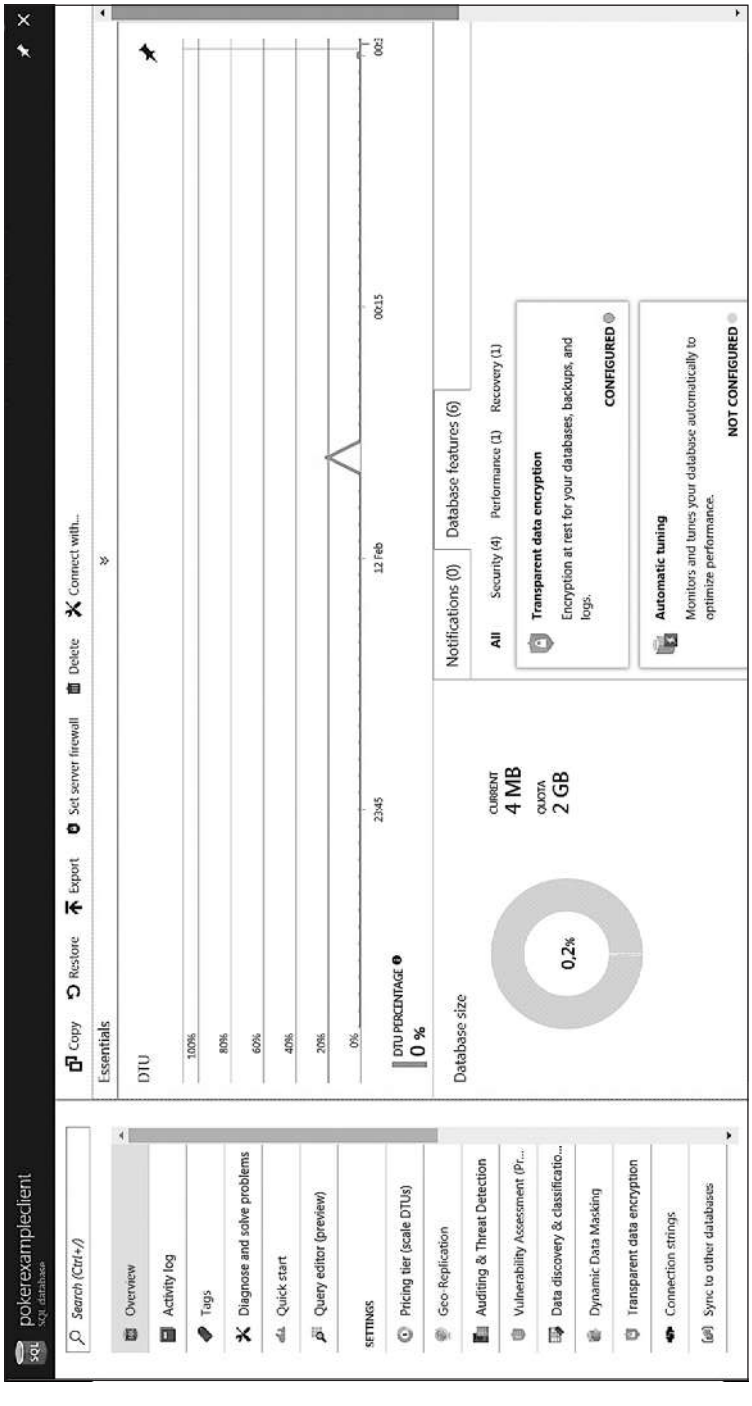


Рис. 5.6. Общая страница управления и мониторинга Azure SQL

Чтобы получить доступ к базе данных извне, необходимо прежде всего сконфигурировать фаервол (рис. 5.7). Ссылка доступа к ней находится в верхней части портала (Set server firewall).

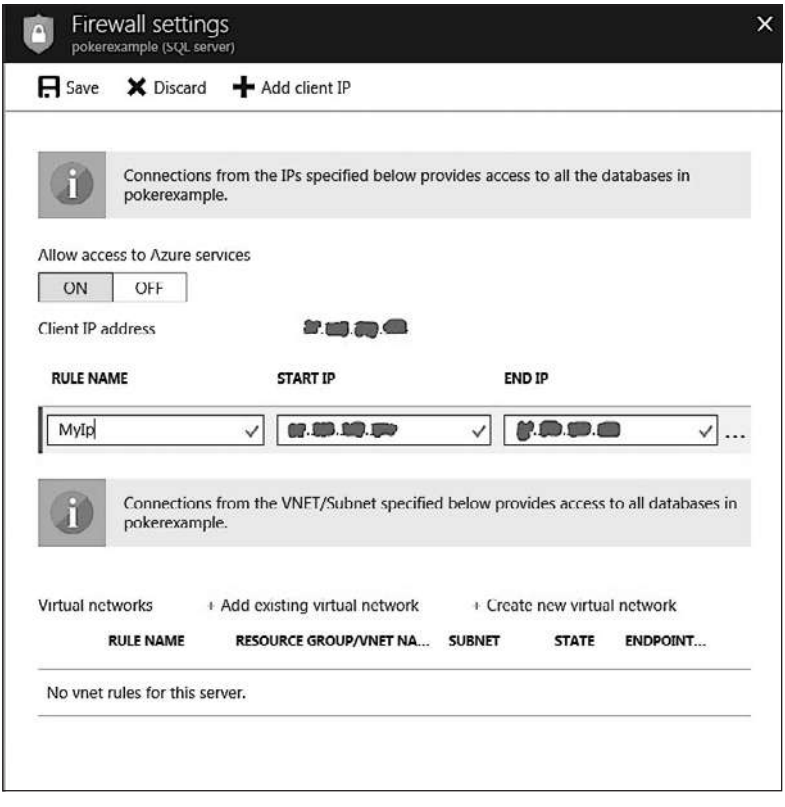


Рис. 5.7. Фаервол сервера Azure SQL

Чтобы получить доступ к БД извне, нужно получить строку подключения с учетными данными пользователя. Для этого следует нажать ссылку Connecting string на левой навигационной панели заглавной страницы (рис. 5.8).

Самые главные показатели мониторинга БД — DTU и размер базы. Мониторинг использования DTU позволяет анализировать закономерности в функционировании БД, например суточные пики и провалы активности или загруженность базы по дням недели. Этот мониторинг очень важен не сам по себе, а ввиду того, что позволяет определить, является текущий ценовой уровень оптимальным или необходимо его изменить. Если конкретнее, то уровень применения DTU должен, с одной стороны, быть таким, чтобы база была максимально загружена, а с другой — чтобы случайные пики не вызывали 100-процентного использования ресурсов DTU. И здесь весьма уместен мониторинг DTU.

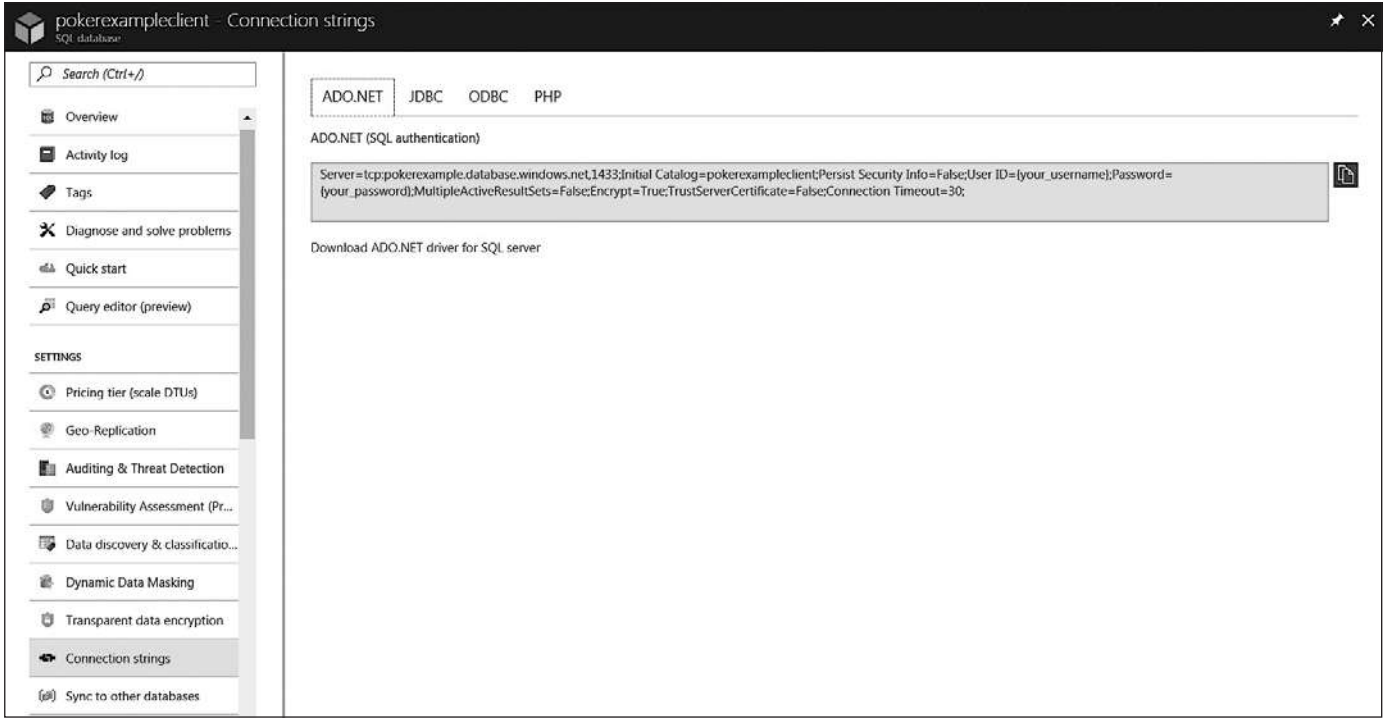


Рис. 5.8. Получение строки подключения для базы данных

Однако возможен ряд вариантов. Если БД в целом равномерно загружена без ярко выраженных пиков и провалов, то рекомендуется ценовой уровень, допускающий использование около 70–80 % DTU. Теперь рассмотрим случай, когда база данных имеет ярко выраженные и узкие пики (spikes). Подобная ситуация неприятна тем, что ради обеспечения достаточной производительности БД необходим ее ценовой уровень, слегка превышающий уровень этих пиков. Но если такие пики достаточно редки, то ресурсы базы задействуются достаточно нерационально: бóльшую часть времени она работает существенно недогруженной (с малым уровнем DTU), а во время пиков — зачастую перегруженной (DTU близка к 100 %). Эта нерациональность в конечном итоге приводит к неоправданно большому счету за облачные ресурсы. Логичнее и эффективнее всего в подобном случае использовать сервис *Query Performance Insight* (рис. 5.9), постараться определить, какие запросы в базе приводят к таким пикам, и попытаться устранить их за счет оптимизации запросов, список которых выдает упомянутый сервис.

Наряду с этим доступна возможность автоматической настройки производительности с помощью сервиса *Automatic Tuning* (рис. 5.10).

Автоматическая настройка производительности состоит в динамическом добавлении/удалении индексов, а также перекомпилировании плана исполнения. Каждый акт срабатывания автоматической настройки отображается в логах. Помимо этого, Microsoft предоставляет целый ряд сервисов подстройки и автоматической оптимизации базы данных (Performance overview, Performance recommendations и пр.), на которые следует обратить внимание при реальном функционировании БД.

Кроме того, через веб-портал доступен еще один из сервисов Azure SQL — *Azure SQL Query Editor*, который позволяет прямо на веб-портале писать запросы SQL к БД и редактировать данные в браузере. Встроенный редактор запросов выглядит следующим образом (рис. 5.11).

Сейчас он доступен в стадии Preview, а потому вполне возможно, что через некоторое время внешний вид и функциональность могут измениться. Прежде чем попасть на эту страницу, нужно войти в базу через ввод учетных данных после нажатия ссылки Login. Если не созданы новые пользователи, можно применить учетные данные, полученные на шаге конфигурирования сервиса (см. рис. 5.4).

Как указывалось выше, возможно редактирование данных в таблице на отдельной вкладке (рис. 5.12).

Теперь рассмотрим очень важный случай работы нескольких баз данных Azure SQL на одном сервере. В реальных проектах нагрузка на базы может быть неравномерной не только из-за наличия пиков, обусловленных проблемами с запросами, но и из-за периодических пиков запросов пользователей. Если им соответствует строгая периодичность (скажем, большая нагрузка днем и маленькая ночью), то можно настроить автоматическое масштабирование базы данных по расписанию, например, с помощью сервиса Azure Automation (описание процесса настройки выходит за рамки тематики книги).

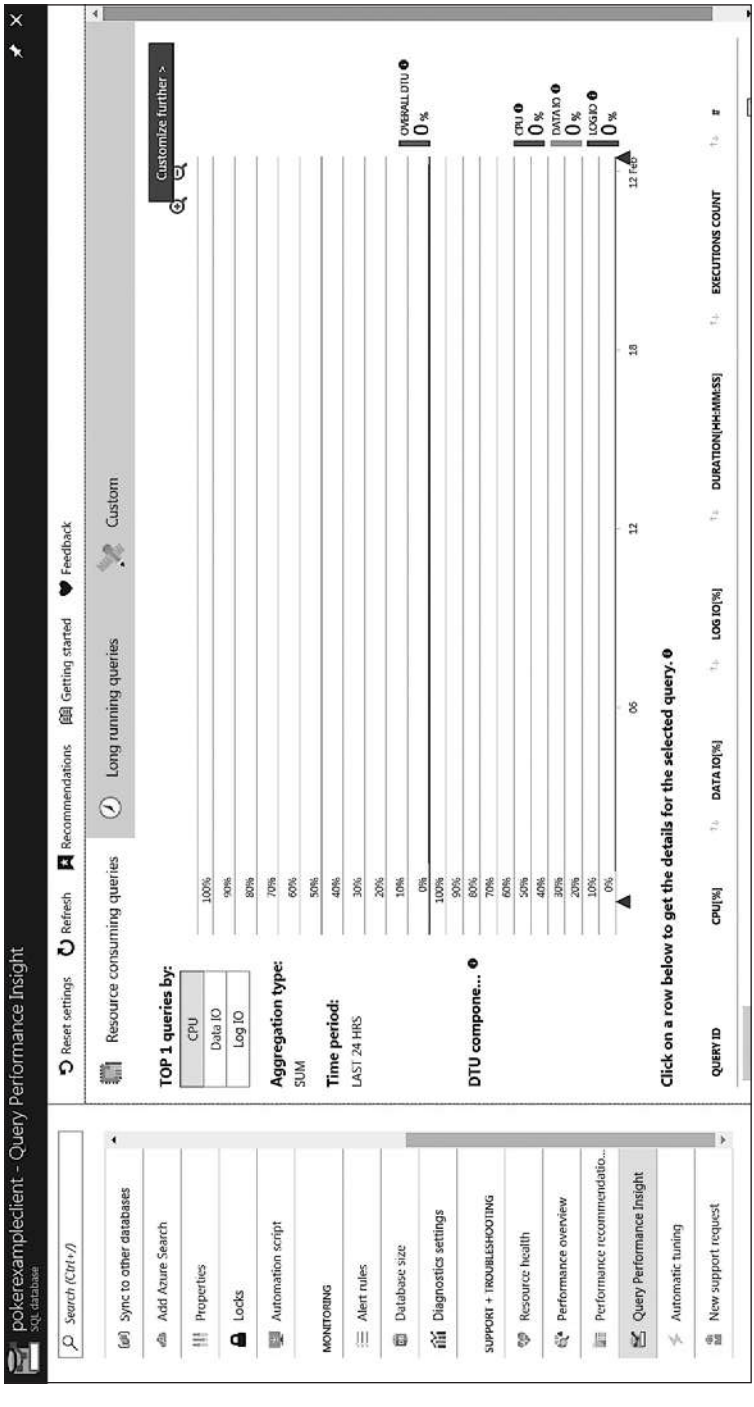


Рис. 5.9. Внешний вид вкладки сервиса Query Performance Insight

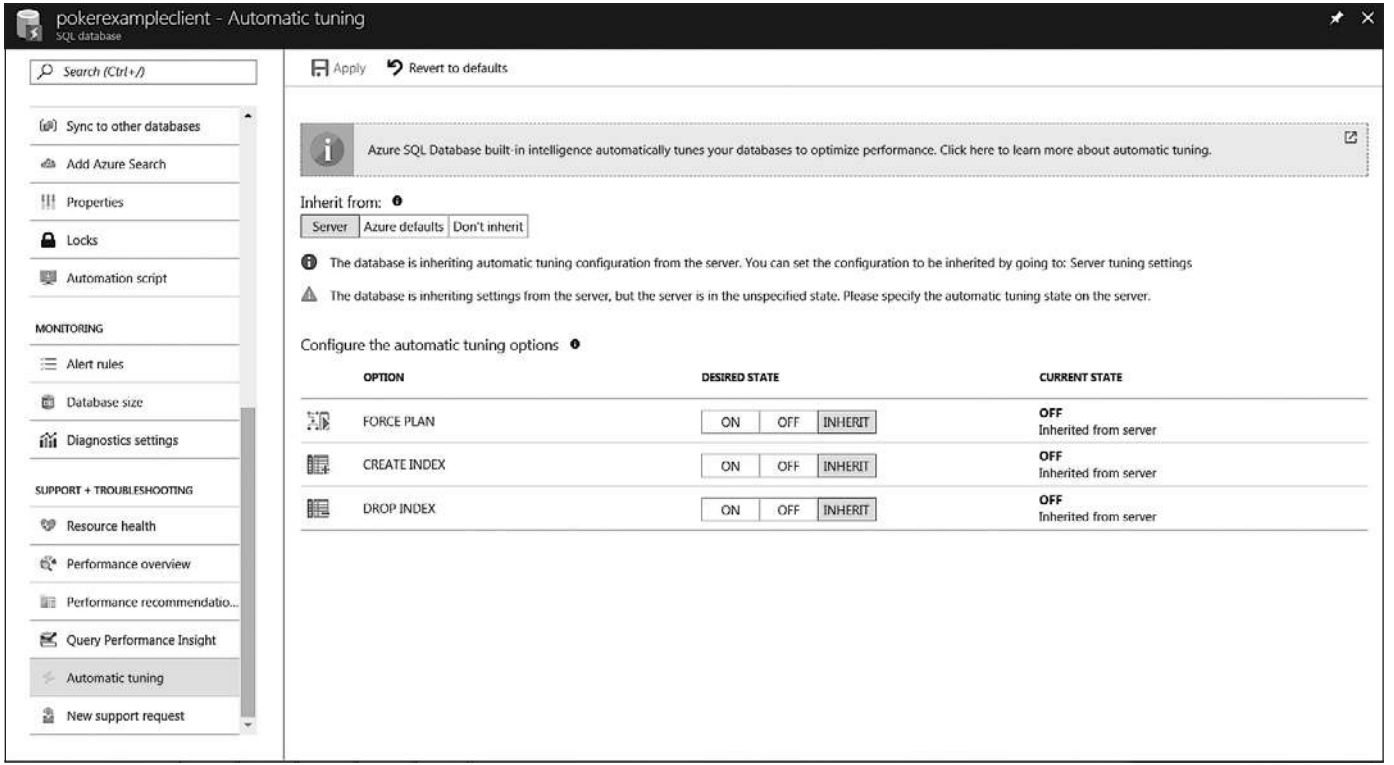


Рис. 5.10. Сервис автоматической оптимизации производительности

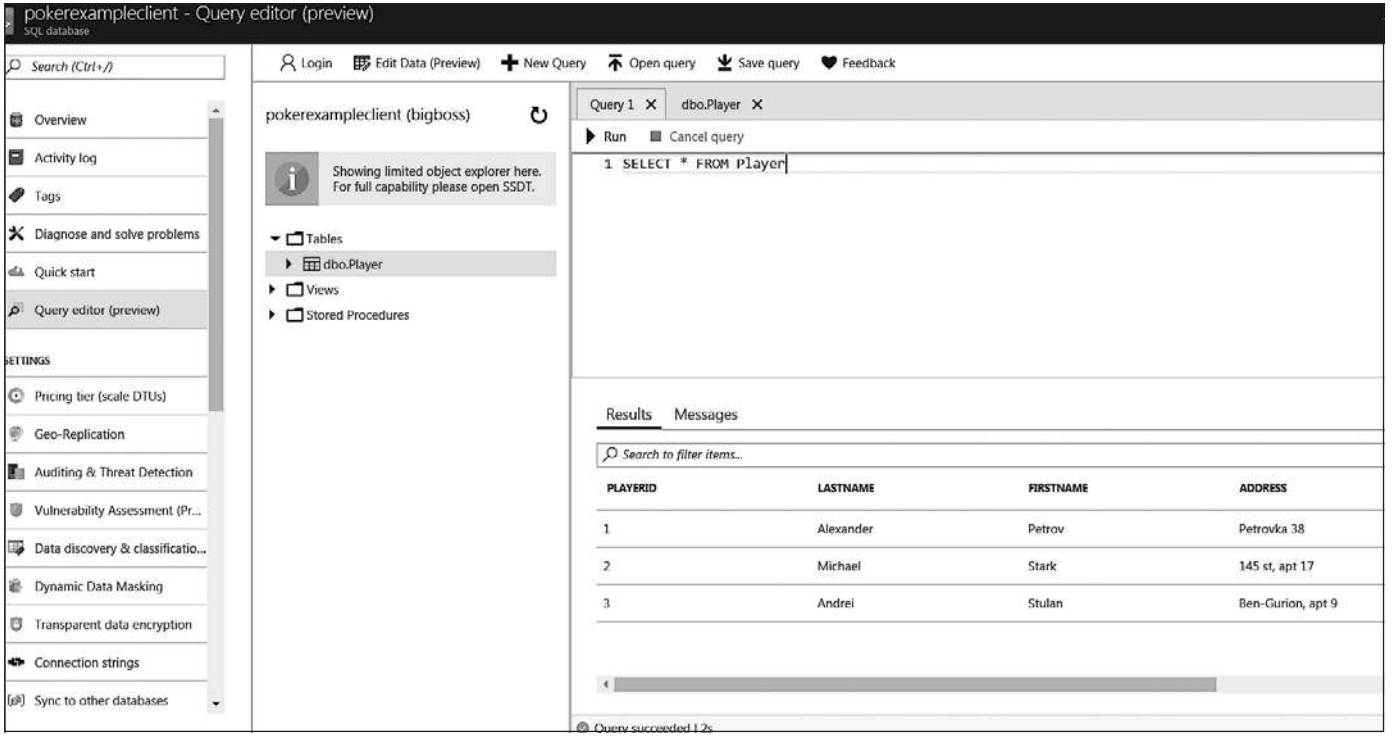


Рис. 5.11. Встроенный редактор запросов Azure SQL

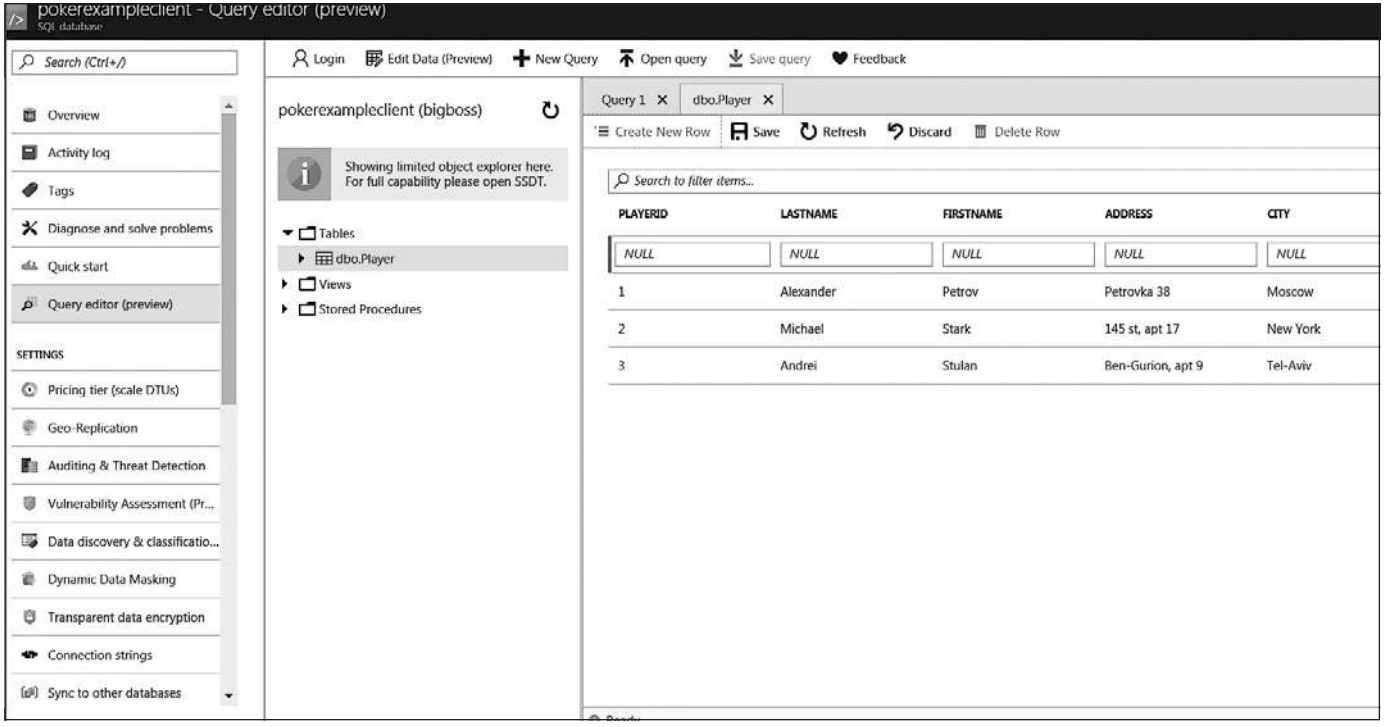


Рис. 5.12. Вкладка редактирования данных в таблице

Еще один случай — некоррелированные запросы могут послужить причиной использования Azure Elastic Database Pool, который встроен в состав Azure SQL Server и служит для объединения баз в один пул и назначения всем им общих разделяемых ресурсов сервера. Рассмотрим ситуацию более детально на примере SaaS-приложения, созданного для оказания неких услуг зарегистрированным в нем пользователям (допустим, это облачная CRM-система). Каждому из пользователей выделяется своя БД определенного уровня производительности, который выражается конкретным значением DTU. Каждый уровень имеет определенную месячную стоимость, которая в конечном итоге скажется на прибыли владельца SaaS: она будет равна суммарной месячной плате всех пользователей за вычетом расходов на облачные сервисы, лежащие в основе архитектуры системы. Теперь предположим, что активность пользователей, выражающаяся в применении DTU каждой базы, носит некоррелированный характер, то есть пользователи нагружают свои базы в различные случайные временные промежутки. Это приводит к появлению моментов времени, когда применение DTU базы конкретного пользователя очень мало, и моментов, когда оно велико (рис. 5.13).

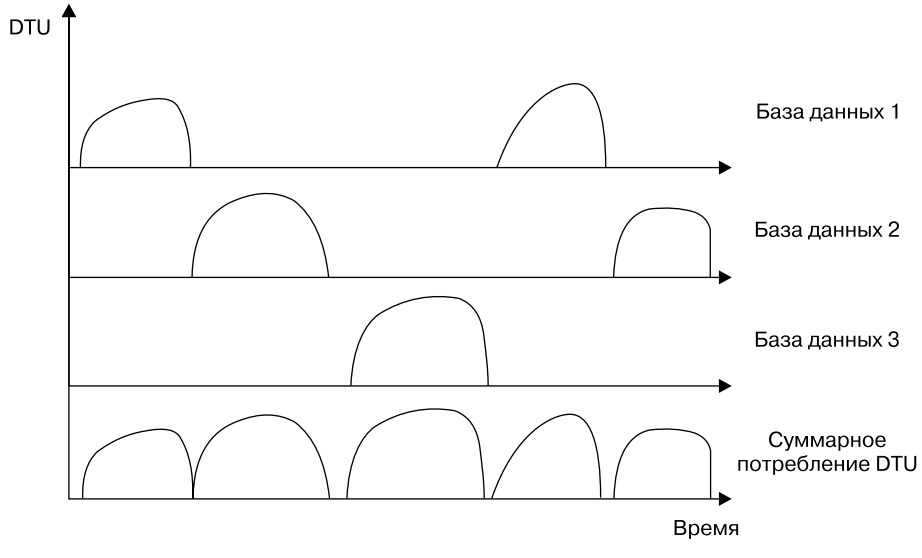


Рис. 5.13. Базы данных, подходящие для объединения в Elastic Pool

Таким образом, в момент малого использования DTU базы простаивают — но Azure взимает плату за них. А если взять несколько БД, у которых интервалы активности одних баз совпадают с интервалами недогруженности других, объединить их в общий пул БД и распределить общие ресурсы DTU между базами таким образом, чтобы суммарная нагрузка была равномерной во времени? (Плата за ценовой уровень при этом будет существенно меньше, чем суммарная плата за все базы данных.) Именно такая идея лежит в основе сервиса Azure Elastic Database

Pool. Вдобавок существует ряд рекомендаций относительно выбора оптимального ценового уровня. Выбор eDTU (elastic DTU) производится с помощью приближенного соотношения:

Суммарное eDTU = MAX(<*Общее количество баз данных* × *Среднее использование eDTU*>, <*Количество баз данных одновременно достигающих пиковых нагрузок* × *Пиковый уровень DTU базы*>)

Далее для определения минимального объема хранилища необходимо просуммировать объемы всех отдельных баз данных. Затем следует выбрать ценовой уровень, дающий максимальное значение eDTU, полученное исходя из формулы и размера хранилища.

Чтобы добавить Elastic Database Pool, нужно выйти на начальную страницу сервера pokerexample. Для добавления нового Elastic Pool на этой странице следует нажать на ссылку + Elastic pool (рис. 5.14).

На рис. 5.15 показана страница создания и конфигурирования Elastic Pool.

Дальнейшие шаги по конфигурированию пула показаны на рис. 5.16.

Добавление баз данных в пул и удаление их из него может происходить динамически без остановки и передеплаивания баз данных. Для этого в панели управления есть графики мониторинга и сервис выдачи рекомендаций по включению баз данных в пул и исключению из него. Детальное рассмотрение этого вопроса выходит за рамки данной книги.

Кроме того, существует сервис Elastic Query, позволяющий выполнять задания, общие для всех баз данных из Elastic Database Pool. Суть его в том, что группа баз объединяется и управляется централизованно с общего специализированного головного сервера ВМ. Взаимодействуя с этим сервером программно или через Azure Portal, можно управлять всеми подключенными БД с помощью создания заданий (jobs). Задания могут быть такими:

- административные (согласованное изменение схемы, выполнение перекомпиляции индексов);
- периодическое обновление данных или их сбор для систем BI, в том числе для выполнения анализа большого объема информации.

Сервис Elastic Database jobs содержит следующие компоненты:

- головной сервер, размещаемый на экземпляре Azure Cloud Service Worker Role. По сути, это специализированное ПО, размещаемое (по состоянию на настоящее время) на виртуальной машине Azure Cloud Service. Для обеспечения высокой доступности рекомендуется создавать минимум два экземпляра ВМ;
- управляющую (головную) БД — экземпляр Azure SQL, служащий для хранения метаданных всех подключенных баз;
- экземпляр сервиса Azure Service Bus, служащий для объединения и синхронизации всех компонентов;
- экземпляр облачного хранилища Azure Storage Account, служащий для хранения журналов всей системы.

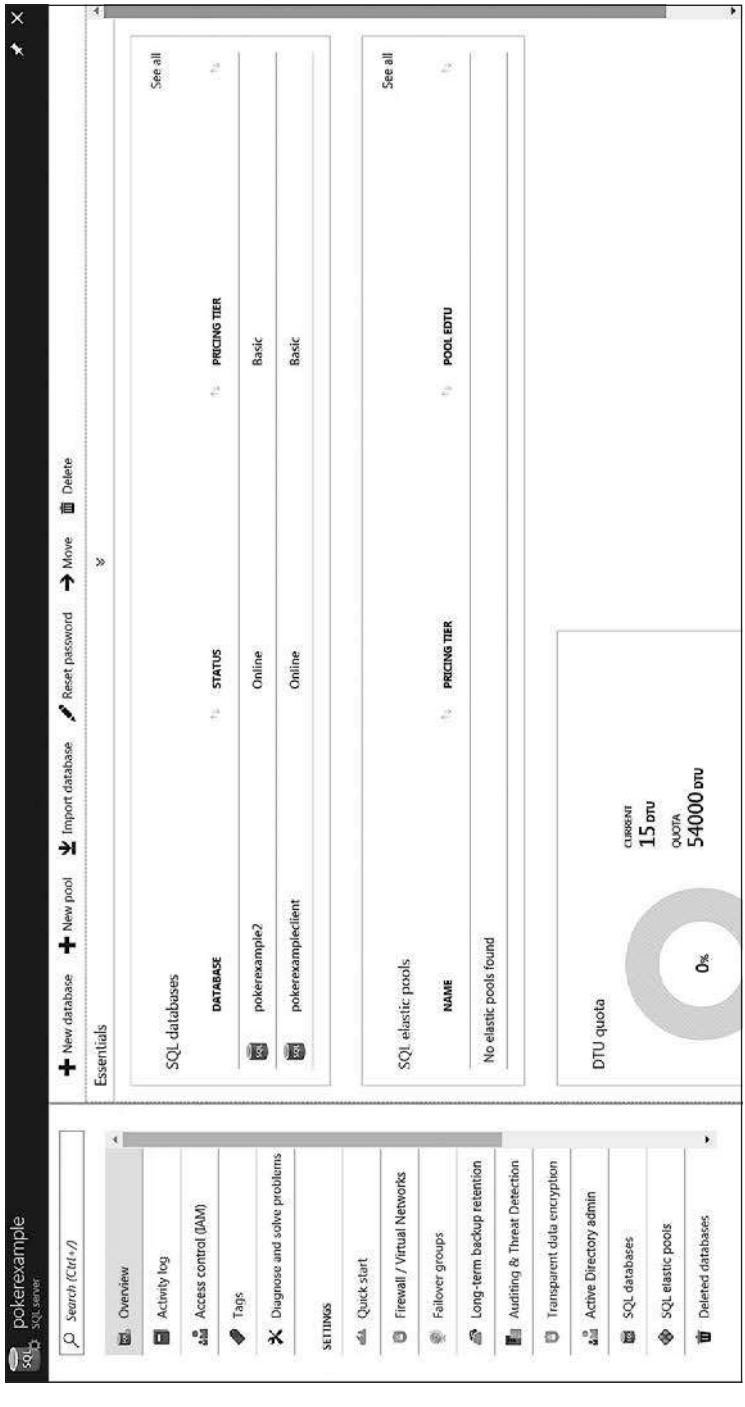


Рис. 5.14. Стартовая страница сервера со списком доступных баз данных

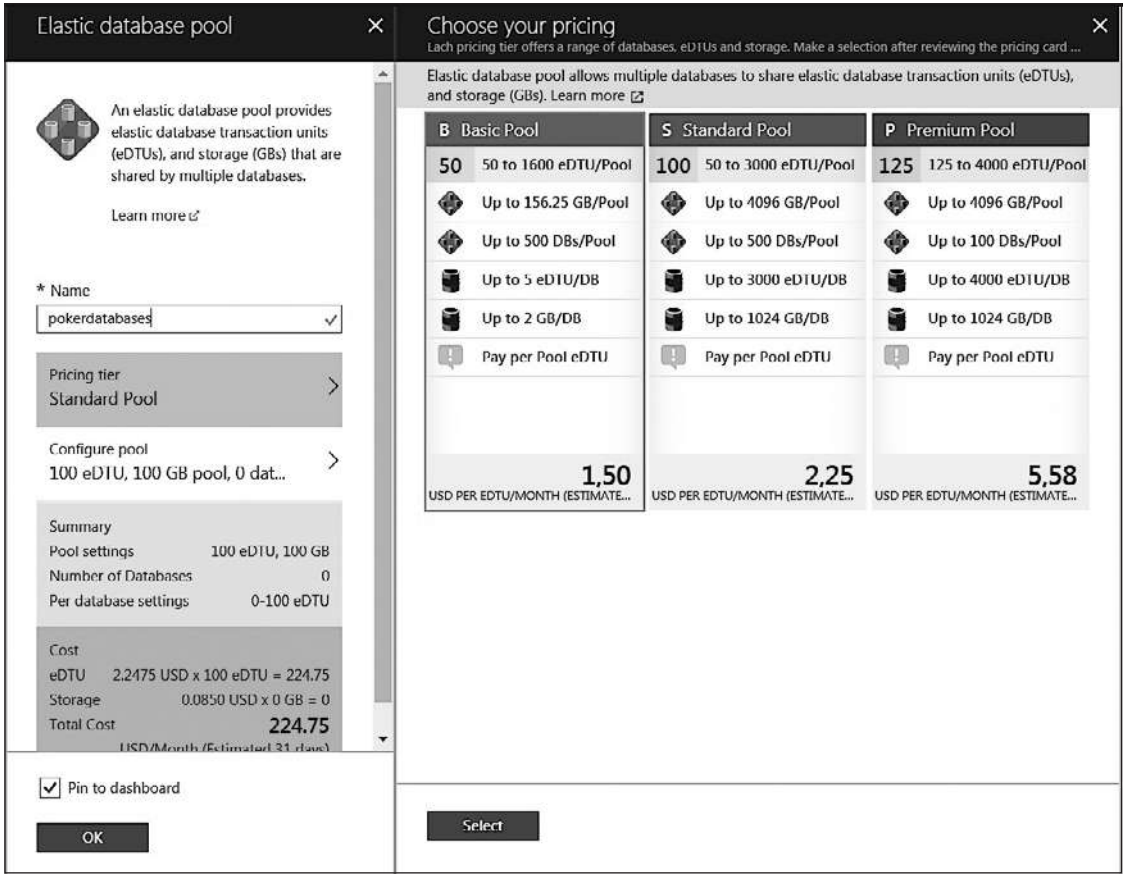


Рис. 5.15. Страница создания и конфигурирования Elastic Pool

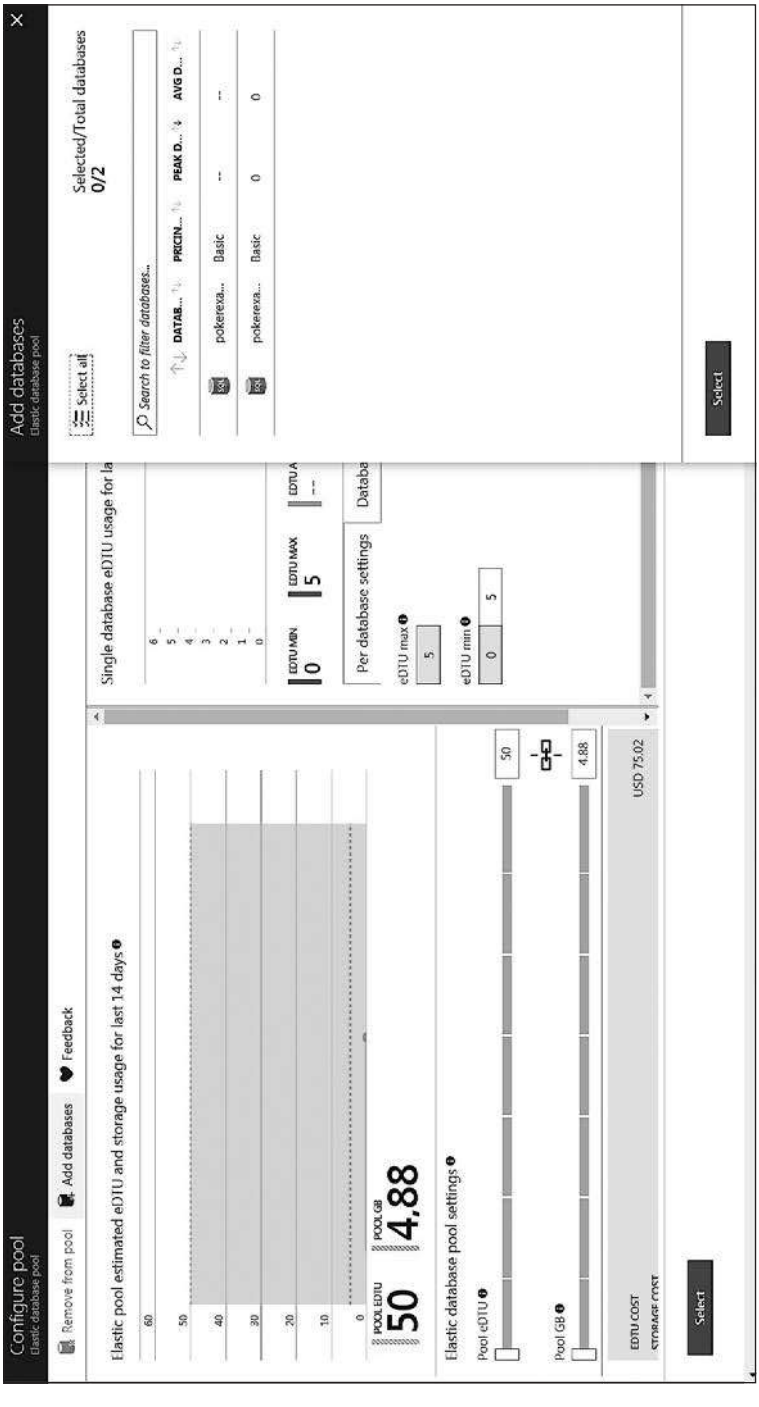


Рис. 5.16. Дальнейшее конфигурирование пула

Настройка, использование и администрирование всей этой системы — довольно сложная тема и в данной книге подробно рассматриваться не будет.

Очень важное свойство Elastic Database Pool — возможность реализовывать сценарии разбиения одной большой БД на меньшие, но выполнять запросы в разделенной на фрагменты (shard) базе так, будто это одна монолитная база (рис. 5.17). Рассмотрим данный механизм подробнее. Нужда во фрагментации БД появляется в случае, когда ее размер становится чрезмерно большим для размещения на одном экземпляре Azure SQL (в настоящее время это более 1 Тбайт для ценового уровня Premium). Конечно, можно разбить большую БД на меньшие логически, проведя анализ ее структуры (схемы). Как уже указывалось ранее, Azure SQL Server — логическая группировка экземпляров Azure SQL Database, а не физическое объединение на одном сервере. И прямые запросы к объектам одной базы из другой возможны, только если объекты являются внешними таблицами или базы объединены в Elastic Database Pool и используются запросы Elastic Database Query или транзакции Elastic Transactions. В таком случае необходимо применить фрагментацию (sharding) базы данных. По сути, это горизонтальное масштабирование, отличное от вертикального — увеличения размера базы в масштабе CPU, оперативной памяти, IOPs и пр. Разделение одного большого хранилища на несколько хранилищ меньшего масштаба и их параллельная обработка с последующим сложением результатов последней — ключевая концепция всех технологий обработки больших данных, с которой мы будем не раз встречаться далее. Фрагментированная БД концептуально во многом близка к реляционному хранилищу данных DWH, но требует бóльших усилий при создании и использовании.

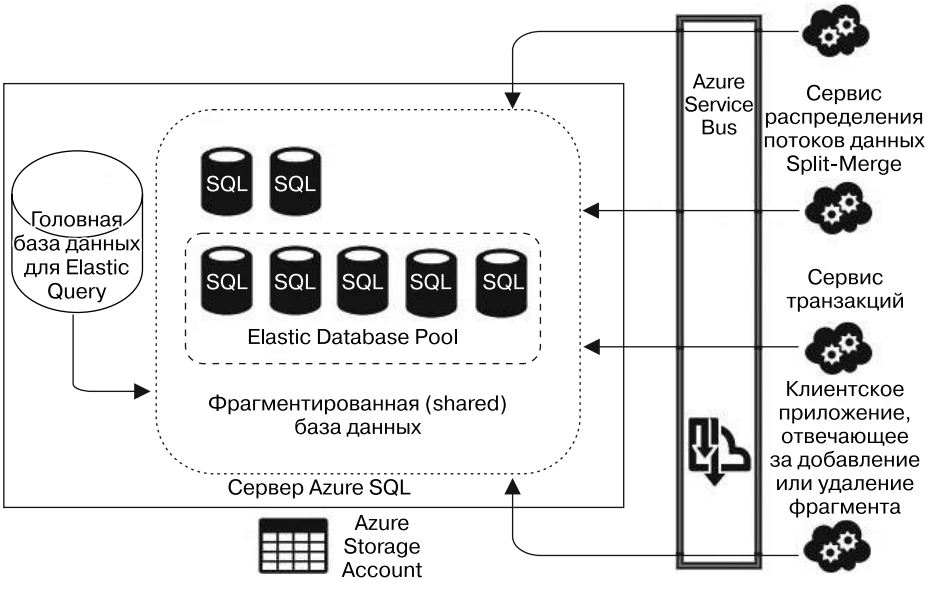


Рис. 5.17. Общая структура фрагментированной базы данных, построенной на основе Elastic Database Pool

Теперь рассмотрим случай, когда все же необходимо выполнить запрос к данным, расположенным в разных базах. При наличии простого набора БД без фрагментирования нужно выбрать одну головную базу и создать в ней внешние источники данных и внешние таблицы (reference table), являющиеся отражениями реальных таблиц, размещенных в других БД. Можно также не использовать специализированную головную базу, а создавать таблицы (см. информацию на сайте https://docs.microsoft.com/en-us/azure/sql-database/sql-database-elastic-query-getting-started-vertical) в каждом экземпляре Azure SQL Database.

Недостаток такого подхода в том, что при смене схем таблиц в базах необходимо синхронно менять схемы во внешних таблицах (при отсутствии головной БД нужно проделать очень большую работу по смене схемы во всех внешних таблицах во всех базах, где присутствуют эти внешние таблицы). Кроме того, у реализации T-SQL в Azure SQL есть ограничения на типы данных для внешних таблиц (например, они не поддерживают Foreign Key и тип `nvarchar(max)`). И в настоящее время присутствует целый ряд ограничений на выполнение подобных запросов в базы данных, где эти запросы будут выполняться, — скажем, невозможно выполнить экспорт БД в *BACPAC*-файл при наличии в ней ссылок на внешние таблицы (https://docs.microsoft.com/en-us/azure/sql-database/sql-database-elastic-query-overview). Ну и не следует забывать и о снижении производительности для таких запросов. Как же все это выглядит?

Предположим, у нас две базы данных: First и Second. Теперь допустим, что из базы Second нужно выполнить запрос к базе First. Для этого в Second следует создать учетные данные (credentials) (рис. 5.18), которые будут служить для доступа к БД First.

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD = '<парОль>';
CREATE DATABASE SCOPED CREDENTIAL FirstDBQueryCred   -- Это имя учетной записи
WITH IDENTITY = '<ИмяПользователя>',
SECRET = '<парОль>';
```

Рис. 5.18. Создание учетных данных для базы

Затем в базе Second нужно создать внешний источник данных, который будет использоваться для связи с таблицами внешней БД (рис. 5.19).

```
CREATE EXTERNAL DATA SOURCE FirstDatabaseDataSource WITH
(
    TYPE = RDBMS,
    LOCATION = '<имя_вашего_сервера>.database.windows.net',
    DATABASE_NAME = 'First',
    CREDENTIAL = FirstDBQueryCred
)
```

Рис. 5.19. Создание внешнего источника данных с учетными данными

После этого в базе Second надо создать внешнюю таблицу — отражение аналогичной таблицы, размещенной в базе First (рис. 5.20).

```
CREATE EXTERNAL TABLE [dbo].[TableFromFirstDatabase]
( [KeyFieldID] [int] NOT NULL,
  [DataFicld] [varchar](50) NOT NULL
)
WITH ( DATA_SOURCE = FirstDatabaseDataSource)
```

Рис. 5.20. Создание внешней таблицы

И все! Теперь в базе Second мы можем делать запросы к таблице из базы First как к обычной таблице, размещенной в базе Second. Стоит иметь в виду, что эти таблицы связаны только по данным — любые изменения схемы в одной БД никак не отразятся на схеме таблицы в другой.

Внешние источники данных — очень мощный инструмент MS SQL, который допускает также подключение доступных файлов типа CSV. Непосредственно для сервиса Azure SQL существует проблема, связанная с производительностью запросов к внешним источникам. Действительно, как уже было сказано, сервер Azure SQL и базы данных Azure SQL — понятия логические. Сервер — это только *логическая* группа независимых БД, поэтому передача данных от внешних серверов занимает значительно больше времени, чем при наличии у физического сервера MS SQL доступа к файлам, физически доступным по путям операционной системы.

5.3. AWS RDS

> Как велико то, что нас объединяет, и как ничтожно то, что нас разделяет.
>
> *Станислав Лем. Магелланово облако*

Сервис реляционных баз данных от Amazon называется AWS RDS — Amazon Web Services Relational Database Service. Он значительно отличается от сервиса Azure SQL как по своей «философии», так и по составу. Прежде всего, основными логическими элементами этого сервиса является *экземпляр*, или *инстанс RDS* (RDS Instance). Сам экземпляр, по сути, представляет виртуальную машину, на которой размещается сервер базы данных и к которой подключен VHD. Типы этой виртуальной машины похожи на обычные EC2, но имеют ряд существенных отличий. Прежде всего, отсутствует прямой консольный доступ к экземплярам, они не отображаются в консоли в списке EC2 и недоступны для размещения в них каких-либо иных приложений; невозможно применить модель Spot или Provisioned Instance; невозможно конфигурировать RAID-структуры. Эти экземпляры представляют собой вычислительные модули, служащие для выполнения запроса к данным, хранящимся на подключенных к ним виртуальных дисках, расположенных в хранилище S3. Могут быть использованы три типа диска: Magnetic, General purpose SSD

и Provisioned IOPS SSD. Изменяя тип экземпляра и виртуального диска, можно гибко подстраивать их под требования к цене и производительности для конкретной базы. При этом можно выбрать экземпляры с расширенной памятью (memory optimized), более производительным CPU (CPU optimized), более производительным интерфейсом ввода-вывода (IO optimized) или общего назначения (general purpose). Достаточно большое количество типоразмеров экземпляров позволяет перекрыть довольно широкий диапазон нагрузок, а возможность выбирать еще и тип и размер диска обеспечивает гибкость при построении систем с заданными соотношениями «цена/производительность».

Очень важный параметр экземпляра — *тип сервера базы данных*: движок (RDS engine). Этот тип определяет конкретную технологию SQL-сервера. В настоящее время поддерживаются следующие типы движков: MySQL, PostgresSQL, MariaDB, Oracle, MS SQL (Express и Standard Edition) и Aurora. Выбор конкретного движка не только влияет на выбор синтаксиса SQL, но и диктует набор и значение некоторых параметров самого экземпляра (например, типа экземпляра, типа диска и др.). Далее, в экземпляре можно создать одну или несколько БД. Необходимо понимать, что эти базы данных будут разделять ресурсы экземпляра и логично размещать в экземплярах базы таким образом, чтобы последние имели некоррелированные пики активности и обеспечивали по возможности более равномерную загрузку. Само количество БД в экземпляре определяется типом движка. Так, для Oracle база может быть одна, но иметь несколько схем.

Рассмотрим на примере, как создавать экземпляры сервиса RDS (рис. 5.21).

В терминах AWS создание экземпляров баз данных называется запуском (Launch). Чтобы запустить экземпляр, нужно перейти на вкладку Instances (навигационная панель слева) и нажать ссылку Launch DB Instance. В итоге откроется панель конфигурирования, показанная на рис. 5.22.

Обратите внимание: не все экземпляры доступны как Free Tier, то есть даже если вы пробуете Amazon в режиме Free Trial, далеко не все ресурсы будут действительно бесплатными. Чтобы оставить только ресурсы, доступные бесплатно, установите флажок в самом низу формы — Only enable options eligible for RDS Free Usage Tier (рис. 5.23). Дальнейший процесс создания показан на примере движка MySQL. Выбрав движок, нажмите Next.

В появившемся окне откроется форма конфигурирования экземпляра. Эта форма может слегка различаться для различных движков. На рис. 5.23 показана вся форма, сдвинутая скроллом, — слева верхняя часть, справа — нижняя. Прежде всего стоит указать модель лицензии (поле License model), затем уточнить версию движка в рамках выбранного типа (поле DB engine version). Следует заметить, что лицензирование — существенный аспект использования сервисов RDS. Одна часть движков предоставляются под лицензией GPL, а другая — Oracle, MS SQL — требует лицензионных отчислений; вследствие этого плата взимается как за вычислительные ресурсы и ресурсы хранения, так и за лицензию. Кроме того, для Oracle и MS SQL возможна оплата лицензии по принципу BYOL (Bring Your Own License — добавление лицензии, оплаченной вне AWS).

Рис. 5.21. Общий вид панели RDS

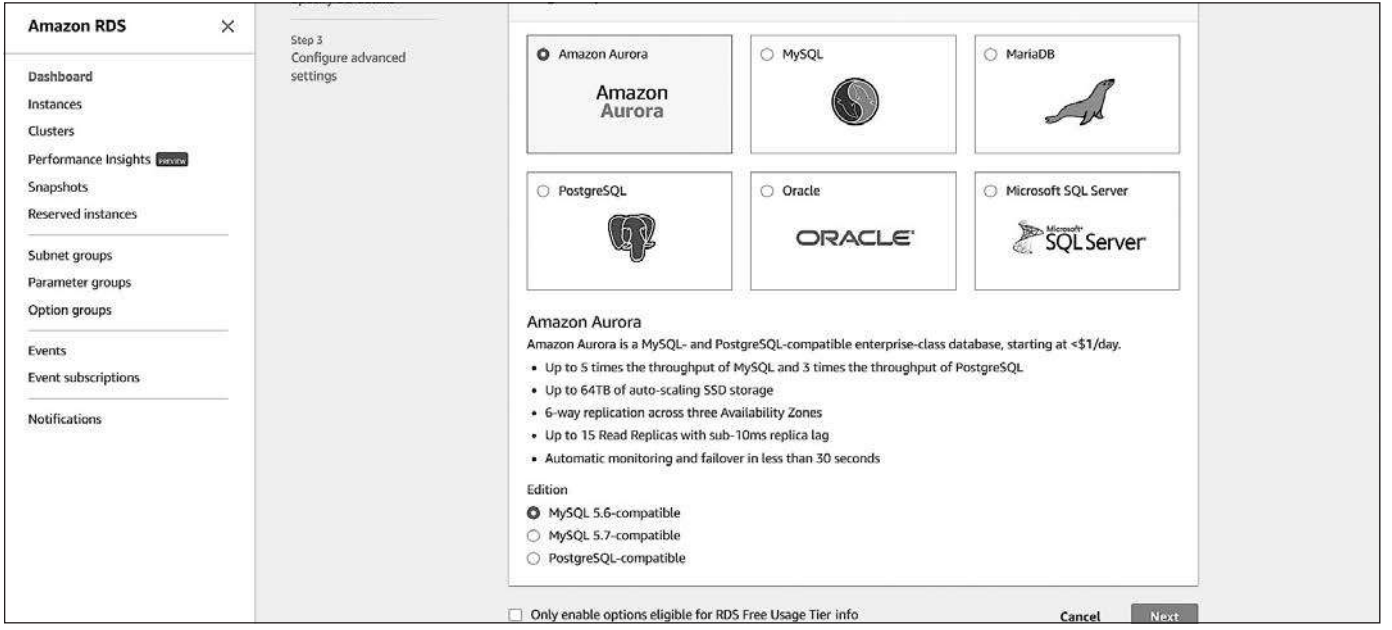


Рис. 5.22. Панель конфигурирования запускаемого экземпляра RDS

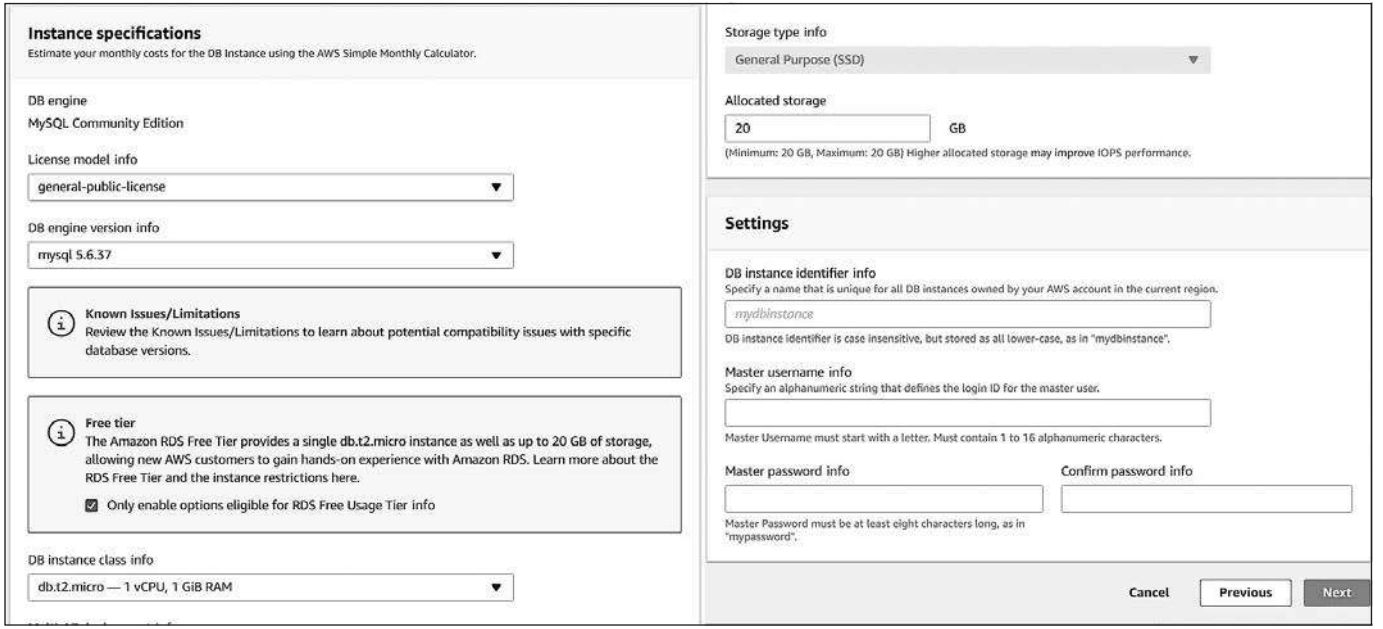


Рис. 5.23. Форма конфигурирования экземпляра RDS

Следующим выбирается тип экземпляра виртуальной машины (DB instance class), которая, по сути, будет лежать в основе этой БД. Данный тип экземпляра определяет уровень производительности базы. Физически данные будут храниться на виртуальных жестких дисках. Тип диска определяется в поле Storage type (для данного случая доступен только SSD), чей размер указывается в поле Allocated storage (20 Гбайт на рис. 5.23). Далее следует указать имя идентификатора базы (DB instance identifier), учетные данные пользователя-администратора и затем нажать Next. После этого откроется панель расширенных настроек (рис. 5.24). Рассмотрим их подробнее.

Сетевые настройки включают в себя выбор виртуальной частной сети (VPC), подсети (subnet), сетевой группы безопасности (security group) и опции публичной доступности (public accessibility). Последний параметр означает наличие возможности подключения к базе данных ресурсов, находящихся вне ее VPC.

Опции базы данных включают указание ее имени (database name), порта доступа (для MySQL по умолчанию это 3306) и ряда других специфичных параметров. Отмечу лишь возможность интеграции с учетными данными пользователей аккаунта AWS — для этого нужно включить опцию Enable IAM DB authentication.

Следующие формы (рис. 5.25) относятся к настройкам шифрования, бэкапам, мониторингу и экспорту логов в сервис AWS CloudWatch. В этих формах все интуитивно понятно, поэтому перейдем к последней и более интересной форме — настройке обновлений операционной системы и движка базы данных (рис. 5.26).

Упомянутый параметр очень важен для того, чтобы обеспечить обновляемость операционной машины экземпляра и движка базы данных. Не забывайте, в AWS экземпляры БД — это не экземпляры бессерверного сервиса, как в случае Azure SQL, а скорее обертка над сервисами IaaS.

Но AWS RDS — это PaaS-сервис, предоставляющий ряд встроенных механизмов администрирования, например механизм резервного копирования и управления сроком жизни резервной копии. Помимо периодически запускаемого резервного копирования, можно создать мгновенный снимок состояния (snapshot) и использовать его для сохранения и восстановления состояния памяти.

Помимо PaaS-возможностей, AWS RDS содержит и опции, наследованные от IaaS. Так, экземпляры могут быть остановлены, запущены и перезапущены. При остановке перестает начисляться плата за вычислительные ресурсы экземпляра, чего не скажешь о плате за виртуальные диски и операции ввода-вывода данных (IOPS) и за хранение бэкапов. Кроме того, операционная система и движок экземпляра требуют обновления и пользователь должен сделать это вручную через консоль. Тип экземпляра может быть изменен вручную или через API таким же образом, как и в случае EC2. Вдобавок для экземпляра MS SQL можно подключить сервис Microsoft Active Directory, чтобы хранить учетные данные пользователя.

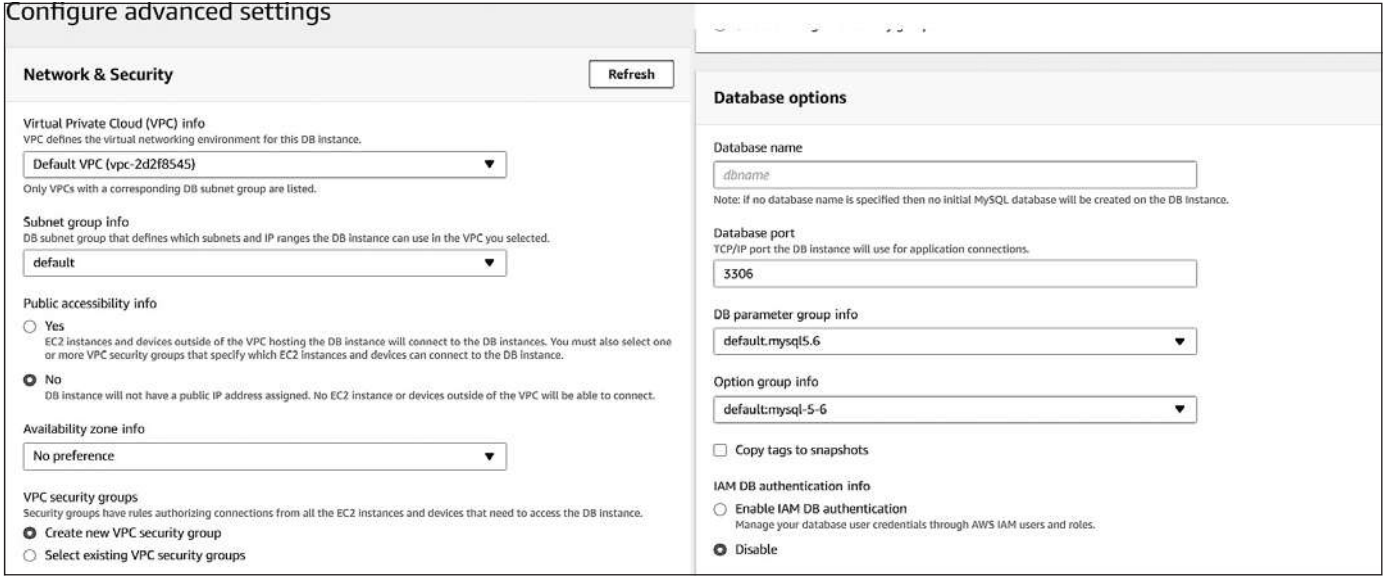


Рис. 5.24. Сетевые настройки и настройки опций базы данных

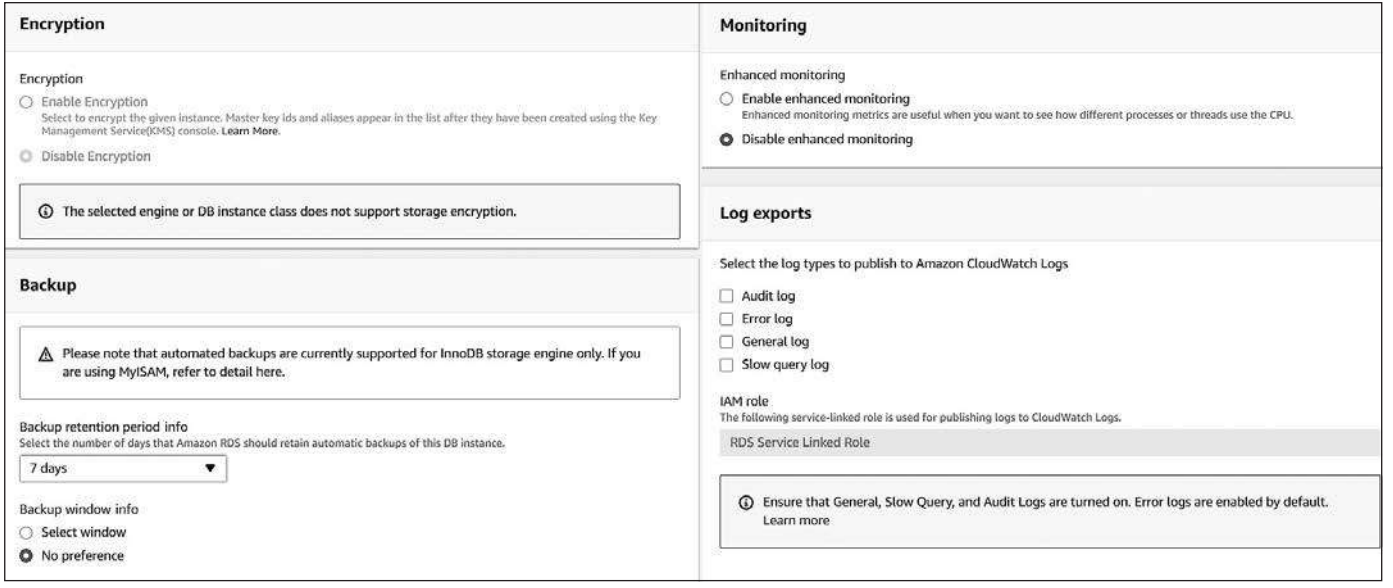


Рис. 5.25. Панель настройки шифрования, бэкапа, расширенного мониторинга и экспорта логов

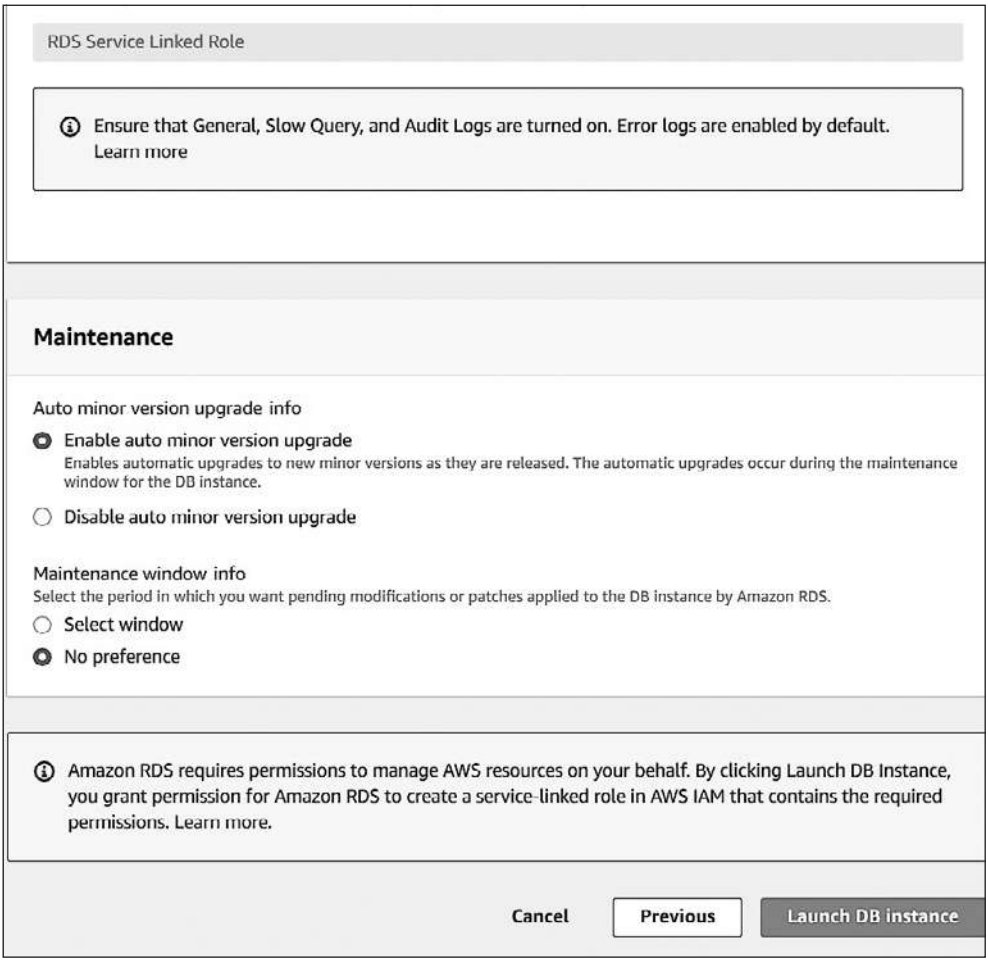


Рис. 5.26. Панель настройки автоматической установки патчей и обновлений

Мониторинг применения ресурсов экземпляров производится с помощью стандартного сервиса CloudWatch. В настоящее время AWS RDS не предоставляет общего механизма анализа влияния запросов на использование ресурсов, для каждого движка AWS рекомендует задействовать встроенные в этот движок механизмы анализа производительности.

Экземпляры движка Aurora создаются как *кластеры* — отказоустойчивые комбинации из нескольких экземпляров. Они состоят из главного экземпляра (primary instance) и реплик, поддерживающих только операции чтения (readonly replica) (рис. 5.27).

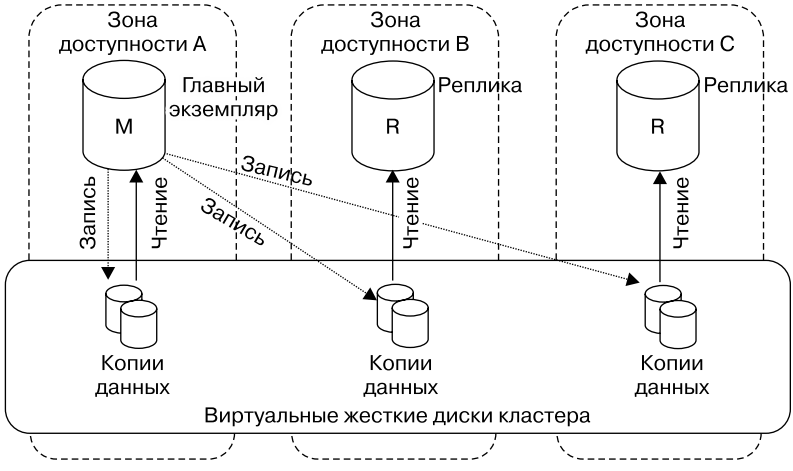


Рис. 5.27. Архитектура сервиса Aurora

5.4. Резюме

Итак, мы рассмотрели сервисы РБД от Microsoft (Azure SQL) и от Amazon (AWS RDS). Несмотря на то что оба сервиса относятся к PaaS и предоставляют базы данных как сервис, их философии и способы управления, создания и мониторинга значительно различаются. AWS RDS, по сути, обертка над специализированными IaaS с предустановленными серверами баз данных и предоставлением ряда функций конфигурирования, резервного копирования и администрирования из веб-консоли или через API таким же образом, как это происходит у PaaS. Кроме того, для экземпляров нет обобщенной характеристики использования ресурсов, а есть стандартные метрики — применения CPU, памяти, ввода-вывода. Это позволяет более детально подойти к выбору типа экземпляра и виртуального диска. Для Azure SQL выбор ценового уровня носит некоторую абстрактность — можно сказать, что двойная разница величины DTU приводит примерно к такой же разнице в производительности. Кроме того, такая абстракция позволяет использовать встроенную систему анализа влияния запросов на производительность (query insight), что, очевидно, практически невозможно в случае экземпляров AWS RDS.

В отличие от AWS RDS сервис AzureSQL больше похож на PaaS, даже скорее на SaaS — у него есть встроенная система мониторинга, контроля доступа через списки запрещенных и разрешенных диапазонов IP, встроенный редактор кода, нет связи с IaaS-сервисами Azure и отсутствуют кнопки перезагрузки и включения-выключения. Данное обстоятельство, а также то, что серверы Azure SQL являются чисто логической группировкой БД, вынуждает использовать Elastic Database Pool для построения систем из многих баз, нагружаемых в разное время. Для AWS RDS в таком сервисе нет необходимости: базы могут быть размещены на одном экземпляре и его ресурсы будут разделены между ними.

Обе системы предоставляют сервисы периодического резервного копирования, создания реплики и географического разнесения.

И конечно, самое важное различие — Azure SQL поддерживает только подмножество T-SQL и является реализацией MS SQL в облаках, а AWS RDS представляют целый ряд различных движков. Эта разница частично нивелируется появлением MySQL и PostgresSQL у Azure в виде сервисов PaaS. К несомненным преимуществам сервиса Azure SQL можно отнести наличие многих встроенных сервисов анализа и оптимизации производительности, в том числе автоматической оптимизации. Эти сервисы действительно полезны, и я очень рекомендую хотя бы ознакомиться с ними. Вероятно, Microsoft стремится построить облачную базу данных с возможностью оптимизации структуры индексов и планов динамического выполнения.

В случае же RDS построить подобные системы весьма непросто ввиду многообразия движков, которые имеют максимальную совместимость с традиционными движками баз, что позволяет БД гораздо проще мигрировать из локальных серверов напрямую в RDS. В отличие от этой концепции Azure SQL не полностью совместим с Microsoft SQL и поддерживает только подмножество языка T-SQL, что делает миграцию в облака более трудной, но выполнимой задачей. Я лично выполнял проект по миграции информационной системы из локальных серверов в облачную среду с максимальным использованием возможностей PaaS. Сервер MS SQL 2008 был успешно перенесен в Azure SQL, что потребовало изменения архитектуры и более 1000 объектов (из примерно 5000) в базе (триггеров, хранимых процедур, пользовательских функций, представлений и пр.) и заняло около полугода. Но в итоге после всех проведенных операций доступность, надежность и производительность сервера возросли, а стоимость обслуживания существенно упала (исключились затраты на администрирование, аренду сервера, аренду серверной комнаты с климат-контролем, лицензию, хранение бэкапа).

6 Нереляционные базы данных

Там, в мозгу, ведь нет никаких слов, чувств.

Воспоминание человека — образ, записанный языком нуклеиновых кислот на макромолекулярных апериодических кристаллах.

Станислав Лем. Солярис

6.1. Общий обзор баз данных NoSQL

Описанные в предыдущей главе реляционные базы данных имеют ряд неоспоримых преимуществ, среди которых простота и удобство построения структуры на основе моделирования сущностей реального мира, нормализованность и отсутствие избыточности информации. Однако этим базам свойственны недостатки, ограничивающие применение РБД для хранения больших данных. Прежде всего, это невозможность существенного горизонтального масштабирования; единственное, что могут предложить реляционные базы, — топологию ведущей БД (master) и пары реплик, сконфигурированных только для чтения данных. Такое разделение на одну ведущую и реплики, доступные только для чтения, обусловлено тем, что реляционная модель данных требует их полной согласованности для всех реплик, чтобы обеспечить принципы ACID, лежащие в основе модели согласованности реляционной БД.

С другой стороны, очень многие направления современных информационных систем, оперирующих большими данными, не нуждаются в обеспечении строгой согласованности и транзакционности, но требуют очень быстрого доступа к произвольному подмножеству хранимой информации и возможности хранения гигантских объемов данных — сотен терабайт.

Примерные направления приведены ниже.

- IoT — поток телеметрии с устройств часто представляет собой поток сообщений, каждое из которых включает произвольный набор полей полезной нагрузки.

Часто очень удобно хранить подобные данные в виде таблицы, допускающей отсутствие схемы (то есть каждая строка может содержать свои наборы колонок, общими являются поля ключей и временной метки). Данных может быть очень много, и они должны быть доступны для извлечения запросом, выполняющимся за минимальное время. Нереляционные базы как раз предоставляют тип хранилища, обеспечивающий работу с таким видом информации, — *табличные базы данных*, называемые еще *базами данных типа «ключ — значение»*.

- Социальные сети или блог-платформы. В этом случае можно столкнуться с трудностями двух видов. Во-первых, пользователи социальных сетей связаны друг с другом связями типа «друзья», «последователи», «состоящие в одной группе», «подписчики». Во-вторых, пользователи генерируют контент, состоящий из сообщений, постов, комментариев, оценок и др. В силу особенности предметной области для хранения данных пользователей наиболее удобны так называемые *графовые БД*, то есть базы, в которых информация представлена в виде графа. Теперь вернемся к генерируемому пользователем контенту. Помимо связей с многими пользователями (в качестве примера можно привести пост, содержащий много комментариев разных пользователей и оценок этих комментариев), этот контент может включать медиафайлы: фото, видео, музыку. И совершенно очевидно, что пользователи социальных сетей ожидают максимального быстродействия: то есть буквально пары секунд на перезагрузку страницы, чтобы увидеть свой комментарий и сотни других вне зависимости от наличия или отсутствия в них медиафайлов. С таким вызовом прекрасно справляются *документоориентированные БД*. Информация в них хранится в виде JSON-документов (в данном случае это логическое понятие единицы хранения информации), включающих в том числе метаданные других документов, медиафайлов.
- Системы рекомендации контента. В данном случае задача состоит в том, чтобы в существующую систему (социальную сеть, блог, интернет-магазин, сайт с фото- или видеоконтентом) встроить подсистему, предоставляющую пользователю ссылку на потенциально интересный или полезный товар (статью, контент). Для этого необходимо хранить информацию об истории посещения пользователем страниц, а также об истории посещения страниц других пользователей и определять сходные паттерны в поведении. Существует достаточно много алгоритмов такой рекомендации, но все они строятся на основе информации о просмотренных пользователями страницах и выявления закономерностей в этом процессе. Данная информация может храниться как в БД «ключ — значение», так и в графовой.

Как видите, зачастую большие данные гораздо удобнее хранить в виде, отличном от реляционного. Однако их прямой анализ может быть затруднен, поскольку подобные хранилища приспособлены для хранения и выборки данных с целью отображения. В таком случае необходимо применить сервисы копирования и трансформации данных для доставки последних в другие хранилища, более

приспособленные для целей аналитики (эти темы подробно будут освещаться в других частях книги).

Теперь подробнее разберем основные типы нереляционных баз данных и концепции, лежащие в их основе.

В настоящее время в облачных средах Microsoft Azure и Amazon Web Services реализованы четыре типа нереляционных баз данных: «ключ — значение», графовые, документоориентированные и семейства столбцов.

Итак, первый и наиболее распространенный тип нереляционной БД — *база данных типа «ключ — значение»*. Строго говоря, к этому типу можно отнести не только табличные базы, но и те, в которых информация хранится и доступна по ключу, например Redis, Memcache. Но подобные базы чаще всего служат для кэширования запросов к основной, но более медленной БД и содержат только временную информацию, действительную в течение ограниченного промежутка времени, потеря которой совершенно некритична для системы. Мы не будем рассматривать эти БД, поскольку они не используются для непосредственного хранения больших данных.

Итак, в средах Azure и AWS базы данных типа «ключ — значение» представлены как отдельными PaaS-сервисами (например, Azure Table Storage, AWS DynamoDB), так и в виде PaaS/IaaS (AWS EMR HBase, Azure HDInsight HBase). Рассмотрим базовые сущности, из которых состоят БД типа «ключ — значение», реализованные в модели PaaS.

Верхний логический элемент хранилища «ключ — значение» — *аккаунт* (account) (для Azure это может быть Azure Storage Account). Аккаунт состоит из *таблиц* (table), которые представляют собой контейнеры для хранения единиц информации. Таблица состоит из *сущностей* (entity) (по сути, являющихся строками), которые имеют такое название, чтобы подчеркнуть: в отличие от строк таблицы реляционной БД сущности могут состоять из произвольного числа *свойств* (properties). Но ряд свойств обязательно содержится во всех сущностях: *ключ раздела* (partition key), *ключ строки* (row key) и *временная метка* (timestamp). Два ключа необходимы в силу специфики хранения данных этих таблиц в облачных средах. Дело в том, что таблицы хранятся на SSD-дисках (напрямую или в массиве RAID) — в разделах — и те строки таблицы, которые имеют одинаковый ключ раздела, будут храниться физически вместе на общих дисках или в RAID. Напротив, ключ строки — уникальный идентификатор сущности в пределах раздела, то есть комбинация «ключ раздела — ключ строки» уникальна. Временная метка показывает время последней модификации сущности. Индексация сущностей происходит автоматически — очевидно индексируются ключевые поля. Это обстоятельство требует тщательного продумывания структуры такой таблицы для целей хранения/добавления/обновления/выборки данных. Дело в том, что базы типа «ключ — значение» допускают в ряде случаев применение механизма транзакций для обновления группы строк, но этот механизм используется для строк с общим ключом раздела.

Вообще конкретная структура таблиц зависит от финишных требований конечной информационной системы: она должна обеспечивать высокую произво-

дительность при записи или чтении, сущности необходимо связать отношениями «один к одному», «один ко многим» или иерархически.

Рассмотрим следующий тип нереляционной базы данных — *графовую БД*. Он идеально подходит для хранения информации, моделирующей связи между сущностями реального мира. Как уже отмечалось, это могут быть пользователи в социальной сети, связанные отношениями с другими пользователями. В качестве модели такой предметной области выступает *граф* — математический объект, состоящий из *вершин* и *связей* (рис. 6.1).

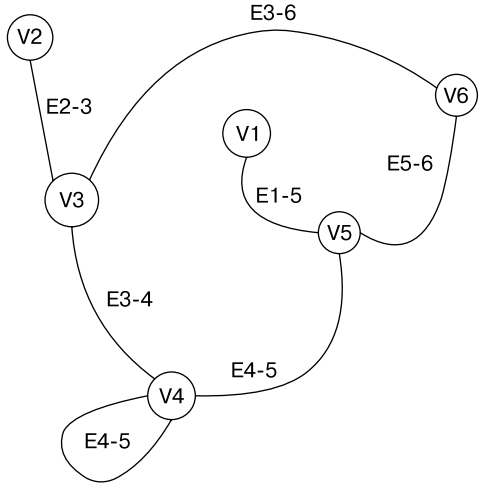


Рис. 6.1. Образец графа, состоящего из вершин V и ребер E

Вершины — точки соединения ребер, причем важен факт соединения вершин ребрами, а не сама форма графа (это называется изоморфизм). Каждая вершина может иметь идентификатор (имя) и набор свойств, относящихся к сущности, которую представляет эта вершина. Каждая связь (ребро графа) может иметь свойство или вес. Свойством может описываться тип связи («друг», «последователь», «родственник» и пр.), а весом — некая величина, характеризующая взаимодействие между вершинами. Например, если вершинами представить предприятия разветвленного концерна, то веса могут моделировать, скажем, величину транспортного потока товаров между предприятиями или финансовые потоки.

Графовые БД предоставляют специальные языки запросов, удобные для выборки информации, относящейся к графу. Более подробно этот вопрос будет рассмотрен в части III. В облачных средах эти базы представлены у Azure (Azure CosmosDB Graph API) и с недавнего времени у Amazon (AWS Neptune). Графовые БД очень удобны для хранения информации в различного рода социальных сетях. Узлами сети будут являться ее пользователи, а ребрами — отношения между ними: «друзья», «подписчики», «члены группы» и др. Традиционная РБД

в подобном случае неудобна в использовании, поскольку представление информации о графовой структуре часто требует денормализованного представления в таблицах, что противоречит самой концепции РБД. А вот графовая база данных здесь будет весьма кстати.

Но важным является не только хранение информации, но и возможность извлечения ее части. Например, для той же социальной сети можно использовать концепцию ленты для каждого пользователя, то есть списка новостей и публично доступных сообщений и событий, опубликованных его друзьями. Для этого в графовой базе данных необходимо выбрать все смежные вершины, то есть все вершины, находящиеся на расстоянии одного ребра от вершины текущего пользователя. Далее для каждого из этих пользователей извлекается список доступных событий (с помощью, скажем, фильтрации «за последний час», «за последний день»), которые комбинируются в один общий список, сортируются по времени публикации в порядке от самых свежих к более поздним и выдаются в ленту. Возможен и несколько другой принцип построения ленты: все события публикуются в один общий для всех пользователей список; из него для каждого конкретного пользователя извлекаются только события, опубликованные его друзьями, группами, на которые он подписан, и пр. Как видите, в обоих случаях наиболее логичный выбор типа хранилища ленты — это базы данных типа «ключ — значение». Ключом раздела можно выбрать идентификатор пользователя, опубликовавшего событие, ключом строки — идентификатор сообщения, включающий временную метку (очевидно, одновременно он будет являться ключом сортировки), значение — ссылка на контент.

Следующий популярный тип нереляционных баз данных — *документоориентированная БД*. Рассмотрим, из каких элементов состоит такая база (рис. 6.2).

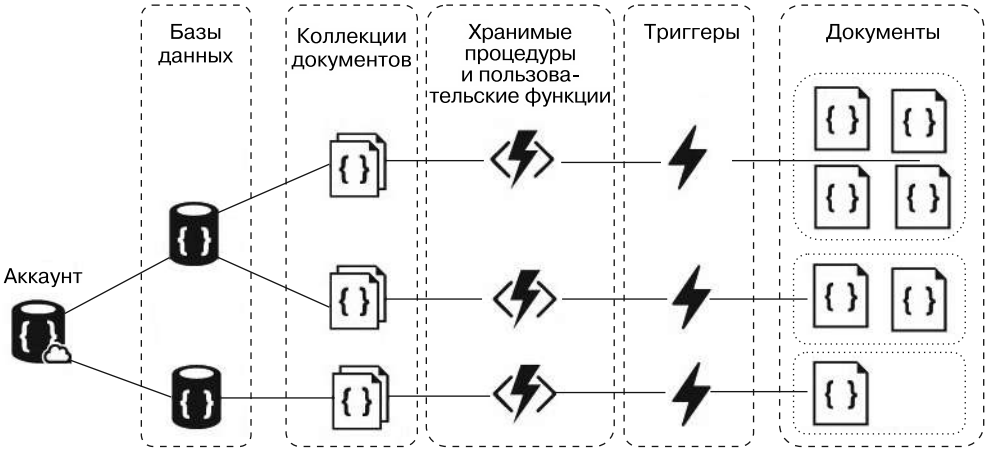


Рис. 6.2. Состав документоориентированной базы данных на примере DocumentDB

Прежде всего, это *аккаунт*, который содержит одну или несколько *баз данных*. База, в свою очередь, состоит из *пользователей*, которые характеризуются набором привилегий и прав доступа, а также из *коллекций*. В последние включены хранимые процедуры, пользовательские функции, триггеры, документы и прикрепленные файлы.

Рассмотрим элементы коллекции. Самый главный элемент — это документ, хранящий информацию в формате JSON. Набор полей в документе может быть произвольным, но, как и в случае с базами типа «ключ — значение», в этом документе есть ряд полей обязательных и добавляемых по умолчанию. Представление информации в формате JSON очень удобно, поскольку ее структура (набор полей, типы данных и др.) может быть произвольной. Кроме того, информация, хранящаяся в этом формате, очень легко интегрируется в одностраничные веб-приложения (single page application, SPA). В частности, для построения систем с единым языком на разных уровнях очень удобны технологии ReactJS, AngularJS (для клиентской части), NodeJS (для серверной) и документоориентированная база данных типа MongoDB. Существует целый стек технологий, «насквозь» состоящий из JSON/JavaScript. Это MEAN — *M*ongoDB, *E*xpressJS, *A*ngularJS, *N*odeJS.

Кроме собственно хранения информации в формате JSON, документоориентированная база данных предоставляет средство выполнения запросов. В качестве языка запросов может выступать JavaScript, нотация JSON, а в ряде случаев и SQL (в частности, поэтому в облачной среде Azure база данных DocumentDB в составе Azure CosmosDB называется SQL API — не стоит путать ее с сервисом AzureSQL!). Набор операторов языка запросов, объединенных в единую именованную группу, называется хранимыми процедурами, пользовательскими функциями или триггерами. Для триггеров и хранимых процедур гарантируется выполнение ACID-принципов обновления информации в базе данных для одновременного обновления как одного, так и многих документов. Документоориентированные БД очень удобны, когда необходимо хранить информацию в виде отдельных документов, представляющих каждый некое событие, при этом различные документы могут иметь различный набор полей, но должны иметь возможность участвовать в основной выборке.

Нереляционные базы данных типа *семейства столбцов* (column family) обеспечивают хранение информации в виде разреженной матрицы, у которой строки и столбцы используются как ключи.

Обратимся к нашему умозрительному примеру с социальной сетью. Помимо пользователей и контента, важной составляющей любой сети являются комментарии, которые могут быть добавлены и к контенту, и к другим комментариям. При этом возможны различные уровни комментирования. Скажем, контент прокомментировал пользователь А, его сообщение прокомментировал пользователь Б, пользователь А ответил на комментарий пользователя Б, на что ему пришли комментарии от пользователей Г, Д, К, и одновременно его первоначальный комментарий

прокомментировал пользователь Ж. Очевидно, что в этом случае графовая и тем более реляционная базы данных не очень удобны для представления информации. Гораздо удобнее представить комментарии как цепочку связанных JSON-документов или один большой такой документ. Каждый уровень комментирования соответствует уровню вложенности объектов JSON.

Таким образом, наша гипотетическая виртуальная сеть может состоять из нереляционных баз данных типа «ключ — значение» (для представления общей ленты), графовой (для представления связей между пользователями) и документоориентированной (для хранения комментариев к контенту). Конечно, это чрезвычайно упрощенный пример, но он дает наглядное представление и частично отвечает на вопрос, а зачем, собственно, нужны нереляционные БД в реальной жизни.

Рассмотрим более подробно сервисы нереляционных баз данных, предоставляемых облачными средами Microsoft Azure и Amazon Web Services.

6.2. Сервисы нереляционных баз данных от Azure

В настоящее время Azure предоставляет два сервиса нереляционных хранилищ данных: традиционный Azure Table Storage и Azure CosmosDB. Первый — хранилище типа «ключ — значение». Второй — набор глобально распределенных нереляционных хранилищ, объединенных общей концепцией управления и создания, к которым в настоящее время относятся базы данных DocumentDB и MongoDB (документоориентированные), Gremlin (графовая база данных), Table (таблица типа «ключ — значение») и Cassandra (база данных, относящаяся к типу семейства столбцов).

Начнем с рассмотрения Azure Table Storage. Этот сервис является встроенным в Azure Storage Account и создается на общей панели, содержащей остальные сервисы (Blob, File и Queue). Общая панель доступа к Azure Table Storage такая же, как и для других сервисов, включенных в Azure Storage. Она очень проста и содержит минимальные опции для конфигурации (рис. 6.3).

Это хранилище поддерживает протокол OData для выборки данных на основе протокола HTTP, так что таблица имеет конфигурации для настройки CORS и политики доступа. Сам сервис очень минималистичен, и основное конфигурирование будет происходить не на портале, а с помощью кода (этот вопрос более подробно мы рассмотрим в части III, где будут приведены примеры кода). Пока же опишу основные концепции и паттерны данного хранилища.

Само по себе хранилище Azure Table Storage предназначено для хранения больших объемов (в настоящее время это 500 Тбайт на один Storage Account) структурированных нереляционных данных. Под структурированностью здесь понимается тот факт, что все сущности («строки») представляют собой набор пар «ключ — значение» (причем не обязательно, чтобы он был одинаковым во всех

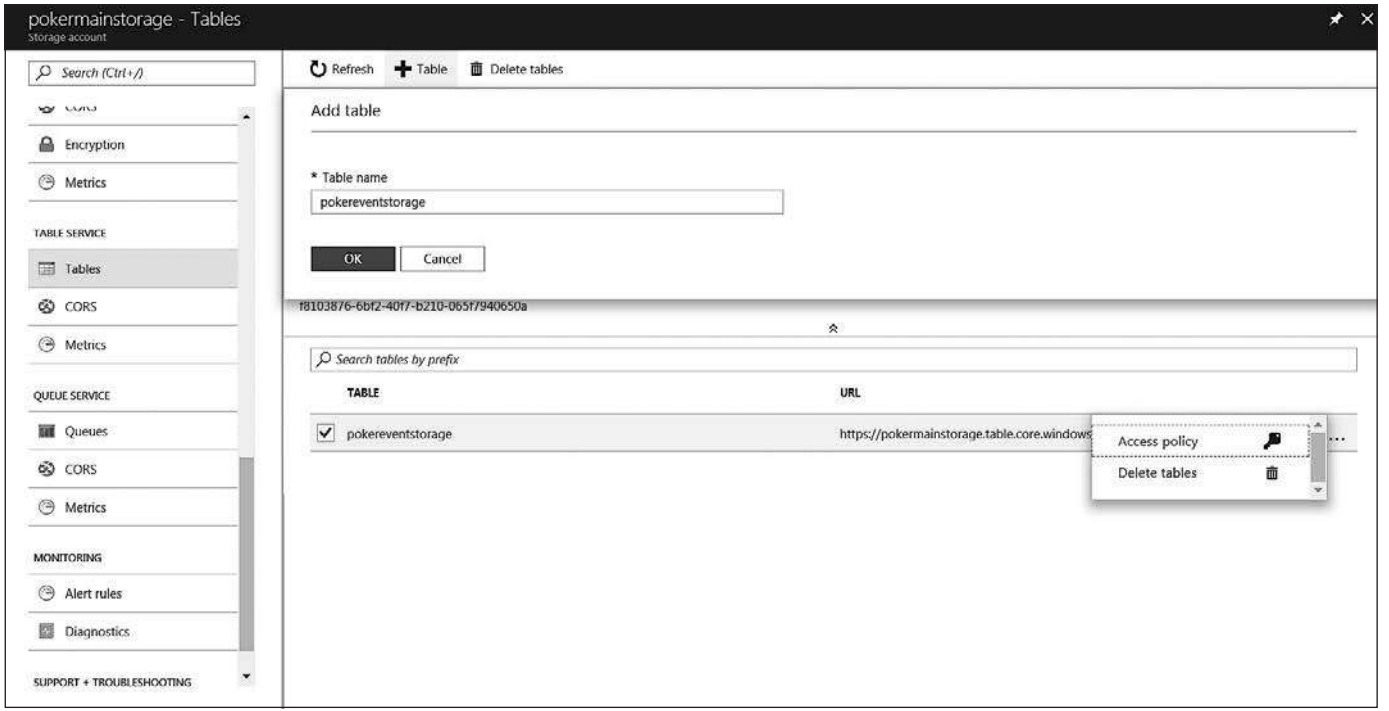


Рис. 6.3. Внешний вид общей панели хранилища Azure Table Storage

строках). По сути, это огромные таблицы, которые нецелесообразно размещать в РБД. В качестве примера можно привести сырые потоковые данные, помещенные в таблицу сервисом Stream Analytics Job (о нем более подробно см. в части IV), которые в итоге должны быть агрегированы сервисом ETL (например, сервисом Azure DataFactory в комбинации с Azure HDInsight Spark) и помещены в реляционное хранилище данных (Azure SQL DataWarehouse) или просто в обычную РБД. При этом желательно сохранить все сырые данные, чтобы в последующем можно было выполнить их интерактивный или интеллектуальный анализ. Azure Table Storage позволяет выполнить такую операцию ввиду того, что способен легко интегрироваться с сервисами копирования, трансформации и потокового анализа данных. Кроме того, поддержка протокола OData для прямого доступа и наличие SDK позволяет получить прямой доступ к табличному хранилищу извне. При этом выполнение высокопроизводительных запросов возможно благодаря встроенному механизму кластерных индексов.

Главные компоненты Azure Table Storage показаны на рис. 6.4.

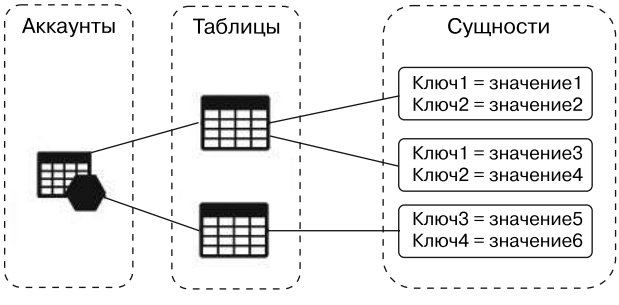


Рис. 6.4. Базовые компоненты сервиса Azure Table Storage

Итак, Storage Table Account включает в себя набор *таблиц*, каждая из которых вмещает набор *сущностей*, то есть строк. Каждая сущность состоит из набора *свойств* и имеет суммарный размер не более 1 Мбайт. В свою очередь, свойство — это пара «имя — значение». Каждая сущность может иметь до 256 свойств, три из которых обязательны: ключ раздела (partition key), ключ строки (row key) и временная метка (timestamp). Временная метка соответствует времени последней модификации этой сущности. Как уже указывалось, ключ строки должен быть уникальным в пределах раздела, а комбинация «ключ раздела — ключ строки» — уникальной глобально. Для набора, состоящего из сущностей с одинаковым значением ключа раздела, значения могут быть очень быстро выбраны с помощью запроса. Операции вставки/удаления/обновления в данном случае тоже выполняются быстрее. Это возможно потому, что на оба ключа создаются кластерные индексы. Никакие другие индексы создать нельзя.

Типы данных, которые поддерживаются Table Storage, полностью соответствуют типам данных, доступных в протоколе OData (табл. 6.1).

Таблица. 6.1. Типы данных стандарта OData

Тип данных OData	CLR-тип, доступный в коде	Примечание
Edm.Binary	byte[ ]	Массив байтов размером до 64 Кбайт
Edm.Boolean	Bool	Булево значение
Edm.DateTime	DateTime	64-разрядное значение, представляющее собой временную метку UTC
Edm.Double	Double	64-разрядное число с плавающей точкой
Edm.Guid	Guid	128-разрядный глобальный идентификатор
Edm.Int32	int / Int32	32-разрядное целое
Edm.Int64	long / Int64	64-разрядное целое
Edm.String	String	Строковая величина с кодировкой UTF-16. Максимальный размер строки — 64 Кбайт

Тип по умолчанию — строковый. Для хранения более сложные типы данных должны быть сериализованы в XML или JSON и помещены в строковый формат. Второй вариант — сериализация в двоичный формат и помещение его в битовый массив.

Модели хранения данных нереляционных хранилищ типа «ключ — значение» не ограничиваются структурами типа «одна большая таблица» или «набор несвязанных таблиц». Можно моделировать отношения и построение таблиц, оптимизированных для решения определенного круга задач (быстрые запросы на выборку, быстрая модификация и пр.). Прекрасный документ, описывающий паттерны и антипаттерны применения Table Storage, находится по адресу https://docs.microsoft.com/ru-ru/azure/cosmos-db/table-storage-design-guide. (Я предпочитаю англоязычную версию статьи и вообще документацию на этом языке, поскольку, на мой взгляд, перевод порой очень хромает стилистически, но вы можете и не знать английского, поэтому я отсылаю к русскоязычной версии.)

Рассмотрим теперь базы данных от сервиса Azure CosmosDB. Он является относительно новым и предоставляет единую программную модель для доступа к нереляционным базам разных типов:

- DocumentDB, она же SQL API, — документоориентированная база данных с возможностью выполнения запросов с помощью как SQL, так и JavaScript;
- MongoDB — облачный сервис хорошо известной базы с таким же названием;
- Graph API — сервис графовой базы данных, называемой еще Gremlin;
- Table API — дальнейшее развитие базы данных типа «ключ — значение», почти полный аналог Azure Table Storage;
- Cassandra — сервис, являющийся адаптацией Apache Cassandra.

Указанные выше базы собраны в единый сервис, предоставляющий всем им ряд общих уникальных черт и свойств. Прежде всего, это *глобальная доступность* — все

перечисленные БД можно реплицировать во *все* регионы, то есть создать копии баз во всех регионах одновременно! Ну или только в выбранных. При этом в зависимости от географического местоположения клиента его запрос будет направлен к ближайшему дата-центру. Уровень согласованности данных можно настроить вплоть до уровня сильной согласованности (strong consistency). Кроме того, полоса пропускания и размер базы данных могут быть настроены независимо или сконфигурированы на автоматическое расширение. Уровень доступности для этого сервиса (для чтения) гарантируется на уровне 99,99 % применительно к одиночной БД без географической репликации и 99,999 % для репликации в нескольких географических регионах. При этом медианное время задержки в документации указывается равным 5 мс.

Познакомимся подробнее с сервисом Azure CosmosDB и его возможностями, а затем перейдем к частным типам БД. Прежде всего необходимо создать аккаунт CosmosDB, в рамках которого мы будем создавать базы, а позже и сущности хранения в них. Для создания аккаунта нужно в левом верхнем углу портала нажать на ссылку + New и в появившемся окне поиска выбрать CosmosDB (рис. 6.5).

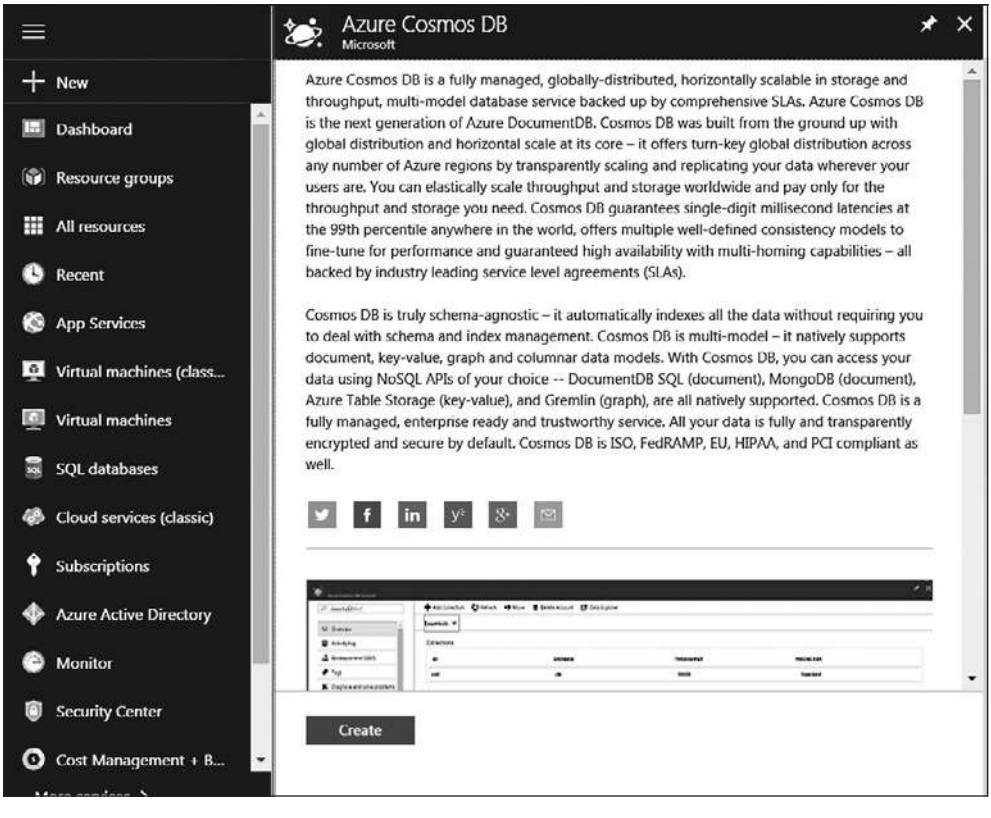


Рис. 6.5. Начальная страница сервиса Azure CosmosDB

После нажатия кнопки Create (Создать) появится следующая форма (рис. 6.6).

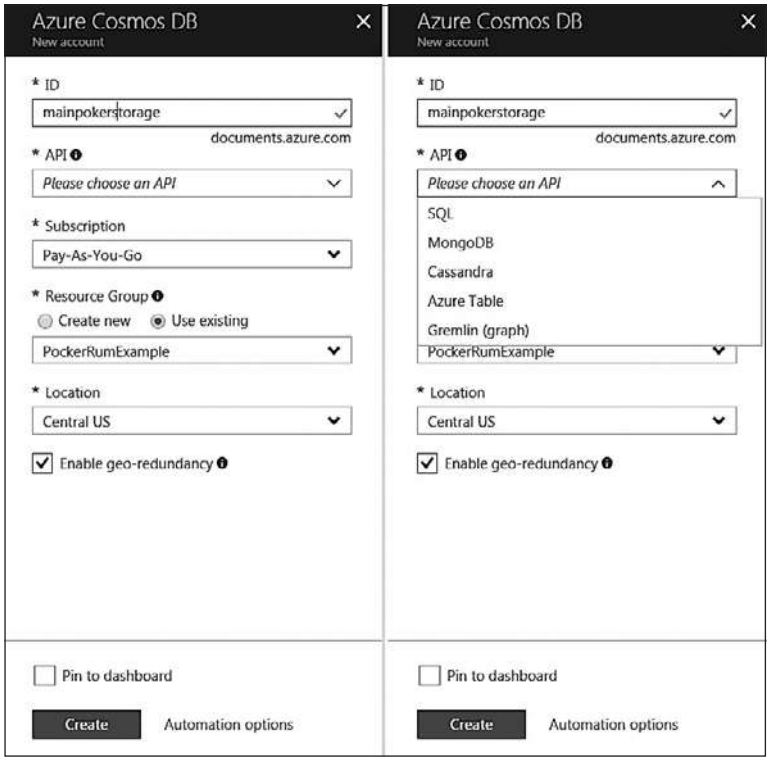


Рис. 6.6. Форма создания аккаунта CosmosDB

Прежде всего здесь необходимо указать идентификатор аккаунта (ID), выбрать его тип (API), группу ресурсов (Resource Group), местоположение и включить опцию географического дублирования (Enable geo-redundancy). Список доступных API в развернутом виде представлен на правой половине рисунка. Вас не должна смущать аббревиатура SQL: имеется в виду база данных SQL API, которая, в свою очередь, соответствует DocumentDB (вероятно, такое странное название было дано в связи с тем, что DocumentDB допускает выполнение запросов к данным, написанным с использованием SQL-подобного синтаксиса). Таким образом, при создании нужно выбрать SQL API и нажать кнопку Create (Создать).

После создания аккаунта CosmosDB типа SQL API будет доступна стартовая страница, показанная на рис. 6.7. Для всех типов поддерживаемых баз данных она выглядит примерно одинаково. С целью повышения наглядности на рис. 6.7 показаны два положения скролла левой навигационной панели: крайнее верхнее и крайнее нижнее. Пока что это пустой аккаунт, в котором нет коллекций. Добавить коллекцию можно прямо на стартовой странице, но мы пойдем на вкладку Overview (рис. 6.8).

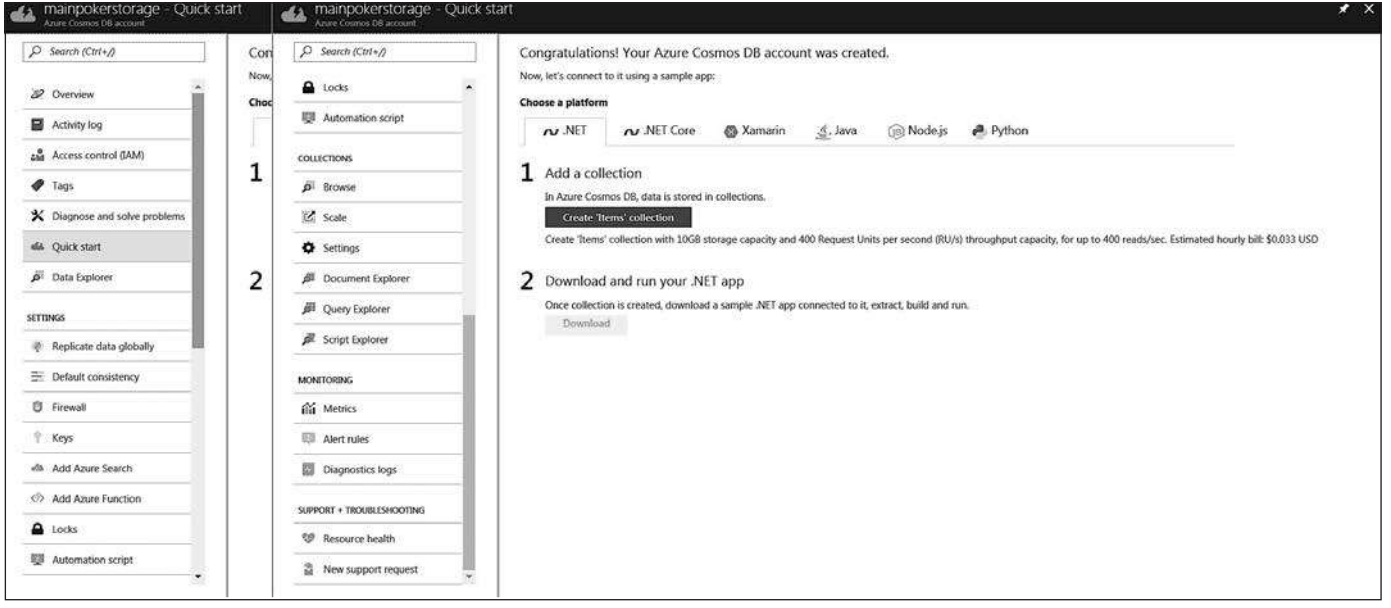


Рис. 6.7. Стартовая страница сервиса CosmosDB

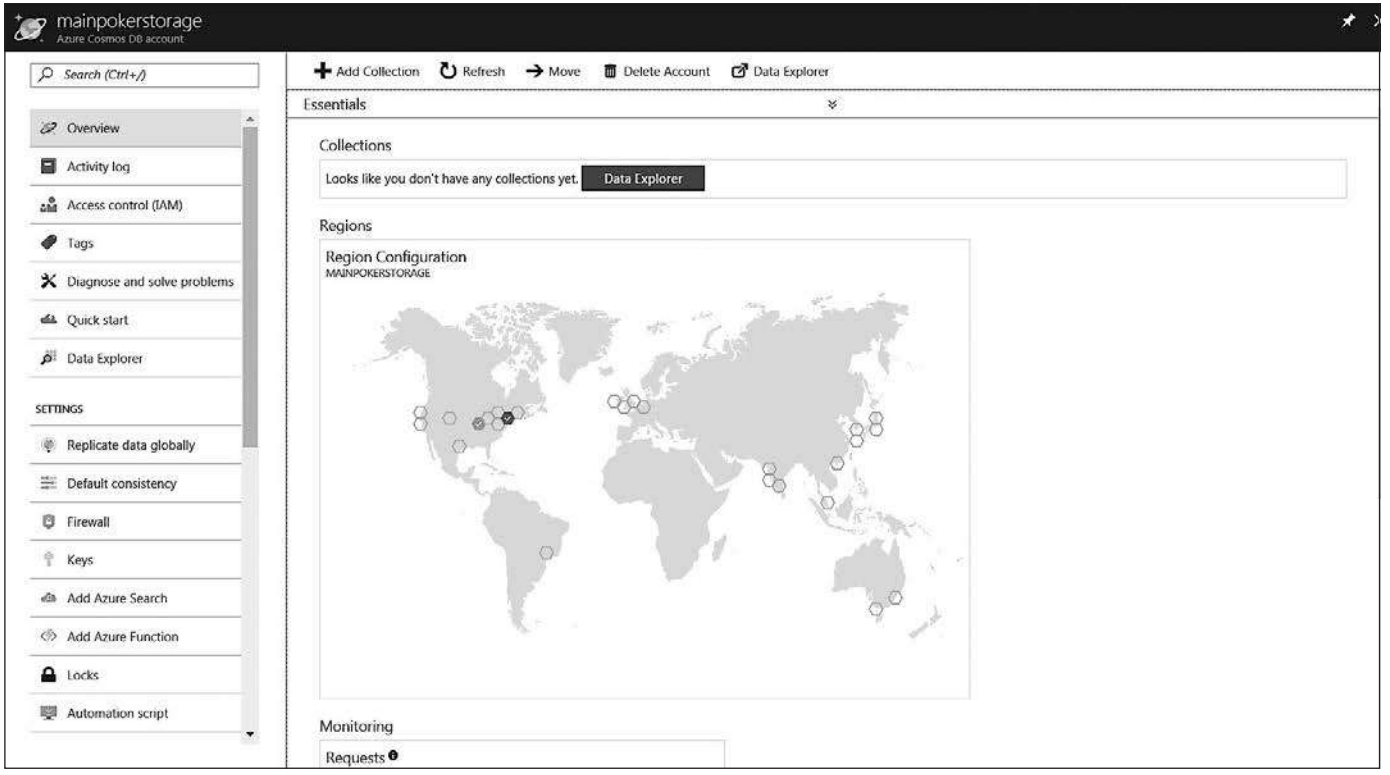


Рис. 6.8. Страница обзора аккаунта

Перейдя на вкладку Overview, можно видеть на карте доступные для репликации регионы (прозрачные шестиугольники) и регионы, использованные в настоящий момент (закрашенные шестиугольники). Кроме карты, на этой странице доступна панель мониторинга, которая размещается внизу экрана и не поместилась на снимке.

Чтобы добавить коллекцию, необходимо нажать ссылку + Add Collection в верхней части экрана, после чего откроется форма, показанная на рис. 6.9. Поскольку у нас еще нет баз данных, нужно создать первую базу данных (Database id) — `PockerGameData` и в ее рамках создать коллекцию (Collection id) — `GameEvents`.

Далее следует выбрать размер хранилища (Storage capacity). Он может быть фиксированным (опция Fixed (10 GB)) и неограниченным (опция Unlimited), то есть автоматически масштабируемым. Кроме того, нужно выбрать производительность (выражена в терминах единиц чтения в секунду) из заданного диапазона. Минимальное значение равно `400`. Далее можно добавить в коллекцию документов уникальный ключ (ссылка + Add unique key) и нажать OK. Созданная база данных и коллекция показана на рис. 6.10.

Теперь вернемся к рис. 6.2, поясняющему структуру DocumentDB (а это не что иное, как SQL API). Аккаунты CosmosDB точно так же состоят из одной или нескольких баз данных, которые, в свою очередь, состоят из одной или нескольких коллекций. Эти два элемента сервиса CosmosDB SQL API мы и создали на предыдущем шаге. Каждая коллекция, в свою очередь, состоит из хранимых процедур (stored procedure), определяемых пользователем функций (user defined functions), триггеров (triggers) и собственно документов (documents). Разберем подробнее, что такое документ в терминах DocumentDB, на примере его создания. Для этого на вкладке Data Explorer перейдем к нашим созданным базе данных и коллекции и нажмем ссылку Documents. На появившейся вкладке нажмем на ссылку New Document, в результате изображение на экране будет выглядеть следующим образом (рис. 6.11).

Итак, после нажатия на ссылке New Document создается JSON-файл с единственным обязательным полем, равным `id`. Значение этого поля содержит призыв на английском, который переводится как «замени новым document_id». Это первичный ключ, который должен быть уникальным. В данный документ можно добавить произвольное количество полей, определяемых пользователем. Необходимые изменения, внесенные в документ, показаны в средней части рисунка: первичный ключ равен `1`, а пользовательское поле `"custom_key"` — `2`. Далее следует нажать ссылку Save. Список созданных документов, для которых первоначально отображаются только ключи, приведен прямо на этой вкладке (под символом id). Если нажать один из этих ключей, то в окне отобразится документ, содержащий наши поля (ключ и пользовательское поле) и еще ряд служебных полей, что видно в нижней части рисунка. Итак, документы — это элементарные порции информации, хранящиеся в DocumentDB в формате JSON. Как было указано выше, большие данные чаще всего представляют собой большое количество элементарных порций данных, хранящихся в том или ином виде, так вот в DocumentDB элементарной порцией является документ JSON.

Теперь приведем один пример применения SQL-подобного запроса для выборки данных (более подробно применение SQL для построения запросов к нереляционным базам данных будет описано в части IV). Сначала нужно добавить несколько документов с различными значениями пользовательского поля (рис. 6.12).

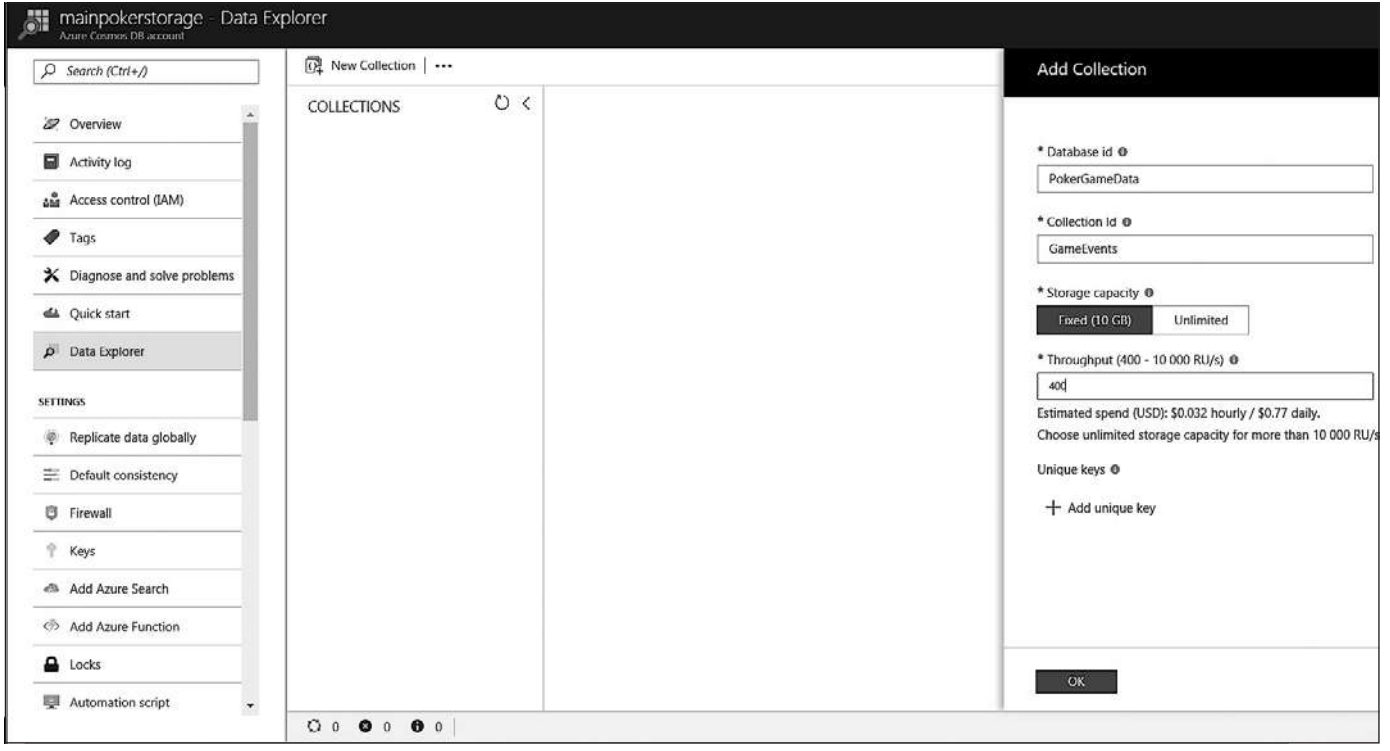


Рис. 6.9. Форма создания базы данных DocumentDB и добавление новой коллекции документов

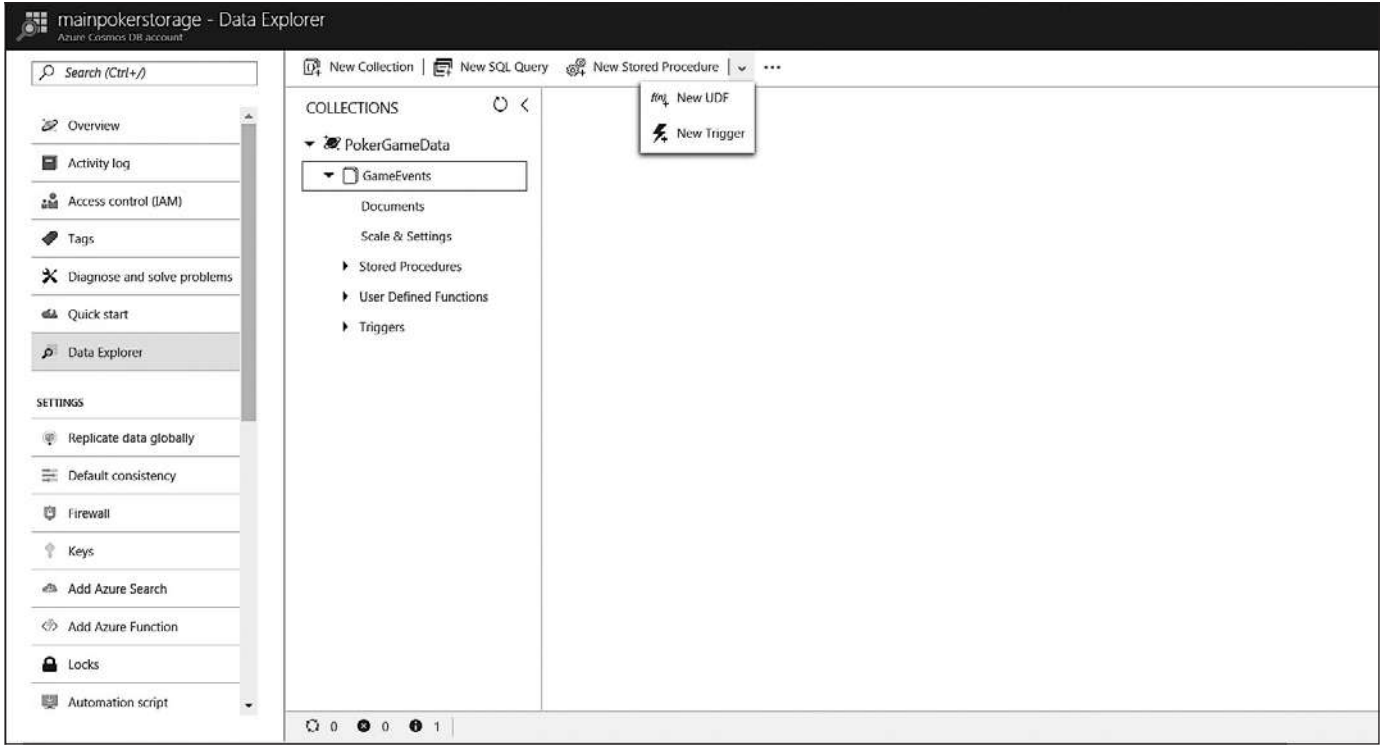


Рис. 6.10. Внешний вид вкладки Data Explorer с созданной базой данных и коллекцией

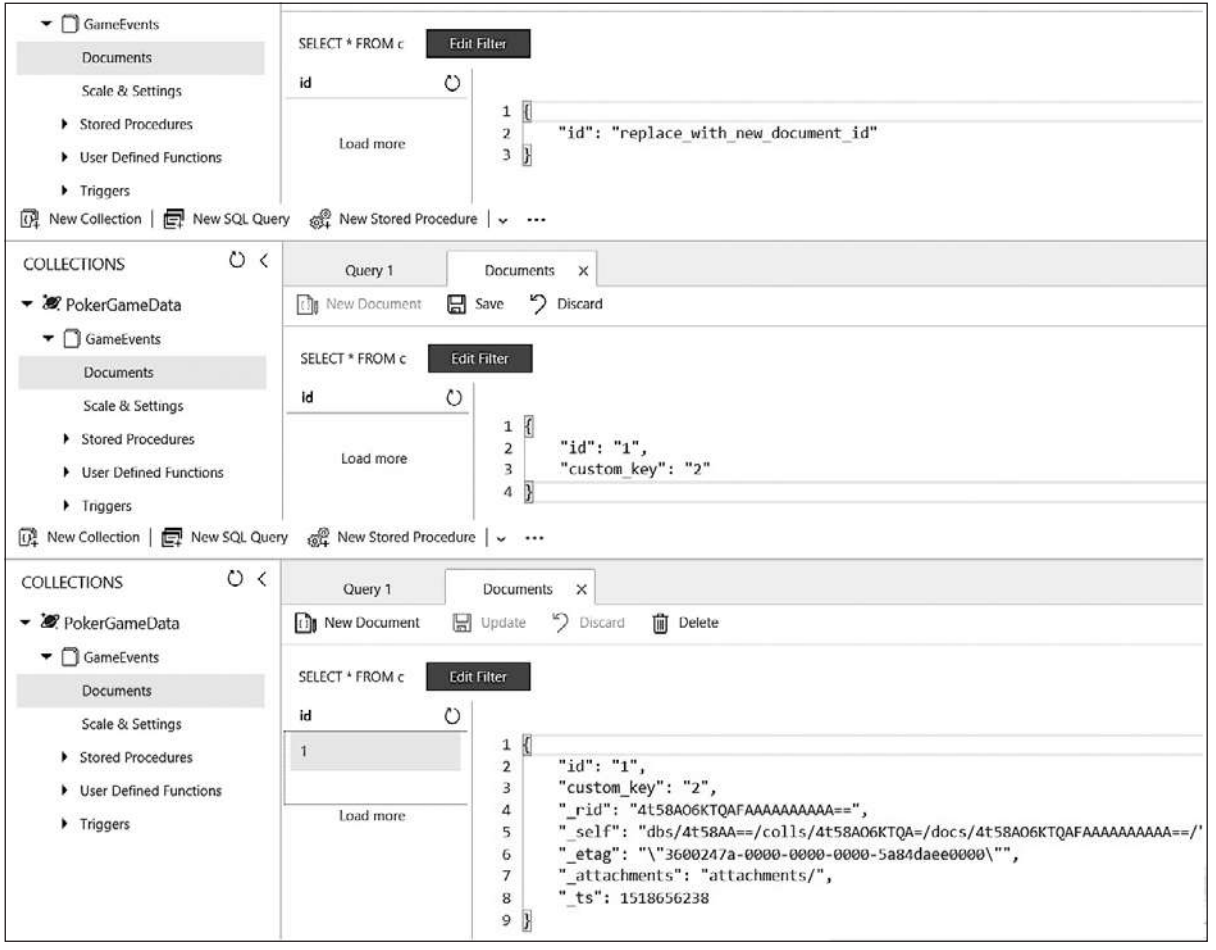


Рис. 6.11. Последовательность создания документа

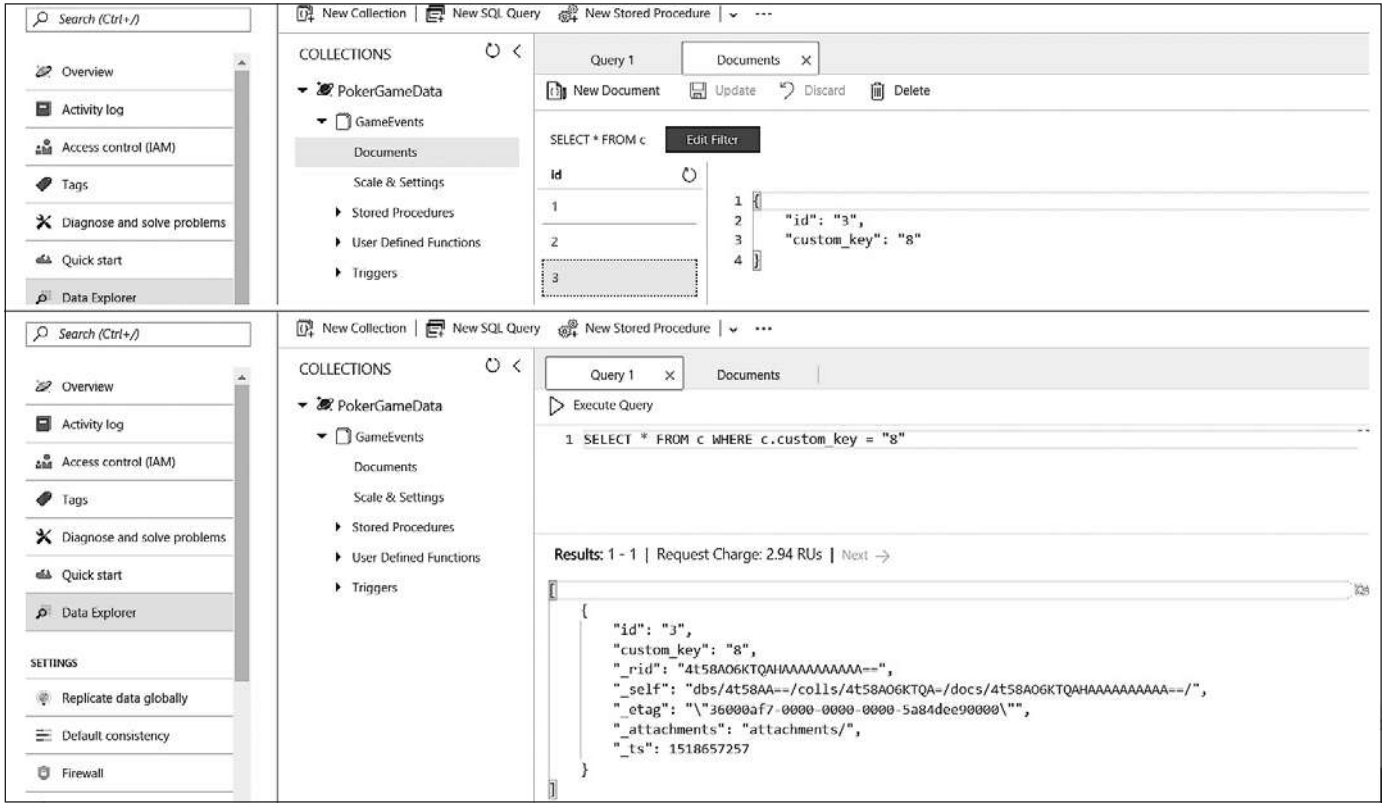


Рис. 6.12. Пример выполнения SQL-запроса к JSON-документам

Для этого типа базы данных доступна «песочница», которая находится по адресу https://www.documentdb.com/sql/demo.

Стоит отметить, что, помимо JSON-документов, документоориентированная база данных позволяет сохранять приложения (attachments) в виде двоичных файлов размером до 2 Гбайт, которые могут быть медиафайлами, архивами и др. (см. свойство `"_attachments"` в нижней части рис. 6.12.).

Пользовательские функции и хранимые процедуры — написанные на языке JavaScript элементы расширения стандартного синтаксиса запросов, которые будут подробнее описаны в части III. Аналоги этих элементов — хранимые процедуры и функции в реляционных базах данных.

Триггеры представляют собой программные конструкции, написанные на языке JavaScript, вызываемые при выполнении операций на документах. Триггер может выполняться непосредственно перед созданием документа (pre-trigger) и после (post-trigger). Очень интересным и важным примером использования триггеров является интеграция с Azure Function — сервисами бессерверного исполнения кода, который может выполняться в ответ на внешний сигнал. Для CosmosDB возможны следующие сценарии запуска (в настоящее время реализованы только для SQL API и Graph API).

- Применение *потока событий изменения* (change feed) — перехват Azure Function событий изменений коллекции (добавление/удаление/изменения). При этом поток событий активизирует пользовательский триггер, который, в свою очередь, запускает Azure Function. Данный подход полностью соответствует концепции EventDriven (рис. 6.13).

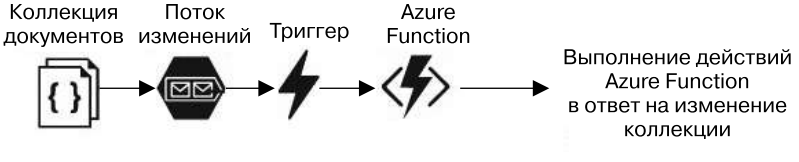


Рис. 6.13. Использование потока событий изменений для вызова Azure Function

- Применение *привязки входа* (input binding) — при этом Azure Function читает данные из базы при запуске сторонним триггером (рис. 6.14).

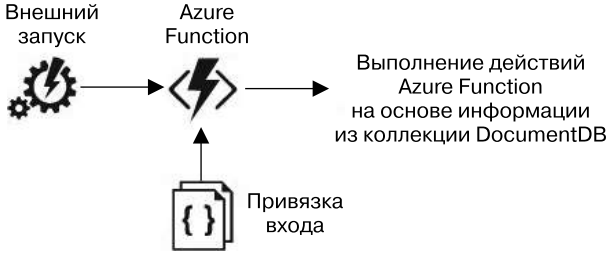


Рис. 6.14. Использование Azure Function с привязкой входа

- Применение *привязки выхода* (output binding) — в этом случае Azure Function записывает данные в базу в ответ на запуск триггером (рис. 6.15).

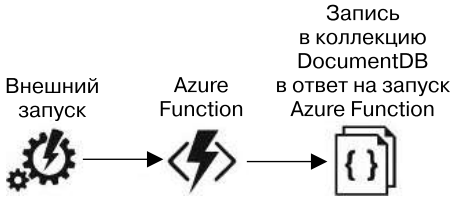


Рис. 6.15. Использование Azure Function с привязкой выхода

Эти паттерны полезны в реальных сценариях и позволяют строить сложные приложения, ориентированные как на события (Event Driven), так и на данные (Data Driven). В части III мы подробно разберем использование этих паттернов на примере покер-рума. А пока двинемся дальше и рассмотрим, какие еще важные опции доступны в сервисе CosmosDB.

Прежде всего, это настройки глобальной репликации (рис. 6.16) и конфигурирования опции отказоустойчивости (рис. 6.17).

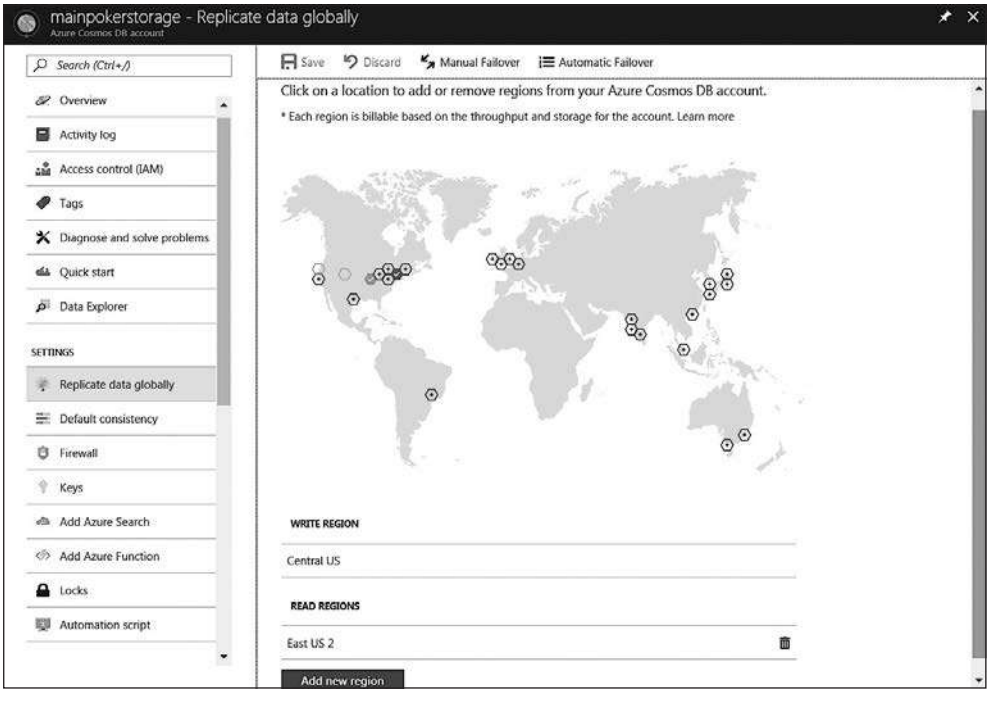


Рис. 6.16. Настройка глобальной репликации

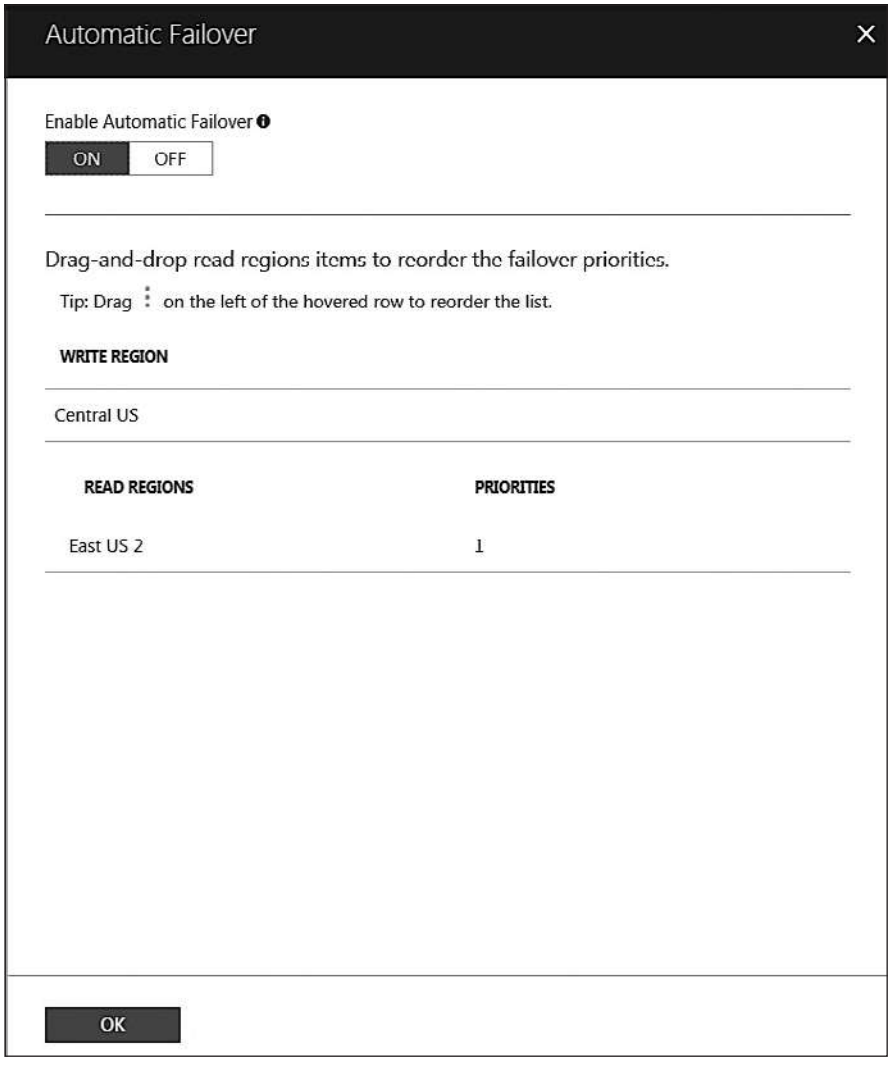


Рис. 6.17. Страница конфигурирования опций отказоустойчивости

Для настройки опций отказоустойчивости необходимо указать резервные головные регионы. Этим регионам назначается приоритет. Данный приоритет будет задавать порядок, который станет использоваться при переключении отказавшей головной реплики на резервную из списка.

Следующий важнейший параметр — настройка уровня согласованности (вкладка Default consistency) (рис. 6.18).

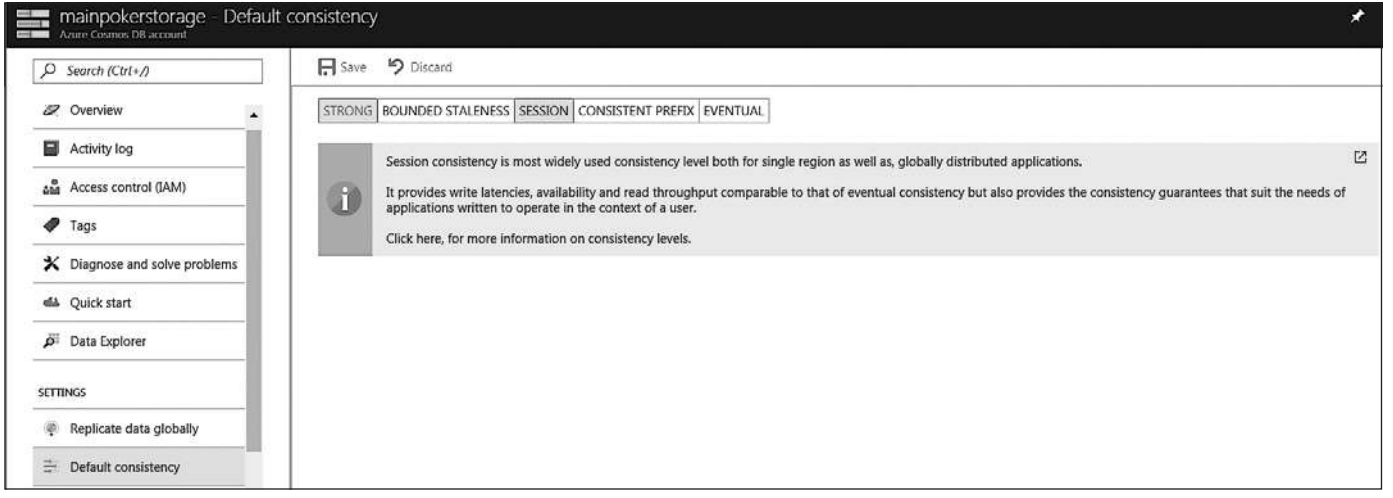


Рис. 6.18. Настройка уровня согласованности

Выбор уровня согласованности зависит от требований конкретной информационной системы. Более высокий (сильный) уровень влечет более высокую цену и увеличение времени записи.

Пользовательский интерфейс графовой базы данных несколько отличается от интерфейса CosmosDB SQL API, в том числе вкладкой DataExplorer. Первое отличие состоит в том, что вместо коллекций создаются графы. А элементарные сущности графа — вершины. Ссылка New Vertex создает новую вершину, к которой можно добавлять метку (label) и ряд пользовательских свойств, представляющих набор «ключ — значение». На рис. 6.19 это свойство Rate. Для каждой вершины есть настраиваемые свойства source и target, определяющие ориентированные ребра графа, которым также можно добавить метку.

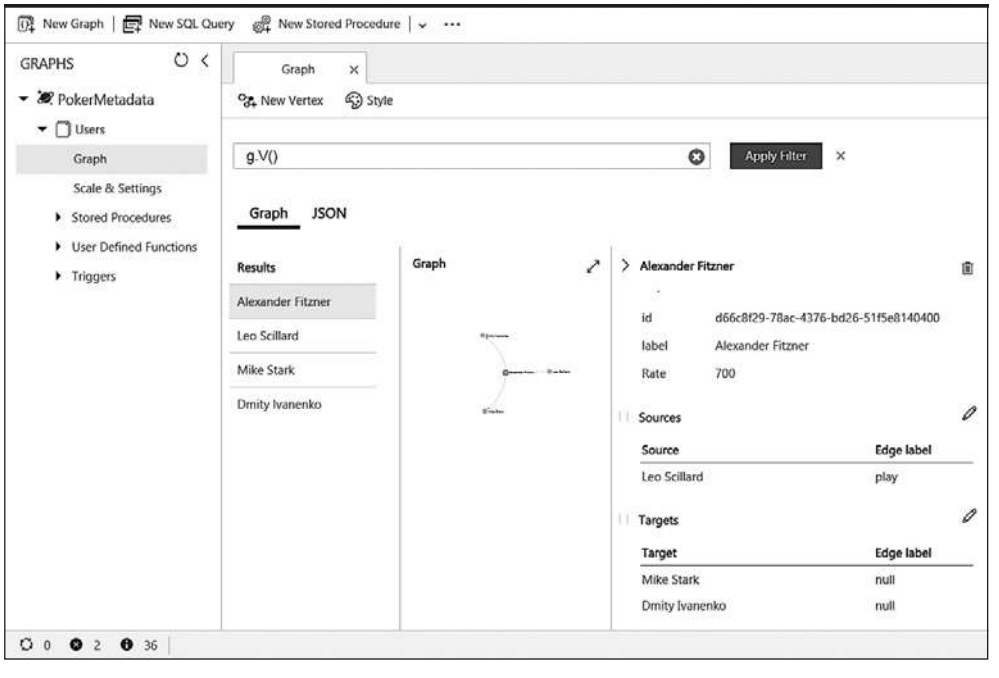


Рис. 6.19. Пользовательский интерфейс вкладки DataExplorer графовой базы данных

Вероятно, вы уже обратили внимание на специфический синтаксис запроса фильтрации. Он будет рассмотрен более детально в части III. Но самое интересное — возможность применения SQL-запросов к графовым данным (рис. 6.20).

Ввиду ограниченности объема книги мы не будем подробно рассматривать аккаунты типа MongoDB и Cassandra. CosmosDB Tables во многом похожа на Azure Table Storage.

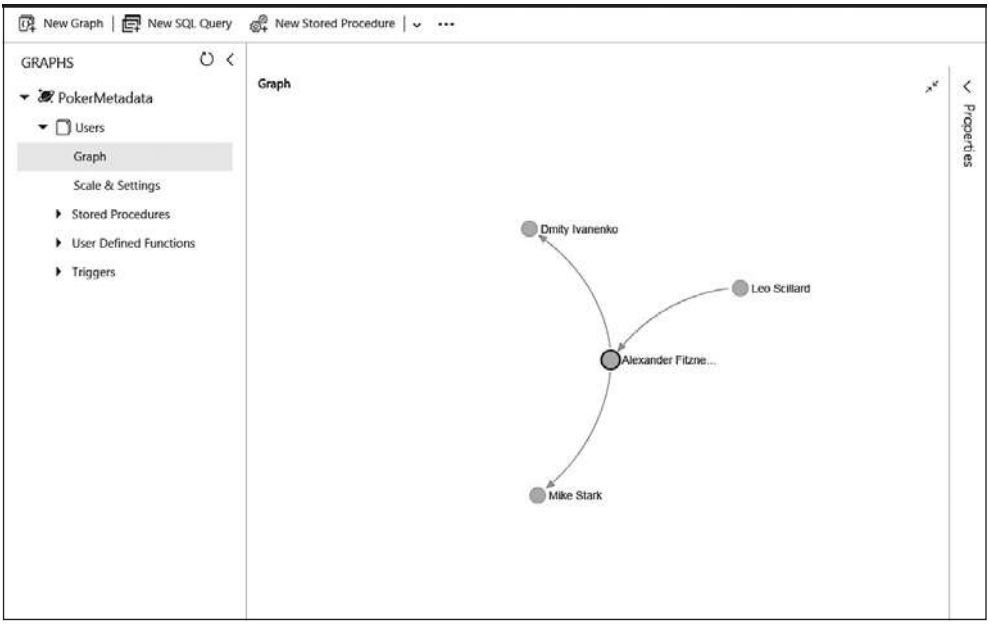


Рис. 6.20. Развернутая палитра графа

6.3. Сервисы нереляционных баз данных от AWS

> Первые попытки человека преодолеть сопротивление того, что ему неведомо, часто бывают неудачны. И однако они необходимы, как показывает история.
>
> *Станислав Лем. Магелланово облако*

Облачный провайдер AWS в настоящее время предоставляет два сервиса нереляционных БД: AWS DynamoDB и AWS Neptune. Изучим их подробнее.

Сервис нереляционной базы данных DynamoDB есть реализация БД типа «ключ — значение» от Amazon. Главные компоненты этого сервиса приведены ниже.

- *Таблица* (Table) — это основная коллекция данных в DynamoDB.
- *Элемент данных* (Item) — по сути, группа атрибутов, отличающаяся от всех других элементов данных таблицы.
- *Атрибут* (Attribute) — элементарная и неделимая единица хранения данных, представляющая собой коллекцию объектов типа «ключ — значение».

- *Первичный ключ* (Primary Key) — идентификатор элемента данных, уникальный в пределах таблицы. В DynamoDB может быть двух разновидностей:
  - *ключ раздела* (Partition Key) — простой первичный ключ, состоящий только из одного атрибута — собственно ключа раздела. DynamoDB использует значение этого ключа как вход внутренней хеш-функции, определяя физический раздел диска, где хранится элемент данных. В этом случае ключ однозначно определяет физическое положение одного элемента данных;
  - *ключ раздела и ключ сортировки* (Partition Key и Sort Key) — составной первичный ключ, состоящий из этих двух атрибутов. В данном случае ключ раздела также служит для указания физического раздела, где хранятся несколько элементов данных, и эти элементы хранятся в отсортированном виде.
- *Вторичный индекс* (Secondary Index) — для таблицы можно вводить один или несколько вторичных индексов, позволяющих создавать высокопроизводительные запросы по полям, отличным от первичного ключа. Вторичные индексы могут быть следующих видов:
  - *глобальный вторичный индекс* (Global Secondary Index) — индекс, у которого ключ раздела и ключ сортировки отличаются от таковых у таблицы;
  - *локальный вторичный индекс* (Local Secondary Index) — индекс с тем же ключом раздела, что и основная таблица, но с иным ключом сортировки.
- *Потоки DynamoDB* (DynamoDB Streams) — поток событий DynamoDB, генерируемый в ответ на изменение, добавление и удаление элементов данных. Можно использовать для запуска сервиса AWS Lambda таким же образом, как и в случае потока событий изменений для сервиса Azure CosmosDB.

Посмотрим на практике, как работать с AWS DynamoDB. Стартовая страница сервиса выглядит так (рис. 6.21).

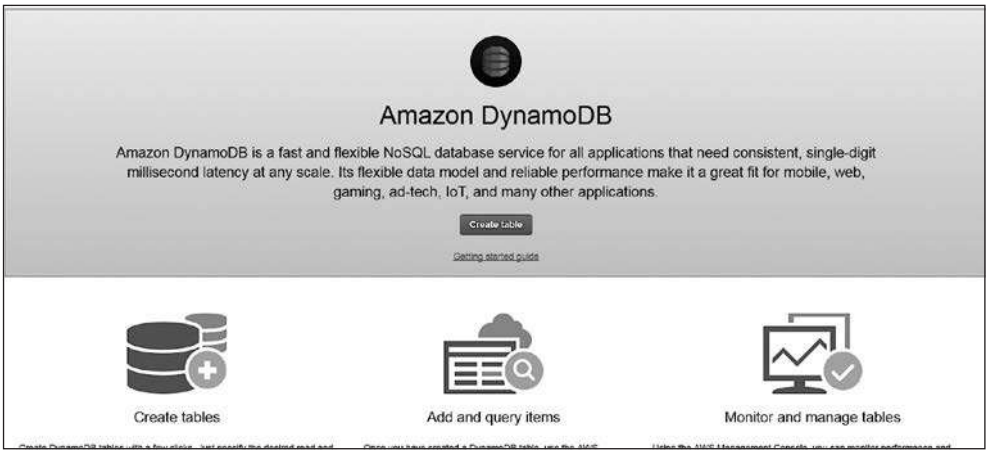


Рис. 6.21. Стартовая страница создания сервиса AWS DynamoDB

Чтобы создать таблицу, следует нажать ссылку Create tables, после чего откроется страница, показанная на рис. 6.22. При создании таблицы нужно указать ее имя, первичный ключ (мы не добавляли ключ сортировки) и тип ключа — строковый.

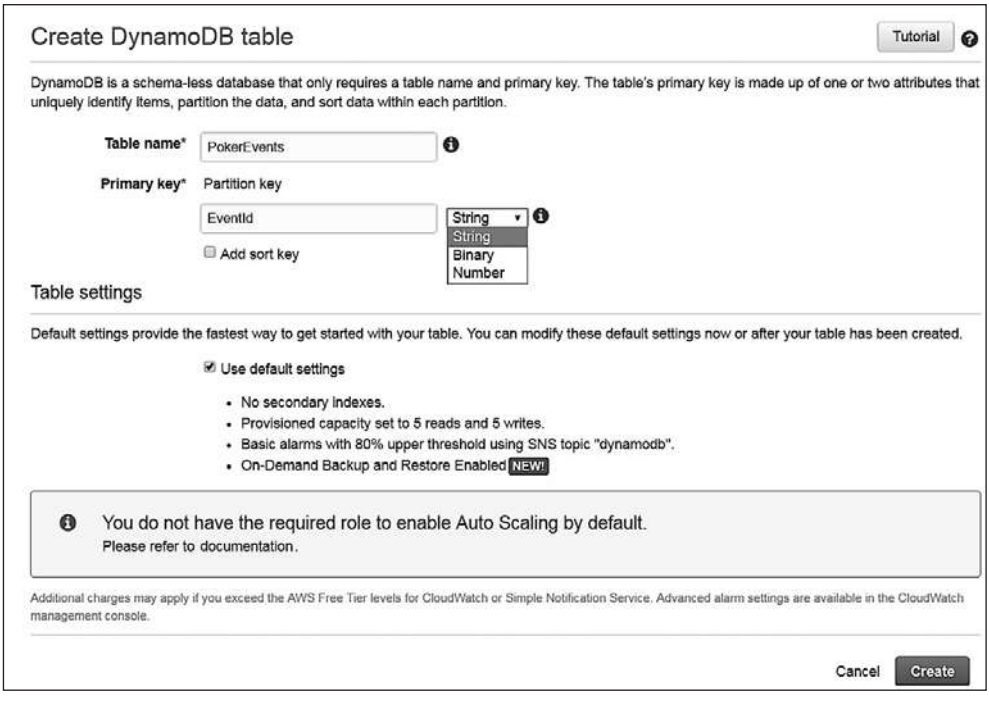


Рис. 6.22. Страница создания и конфигурирования таблицы

Спустя некоторое время мы можем увидеть панель созданной таблицы (рис. 6.23).

Названия элементов управления позволяют оценить основные возможности сервиса DynamoDB, часть которых значительно отличается от таковых у CosmosDB и Azure Table Storage. Прежде всего, это Time to live attribute — механизм ограничения времени жизни элементов данных. По истечении настраиваемого параметра элемент будет автоматически удален из таблицы. Чтобы сконфигурировать это время жизни, следует нажать на ссылку Manage TTL. В результате откроется следующее окно (рис. 6.24).

Alarm — это сигнал сервиса CloudWatch, который возникает при наступлении определенного условия. Стоит пояснить, что имеется в виду. С каждой таблицей DynamoDB ассоциированы метрики, то есть численные величины, характеризующие те или иные параметры таблицы: общее количество запросов, количество запросов с уменьшенным временем ответа (дросселированных, throtled), количество ошибок и пр. Часть этих метрик отображается в виде графиков на вкладке

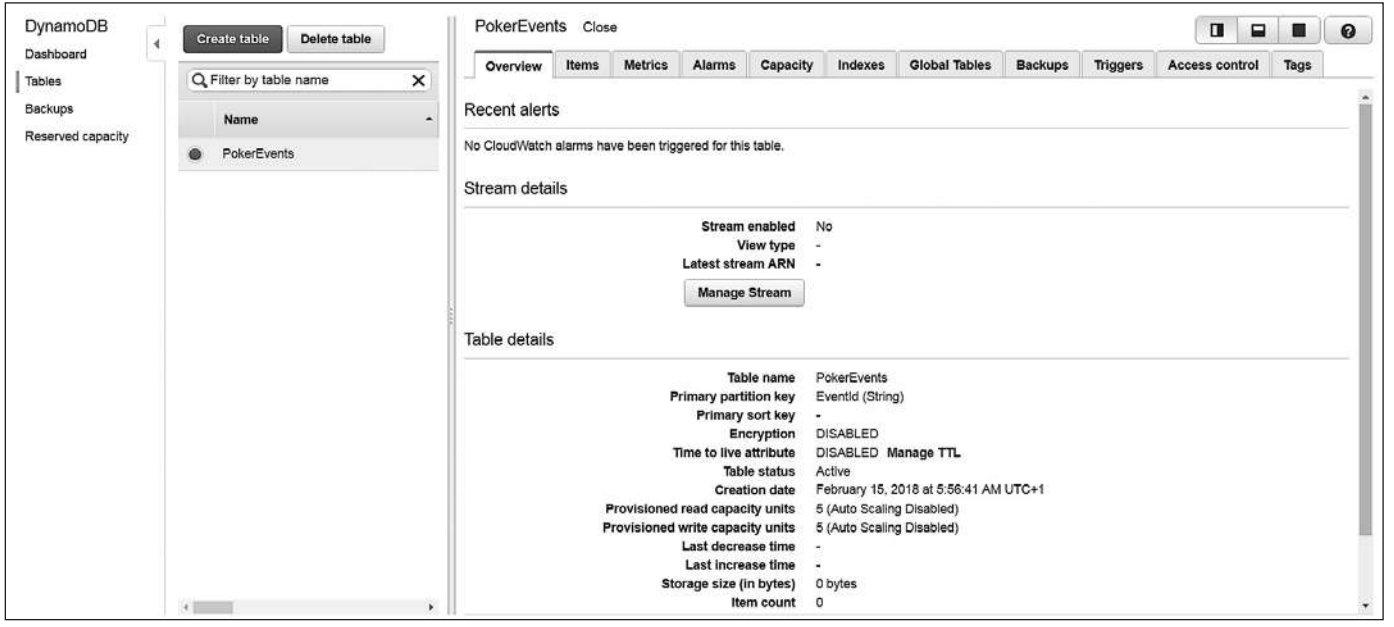


Рис. 6.23. Страница с панелью созданной таблицы

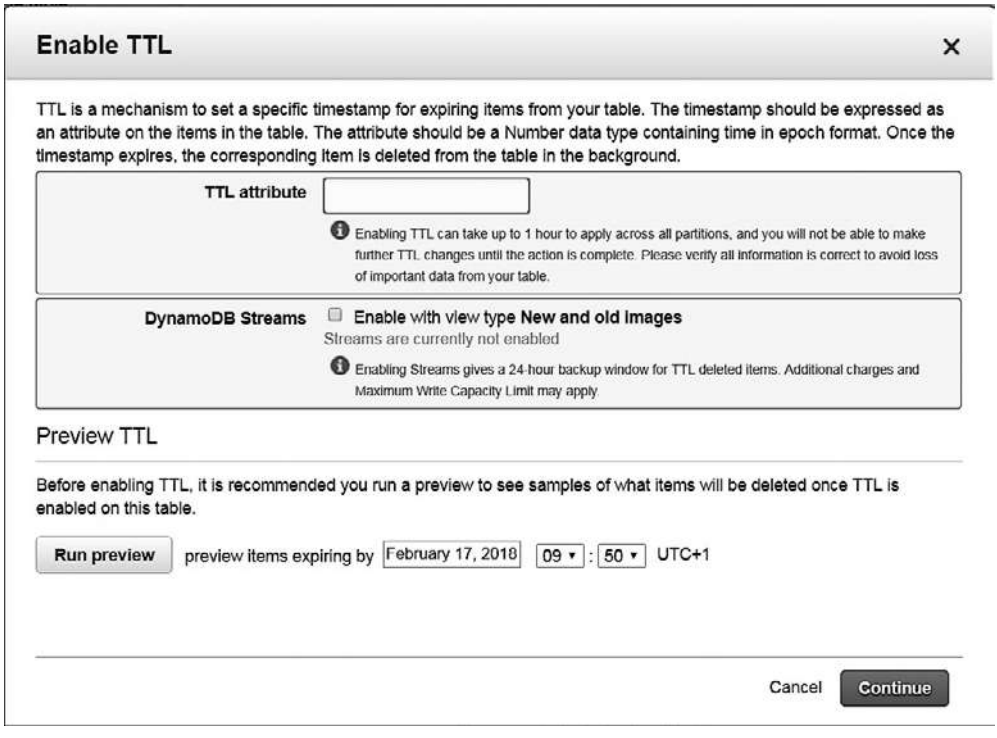


Рис. 6.24. Форма конфигурирования времени жизни элементов данных таблицы

Metrics (в правой нижней части рис. 6.25 приведен развернутый список доступных метрик). Создание сигнала Alarm требует выполнения определенного условия, связанного с определенной метрикой. Например, задержка (latency) превысила определенное значение.

Как указывалось выше, DynamoDB позволяет создавать вторичные индексы. Сделать это можно на вкладке Indexes (рис. 6.26).

Global Tables — это механизм построения географически распределенных высокодоступных и отказоустойчивых кластеров, относящихся к одному аккаунту. Данная система состоит из *реплик* — одиночных таблиц, которые представляют собой копии головной таблицы, расположенные каждая в своем регионе. При этом все реплики действуют как одна общая таблица. Данные, добавленные, измененные или удаленные на одной физической реплике таблицы, автоматически распространяются на другие реплики с помощью механизма потоков DynamoDB (DynamoDb Stream); соответственно, потоки должны быть сконфигурированы до создания Global Tables. Читателя, заинтересованного в подробностях модели согласованности данных, я отсылаю к документации AWS DynamoDB.

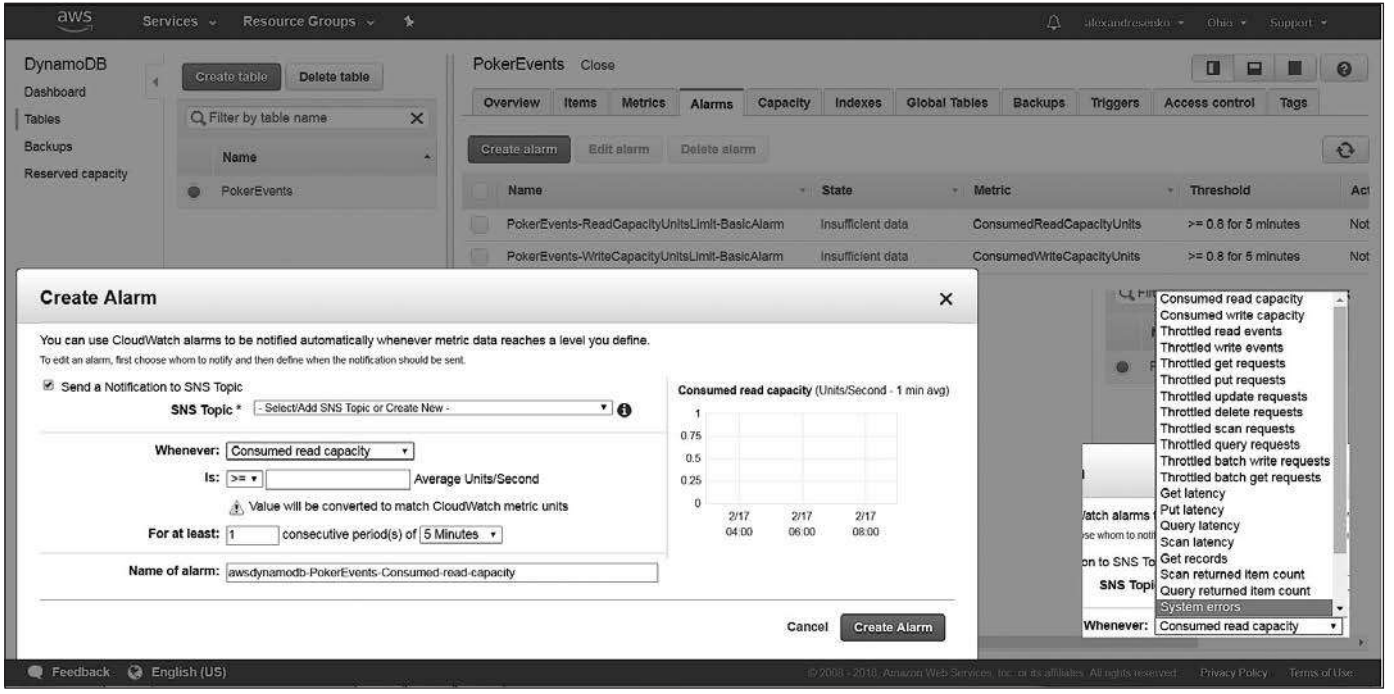


Рис. 6.25. Создание CloudWatch Alarm для таблицы

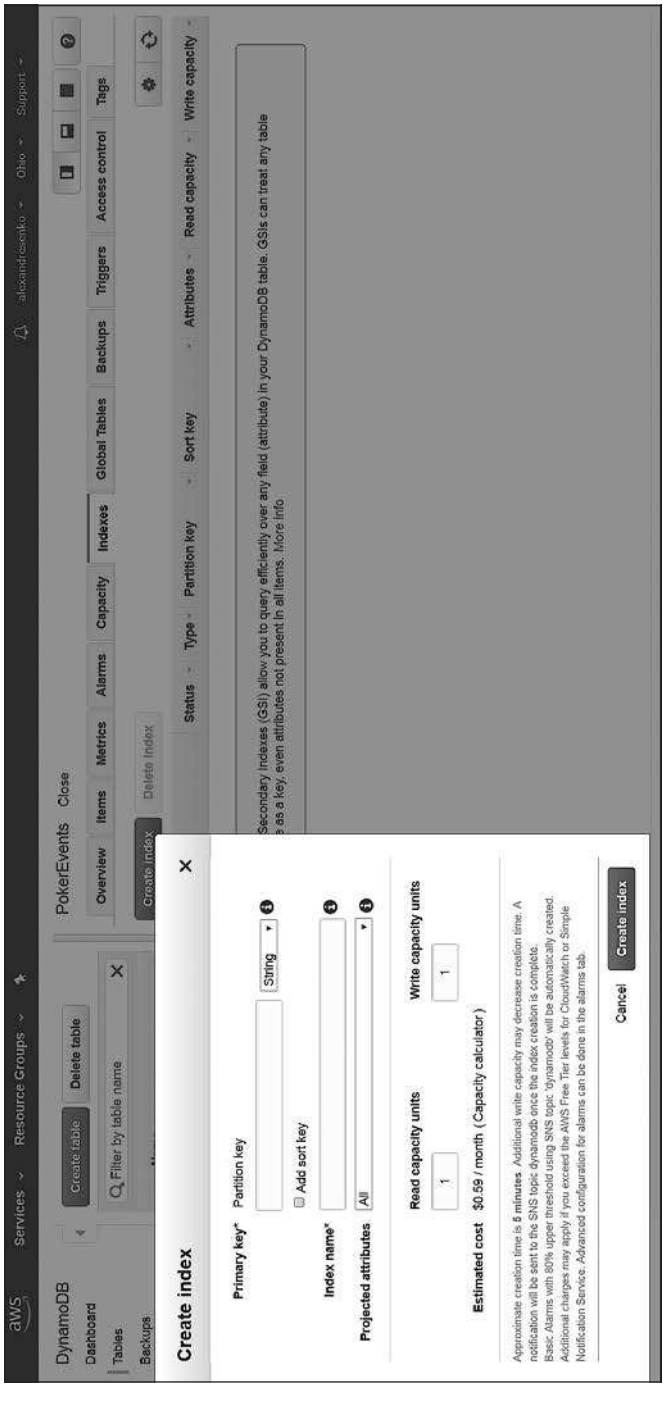


Рис. 6.26. Вкладка создания индексов

Кроме того, у DynamoDB имеется возможность прямой интеграции с AWS Lambda (вкладка Triggers), создания бэкапов и восстановления данных из них (вкладка Backups) и недавно добавленная опция шифрования данных.

Для добавления данных в таблицу и их обзора в ней следует перейти на вкладку Items (рис. 6.27).

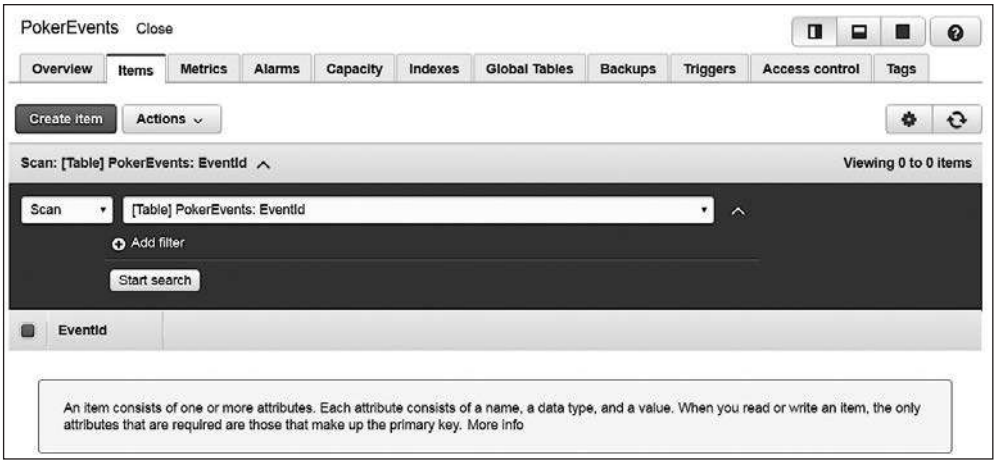


Рис. 6.27. Вкладка Items, обеспечивающая добавление и выборку элементов данных

Чтобы добавить элемент данных, следует выбрать ссылку Create item и ввести нужные поля в появившуюся форму (напоминаю: EventId — это первичный ключ таблицы, указанный при ее создании) (рис. 6.28, 6.29).

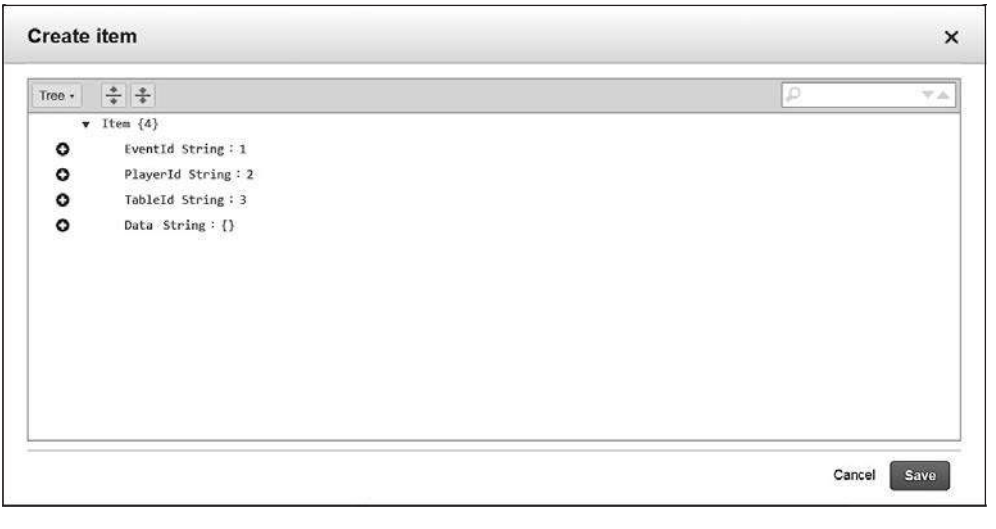


Рис. 6.28. Создание элемента данных DynamoDB — текстовый вид

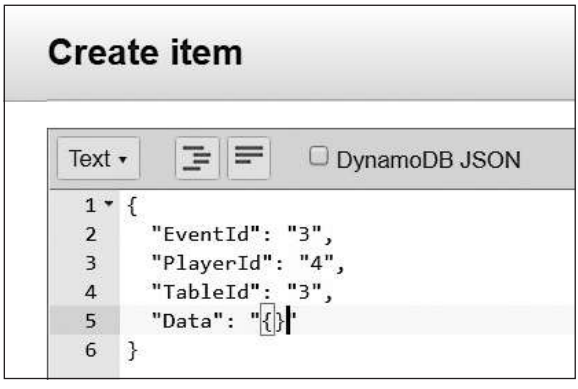


Рис. 6.29. Создание элемента данных DynamoDB — вид JSON

В элементе данных мы создали ряд полей в дополнение к первичному ключу EventId. Именно здесь наглядно видна уникальность NoSQL: по сути, каждый элемент данных может иметь свой набор полей — обязательным является поле первичного ключа и проиндексированные поля, то есть это в чистом виде концепция БД без схемы — schemaless. В итоге вид вкладки элементов данных имеет следующий вид (рис. 6.30).

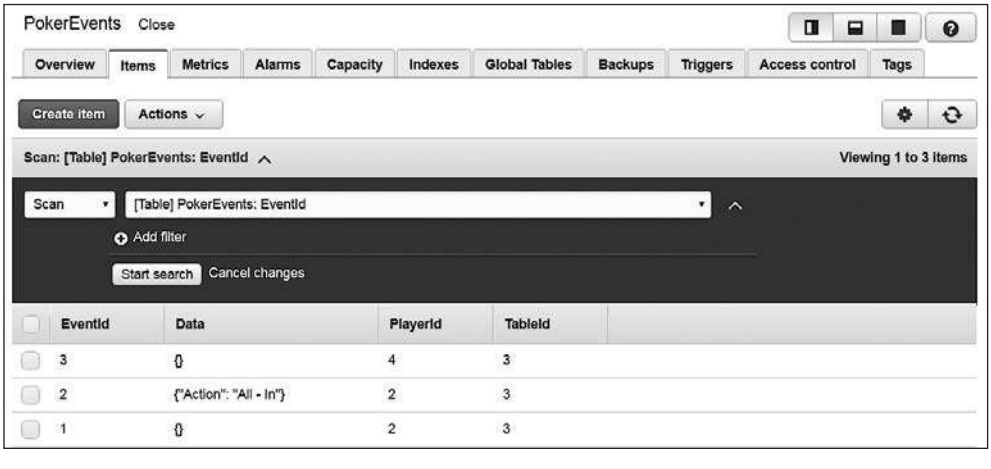


Рис. 6.30. Вид добавленных элементов в таблицу DynamoDB

Более подробно вопросы программного добавления элементов и выборки их из таблицы будут рассмотрены в части III.

Помимо DynamoDB, у AWS в настоящее время есть планы по предоставлению сервиса графовой базы данных — AWS Neptune. Сейчас этот сервис находится в стадии закрытого просмотра, доступен только по специальному запросу. На мо-

мент сдачи книги данный сервис еще не вышел из стадии preview, и я не рискнул включить материал по нему, поскольку продакшен-версия может весьма отличаться от доступной для preview.

6.4. Резюме

Теперь подведем итоги обзора сервисов нереляционных БД, сравнив их между собой. Итак, эти базы предлагают широкий спектр моделей хранения данных, в ряде случаев более соответствующих модели предметной области. Например, графовые базы подходят для предметной области в виде социальных сетей. Документоориентированные базы и базы семейства столбцов отлично служат для хранения объектов, плохо поддающихся нормализации: генерируемого пользователями контента, типа комментариев в социальных сетях и т. п. Базы данных табличного типа идеальны для хранения многих однотипных записей о событиях в информационной системе, логов, данных телеметрии подключенных устройств и пр.

Следующее важнейшее преимущество нереляционных баз данных — их масштабируемость и опция одновременного обслуживания огромного количества запросов. Это возможно благодаря простоте репликации и отсутствию требования нормализации. По сути, данные хранятся в виде атомарных автономных единиц: строк, ячеек матрицы или документов JSON. Эти единицы хранения могут быть построены таким образом, что становятся доступными напрямую, без применения сложных запросов с соединениями таблиц по ключу или их объединения. В ряде приложений важным свойством является отсутствие схемы, то есть возможность хранения в единой логической единице базы данных (например, в таблице) объектов с совершенно разными наборами свойств.

Из всех указанных преимуществ вытекают и существенные недостатки такой модели по сравнению с традиционными РБД. Прежде всего, возникают проблемы с целостностью данных в распределенной системе, и невозможно обеспечить *сильную согласованность*, то есть всегда присутствует неоднозначность и потеря информации при одновременном обновлении одного и того же элемента данных для их разных реплик. Из-за отсутствия в общем случае сильной согласованности возможности ACID-транзакций, как правило, не обеспечиваются встроенными механизмами базы данных, а должны быть обеспечены всей системой в целом. Обычно в лучшем случае для кластеров NoSQL согласованность может быть итоговой. Но это проблема не NoSQL, а любой распределенной системы хранения данных, и она носит название «теорема CAP». Типичная формулировка данной теоремы состоит в том, что в любой реализации распределенной системы хранения и вычислений можно обеспечить не более двух из трех следующих свойств:

- *согласованность данных* — во всех узлах системы в каждый момент времени данные не противоречат друг другу;

- *доступность* — любой запрос к системе должен завершаться корректным ответом (возможно, при этом данные будут противоречащими);
- *устойчивостью к разделению* — способность системы к расщеплению на независимые фрагменты без уменьшения доступности.

Как правило, в нереляционных БД согласованность данных приносится в жертву масштабируемости и доступности.

Следующий недостаток нереляционных БД — трудности построения аналитических запросов на выборку данных. Документоориентированные базы, семейство столбцов и табличные БД идеальны для хранения многих порций элементарных данных и выборки их отфильтрованного подмножества. Аналитические запросы, помимо фильтрации, требуют возможности объединения данных из разных коллекций, выбора подмножества полей, агрегации и выполнения арифметических операций.

Сервис CosmosDB SQL API создан в попытке нивелировать эти недостатки — он предоставляет мощный механизм запросов на языке SQL и JavaScript (в виде хранимых процедур и функций), а весь CosmosDB SQL обеспечивает настраиваемый уровень согласованности (вплоть до сильной согласованности).

CosmosDB Table и Azure Table Storage есть чистые хранилища типа «ключ — значение», которые могут предоставлять также доступ по протоколу OData, без применения специальных Azure Storage SDK.

В отличие от сервисов Azure Table Storage CosmosDB Table API, AWS DynamoDB не предоставляет встроенного механизма запроса, сильной согласованности или прямого доступа к конечной точке OData. Но зато DynamoDB позволяет регулировать время жизни элемента данных и создание резервных копий таблицы.

Несколько особняком стоят графовые базы данных. Их основная цель — предоставить возможность моделирования предметной области в виде сетевой структуры, которая при наличии большого количества узлов достаточно трудно поддается представлению в реляционном виде. Граф может быть представлен в матричном виде — в виде матрицы инцидентности и матрицы смежности, для чего могут подойти базы данных семейства солбцов с их матричной структурой. Однако наиболее удобное описание в плане наглядности, специализированного языка запросов и др. возможно только в графовых базах данных.

7 Реляционные хранилища больших DWH

Должен вам сказать, что мы вовсе не хотим завоевывать космос. Мы хотим расширить Землю до его границ. Мы не знаем, что делать с другими мирами. Нам не нужно других миров...

Станислав Лем. Солярис

7.1. Общий обзор реляционных хранилищ данных

Как уже описывалось в главе 5, реляционная база данных предназначена для выполнения двух основных функций: OLAP и OLTP. Преимущественно РБД используются для целей OLTP. Это приводит к тому, что, с одной стороны, обычные реляционные базы имеют ограничения по размеру, а с другой — выполнить высокопроизводительные запросы к большим массивам данных может быть затруднительно.

Рассмотрим пример. Предположим, есть база данных SQL, которая служит как хранилище данных веб-приложения, непрерывно обновляемых пользователями. Кроме того, необходимо периодически строить отчеты в системах бизнес-аналитики. Для этого нужно выполнить запросы, очень сильно нагружающие SQL-сервер, поскольку они будут содержать операции группирования, совершения математических операций, просмотра множества таблиц и соединения в одном запросе результатов просмотра и пр. И тут возможна ситуация, когда запросы от подсистемы бизнес-аналитики могут очень мешать работе веб-приложения и даже блокировать ее из-за зависания базы данных во время выполнения BI-запросов. Это происходит потому, что для выполнения таких запросов требуются гораздо бóльшие вычислительные возможности, чем для обычной работы приложения.

Проблема нехватки ресурсов баз может обостриться ввиду необходимости выполнять запросы к таблицам из разных БД. Она частично устраняется с помощью поднятия ценового уровня базы (для Azure SQL) или типа экземпляра (для AWS RDS) перед запуском всех BI-запросов и опускания после их завершения. Но такой подход ограничен тем, что нужно разделить по времени периоды работы сервисов бизнес-аналитики и основного приложения. Кроме того, нерешенной является проблема выполнения запросов среди многих таблиц многих внешних БД. Да и предельно высокий ценовой уровень одиночной базы может оказаться недостаточным. Как быть в этом случае? Вот тут-то на помощь и приходит DWH (Data Warehouse) — «склад данных» — реляционное хранилище данных, оптимизированное для хранения огромных массивов данных и выполнения запросов к ним (рис. 7.1).

Рис. 7.1. Наполнение хранилища Data Warehouse и доступ сторонних сервисов к нему

Высокая производительность аналитических запросов достигается благодаря массивно-параллельной архитектуре и распараллеливанию запроса для выполнения несколькими серверами одновременно. В данном случае тоже необходимо разместить информацию из различных БД в одном общем сервисе — «складе» (DWH). Преимущество такого подхода в том, что SQL DWH позволяет использовать стандартные SQL-запросы, хранимые процедуры и др., а следовательно, такое хранилище совместимо со всеми стандартными средствами и инструментами BI.

Реляционное хранилище данных (SQL DWH) представляет собой массивно-параллельное хранилище, состоящее из кластера серверов трех типов: головного (управляющего), узлов вычисления и томов хранения (рис. 7.2).

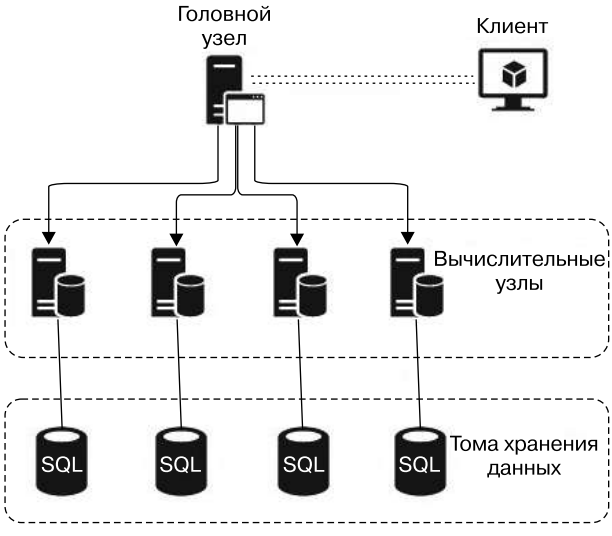


Рис. 7.2. Общая архитектура реляционного хранилища данных

Клиент (а точнее, программный клиент) может взаимодействовать с реляционным хранилищем только через головной узел. Последний отвечает за распределение данных по вычислительным узлам, в ряде случаев являющимся одновременно и узлами хранения. Кроме того, головной узел компилирует запросы языка SQL в команды кластеру, которые обеспечивают параллельное выполнение, а затем собирает результат вместе и выдает назад клиенту. Реляционное хранилище данных, в отличие от реляционных баз данных, оптимизировано только для целей OLAP. Выполнение транзакций в DWH возможно, но это несвойственно хранилищу, основная задача которого — хранить информацию и предоставлять доступ к ней с помощью интерфейса SQL. При этом производительность запросов на чтение данных существенно выше, чем для обычной БД за счет параллельного выполнения и иного физического представления данных, другого плана выполнения запросов и индексов.

Рассмотрим вопрос параллельного выполнения запросов. Необходимо разделить данные между вычислительными узлами. Это можно сделать следующими способами. Во-первых, в каждом узле разместить идентичные копии данных и разделить запрос таким образом, чтобы каждая его часть в вычислительном узле обрабатывала свой набор строк. Как уже отмечалось, за разделение запроса на подзапросы и соединение их результатов отвечает головной узел (см. рис. 7.2). Такая архитектура очень надежна: утеря одного узла никак не повлияет на информацию в хранилище. Недостаток данной архитектуры в том, что из-за ограниченного объема дисков одного узла в нем невозможно хранить очень большие объемы данных. Во-вторых, можно размещать данные в различных узлах без дублирования.

При этом вероятно как псевдослучайное распределение строк данных по узлам, так и последовательное, в порядке роста значения ключа. В этом случае возможно хранение и обработка гораздо бóльших объемов данных, чем в случае их полного копирования в каждом узле.

Итак, реляционное хранилище данных применяется, когда необходимо получить централизованное хранилище информации, позволяющее выполнять запросы с помощью стандартного синтаксиса SQL, поддерживаемого большинством систем BI. Для этого нужно извлечь данные из внешних источников (extract), преобразовать их к виду, удобному для добавления в хранилище с необходимой обработкой, фильтрацией и пр. (transform) и загрузить их в хранилище (load). Такой подход называется ETL и является традиционным. Он очень удобен в случае, когда внешние источники данных очень разнородны и все вместе или по отдельности не поддерживают встроенный механизм выполнения аналитических запросов. Недостаток ETL состоит в том, что для наполнения реляционного хранилища обязательно нужен сервис копирования и трансформации. Критически важный момент для ETL — метаданные о сторонних источниках: что в них содержится, в каком формате и др. Эти метаданные в облачных средах могут быть представлены в специальных сервисах — каталогах данных (data catalog) (будут подробнее описаны в части IV).

Рассмотрим логическую структуру информации в реляционном хранилище. Ниже представлены несколько принципов организации реляционных хранилищ данных.

- Данные в хранилище должны быть объединены в соответствии с категориями предметной области, а не их физическими источниками.
- Централизованное хранилище должно хранить в своем составе данные, относящиеся ко всей системе в целом, а не только к отдельным аспектам. Только в таком случае можно построить комплексные аналитические запросы.
- Данные в хранилище содержатся и поступают извне, но не создаются.
- Говорить о корректности данных в хранилище можно только с привязкой их к конкретному временно́му промежутку.

Существует две стратегии наполнения хранилища данными: стратегия *полного обновления* (каждый раз происходит полное обновление данных из внешних источников) и *инкрементального* (обновления только тех данных, которые были изменены в процессе работы OLTP-подсистемы). В плане реализации, безусловно, проще стратегия полного обновления, в то время как инкрементальное требует использования сложных ETL-процедур замены обновленных элементов, вставки новых и удаления тех, что были удалены в OLTP. Выбрать ту или иную стратегию можно только путем совместного целостного анализа хранилища данных, их источников и характера их изменений.

Есть две общие модели построения реляционных хранилищ: *нормализованные хранилища* и *хранилища с измерениями*. Построение нормализованного хранилища

в общем не отличается от построения нормализованной реляционной базы данных. Информация хранится в связанных таблицах в третьей нормальной форме, и, как следствие, требуется выполнение выборки из многих таблиц, что потенциально ведет к усложнению запросов и падению производительности. Хранилище с измерениями — это денормализованная форма хранения информации. В свою очередь, возможны две архитектуры хранилища с измерениями — в виде схем *«звезда»* и *«снежинка»*.

В схеме «звезда» информация хранится в таблицах двух типов: центральной *таблицы фактов* и многочисленных *таблиц измерений* (рис. 7.3).

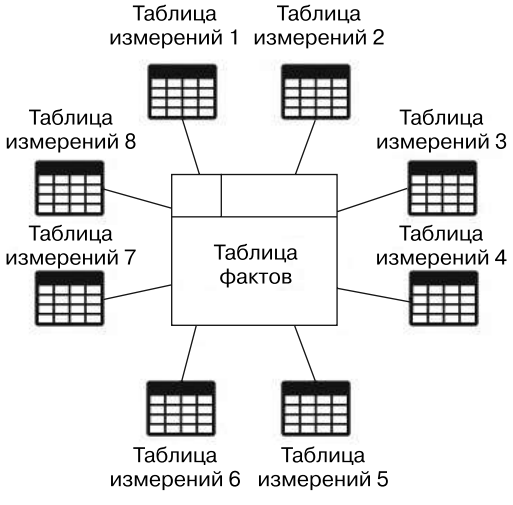


Рис. 7.3. Архитектура хранилища данных в виде схемы «звезда»

В центре звездообразной структуры находится таблица фактов, содержащая информацию, которая будет анализироваться. Она может включать факты (а по сути, записи), связанные с событиями или состоянием объекта, с транзакциями и пр. Таблица содержит первичный ключ, числовые поля, характеризующие данный факт, и внешние ключи, ссылающиеся на таблицу измерений. Числовые поля должны включать аддитивные значения, подлежащие анализу (в том числе суммированию, вычислению среднего значения и др.). Таблицы измерений нужны для хранения атрибутов фактов (как правило, текстовых), которые не могут быть использованы для численных операций со строками таблицы фактов. Например, в таблице фактов с данными о проведенных покерных играх каждая строка содержит сумму выигрыша игрока, внешний ключ на таблицы измерений игроков и игр. В схеме «звезда» данные существенно денормализованы для оптимизации запросов.

Иногда требуется добавить элемент нормализации и расщепить некоторые таблицы изменений на несколько таблиц — это схема «снежинка» (рис. 7.4).

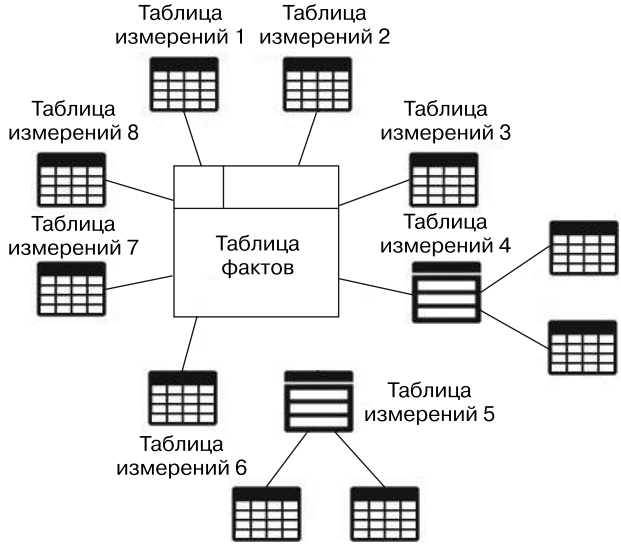


Рис. 7.4. Архитектура хранилища данных в виде схемы «снежинка»

Выбор той или иной структуры реляционного хранилища данных и построение запросов (OLAP — кубы, DataMart — витрины данных) выходит за рамки данной книги. В следующих разделах подробнее рассмотрим реализации хранилищ данных от Azure SQL DWH и AWS RedShift.

7.2. Azure SQL DWH

> Это мы говорим, будто мы выдумываем. На самом деле все давным-давно выдумано. Кто-то давным-давно все выдумал, сложил все в ящик, провертел в крышке дыру и ушел...
>
> *Аркадий и Борис Стругацкие. Трудно быть богом*

Реляционное хранилище Azure SQL DWH — облачный сервис от Microsoft Azure, построенный на основе сервиса Azure SQL. Этот сервис использует технологию Microsoft PolyBase для построения запросов к данным, расположенным как в хранилище Azure Blob Storage, так и в базах данных Azure SQL. Рассмотрим основные концепции сервиса Azure SQL DWH.

Архитектура хранилища Azure SQL DWH соответствует представленной на рис. 7.2. Как головной узел, так и вычислительные узлы содержат экземпляры си-

стемного сервиса DMS (Data Movement Service), отвечающего за передачу данных между узлами. Данные физически хранятся в Azure Storage. Такое физическое разделение их хранения и параллельного выполнения запросов обеспечивает возможность независимого масштабирования объемов хранения и вычислительных мощностей кластера.

Непосредственно из Azure Storage данные разделяются по вычислительным узлам с помощью одного из трех видов *распределения* (distributions). Каждый запрос разделяется на 60 меньших фрагментов, каждая из которых выполняется на вычислительном узле. Каждый вычислительный узел работает с 1 до 60 фрагментов в зависимости от его ценового уровня. На наивысшем уровне (максимальная производительность вычислительного узла) используется одно распределение на узел, при минимальных ресурсах — все распределения на один узел.

Существует три возможных паттерна данных, влияющих на производительность запросов:

- распределение с помощью хеш-функции (hash);
- последовательное, или циклическое, распределение (round robin);
- репликация (replicate).

Идея использования *хеш-функции* для распределения строк из исходной таблицы по вычислительным узлам состоит в том, что строки равномерно, независимо и случайно разделяются по вычислительным узлам (рис. 7.5). Это позволяет строить оптимальные запросы с помощью соединения таблиц и агрегации данных. При этом каждая строка таблицы относится к одному распределению.

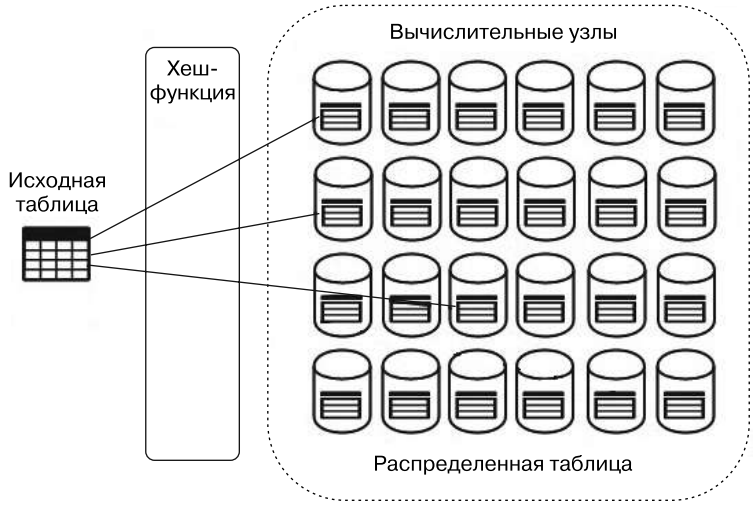


Рис. 7.5. Распределение данных таблицы по вычислительным узлам с помощью хеш-функции

Циклическое распределение строк по вычислительным узлам подразумевает последовательное равномерное наполнение узлов со случайным выбором начального узла. Такое распределение обеспечивает высокую производительность операций выборки данных в случае, когда в соединении таблиц нет нужды, поскольку оно потребует выполнения дополнительных «перетасовок» строк между узлами.

Репликация применяется при наличии таблиц небольшого размера и состоит в том, что такая таблица копируется на все вычислительные узлы (рис. 7.6). Это позволяет добиться высокой производительности любого типа запроса и одновременно увеличить надежность системы, поскольку отказ одного вычислительного узла или более повлияет лишь на производительность.

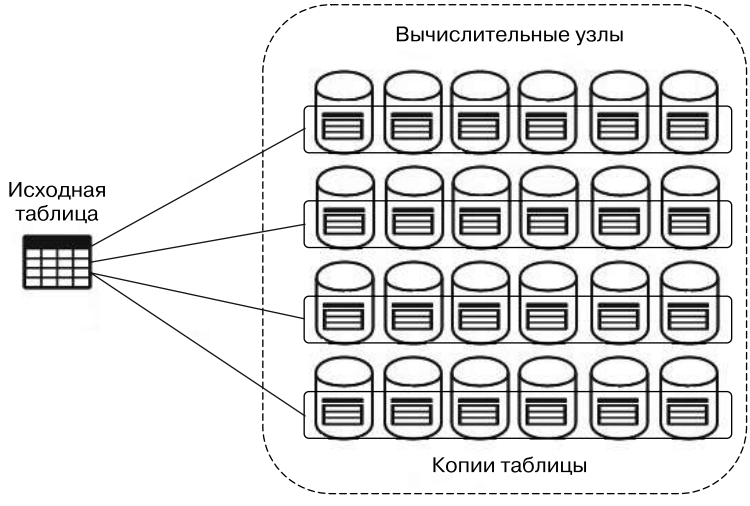


Рис. 7.6. Репликация таблицы по вычислительным узлам

Теперь посмотрим, как работать с Azure SQL DWH из веб-портала. Для этого во вкладке добавления ресурсов в строке поиска следует написать `SQL Data Warehouse` (рис. 7.7).

После нажатия кнопки Create (Создать) откроется следующая форма настройки реляционного хранилища Azure SQL DWH (рис. 7.8). Здесь нужно указать имя базы данных (Database name), выбрать ресурсную группу (Resource group), сервер (Server) и ценовой уровень производительности (Performance tier).

Для реляционного хранилища можно настроить ценовой уровень по двум разным критериям: гибкости (Optimized for Elasticity) и вычислительным возможностям (Optimized for Compute) (рис. 7.9). Дадим пояснение величинам, используемым для оценки производительности реляционного хранилища — DWU и cDWU:

- DWU (Data Warehouse Unit) представляет собой абстрактную величину, описывающую в нормализованном виде доступные вычислительные ресурсы: CPU, память и полоса пропускания дисков данных — IOPS;

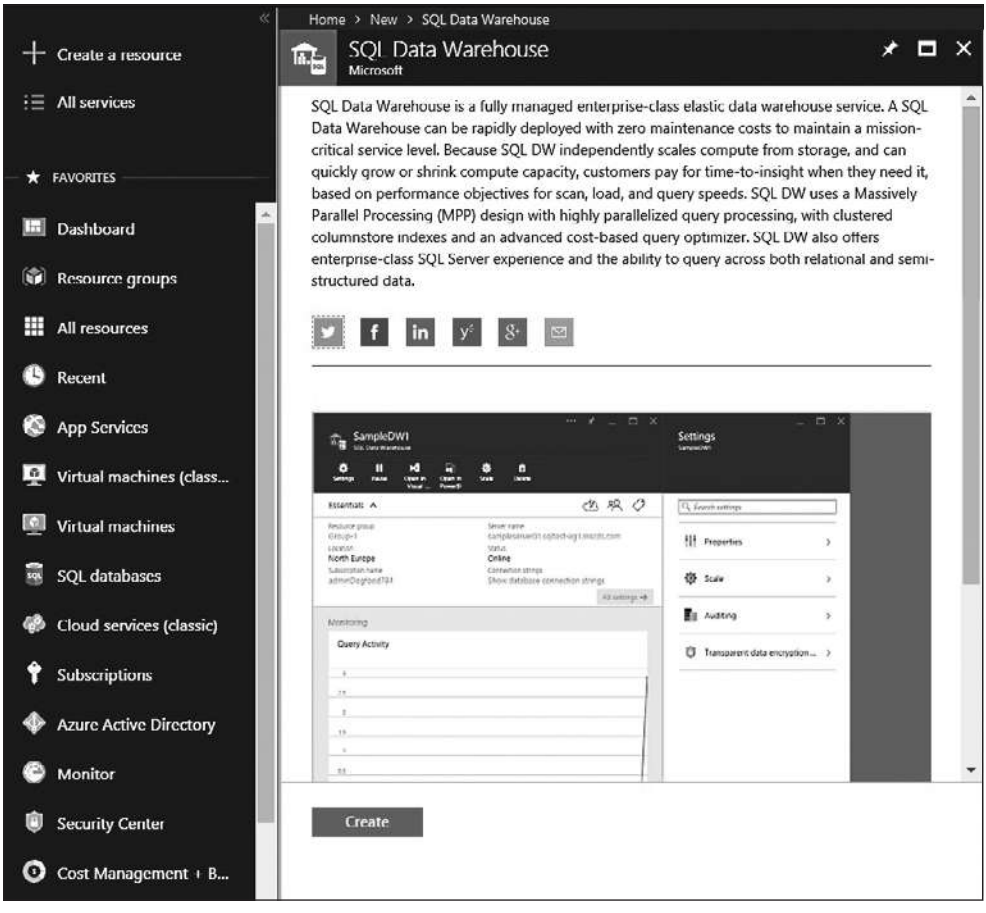


Рис. 7.7. Внешний вид вкладки создания хранилища данных

- cDWU (compute Data Warehouse Unit) — с одной стороны, описывает скорость чтения информации из Azure Storage, а с другой — скорость выполнения запросов.

После создания Azure SQL DWH панель будет выглядеть следующим образом (рис. 7.10). Она отображает основные возможности Azure SQL DWH, которые близки к таковым у базы данных Azure SQL. На панели представлены отдельный сервис загрузки данных —Load Data (поддерживается загрузка с помощью RedGate и Azure DataFactory), интеграция с сервисом SSAS (ссылка Model and Cache Data), сервис бизнес-аналитики (ссылка Open in PowerBI) и, конечно, сервисы мониторинга (Monitoring), масштабирования (Scale), выполнения запросов в онлайн-редакторе (ссылка Open Editor). Кроме того, доступны опции бэкапа, аудита и шифрования.

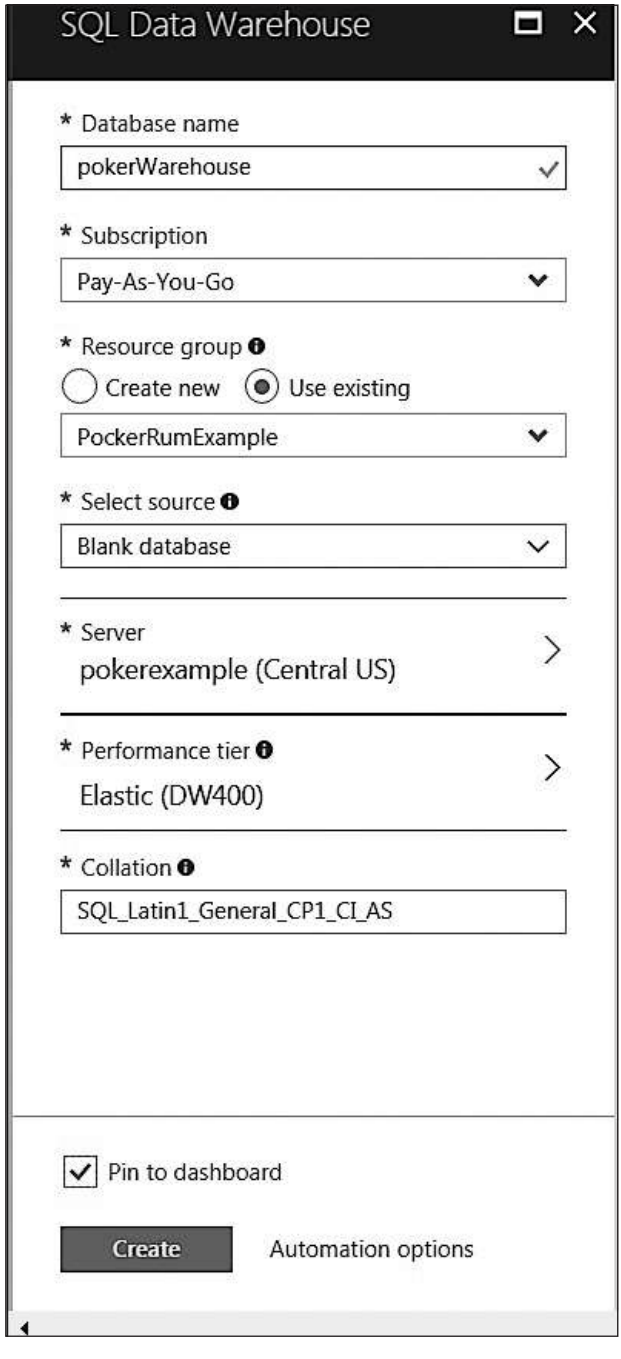


Рис. 7.8. Форма конфигурирования Azure SQL DWH

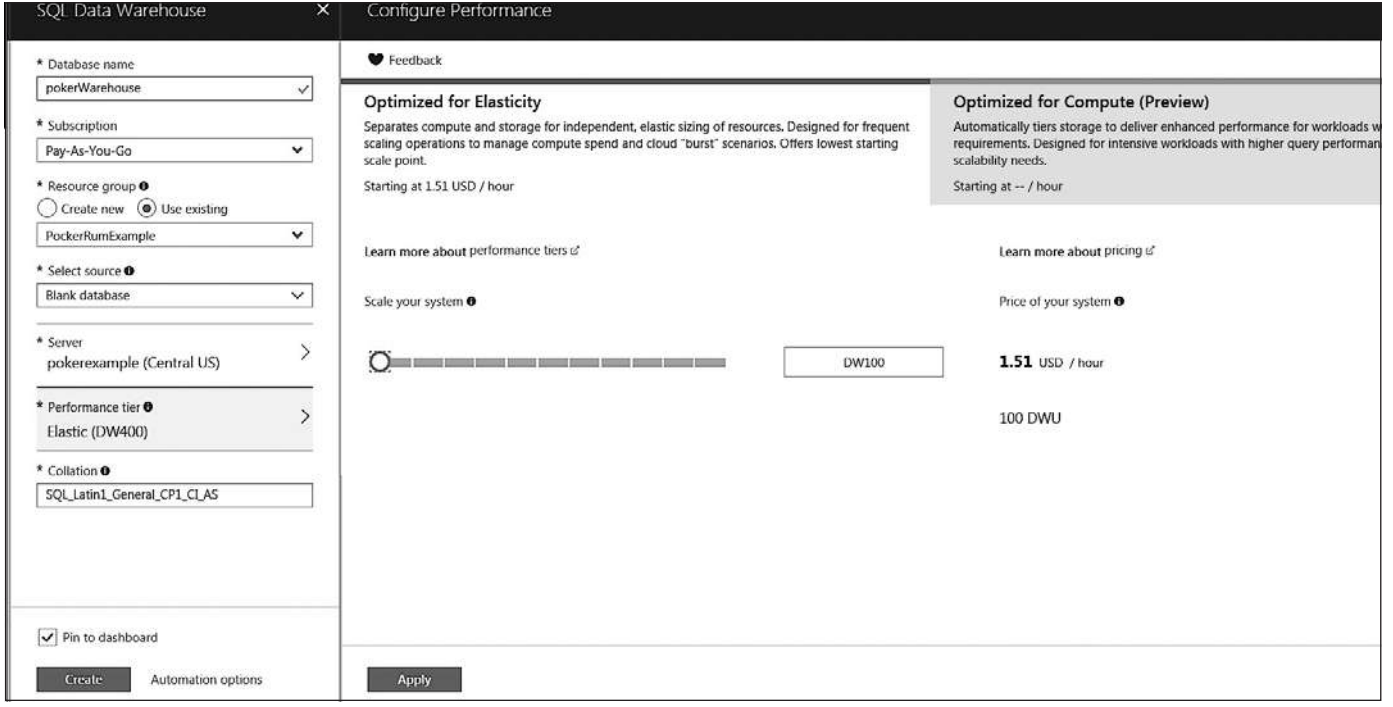


Рис. 7.9. Настройка ценового уровня Azure SQL DWH

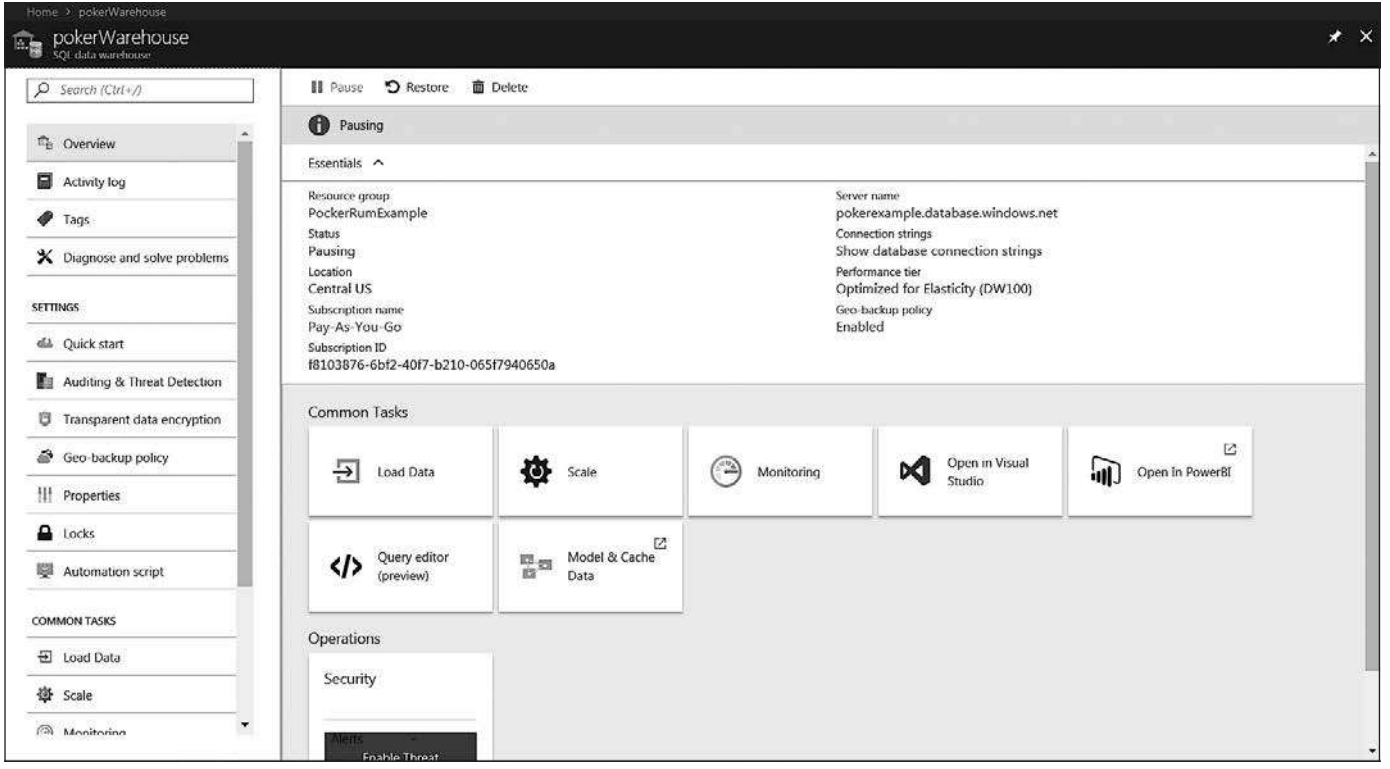


Рис. 7.10. Внешний вид общей панели мониторинга Azure SQL DWH

Внимательный читатель, наверное, обратил внимание на информационное сообщение Pausing. Дело в том, что после создания DWH я сразу нажал ссылку Pause, чтобы остановить вычислительные узлы и снизить расходы. Таким образом, узлы можно остановить и запустить в любой момент времени.

В качестве движка сервиса Azure SQL DWH используется Microsoft SQL DWH и, соответственно, язык запросов T-SQL. Чтобы подключить внешний клиент, следует задействовать строку подключения, доступную по ссылке Connection String. Наиболее полная интеграция возможна со средствами бизнес-анализа и аналитики, предоставляемыми корпорацией Microsoft.

7.3. AWS RedShift

> — А, проклятый бледнолицый, как ты смеешь являться в лагерь Вождя Краснокожих, грозы равнин?
>
> *О'Генри. Вождь краснокожих*

Сервис AWS RedShift представляет собой сервис реляционного хранилища от AWS. Архитектура этого сервиса в целом совпадает с приведенной на рис. 7.2, но имеет ряд отличий. AWS RedShift состоит из ведущего (головного) узла и вычислительных узлов. Клиентское приложение взаимодействует с ведущим узлом с помощью стандартных протоколов PostgreSQL JDBC и ODBC.

Вычислительные узлы AWS RedShift DWH состоят из разделов (node slices). Каждый из разделов представляет собой выделенную порцию вычислительных ресурсов: CPU, памяти, IOPS. Ведущий узел разделяет данные, взятые из хранилища, между вычислительными узлами, а точнее, между разделами вычислительных узлов. Это и позволяет выполнять запросы в параллельном режиме. Количество разделов определяется размером вычислительного узла, а точнее, типом экземпляра.

Узлы кластера содержат одну или более БД, в которых хранятся данные. Все эти базы взаимодействуют с ведущим узлом, который и занимается распараллеливанием запроса.

Чтобы обеспечить высокую производительность выполнения запроса, помимо массивно-параллельного выполнения запросов данных используются различные механизмы. Во-первых, это *столбцовое, или колоночное, хранилище* (Columnar Store) — способ организации хранения информации в таблице, обеспечивающий значительное снижение общего числа операций чтения-записи. При этом каждая строка в типичной БД сохраняется в виде блока на диске. В столбцовом хранилище в едином блоке хранятся данные нескольких столбцов. Сам движок AWS RedShift DWH преобразует данные из такой формы представления в табличную. Увеличение скорости работы в подобном случае достигается за счет того, что каждый блок хранит информацию для нескольких строк таблицы, например

для *n*, соответственно, ускорение будет равно приблизительно *n*, так как за один акт чтения блока читаются данные для ячейки у *n* строк. Чтобы добиться повышения общей производительности кластера, необходимо определить один столбец как *ключ распределения* (distribution key), благодаря которому данные будут распределяться по узлам кластера. Выбор правильного ключа обеспечит высокую производительность запросов.

Во-вторых, это *кэширование результатов* (Result Caching) в головном узле. При этом перед выполнением запроса проверяется состояние кэша. Нужно учитывать, что кэширование эффективно для запросов, возвращающих небольшое количество результатов. По умолчанию оно включено. И наконец, головной узел обеспечивает компилирование и оптимизацию кода запроса, кэширует скомпилированный результат исполнения в вычислительных узлах, обеспечивая ускорение сложных запросов. Такой механизм приводит к тому, что выполнение одного и того же запроса во второй и третий раз займет меньшее время, чем в первый.

Теперь рассмотрим, как создавать и работать с сервисом с AWS RedShift с помощью консоли AWS. Общий вид панели создания кластера показан на рис. 7.11.

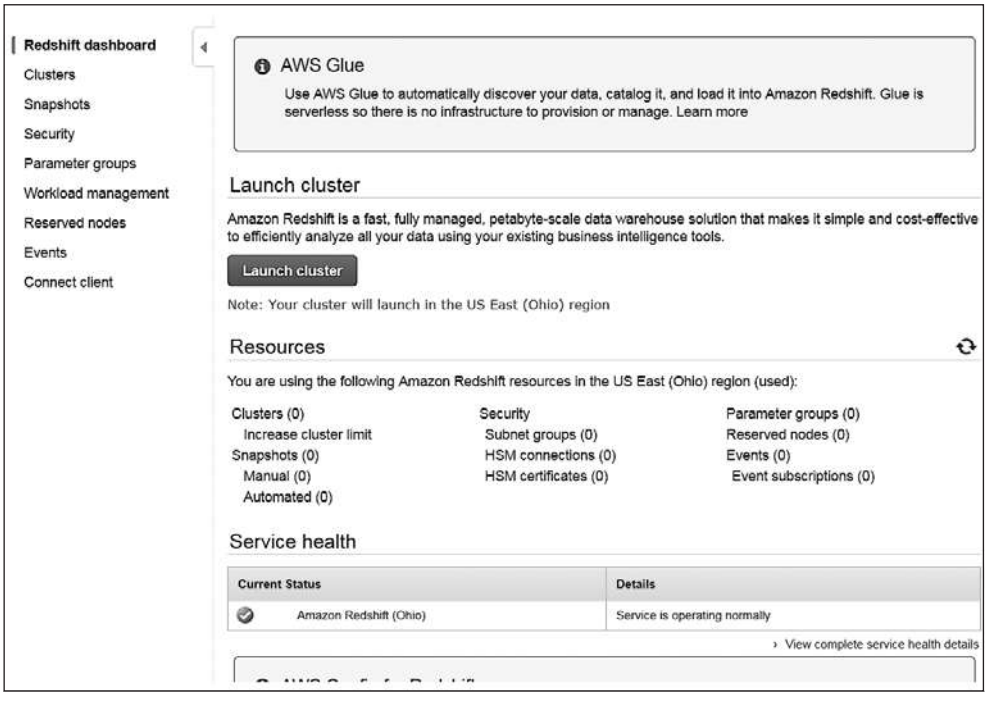


Рис. 7.11. Общий вид панели создания кластера

Чтобы начать создание кластера, следует выбрать ссылку Launch cluster. В результате откроется панель конфигурирования кластера, показанная на рис. 7.12.

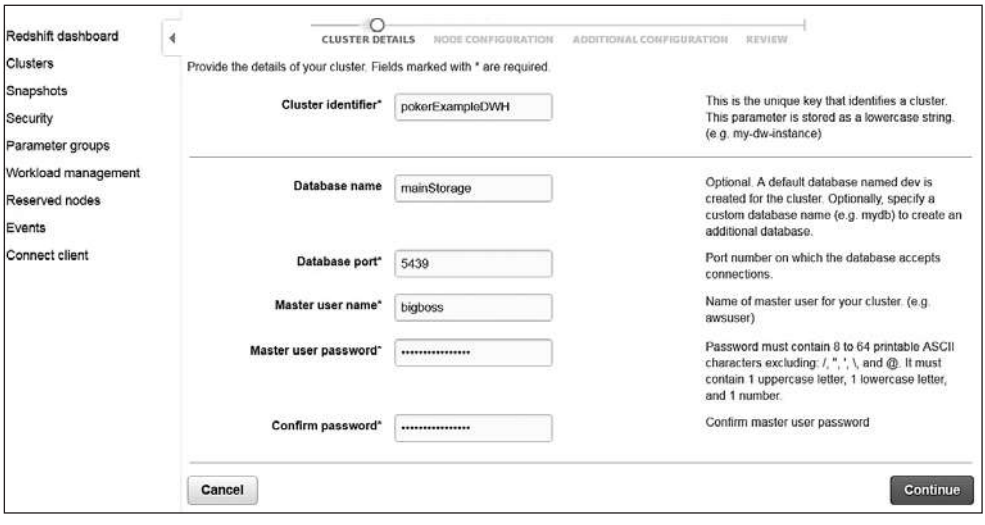


Рис. 7.12. Панель общего конфигурирования кластера

Первый шаг конфигурирования кластера в пояснениях не нуждается (обратите внимание на порт `5439`, ведь в основе кластера — PostgresSQL). На этой панели производим, по сути, конфигурирование сервера для данного кластера. Непосредственно конфигурирование кластера происходит на следующем шаге (рис. 7.13.).

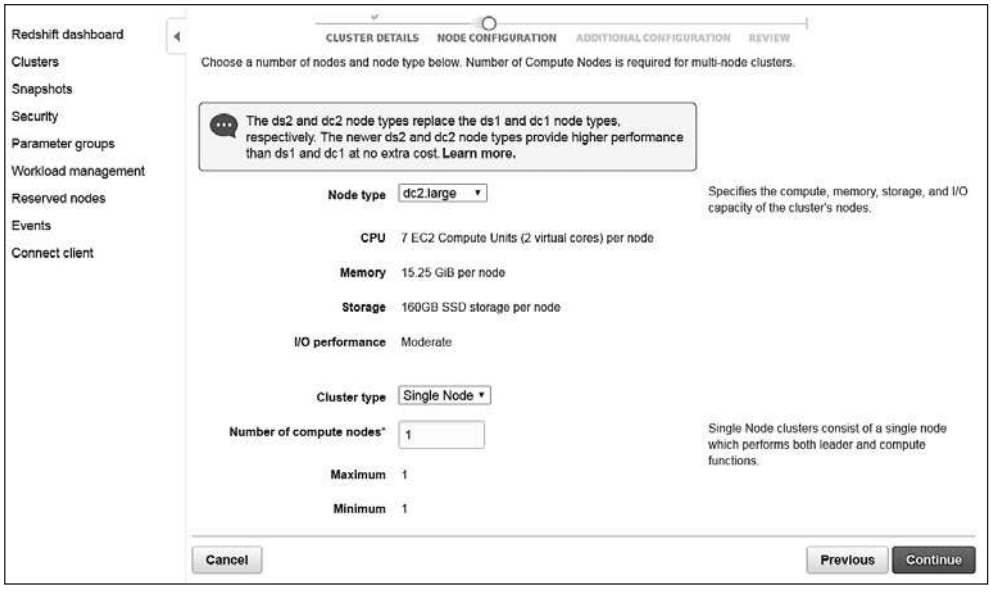


Рис. 7.13. Панель конфигурирования кластера

На этой панели прежде всего необходимо выбрать тип экземпляра вычислительного узла (Node type), тип кластера (Cluster type), который может быть Single Node или Multi Node. В случае Single Node кластер состоит из одного вычислительного узла и одного головного, в случае Multi Node — из головного узла и вычислительных узлов, количеством от 2 до 32.

На следующей панели конфигурируются параметры кластера, относящиеся к построению его сетевой инфраструктуры: VPC, группа безопасности (VPC Security groups), публичный IP-адрес, мониторинг CloudWatch. И наконец, выбирается IAM-роль и шифрование (рис. 7.14).

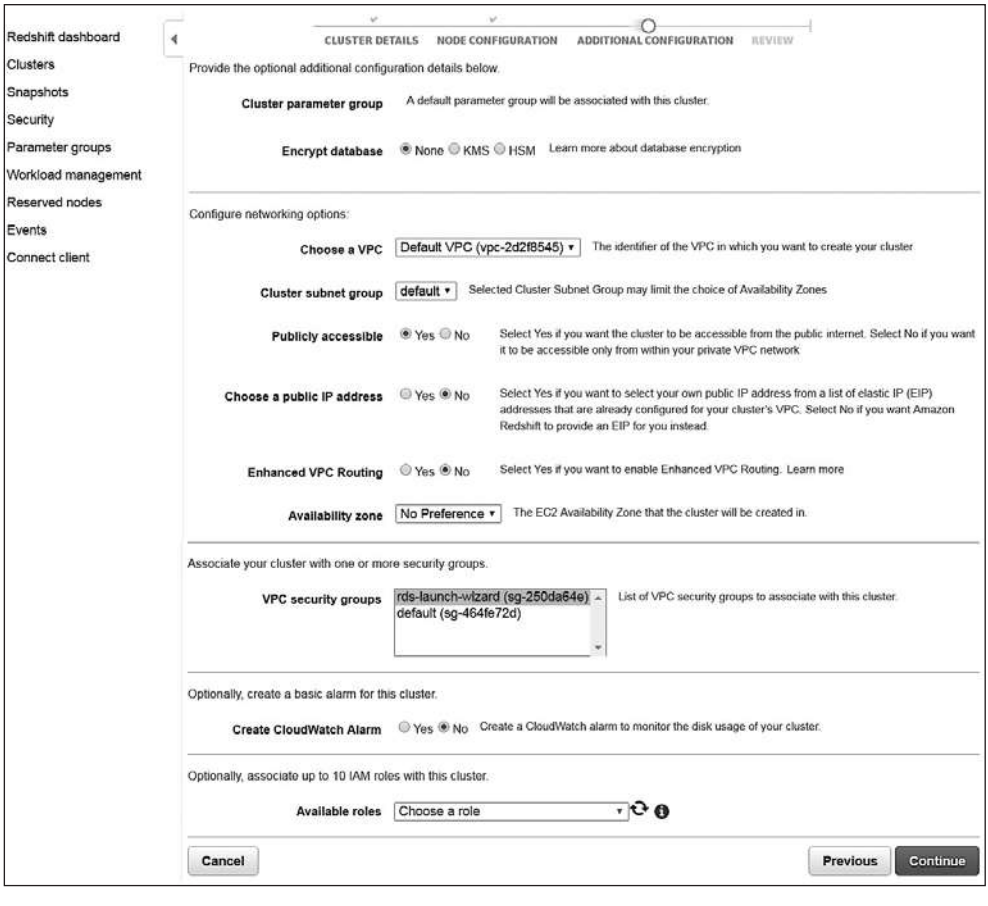


Рис. 7.14. Панель конфигурирования узлов кластера

Далее (рис. 7.15) будет доступна страница, на которой приведены параметры создаваемого кластера; если они удовлетворяют пользователя, то кластер можно создавать.

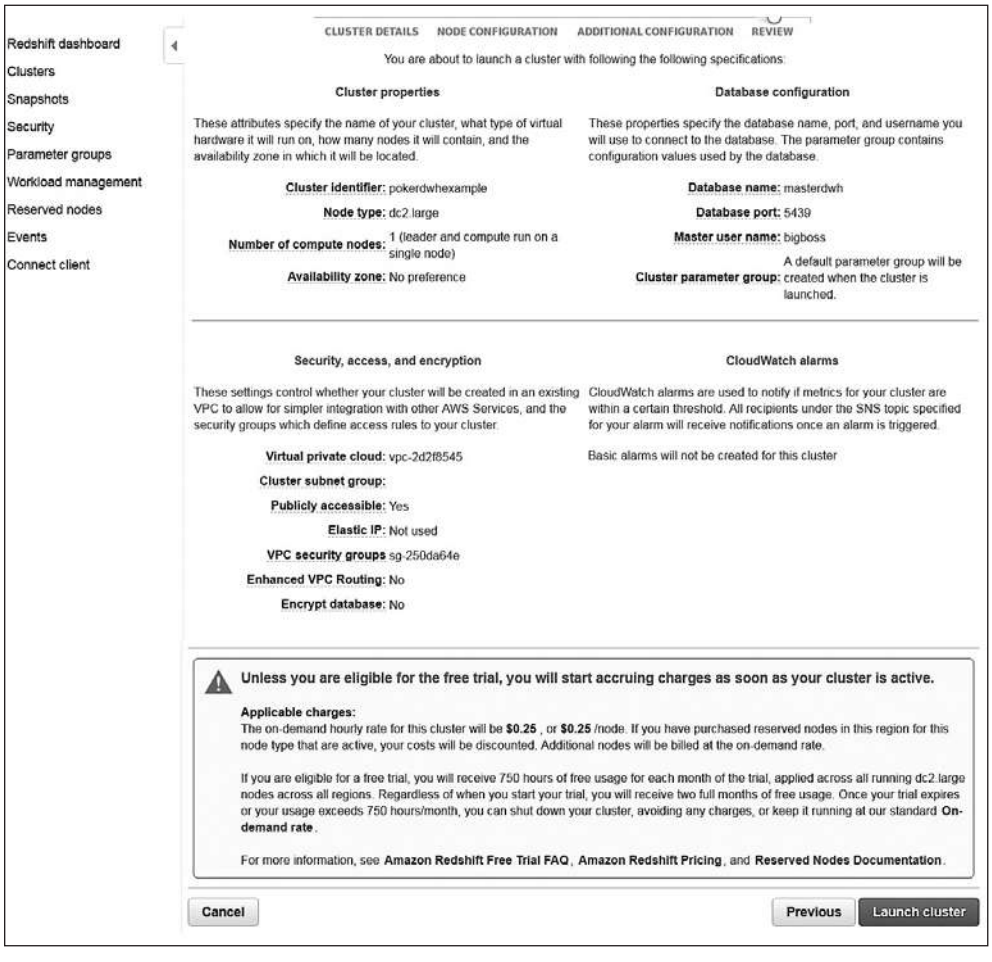


Рис. 7.15. Панель обзора конфигурирования кластера

После создания кластера на панель доступных кластеров добавляется запись. В развернутом виде она показана на рис. 7.16.

Рассмотрим подробнее основные возможности кластера AWS RedShift. Чтобы подключиться к нему, нужно перейти к вкладке Connect client, где указаны соответствующие инструкции. Можно создать зарезервированные экземпляры в качестве типов вычислительных узлов (вкладка Reserved nodes).

Очень интересная возможность кластера — *менеджер нагрузки* (вкладка Workload management), который определяет правила и очередность выполнения нескольких длительных запросов в условиях их конкуренции за ресурсы. Пользователь сам должен конфигурировать менеджер и устанавливать приоритеты выполняемых заданий.

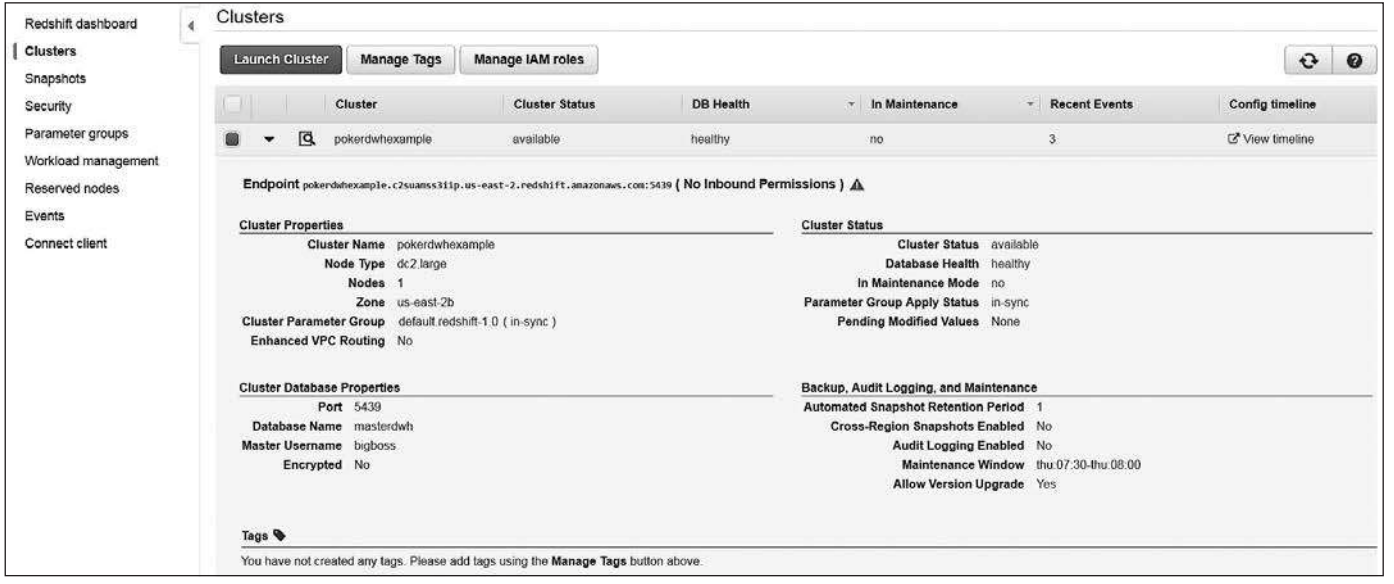


Рис. 7.16. Внешний вид панели созданного кластера

На вкладке Parameter groups приведен набор параметров, относящихся к кластеру. На вкладке Security указываются параметры безопасности сетевых ресурсов и обеспечивается возможность подключения аппаратного модуля шифрования (Hardware Security Module, HSM). Кроме того, можно создать мгновенные снимки (вкладка Snapshots).

7.4. Резюме

В текущей главе мы рассмотрели реализации реляционных хранилищ данных от Microsoft и Amazon. Различаются они лежащими в их основе движками: для Azure SQL DWH это Microsoft SQL DWH + Microsoft Poly, а для AWS RedShift — PostgresSQL, что и обусловливает различия в языках запросов, лежащих в их основе. Кроме того, есть ряд принципиальных различий в способах хранения информации в DWH. Для Azure таким хранилищем является Azure Storage, а для AWS RedShift — базы данных в узлах. И поэтому кардинально различаются способы разделения таблиц по узлам кластера.

Следующее принципиальное различие — использование разного подхода к конфигурированию узлов кластера. Для Azure SQL DWH конфигурирование выполняется с помощью абстрактного обобщенного показателя DWU; соответственно, метрики применения ресурсов собраны в этот один показатель. Для AWS RedShift выбор ограничен конкретными типами экземпляров вычислительных узлов. И соответственно, метрики будут характеризовать реальные величины, связанные с хорошо известной и наглядной интерпретацией (в данном случае — использование CPU, оперативной памяти, операций ввода данных на диск и вывода с него). Однако на самом деле выбор хранилища — вопрос удобства и личного предпочтения: Azure SQL DWH сделано с бóльшим приближением к концепции serverless, чем AWS RedShift. В то же время AWS RedShift содержит многие элементы конфигурирования из области IaaS-сервисов.

Очень полезными компонентами сервиса AWS RedShift являются WLM — WorkLoad Manager и Hardware Security Module, обеспечивающий возможность конфигурирования и подключения устройств аппаратного шифрования. Несмотря на многие общие черты, сравнить эти сервисы напрямую затруднительно. Это примерно то же, что сравнить Microsoft SQL и PostgresSQL, — оба являются мощными масштабируемыми реляционными хранилищами промышленного уровня, способными обеспечить хранение и анализ терабайтных и петабайтных объемов данных.

8 Хранилища данных типа Data Lake

Море — это вечное движение и любовь, вечная жизнь.

Жюль Верн. Двадцать тысяч лье под водой

8.1. Общий обзор специализированных облачных хранилищ больших данных

В предыдущих главах мы рассмотрели облачные хранилища файлов и СУБД различного типа: как реляционные, так и нереляционные. Это достаточно традиционные сервисы, хорошо подходящие для создания информационных систем широкого назначения, включая масштабируемое файловое хранилище, базу данных телеметрии IoT-устройств, масштабируемое хранилище логов и т. д. Но в то же время у всех описанных сервисов есть ряд существенных ограничений. Рассмотрим их по порядку.

Облачные хранилища файлов допускают хранение огромного количества файлов, суммарный объем которых исчисляется многими терабайтами и даже петабайтами. Но в хранилищах эти файлы просто хранятся, а как быть, если нужно провести общее исследование информации в них? Например, для многих сотен файлов логов суммарным объемом несколько гигабайт сгруппировать IP-адреса запросов, чтобы определить страну-источник последних. Или совершить более сложные исследования логов с многих сотен серверов для определения корреляции в событиях, узких мест производительности и пр. Для выполнения подобных действий необходим сторонний сервис, которому это под силу, но он должен иметь доступ к каждому из логов. В принципе, у AWS есть такие сервисы: AWS Athena и AWS RedShift Spectrum, и они специально предназначены для анализа информации в файлах с помощью SQL-подобного синтаксиса (более подробно он будет описан далее).

Если задачу анализа лог-файлов общего назначения можно считать решенной, то для более сложных случаев указанные сервисы недостаточно приспособлены. Как быть, если лог-файлов очень много (терабайты) и необходимо получить периодический отчет, применить алгоритмы машинного обучения или выполнить совместный анализ информации из файлового хранилища и, скажем, реляционной базы данных? Для решения указанных проблем нужно переместить эти файлы в хранилища, допускающие подобные манипуляции. Но если требуется применить машинное обучение или выполнить пакетный анализ огромного количества данных, то нужно использовать HDFS-совместимое хранилище. Вообще, как правило, реляционное хранилище данных — последнее звено в цепочке системы анализа больших данных.

Эта цепочка включает:

- прием потоков данных;
- хранение сырых данных в нереляционном хранилище;
- копирование и агрегацию данных в HDFS-совместимое хранилище Data Lake;
- выполнение интеллектуального анализа данных;
- помещение результата в реляционное хранилище.

Реляционное хранилище обеспечивает доступ к информации для сервисов бизнес-аналитики (рис. 8.1). Но в моей реальной практике был пример архитектуры, в которой было два оконечных хранилища: DWH и Data Lake — и каждое из них выполняет свою задачу.

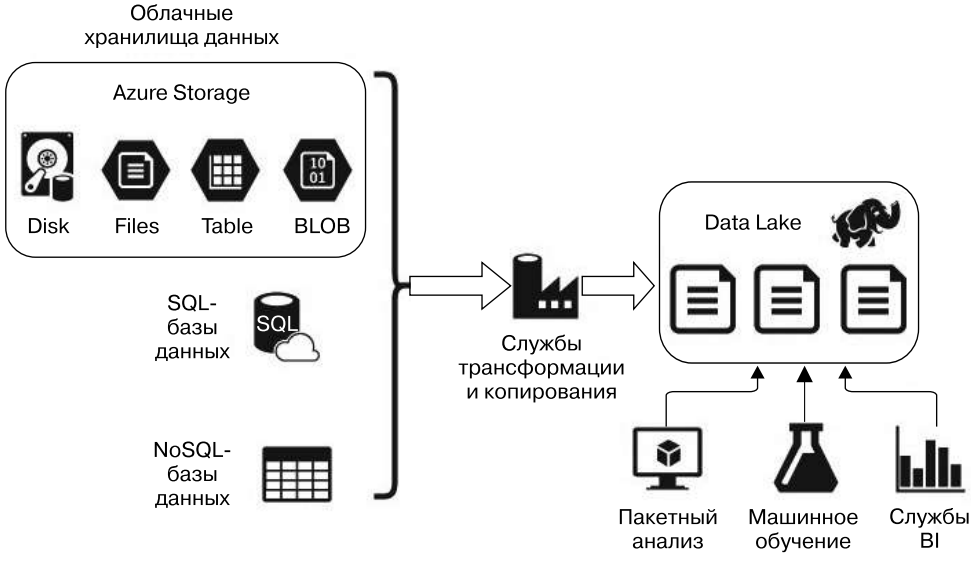


Рис. 8.1. Наполнение хранилища DataLake данными из иных облачных хранилищ и доступ сторонних сервисов к нему

В этом случае можно применить сервисы «из мира» Hadoop (MapReduce, Spark, Pig, Hive и др.) или облачных средств аналитики HDFS-совместимых хранилищ (Azure Data Lake Analytics). Такой подход именуется Data Lake («озеро данных»). Data Lake является хранилищем структурированных, неструктурированных и частично структурированных файлов различного типа и совместно с сервисами экосистемы Hadoop образует «склад данных», который отличается от традиционного SQL DWH, но обладает гораздо большими возможностями, предоставляемыми средствами экосистемы Hadoop.

Data Lake позволяет применять ELT, то есть сначала загрузить файл (в ряде случаев просто копировать), а потом, если надо, преобразовать.

Кроме того, в Data Lake все данные хранятся как файлы в специализированной файловой системе HDFS — каждый файл имеет имя, путь, и эти параметры напрямую фигурируют в запросах обработки данных. В SQL DWH же, как и в любой СУБД, хранение информации полностью отделено от ее логического представления.

Итак, опишем кратко HDFS и поясним, почему она так важна для построения распределенных масштабируемых хранилищ, допускающих массивно-параллельную обработку хранящейся в них информации. HDFS (Hadoop Distributed File System) — распределенная файловая система Hadoop, предназначенная для хранения файлов, которые поблочно распределяются между узлами кластера, лежащего в основе HDFS. Каждый блок (за исключением последнего блока файла, служебного) можно расположить в одном или нескольких узлах одновременно, что определяется коэффициентом репликации. Таким образом, коэффициент определяет количество узлов, в которых может быть размещен блок, и является параметром, настраиваемым на уровне файла. Наличие репликации, а по сути, избыточности данных обеспечивает отказоустойчивость кластера по отношению к отказам отдельных узлов.

Кластер HDFS состоит из узлов двух типов: головного, или *узла имен* (name node), который хранит все метаданные о файлах и репликации их блоков между узлами, а также из *узлов данных* (data node), в которых хранятся блоки файлов. На рис. 8.2 файл 1 реплицирован между узлами данных 1, 2 и 3. Файл 2 реплицирован между узлами 2 и 3, а файл 3 расположен только в узле 4. Узел имен отвечает за выполнение операций с файлами и каталогами (открытие и закрытие файлов, работа с каталогами), а узлы данных непосредственно выполняют операции чтения и записи.

В отличие от обычных файловых систем (например, NTFS) HDFS позволяет только записывать и удалять файл, но не допускает его модификацию. Записывать файл в одно и то же время может только один процесс. При этом организация файлов — традиционная иерархическая с корневым каталогом и набором вложенных каталогов.

Каждый узел, будь то узел имен или узел данных, имеет доступ извне через командную строку, а также содержит веб-сервер, который обеспечивает доступ к веб-порталу, отображающему статус узла и состав файловой системы.

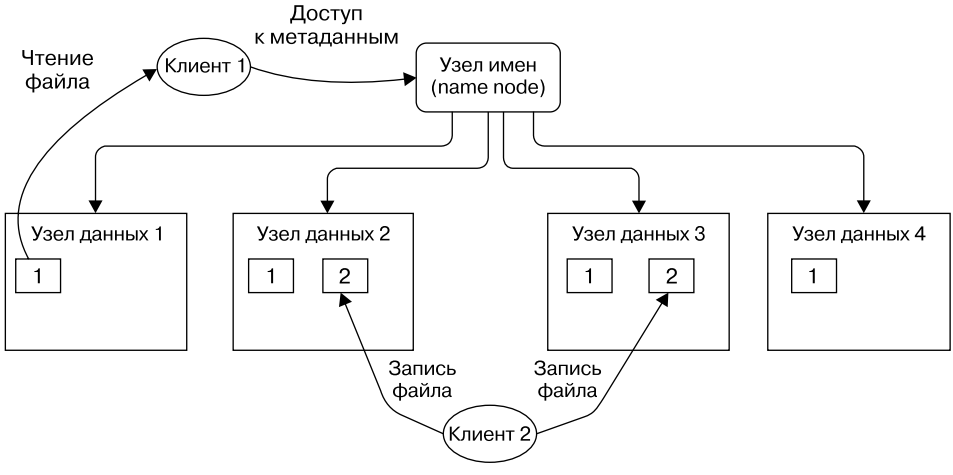


Рис. 8.2. Архитектура HDFS

Такая файловая система лежит в основе большинства сервисов пакетного и интерактивного анализа, сервисов машинного обучения из экосистемы Hadoop, а также составляет основу хранилища типа Data Lake. Рассмотрим подробнее облачные сервисы, реализующие концепции Data Lake.

8.2. Azure Data Lake Store

Azure Data Lake Store представляет собой масштабируемое хранилище данных для целей массивно-параллельной обработки как встроенными сервисами анализа (Azure Data Lake Analytics), так и сервисами экосистемы Hadoop, предоставляемыми сервисами HDInsight. Доступ из кластера HDInsight возможен с помощью REST API, совместимых со стандартом WebHDFS. В отличие от Azure Storage для Azure Data Lake Storage не указываются верхние пределы размеров и количества файлов. Отдельные файлы могут иметь размеры от килобайта до петабайта и сохраняются в виде реплицированных копий, разделенных по кластеру.

Очень важное и полезное свойство этого сервиса — можно хранить файл в том же формате, который использовался при его создании: CSV, LOG и пр. В этом-то и прелесть концепции Data Lake: хранение и анализ файлов без преобразования формата, но с максимально возможной производительностью.

Теперь рассмотрим вопросы безопасности и защиты данных в сервисе Azure Data Lake Store. Прежде всего доступно шифрование данных. Кроме того, имеется возможность интеграции этого хранилища с Azure Active Directory (AD) для обеспечения управления учетными данными и контроля доступа к ресурсам (аутентификация). Интеграция с AD позволяет использовать все преимущества Azure AD, а именно: многофакторную аутентификацию, контроль доступа на основе

ролей (role-based access control) и мониторинг доступа к защищаемым ресурсам. Вообще же, Azure Data Lake Storage поддерживает протокол OAuth 2.0, что позволяет (по крайней мере теоретически) применять отличный от Azure AD сторонний провайдер, поддерживающий OAuth 2.0. Следующая опция Data Lake Store — поддержка прав доступа к файлам и каталогам на основе стандартов POSIX, доступных с помощью протокола WebHDFS.

В качестве внутренней файловой системы в этом сервисе может быть использована Azure Data Lake Store File System, являющаяся адаптацией HDFS; в последней объекты адресуются с помощью префикса `adl://`.

Рассмотрим на практике, как создавать экземпляр Azure Data Lake Store в вебпортале. Прежде всего в строке поиска доступных ресурсов следует написать `Data Lake Store` и в появившейся форме (рис. 8.3) нажать кнопку Create (Создать). Это приведет к открытию формы, показанной на рис. 8.4. Здесь нужно указать имя (Name), выбрать ресурсную группу (Resource group), местоположение (Location) и тип шифрования (Encryption). Доступны три типа шифрования: Do not enable encryption (Не шифровать данные), Use keys, managed by Data Lake Store (Использовать ключи, управляемые Azure Data Lake Store) и Use keys from your own Key Vault (Применить ключи, управляемые сервисом Key Vault и поставляемые пользователем). Особняком стоит настройка моделей оплаты — Pricing package. Я в своем примере выбрал личную подписку Pay-as-You-go.

Рис. 8.3. Первый шаг создания Azure Data Lake Store

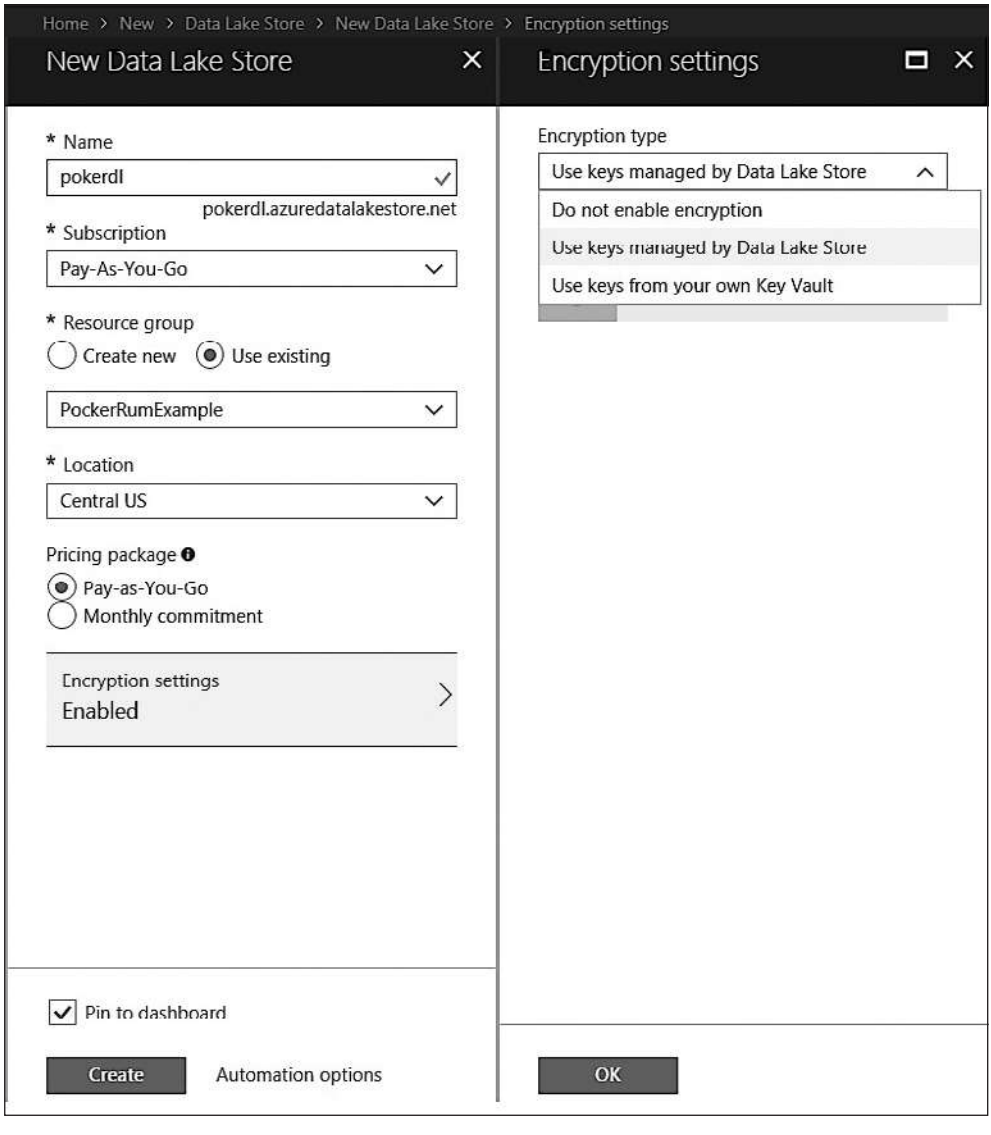


Рис. 8.4. Форма конфигурирования Azure Data Lake Store

Внешний вид общей панели сервиса Azure Data Lake Store выглядит следующим образом (рис. 8.5).

Графический интерфейс этого сервиса не так интересен, как его возможности. Укажу лишь, что оперировать данными на веб-портале можно после нажатия на ссылку Data explorer (Обозреватель данных) (рис. 8.6).

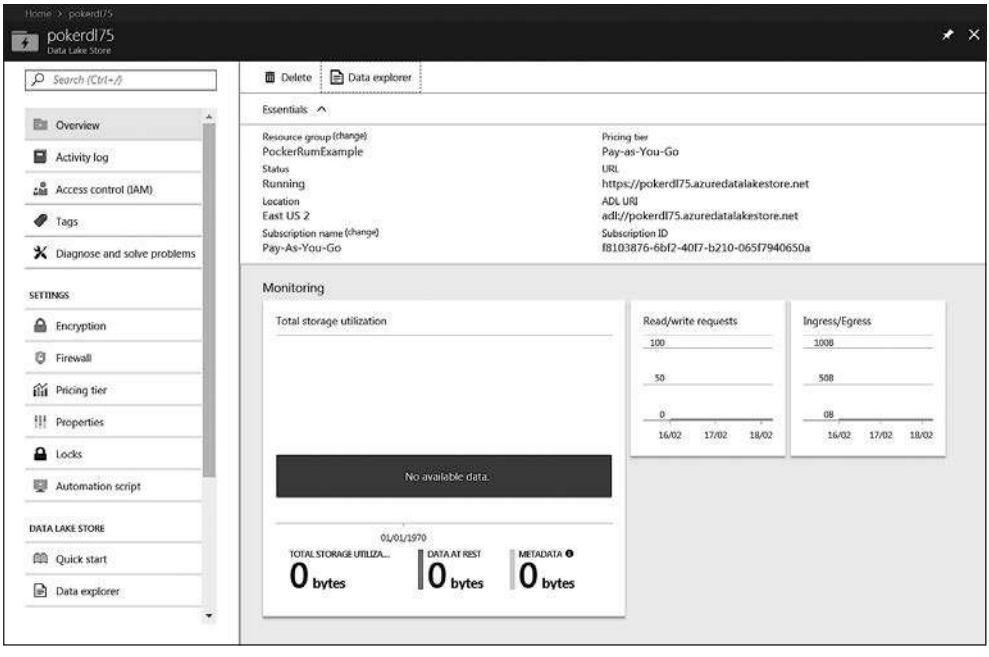


Рис. 8.5. Внешний вид общей панели сервиса Azure Data Lake Store

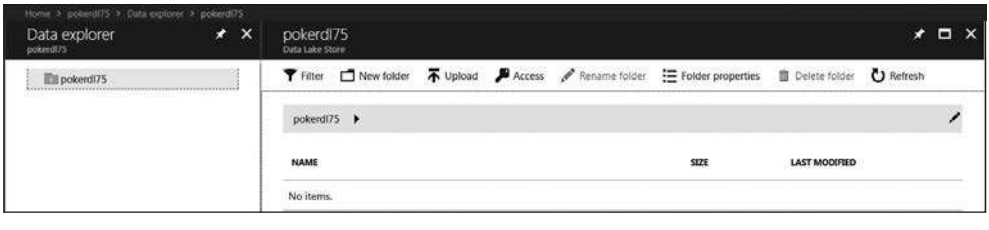


Рис. 8.6. Внешний вид панели Data explorer (Обозреватель данных)

8.3. AWS Data Lake Solutions

В отличие от Azure, облачный провайдер AWS не содержит отдельного сервиса Data Lake (рис. 8.7). Вместо этого AWS предлагает решение, реализованное как стек ресурсов (https://aws.amazon.com/answers/big-data/data-lake-solution/), представленных в виде доступного для развертывания файла CloudFormation. Мы не будем подробно рассматривать это решение, опишем лишь, что оно собой представляет.

Прежде всего, основным хранилищем файлов в данном случае является S3 Bucket. Этот сервис обеспечивает доступность, надежность и криптографическую защищенность. Выполнить быстрый поиск файлов поможет кластер AWS ES — сервис Elasticsearch, который предназначен для индексирования файлов в целях

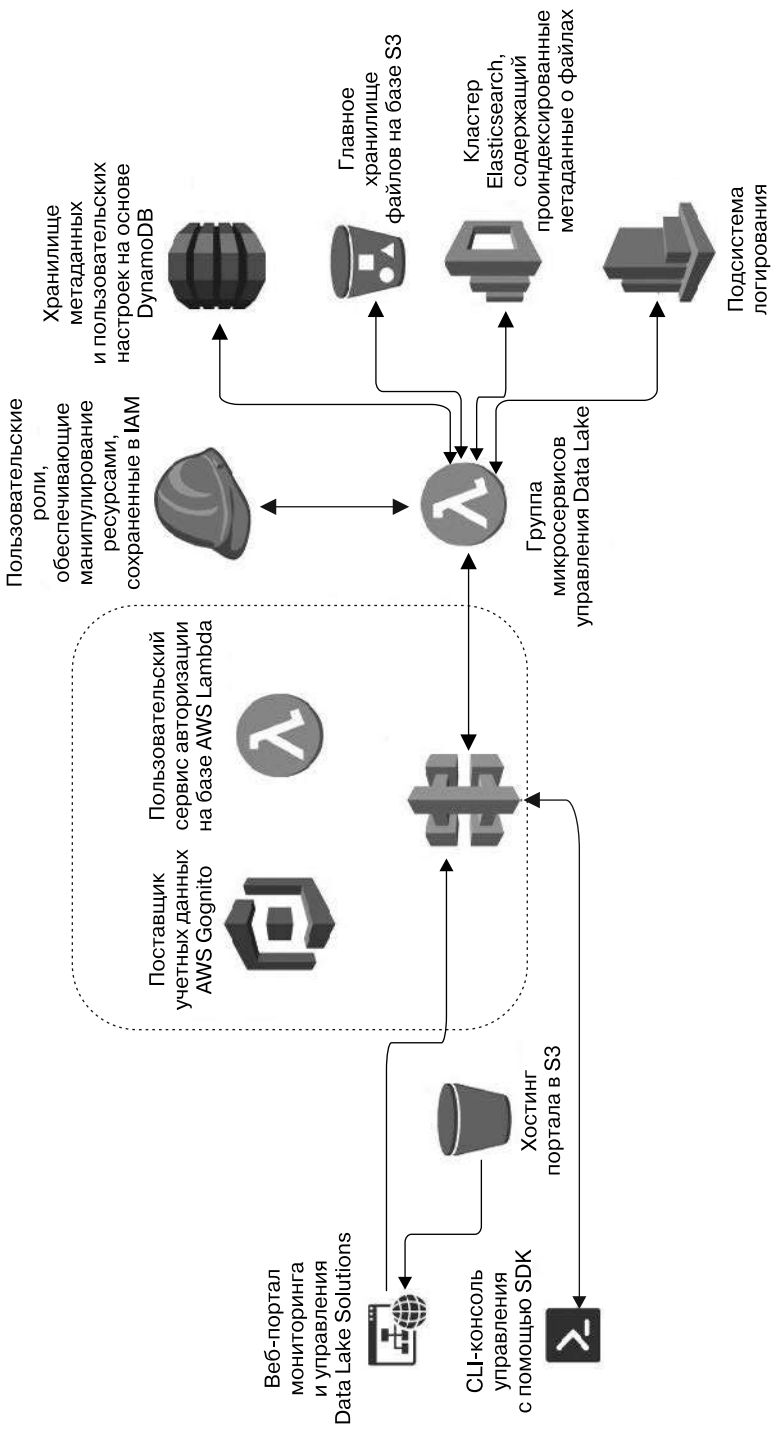


Рис. 8.7. Построение сервиса Data Lake на базе стандартных ресурсов AWS

обеспечения высокой скорости поиска. Общие метаданные и пользовательские настройки хранятся в таблицах DocumentDB. Чтобы получить доступ извне, это решение предоставляет веб-портал управления, построенный с помощью технологии одностраничного приложения. Идея такого приложения состоит в том, что для получения контента и выполнения действий осуществляются запросы к сервисам REST, которые в данном случае доступны через применение API-шлюзов (AWS API Gateway). Все конечные точки этого шлюза защищены с помощью протокола OAuth 2.0 и сервиса AWS Cognito, отвечающего за хранение учетных данных пользователей и предоставление конечных точек OAuth.

Помимо указанной архитектуры, можно задействовать другие сервисы: AWS Glue для перемещения данных Data Lake и AWS RedShift Spectrum для обеспечения аналитики среди файлов, размещенных в AWS S3 Bucket. Более подробную информацию можно получить на сайте https://aws.amazon.com/blogs/big-data/build-a-data-lake-foundation-with-aws-glue-and-amazon-s3/.

8.4. Резюме

Реализации хранилищ Data Lake от AWS и Azure принципиально различаются тем, что Azure Data Lake — отдельный сервис, реализующий HDFS-совместимое хранилище, получить доступ к которому можно с помощью стандартных средств экосистемы Apache Hadoop. Это именно сервис типа PaaS, требующий минимального администрирования со стороны пользователя. Azure Data Lake легко интегрируется с другими сервисами Azure, не требуя участия сторонних программ, написанных пользователем. В отличие от этого AWS предоставляет решение Data Lake, построенное с применением стандартных средств AWS и являющееся системой, готовой к использованию. Тут данные хранятся в AWS S3, что влечет необходимость выполнять дополнительное администрирование (хотя бы на уровне анализа логов), а также прилагать усилия по интеграции с другими сервисами AWS.

Часть III
Доставка BigData в облако

> Видите ли, — сказал он, — мне представляется, что человеческий мозг похож на маленький пустой чердак, который вы можете обставить как хотите. Дурак натащит туда всякой рухляди, какая попадется под руку, и полезные, нужные вещи уже некуда будет всунуть, или в лучшем случае до них среди всей этой завали и не докопаешься. А человек толковый тщательно отбирает то, что он поместит в свой мозговой чердак. Он возьмет лишь инструменты, которые понадобятся ему для работы, но зато их будет множество, и все он разложит в образцовом порядке.
>
> *Артур Конан Дойль. Этюд в багровых тонах*

В предыдущих частях были описаны общие вопросы, посвященные архитектуре облачных систем, а также облачные сервисы для хранения данных в различных форматах и базах. Данные могут поступать в облачные хранилища различными путями. Если данные были сгенерированы иной информационной системой вне облачного аккаунта, то их следует переместить в облачную среду из внешнего хранилища с помощью сервисов трансформации и копирования. Может возникнуть резонный вопрос: а зачем нужно такое перемещение, тем более что в случае больших данных оно будет представлять длительную и дорогостоящую операцию? Дело в том, что для анализа этих данных необходимо размещение их в тех хранилищах, которые будут доступны напрямую из вычислительных узлов кластеров сервисов аналитики с минимальными задержками.

Следующий вариант — прямое поступление больших данных в облачную систему. В этом случае могут быть использованы как средства прямой загрузки того или иного хранилища (например, SDK), так и специализированные сервисы, обеспечивающие возможность приема данных вне зависимости от типа хранилища (например, концентраторы).

Итак, существуют следующие стратегии поступления больших данных в облачные информационные системы.

- *Прямое копирование файлов.* В зависимости от конечной цели копирования и объемов данных выбирается свой тип облачного хранилища. Как правило, облачное хранилище общего назначения отвечает большинству критериев. Когда необходимо обеспечить доступность файлов с виртуальных машин, используется хранение файлов в файловой системе. Но, как правило, конечная цель прямого копирования файлов — анализ информации в файлах. Для этого требуется переместить их в специализированное хранилище, допускающее такой анализ. Для AWS подобным хранилищем может быть напрямую хранилище

общего назначения AWS S3, для Azure — специализированное хранилище Azure Data Lake Store. По сути, описанный подход реализует концепцию Data Lake.

- *Прямое копирование информации*. Информацию можно хранить в базах данных и копировать в однотипные сервисы БД, например из баз данных SQL в сервисы РБД или реляционные хранилища данных. То же самое относится к нереляционным базам данных.
- *Трансформация данных*. Информация из внешних источников преобразуется в другой формат. Существуют различные варианты трансформации, один из которых — последовательное преобразование данных из одного формата в другой. В простейшем случае выполняется одна подобная трансформация, например из CSV-файла в таблицу SQL. В другой ситуации возможна цепочка преобразований. Так, в части I был описан пример системы мониторинга потребления электроэнергии, в которой сырые данные телеметрии сначала хранятся в табличной базе данных, а после перемещаются в реляционное хранилище.

В текущей части мы рассмотрим различные варианты перемещения данных, доступные в облачных провайдерах. Специализированный софт для копирования будет только упомянут, поскольку его детальное описание выходит за рамки книги.

9 Прямая загрузка данных

— На каком основании я опять буду здесь? — тревожно спросил Иван.

Стравинский как будто ждал этого вопроса, немедленно уселся опять и заговорил:

— На том основании, что, как только вы явитесь в кальсонах в милицию и скажете, что виделись с человеком, лично знавшим Понтия Пилата, — как моментально вас привезут сюда, и вы снова окажетесь в этой же самой комнате.

— При чем здесь кальсоны? — растерянно оглядываясь, спросил Иван.

— Главным образом Понтий Пилат. Но и кальсоны также.

Михаил Булгаков. Мастер и Маргарита

У каждого облачного сервиса хранения данных есть конечная точка REST API и программная SDK, которая служит для упрощения доступа к этой точке. Кроме того, каждый сервис поддерживает специфический протокол обмена данными. SDK и конечная точка позволяют загружать данные напрямую в этот сервис. Такой способ позволяет перемещать большие объемы данных, необходимых для последующего пакетного и интерактивного анализа, а также тренировки моделей машинного обучения. Каждый сервис хранения данных имеет специфичный способ добавления данных, и мы рассмотрим эти способы применительно ко всем сервисам, описанным в главе 2.

Поскольку использование SDK не слишком отличается в AWS и Azure, ограничимся рассмотрением Azure SDK на примере консольных приложений .Net Core 2.0.

Если вы не знакомы с этим языком и вообще с языками программирования, то можете пропустить эти разделы или читать «по диагонали». Но я все-таки рекомендую их прочесть, чтобы понимать, как устроены сервисы копирования. Оба облачных аккаунта, AWS и Azure, имеют в своем составе очень мощные сервисы копирования и трансформации данных: AWS Glue и Azure Data Factory. С их помощью очень удобно перемещать информацию между отдельными сервисами внутри облачного провайдера и в ряде случаев между внешними ресурсами и облаком. Их описанию будет посвящена отдельная глава. В этой же рассмотрим сервисы, утилиты и SDK, предназначенные для копирования данных в облако. Следует иметь в виду, что в состав облачных провайдеров входят сервисы, облегчающие миграцию баз данных и локальных ресурсов в облако. Но вопрос миграции довольно специфичен, заслуживает отдельной книги и не относится к тематике нашего издания, а потому здесь не будет рассматриваться. Кроме того, следует помнить: очень многие облачные сервисы могут напрямую добавлять информацию в хранилища того или иного типа. Так, многие сервисы Azure пишут лог-файлы в Azure Blob Storage, а метрики — в Azure Table Storage. Сервисы Azure Event Hub моут «захватывать» сообщения и направлять их копии в хранилище Azure Blob Storage или в Azure Data Lake Store.

9.1. Доставка данных в облачное хранилище общего назначения

Как уже отмечалось в части II, в самом общем виде облачные хранилища делятся на хранилища двоичных объектов (BLOB) и хранилища файлов на основе протоколов сетевых файловых систем (File Storage). В обоих случаях данные можно добавлять в хранилище следующими способами:

- с помощью веб-портала облачного аккаунта, что можно использовать только для добавления малого количества файлов небольшого объема;
- благодаря конечным точкам REST API, представленным в виде программных SDK, которые могут быть применены в программных продуктах.

Для файлового хранилища доступен еще один способ — монтировать его в качестве сетевого к виртуальной машине и работать с ним уже из нее.

Итак, рассмотрим использование Azure Storage SDK на примере приложения .Net Core. Сначала следует установить пакеты NuGet (рис. 9.1).

Код для загрузки одиночного файла с именем `"format.json"` в BLOB-контейнер под именем `"pokerattachments"` выглядит так (рис. 9.2).

Тут не приведен один параметр — строка подключения. Способ ее получения описан в главе 4.

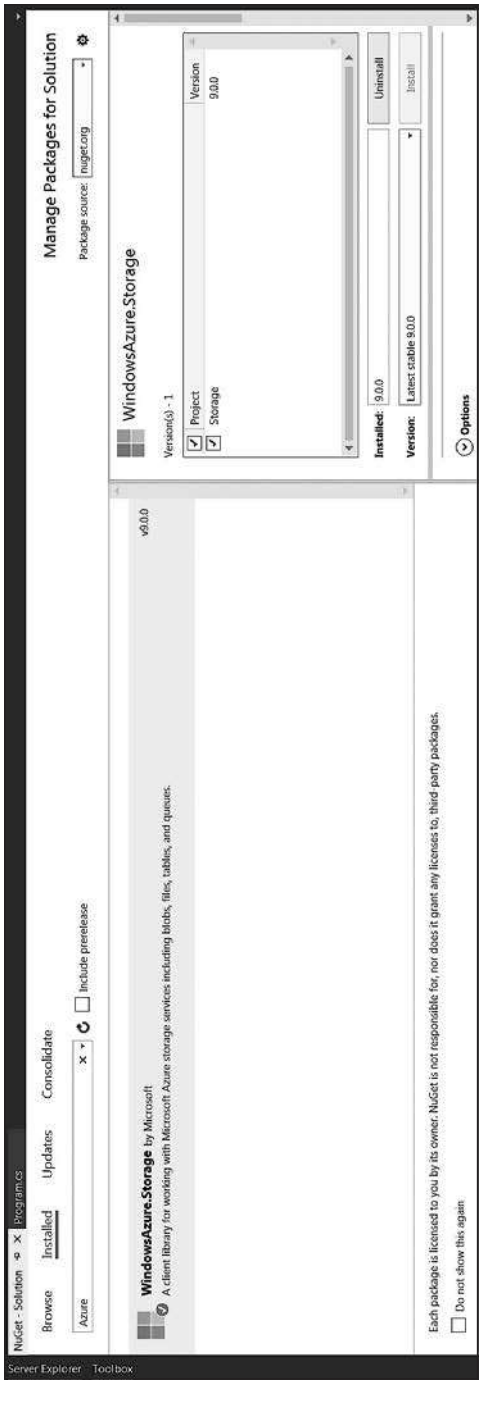


Рис. 9.1. NuGet-пакеты, необходимые для работы с Azure Storage SDK

```
private static async Task MainAsync()
{
    string storageConnectionString = "";

    CloudStorageAccount storageAccount = CloudStorageAccount.Parse(storageConnectionString);

    CloudBlobClient cloudBlobClient = storageAccount.CreateCloudBlobClient();
    var cloudBlobContainer = cloudBlobClient.GetContainerReference("pokerattachments");
    await cloudBlobContainer.CreateIfNotExistsAsync();

    BlobContainerPermissions permissions = new BlobContainerPermissions
    {
        PublicAccess = BlobContainerPublicAccessType.Blob
    };
    await cloudBlobContainer.SetPermissionsAsync(permissions);

    string localPath = @"C:\Users\alexander\Projects";
    string localFileName = "format.json";
    string sourceFile = Path.Combine(localPath, localFileName);

    CloudBlockBlob cloudBlockBlob = cloudBlobContainer.GetBlockBlobReference(localFileName);
    await cloudBlockBlob.UploadFromFileAsync(sourceFile);
}
```

Рис. 9.2. Пример использования Azure Storage SDK для загрузки файлов

Сначала в коде создается класс `CloudStorageAccount`, в котором создается клиент для работы с сервисом BLOB `CloudBlobClient`. Далее в клиенте с помощью метода `.CreateIfNotExistsAsync()` создается BLOB-контейнер. Затем для контейнера указываются права доступа `.SetPermissionsAsync(permissions)`, создается блочный BLOB (метод `.GetBlockBlobReference(fileNAme)`), и файл для него загружается в хранилище.

Теперь рассмотрим, как использовать приложения, рекомендуемые Microsoft, для копирования данных в Storage-аккаунт.

Одно из самых удобных приложений, применяемых для копирования файлов в облачное хранилище, — AzCopy. Оно представляет собой консольную программу, доступную для Windows и Linux. Эта программа очень удобна для конфигурирования. Например, можно указать параметры копирования только тех файлов, которые были изменены после их добавления в хранилище. Ниже представлен пример использования AzCopy для копирования всех текстовых файлов в BLOB-контейнер.

```
AzCopy /Source:<источник> /Dest:https://<аккаунт>.blob.core.windows.net/
<blob-контейнер> /DestKey:<ключ аккаунта> /Pattern:"*.txt"
```

Здесь все файлы с расширением `.txt` копируются из локального пути `<источник>` в Storage Account с именем `<аккаунт>` в контейнер `<blob-контейнер>`. Помимо копирования файлов из локальных источников в BLOB и наоборот, эта утилита может работать с File Storage, копировать файлы между разными аккаунтами, между BLOB и File Storage, и импортировать данные из Azure File Storage.

Подробная документация представлена на сайте https://docs.microsoft.com/en-us/azure/storage/common/storage-use-azcopy.

В своей практике я регулярно использовал эту утилиту. Характерный и показательный пример — копирование большого количества файлов (суммарный объем около 500 Гбайт) из локального файлового хранилища в Azure File Storage. Помимо большого объема переносимой информации, трудность заключалась в том, что приложение непрерывно генерировало новые файлы и изменяло текущие (не все десятки тысяч из 500 Гбайт, но порядка сотни-двух в день). Кроме того, передача по сети такого большого количества файлов с максимальной скоростью могло частично блокировать локальную сеть и помешать нормальному функционированию системы. Чтобы не допустить этого, копирование происходило в выходные, а в рабочие дни в нерабочее время производилась синхронизация, то есть копировались файлы, добавленные локально после копирования основной массы.

Следующий сервис импорта данных в Azure Blob Storage — Microsoft Azure Import/Export Service. Он доступен с общей панели поиска сервисов и предназначен для передачи больших объемов файлов в Azure Blob Storage с локальных дисков. Для этого в Azure создаются специальные задания (import/export job).

Для конфигурации задания импорта данных нужно указать имя, хранилище Azure Storage и сгенерировать файл журнала с помощью утилиты WaImportExportV1 (подробности см. на сайте https://docs.microsoft.com/en-us/azure/storage/common/storage-import-export-service?toc=%2fazure%2fstorage%2fblobs%2ftoc.json).

Эта утилита чем-то похожа на AzCopy, но на выходе выдает JR- или XML-файл. В результате данные с вашего локального диска будут доставлены в Azure Blob Storage. Import/export-сервис позволяет создавать бэкапы данных с локальных дисков и восстанавливать данные из бэкапов. В качестве типов BLOB могут быть Page, Block и Append.

Аналогичный сервис имеется и у Amazon — AWS Import/Export service, работающий по схожей схеме.

Оба этих сервиса требуют большого объема предварительного конфигурирования утилит и локальных дисков, которые будут источниками данных. Подробную информацию можно получить на сайте https://aws.amazon.com/documentation/importexport/.

9.2. Доставка данных в реляционные БД и хранилища

Синхронизировать данные между реляционными и нереляционными базами данных можно с помощью любого стороннего приложения, допускающего синхронизацию/миграцию данных между базами. Довольно часто в среде Azure в качестве такового используется набор утилит RedGate. Кроме того, можно применять сервис Azure Data Factory и относительно новый сервис — Azure Database Migration Service (сейчас он находится в стадии Public Preview). Ниже представлены не-

сколько замечаний, которые позволят увеличить производительность операции добавления данных:

- имеет смысл на время копирования максимально поднять ценовой уровень экземпляра сервиса Azure SQL или взять максимальный размер экземпляра AWS RDS;
- в базе данных — приемнике на время копирования следует отключить опцию аудита и триггеры на добавление/обновление;
- после добавления данных в таблицу стоит обновить индексы и пересчитать статистику.

Сами по себе сервисы реляционных баз данных для внешних клиентов ничем не отличаются от локальных, так что для них действуют все те же правила, рекомендации и подходы, что и для копирования данных в локальные базы данных.

9.3. Доставка данных в нереляционные базы данных

В этом разделе рассмотрим способы прямой программной загрузки данных в нереляционные хранилища данных. Начнем с примера использования Azure Table Storage SDK. Вы уже, наверное, заскучали, читая теорию и про разные сервисы хранения данных. Вспомним пример с покер-румом, описанный в части I. Такая система представляет собой отличную иллюстрацию системы, в которой большие данные получаются за счет накопления большого количества малых порций данных — сообщений или событий. Если пока не рассматривать игровые сервисы, отвечающие за реагирование на эти события, то весьма интересен также вопрос логирования, то есть сохранения всех событий в хранилище, допускающем хранение большого количества данных. Реляционная база данных для этой цели не подойдет ввиду расточительности, поскольку нам не нужны ее транзакционные и реляционные свойства. Нам, по сути, требуется таблица, которая допускает быстрый доступ и оперирует большими объемами хранения, обеспечивая притом низкую стоимость. Всеми этими свойствами обладает Azure Table Storage.

Как и прежде, рассмотрим использование Azure Table Storage на примере консольного приложения, созданного с помощью технологии .Net Core. Для этого прежде всего следует добавить пакеты NuGet (рис. 9.3).

Чтобы иметь возможность добавлять строки в таблицу, нужно определиться с ключами строки и раздела. Предположим, мы будем отслеживать ходы игрока в рамках игры. Для этого в качестве ключа раздела удобно взять идентификатор игрока, в качестве ключа строки — уникальный идентификатор (на самом деле удобнее взять Timestamp, поскольку при анализе игры требуется упорядочивать ходы по времени). В программе элемент данных таблицы представляется с помощью наследуемого от `TableEntity` класса `GameEvent`. В нашем случае он имеет следующий вид (рис. 9.4).

Рис. 9.3. Пакеты NuGet, необходимые для использования Azure Table Storage SDK

```
using Microsoft.WindowsAzure.Storage.Table;

namespace Tables
{
    public class GameEvent : TableEntity
    {
        public GameEvent(string playerId, string eventId)
        {
            this.RowKey = eventId;
            this.PartitionKey = playerId;
        }

        public GameEvent() { }

        public int GameId { get; set; }

        public string Action { get; set; }

        public int ActionValue { get; set; }

    }
}
```

Рис. 9.4. Класс GameEvent, представляющий элемент данных таблицы

В данном классе поле `GameId` представляет собой номер текущей игры. Действие игрока в чисто методических целях, конечно, представляется в строковом поле `Action` (бет, колл, олл-ин и др.), а числовая величина, соответствующая этому действию (например, ставка), — в поле `ActonValue`.

Основной код тестовой программы, которая будет имитировать действия пользователя в виде потока сообщений со случайными параметрами, приведен на рис. 9.5. В этой программе создается экземпляр класса `CloudStorageAccount`, после чего с помощью класса `CloudTableClient` создается таблица GameEvents. Далее с помощью цикла создаются экземпляры класса `GameEvent`, которые, задействуя статический метод `.Insert()` класса `TableOperation` и метод `.ExecuteAsync()` экземпляра таблицы, и осуществляют вставку новой строки в таблицу.

Перед выполнением кода следует вставить строку подключения облачного аккаунта, к которому относится эта таблица. После работы программы можно просмотреть данные в таблице, например, с помощью Microsoft Azure Storage Explorer (рис. 9.6).

Вдобавок к одиночному добавлению новой строки в таблицу можно вставлять группу строк (bulk insert).

```
private static Random _rnd = new Random();

private static async Task  MainAsync()
{
    string cnnStr = "";
    CloudStorageAccount storageAccount = CloudStorageAccount.Parse(cnnStr);
    CloudTableClient tableClient = storageAccount.CreateCloudTableClient();

    CloudTable table = tableClient.GetTableReference("GameEvents");

    await table.CreateIfNotExistsAsync();

    int maxEventsInSeries = 100;

    string[] actions = new string[] { "bet", "all-in", "call", "raise", "fold", "check" };

    for (int e = 0; e < maxEventsInSeries; e++)
    {
        string playerId = _rnd.Next(1,5).ToString();
        string eventId = Guid.NewGuid().ToString();

        GameEvent gameAct = new GameEvent(playerId, eventId);

        gameAct.Action = actions[_rnd.Next(0, 5)];
        gameAct.ActionValue = _rnd.Next(0,500);
        gameAct.GameId = 5;

        TableOperation insertRowOperation = TableOperation.Insert(gameAct);
        await table.ExecuteAsync(insertRowOperation);
    }
}
```

Рис. 9.5. Основное тело программы, имитирующей ходы игрока

С точки зрения нашего реального примера в использовании Azure Table Storage есть один существенный недостаток. Дело в том, что, помимо логирования хода, игровой сервер должен выполнять действия. Azure Table Storage не предоставляет возможностей генерации событий в ответ на добавления записи в таблицу, а поэтому требуется применять отдельный игровой сервер, который будет обрабатывать весь поток событий от игроков и одновременно логировать эти события в базу данных NoSQL. Проблема такого подхода состоит в построении надежного и высокодоступного игрового сервера, который в данном случае будет слабым звеном системы, чей отказ фатально скажется на всей системе.

Следующая проблема применения Azure Table Storage SDK — использование строки подключения уровня Storage Account, которая дает полный контроль над его ресурсами.

О путях решения второй проблемы (общие ключи) мы поговорим позднее, а вот первая проблема имеет решение в сервисе Azure CosmosDB SQL API. В этом случае в качестве хранилища игровых событий будет выступать база DocumentDB.

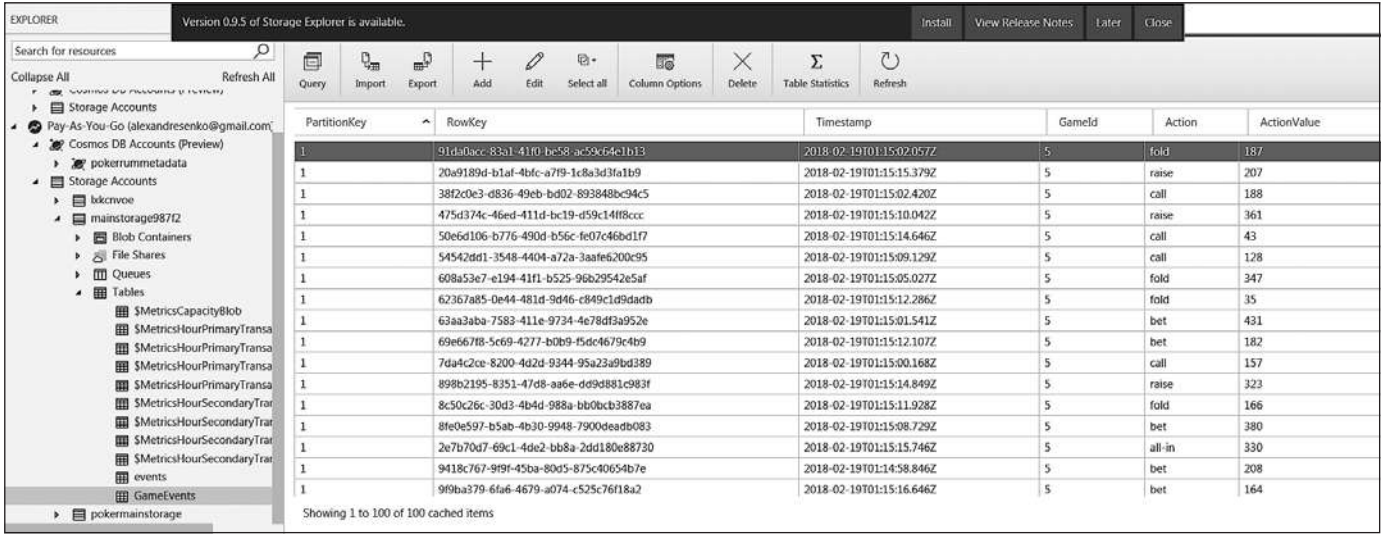


Рис. 9.6. Данные, добавленные в таблицу

Когда игрок делает ход (это может быть ставка, пас и пр.), клиентское приложение с помощью CosmosDB SDK записывает игровую ситуацию этого игрока (то есть текущий ход, карты, ставки и т. д.) в коллекцию, хранящую все подобные игровые события. Для данной коллекции должен быть сконфигурирован триггер, срабатывающий после успешного добавления новой записи. Он запускает бессерверный игровой сервис на базе Function App. Тот, в свою очередь, получив это входное событие (а по сути это есть не что иное, как игровая ситуация с точки зрения одного игрока, сделавшего ход) извлекает из другой коллекции документ (или читает метаданные соответствующей вершины графа, если используется графовая база данных), который описывает игровую ситуацию на предыдущем ходе. Сравнивая текущий ход игрока с предыдущей игровой ситуацией, игровой сервис должен обновить JSON документа игровой ситуации сообразно тому, как этот документ должен выглядеть после того, как пользователь сделал ход. После этого необходимо отправить обновления всем игрокам, направив их приложениям сообщения, обновляющие их UI через шину интеграции. Выполнить данную операцию можно с помощью другой бессерверной функции, которая запускается триггером обновления коллекции игровых ситуаций (рис. 9.7).

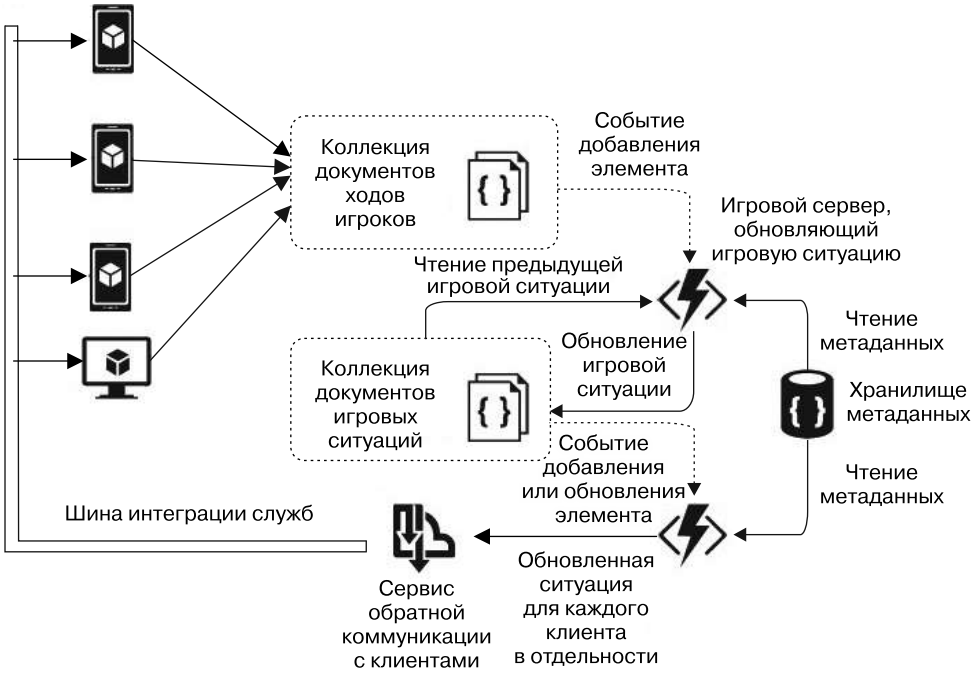


Рис. 9.7. Построение ядра покер-рума на базе архитектуры, ориентированной на события на базе NoSQL и бессерверных игровых сервисов

Итак, мы рассмотрели общую архитектуру игрового приложения, построенного с использованием нереляционной базы данных (в данном случае — документоориентированной CosmosDB SQL API), бессерверных функций (Azure Function App) и шины интеграции сервисов (Service Bus Topic). Все игровые события игроков здесь сохраняются и доступны для последующего анализа и построения отчетов, выявления статистики (более подробно мы изучим этот вопрос в части IV). В такой архитектуре отсутствует центральный игровой сервер, принимающий на себя весь поток сообщений и отвечающий за обновление игровой ситуации и определение того, какое сообщение должно быть послано каждому клиенту в ответ на действие его партнеров. Основные трудности по обработке входящих сообщений берет на себя база данных, клиент должен иметь только ее SDK, ключи и адрес конечной точки. Игровые сервисы в подобном случае разделены на небольшие логически обособленные фрагменты, играющие каждый свою роль (реализована концепция микросервисов). Более удобный сервис, обеспечивающий прием сообщений от многих клиентов (концентратор сообщений, Message Hub), а также шина интеграции сервисов, будут рассмотрены в главе 10. Пока отметим следующее: похожую архитектуру можно реализовать и с помощью сервисов AWS, а именно DynamoDB + Lambda + SNS. Рассмотрим реализацию этой архитектуры на Azure, а потом я скажу пару слов об отличиях этого решения от AWS.

Итак, первоначальным источником сообщений будут являться действия пользователя, которые благодаря коду преобразуются в сообщения формата JSON, посылаемые напрямую в базу данных хранения игровых шагов. Как и прежде, для демонстрации работы с Azure CosmosDB SDK создадим консольное приложение .Net Core. Прежде всего добавим следующие NuGet-пакеты (рис. 9.8).

Код, имитирующий действия пользователя, имеет следующий вид (рис. 9.9).

Данный код очень похож на код добавления новой строки в таблицу Azure Table Storage, только является более простым. А именно: здесь сначала создается экземпляр класса `DocumentClient`, с которым и будут производиться действия. В частности, документ добавляется с помощью метода `.CreateDocumentAsync` этого класса. Я умышленно не стал упрощать код дальше (можно убрать класс `GameEvent`, а вместо него использовать анонимный класс). Параметры `databaseEndpoint` и `accessKey` можно взять с вкладки Keys аккаунта CosmosDB. Остальные параметры в пояснении не нуждаются.

Результат работы этого приложения можно наблюдать непосредственно на вебпортале на вкладке Data Explorer. Для наглядности раскрыт один из документов (рис. 9.10).

Теперь покажем, как подключить экземпляр Azure Function App, запускающийся при добавлении нового документа в коллекцию. Прежде всего следует создать экземпляр сервиса Function App, который будет вызываться при добавлении игровых событий в коллекцию. Для этого в строке поиска ресурсов следует набрать `Function App` (рис. 9.11).

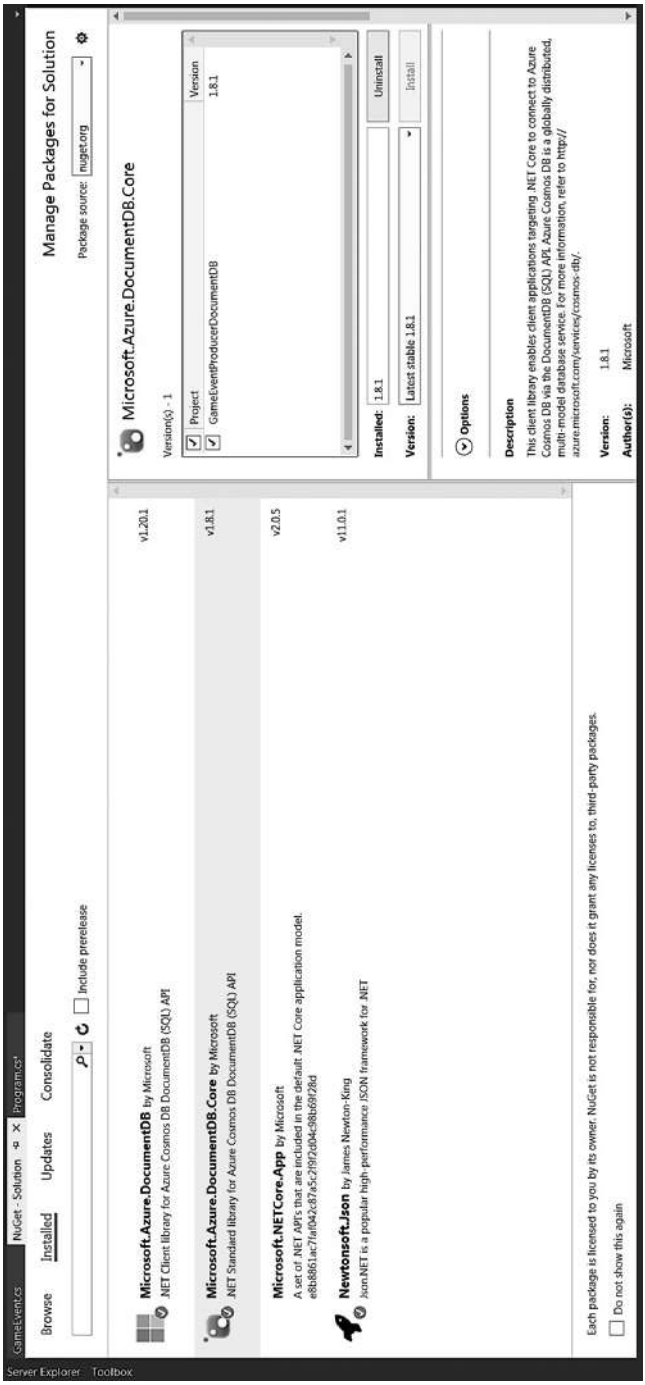


Рис. 9.8. NuGet-пакеты, необходимые для работы с CosmosDB SDK

```
private static Random _rnd = new Random();

public static async Task MainAsync()
{
    string databaseEndpoint = "https://pokerstore789.documents.azure.com:443/";
    string accessKey = "";

    var client = new DocumentClient(new Uri(databaseEndpoint), accessKey);

    string dbName = "mainpokerstorage";
    string collectionName = "gameevents";

    var collectionLink = UriFactory.CreateDocumentCollectionUri(dbName, collectionName);

    int maxEventsInSeries = 100;

    string[] actions = new string[] { "bet", "all-in", "call", "raise", "fold", "check" };

    for (int e = 0; e < maxEventsInSeries; e++)
    {
        string playerId = _rnd.Next(1, 5).ToString();
        string eventId = Guid.NewGuid().ToString();

        GameEvent gameAct = new GameEvent(playerId, eventId);

        gameAct.Action = actions[_rnd.Next(0, 5)];
        gameAct.ActionValue = _rnd.Next(0, 500);
        gameAct.GameId = 5;

        Document created = await client.CreateDocumentAsync(collectionLink, gameAct);
    }

}
```

Рис. 9.9. Код добавления элементов в базу данных DocumentDB

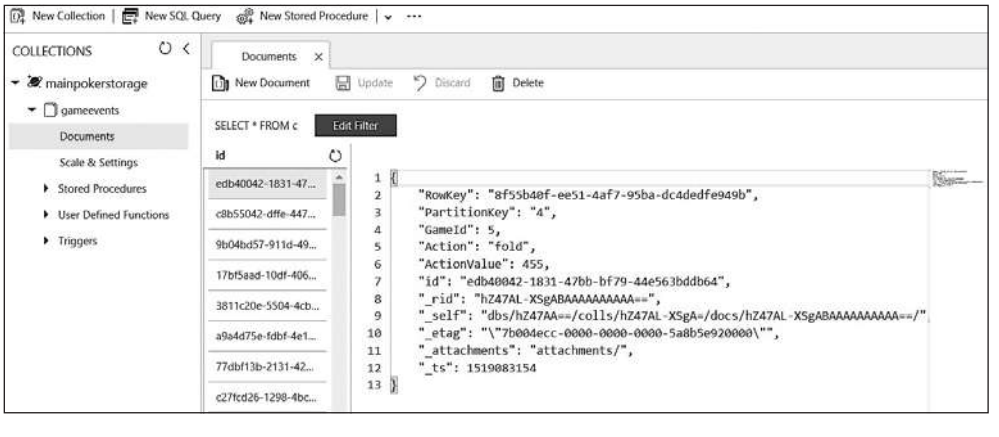


Рис. 9.10. Данные, добавленные в коллекцию

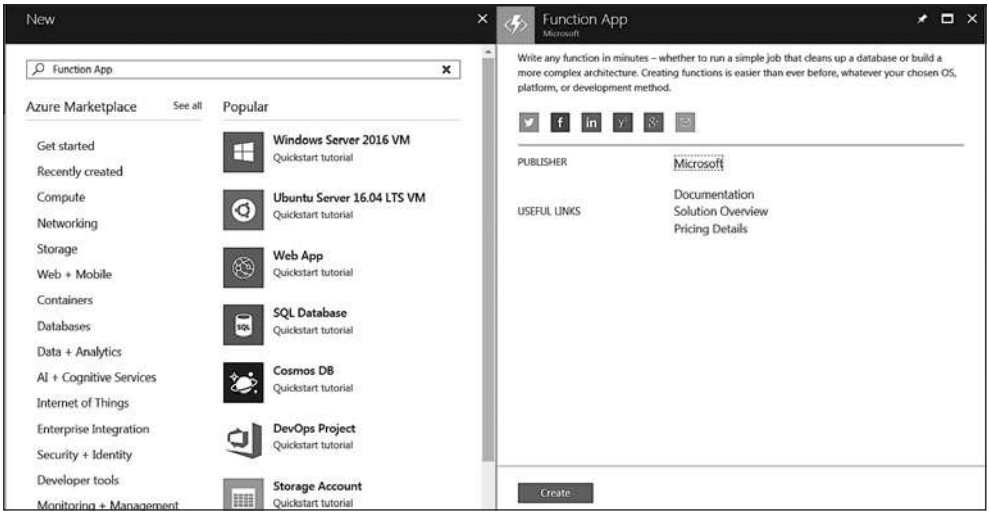


Рис. 9.11. Первый шаг создания Function App

В появившейся форме (рис. 9.12) следует указать параметр App name. Это будет URL сервиса App Service, лежащего в основе Function App и предоставляющего вычислительные возможности. Всего возможны два варианта хостинга функции и режима ее исполнения (Hosting Plan). Первый из них, выбранный нами (Consumption Plan) обеспечивает «истинное бессерверное» исполнение функции, когда плата взимается только за фактическое время работы функции и использование ею ресурсов. Но при этом возможна некоторая задержка запуска функции, которая будет уменьшена при втором варианте хостинга — на выделенном экземпляре App Service, но это уже «нечестная бессерверная» функция. Кроме того, в данной форме конфигурируется Application Insight — сервис мониторинга приложений от Microsoft (к сожалению, я не имею возможности описать его подробнее), который для данного примера отключен.

Общая панель созданного сервиса Function App выглядит следующим образом (рис. 9.13).

Далее к созданному экземпляру Function App следует добавить конкретную функцию, которая будет содержать исполняемый код, и подключить ее к коллекции DocumentDB. Это можно сделать напрямую с вкладки Add Azure Function CosmosDB (рис. 9.14).

Дадим нашей функции имя `gameeventsTrigger` (Name your Azure Function) и выберем язык C#. После успешного создания экземпляра сервиса код функции будет иметь следующий вид (рис. 9.15).

Указанный код содержит все необходимые для работы с CosmosDB пакеты и библиотеки, а также точку входа. Теперь протестируем всю цепочку. Для этого еще раз запустим клиент, генерирующий тестовые данные, и посмотрим на логи работы нашей функции (рис. 9.16).

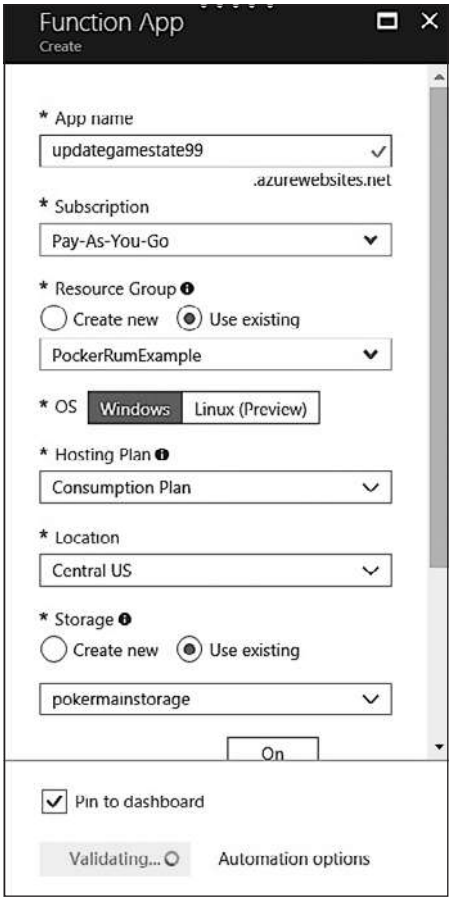


Рис. 9.12. Форма конфигурирования Function App

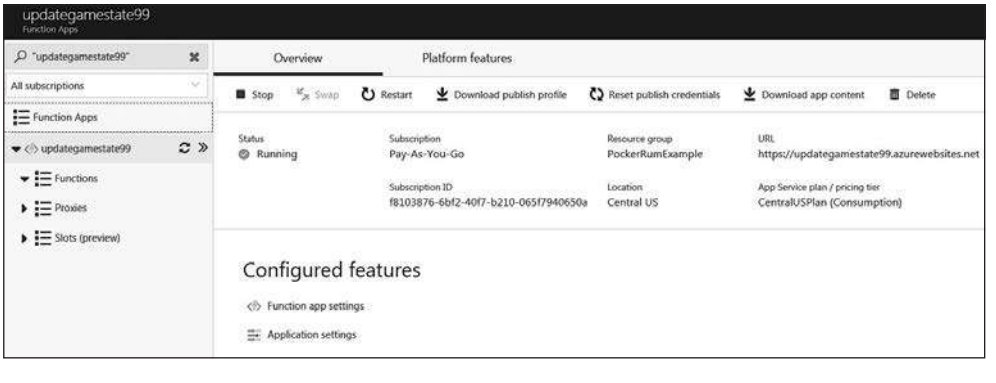


Рис. 9.13. Общая панель созданного сервиса Function App

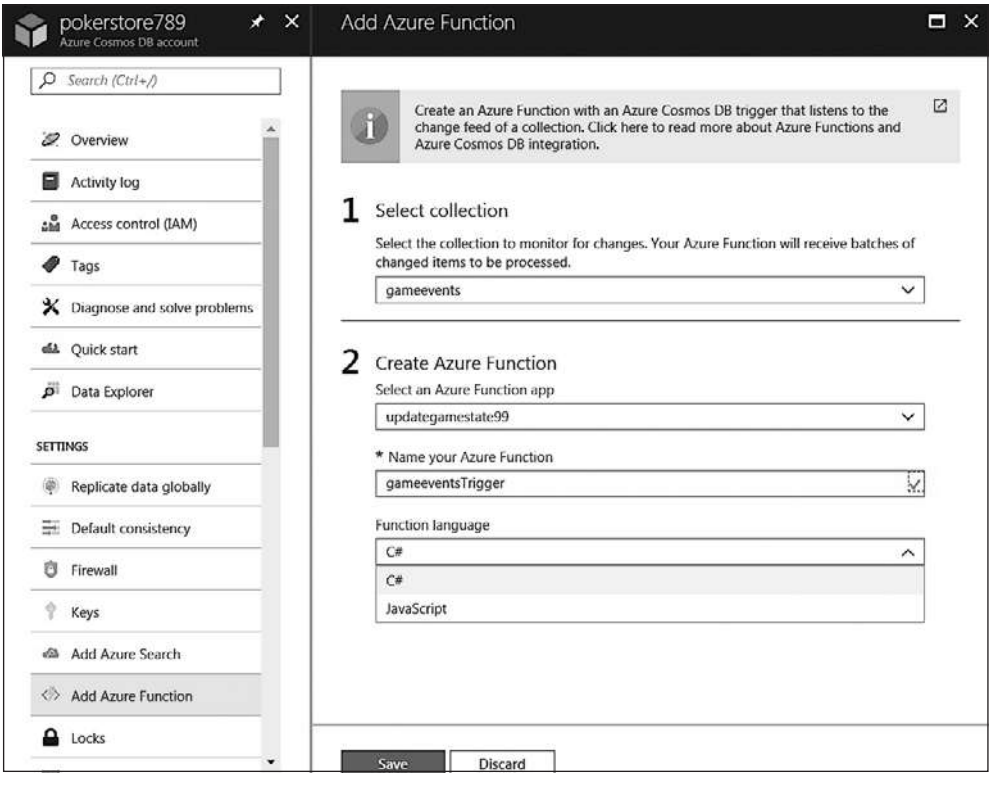


Рис. 9.14. Добавление новой функции Function App к коллекции CosmosDB

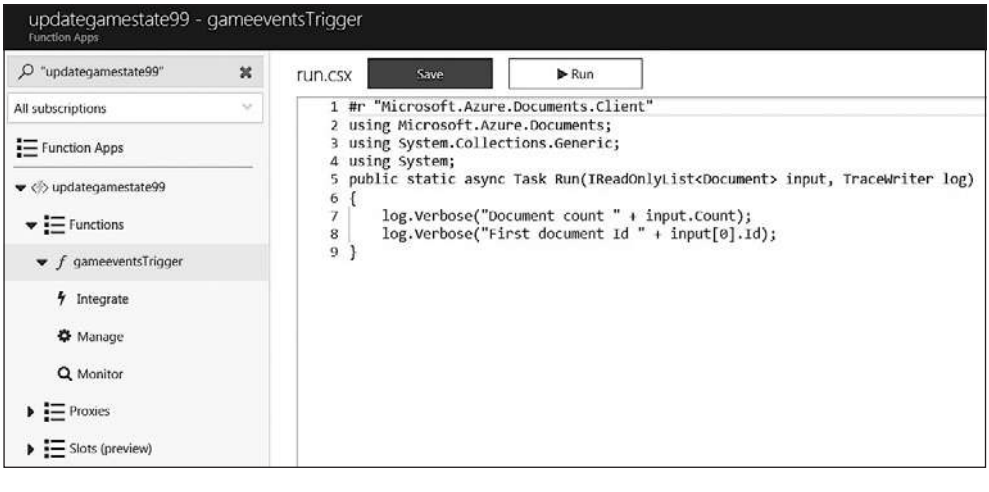


Рис. 9.15. Автоматически сгенерированный код функции

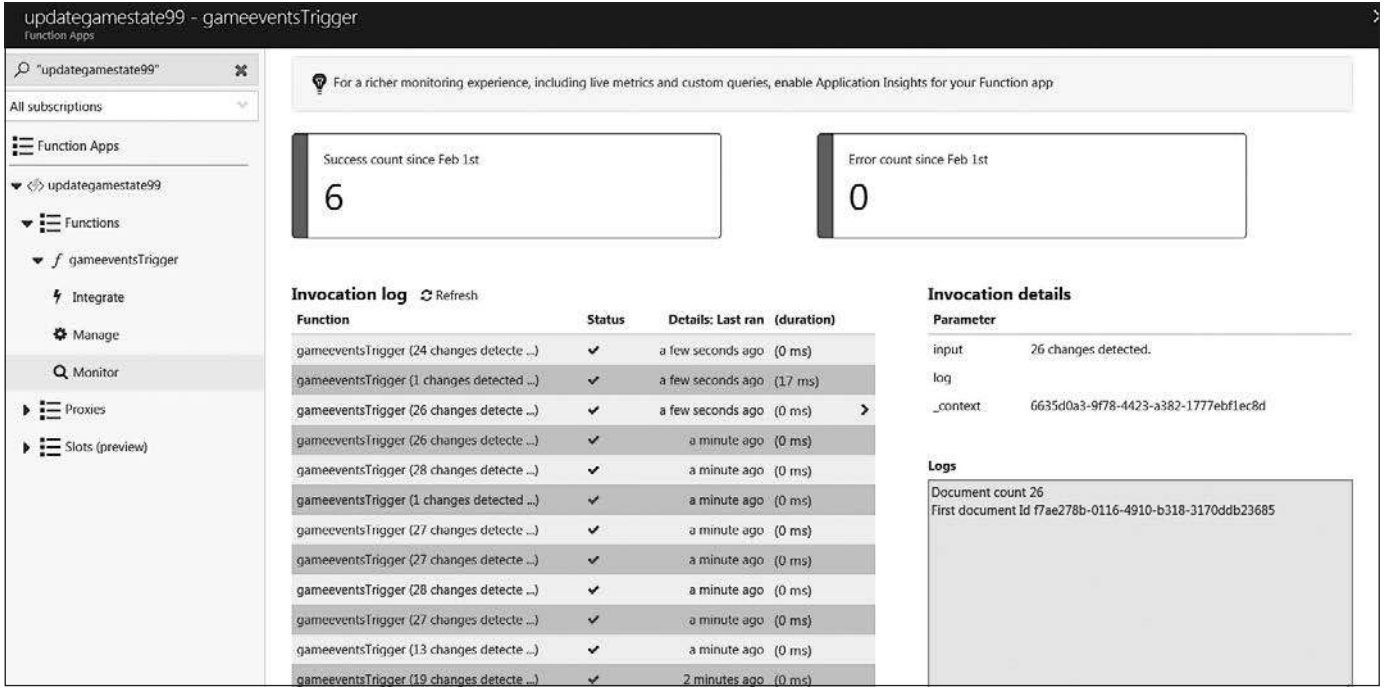


Рис. 9.16. Логи, показывающие работу функции из Function App в связке с триггером CosmosDB

Выбор языка программирования для бессерверных приложений, а также написание для них программ, развертывание и конфигурирование хостинга для них — это тема для отдельной главы. Здесь показан лишь один из вариантов создания функции.

Итак, я показал, как технически строить системы на основе концепции Event Driven с помощью нереляционной базы данных, бессерверной функции и механизма «поток изменения», который используется для ее запуска.

Теперь рассмотрим вопрос о доступе к ресурсам БД. Нам необходимо, чтобы клиент имел ключи с доступом на запись и на чтение, то есть, по сути, полный доступ ко всей базе данных. Потенциально это огромная дыра в безопасности, поскольку злоумышленник, похитив ключ, может иметь полный доступ к базе для чтения и записи. И самое неприятное — со скомпрометированным клиентом в общем-то ничего нельзя поделать, поскольку он компрометирует ключ для всего аккаунта, и ключ следует перегенерировать на уровне аккаунта и распределить среди клиентов, но при этом нельзя исключить факт новой компрометации. Одно из решений указанной проблемы — предоставление временных учетных данных для каждого клиента с помощью стороннего сервиса. Именно так работает сервис AWS Cognito — поставщик удостоверений (Identity Provider) от AWS, имеющий в своем составе подсервис Federated Identities, который позволяет для пользователя из пула пользователей AWS Cognito создавать временные учетные данные для доступа к DynamoDB. Если пользователь себя скомпрометировал, то он просто удаляется или блокируется в пуле пользователей с деактивацией временных ключей доступа. Клиент при этом должен иметь учетную запись в пуле пользователей AWS Cognito и выполнить туда вход.

Решение с использованием стороннего сервиса для разделения постоянных ключей доступа к ресурсам и временных ключей для пользователей позволяет очень гибко управлять правами доступа многих клиентов к данным в DynamoDB. Но при этом остается неявная проблема, связанная с тем, что клиент жестко привязан к типу хранилища, а это весьма ограничивает общую гибкость системы. Для решения данной проблемы существуют отдельные сервисы, называемые концентраторами сообщений и разделяющие процесс приема сообщений и их хранения/обработки. Они будут подробно рассмотрены в главе 10.

9.4. Доставка данных в HDFS-совместимые хранилища

Теперь рассмотрим способы доставки данных в сервисы HDFS-совместимых хранилищ. Помимо сервисов трансформации и копирования данных (таких как Azure Data Factory и AWS Glue), можно использовать следующие сервисы.

Прежде всего, это сервис Azure Import/Export Service, который может работать совместно с Azure Data Lake Store и Azure Blob Storage.

Простейший способ копирования данных — использование утилиты командной строки AdlCopy. Эта утилита предназначена для копирования данных из Azure Blob Storage в Azure Data Lake Store или между двумя Azure Data Lake Store (обратное копирование из Azure Data Lake Store в Azure Blob Storage с помощью AdlCopy невозможно). Синтаксис этой утилиты похож на AzCopy: необходимо указать URL источника и приемника. Далее следует вход в Active Directory, который и защищает Azure Data Lake Store. AdlCopy можно использовать в двух режимах: в виде отдельного приложения (standalone) (то есть на компьютере пользователя) и совместно с Azure Data Lake Analytics (этот сервис мы рассмотрим в части IV).

Для Azure Data Lake Store доступны также средства копирования из экосистемы Apache Hadoop, входящие в состав сервиса HDInsight: Sqoop, DistCp и др.

Кратко рассмотрим DistCp — Hadoop Distributed Copy (рис. 9.17). Эта технология может служить для копирования данных из Azure Blob Storage в Azure Data Lake Store, между различными аккаунтами Azure Data Lake Store, а также между различными папками в пределах Azure Data Lake Store. Кроме того, возможно копирование из другого источника, совместимого с HDFS. Собственно, в SSH-терминале HDInsight команда копирования выглядит как `"hadoop discp <source> <destination>"`.

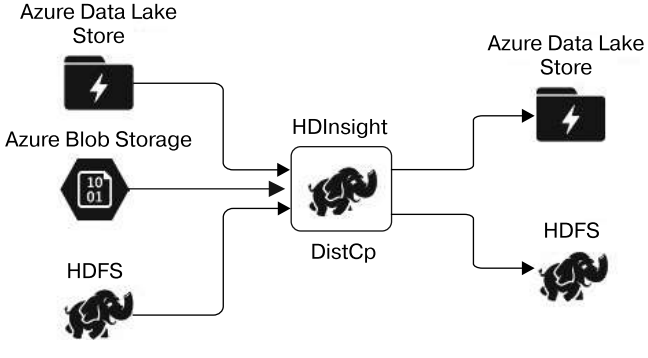


Рис. 9.17. Архитектура процесса копирования с помощью сервиса DistCp

Другой сервис — Apache Sqoop (рис. 9.18) — предназначен для копирования данных из реляционных баз данных и хранилищ данных (DWH) и Azure Data Lake Store и обратно. При этом в качестве источников могут выступать как сервисы РДБ Azure, так и внешние сервисы, поддерживающие JDBC. Данные будут сохраняться в формате CSV.

Более подробно с сервисами, разворачиваемыми в HDInsight, вы познакомитесь в части IV.

Кроме того, Azure Data Store поддерживает SDK, которая позволяет работать из .Net-приложения с помощью Data Lake SDK и получать доступ к файлам и папкам из кода.

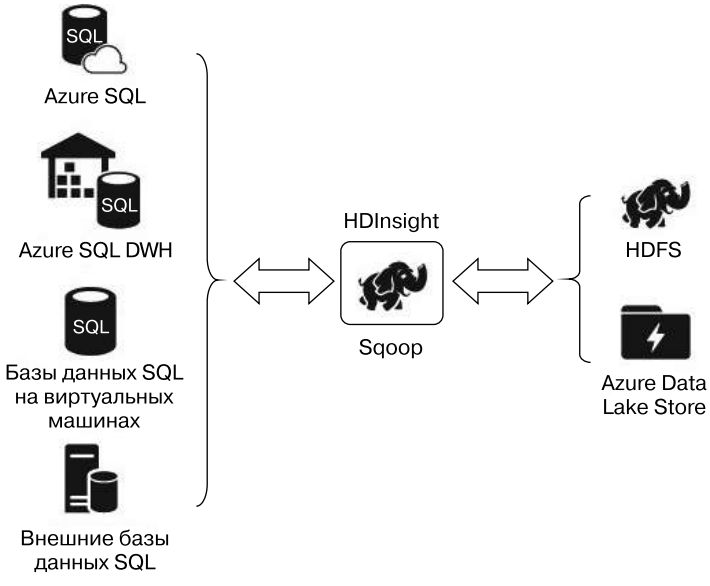


Рис. 9.18. Архитектура сервиса копирования и трансформации Apache Sqoop

Если рассматривать сервисы AWS, то я просто не могу обойти вниманием сервис AWS Snowball, предназначенный для импорта и экспорта большого объема данных из локального хранилища в AWS S3, перемещаемых с помощью физического носителя, минуя интернет-соединение. Что это значит? Идея, лежащая в основе AWS Snowball, очень проста: при наличии терабайтов или петабайтов данных самый быстрый способ переместить их из одной точки в другую — это записать на твердотельный носитель большой емкости, подключенный к физическому хранилищу, затем на машине доставить этот носитель в дата-центр AWS для обратной операции. Скорость передачи данных в подобной ситуации будет гораздо выше, чем у любого из существующих интернет-соединений. Размер физического носителя информации, применяемый для доставки ее в облако, и стоимость такой операции могут весьма различаться, вплоть до доставки петабайтов данных с помощью автомобиля-трейлера.

9.5. Резюме

Итак, мы рассмотрели различные способы доставки данных в облачные хранилища, не относящиеся к стандартным облачным сервисам трансформации и копирования данных. Это могут быть как специализированные сторонние приложения, размещаемые на виртуальных машинах или физических серверах, как относящиеся к экосистеме Hadoop (Apache DisCp и Apache Sqoop), так и отдельные (например, Microsoft SSIS или RedGate). Помимо этого, доступен ряд утилит командной

строки, каждая для своего типа хранилища: AzCopy для Azure Storage Account, AdlCopy для Azure Data Store. Существуют стандартные утилиты для копирования данных из внешних РБД в облачную; для MS SQL в качестве таковой может быть использована sqlcmd.

В общем случае задача доставки данных в облачное хранилище нетривиальна и требует тщательного анализа стратегии, поскольку сама процедура переноса может быть весьма затратной ввиду большого объема переносимых данных. Но это относится к переносу больших данных, находящихся во внешних хранилищах.

Несколько другая ситуация складывается в случае прямой доставки данных, генерируемых внешними приложениями. Здесь можно использовать программные SDK для работы с хранилищем из кода. При большом количестве клиентов, которые должны доставить данные, возникает ряд проблем, обусловленных жесткой привязкой клиентов к SDK конкретного хранилища, а также в связи с необходимостью их прямой авторизации (для предоставления им адекватных прав доступа к ресурсам) и негибкостью архитектуры. Последнее означает, что если клиентское приложение жестко привязано к SDK определенного хранилища, то перестроить архитектуру (допустим, в целях добавления сервиса потокового анализа) будет довольно сложно. Но вместе с тем нереляционные хранилища предоставляют весьма простую SDK и очень мощные сервисы интеграции с бессерверными приложениями, что позволяет на основе подхода Event Driven строить не просто хранилища, но целые системы, задействуя минимальное количество кода.

10 Прямая загрузка потоковых данных

В очередь, сукины дети, в очередь!

Михаил Булгаков. Собачье сердце

10.1. Общая архитектура

В предыдущей главе мы рассмотрели ситуацию, когда множество клиентских приложений должны посылать большое количество сообщений, которые нужно динамически обработать, поместить в хранилище и потом снова обработать уже в нем. При этом необходимо иметь возможность менять логику потока обработки и хранения данных, не прибегая к изменению кода клиентов. И наконец, с точки зрения соображений безопасности клиенты должны иметь право делать только одно — посылать сообщения или принимать их, но никоим образом не читать данные или удалять базы, а также у них не должно быть прямых прав для записи этих данных.

Подобные задачи очень распространены в системах, работающих с подключенными через интернет-соединение IoT-устройствами, а также в системах онлайн-анализа логов. Помимо перечисленных выше требований к нашему выделенному сервису, существует еще два требования, относящиеся к специфике «Интернета вещей» и к обеспечению надежной обработки сообщений. Прежде всего, протокол взаимодействия между клиентом и сервисом-приемником должен быть очень простым, чтобы его можно было реализовать на устройстве с ограниченными вычислительными возможностями и с весьма ограниченной памятью (например, платформы Arduino, Intel Edison, STM32 Discovery и другие «недотягивающие», такие как до RaspberryPi). Следующее требование — надежная доставка сообщений вне зависимости от возможных сбоев сервисов обработки. Это более сильное требование, чем требование высокой надежности. Действительно, для обеспечения общей надежности всей системы необходимо, чтобы надежность всех ее компонентов была достаточно высока и добавление нового компонента не приводило

к заметному росту количества отказов. Помимо отказа в облачной инфраструктуре, может возникнуть ошибка в сервисе, созданном пользователем. И даже тогда сообщение должно быть обработано, как только работоспособность пользовательского сервиса восстановится. Для этого сервис приема потока сообщений должен надежно хранить сообщение до тех пор, пока оно не будет обработано или пока не истечет время его жизни (это необходимо, чтобы предотвратить переполнение памяти при непрерывном потоке сообщений). Сервис, обладающий такими свойствами, называется *концентратором сообщений* (Event Hub). Для IoT-устройств существуют специализированные концентраторы (IoT Hub), которые обладают рядом других свойств, очень важных именно для применения совместно с устройствами «Интернета вещей» (например, двунаправленная коммуникация из одной точки, встроенная маршрутизация сообщений, «цифровые двойники» устройства и ряд других). Однако эти сервисы все-таки специализированные, и мы не будем рассматривать их подробно.

Прежде чем перейти к концепции концентрации сообщений, обратимся к идеям, лежащим в ее основе.

Предположим, у нас есть источник сообщений (например, поступающие от клиента запросы) и сервис, который должен их обрабатывать. Обработка отдельного запроса занимает время и требует затраты вычислительных ресурсов (CPU, памяти, IOPS). Причем во время обработки одного запроса не могут быть обработаны остальные запросы. Чтобы клиентские приложения не зависали в ожидании, пока освободится сервис, необходимо разделить их с помощью дополнительного сервиса, который будет ответственным за промежуточное хранение сообщений в то время, когда они ждут обработки, находясь в очереди. Такое разделение необходимо также для увеличения общей надежности системы. Действительно, клиент посылает сообщение в систему, но обрабатывающий сервис может «упасть», однако сообщение не должно быть потеряно, его нужно сохранить в сервисе, обладающем бо́льшей надежностью, чем обрабатывающий сервис. Простейший вариант такого сервиса так и называется — очередь (queue) (рис. 10.1).

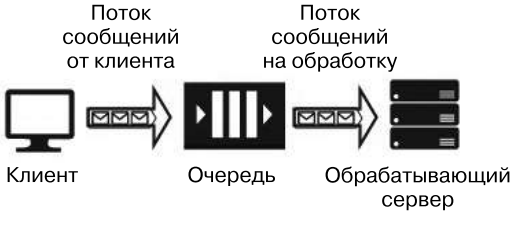


Рис. 10.1. Синхронизация клиента и обрабатывающего сервиса с помощью очереди

Сервис очередей работает следующим образом: клиент знает URL очереди и имеет ключи доступа к ней. Используя SDK или API очереди, клиент помещает в нее сообщение, содержащее в своем составе временну́ю метку, идентификатор и тело сообщения с полезной нагрузкой в JSON-, XML- или бинарном формате.

Программный код сервиса включает в себя цикл, который «прослушивает» очередь, извлекая очередное сообщение на каждом шаге, и если в очереди есть сообщение, то оно извлекается и обрабатывается. При успешной обработке сообщения сервисом оно удаляется из очереди. При возникновении ошибки во время обработки оно не удаляется и может быть обработано повторно, когда новая версия сервиса, с исправленным кодом, будет запущена. Очередь предназначена для синхронизации одного клиента (или группы однотипных клиентов) и ровно одного обрабатывающего сервиса (хотя последний может быть расположен на кластере серверов или на серверной ферме). К облачным сервисам очередей относятся Azure Storage Queue, Azure Service Bus Queue и AWS SQS. К сервисам, размещаемым на виртуальных машинах, можно отнести RabbitMQ, ZeroMQ, MSMQ, IBM MQ и др.

Различные сервисы очередей гарантируют различные виды доставки сообщений:

- как минимум однократную доставку сообщения;
- строго однократную доставку;
- доставку сообщений с сохранением порядка;
- доставку сообщений без сохранения порядка.

Очередь обеспечивает надежную доставку сообщений из одного источника в один обрабатывающий сервис, то есть взаимодействие «один к одному». Но как быть, если необходимо обеспечить доставку сообщений нескольким сервисам? В этом случае нужно использовать сервис под названием «топик» (topic) (рис. 10.2).

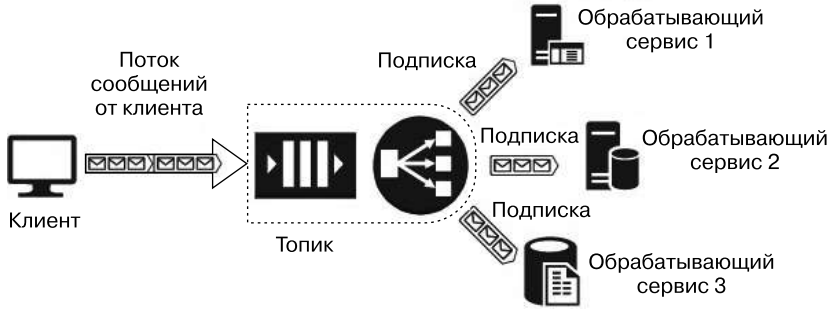


Рис. 10.2. Синхронизация клиента и группы обрабатывающих сервисов с помощью топика

Важный элемент такой архитектуры — «подписки». Это зарегистрированный в разделе путь, по которому направляется сообщение. Сообщения публикуются в топике клиентом и передаются на одну из подписок, из которой извлекаются одним из сервисов и обрабатываются им. Топики обеспечивают архитектуру взаимодействия клиента и сервисов как «один ко многим». В качестве примеров таких сервисов можно привести Azure Service Bus Topic и AWS SNS.

А теперь предположим, что есть большое количество разнородных клиентов, которые должны посылать много сообщений различным сервисам, то есть нужно построить систему взаимодействия «многие ко многим». Конечно, подобную архитектуру можно построить с помощью нескольких разделов, но такое построение немасштабируемо и требует усилий для администрирования и мониторинга. Однако существуют отдельные сервисы — концентраторы сообщений (рис. 10.3).

Рис. 10.3. Синхронизация групп клиентов и обрабатывающих сервисов

Концентратор принимает сообщения от многих клиентов. Все клиенты могут посылать сообщения в одну общую конечную точку сервиса или подключаться раздельно к различным конечным точкам через специальные ключи. Эти ключи позволяют гибко управлять клиентами: отключать некоторые, подключать новые и др. Внутри концентратора тоже есть разделы (partitions). Но в данном случае они могут быть распределены между всеми клиентами в целях повышения производительности (round robin — «с циклическим добавлением») или клиент может публиковать сообщения в один из разделов. С другой стороны, обрабатывающие

сервисы объединены в группы потребителей (consumer group). К одной группе могут быть подключены один или несколько сервисов. Таким образом, концентратор сообщений — наиболее гибкий сервис, который можно сконфигурировать как очередь, раздел или группы очередей либо набор разделов. В общем виде концентратор сообщений обеспечивает схему «многие ко многим» между клиентами и сервисами. К таким концентраторам можно отнести Apache Kafka, Azure Event Hub и AWS Kinesis Stream.

Прежде чем рассмотреть облачные PaaS-сервисы, обратим внимание на очень мощный и известный сервис — Apache Kafka. В облачных средах он может быть доступен в виде дистрибутива, развернутого в кластере виртуальных машин напрямую или с помощью сервиса HDInsight. Итак, Apache Kafka представляет собой сервис, обеспечивающий следующие возможности:

- публикацию и подписку на поток сообщений;
- надежное хранение сообщений;
- применение сторонних сервисов потоковой обработки сообщений.

Физически Kafka запускается в кластере из одного или нескольких серверов. Kafka предоставляет API для взаимодействия с внешними клиентами (рис. 10.4).

Рис. 10.4. Логическая структура Apache Kafka

Рассмотрим эти API по порядку.

- *API производителя* позволяют клиентским приложениям публиковать потоки сообщений в одном или нескольких топиках Kafka.

- *API потребителя* дают клиентским приложениям возможность подписаться на один или несколько топиков и обработать потоки сообщений, доставленных топиками клиентам.
- *API потоковых процессоров* позволяют приложениям взаимодействовать с кластером Kafka в качестве процессора потоковой обработки данных. Источниками для одного процессора может быть один или несколько топиков. При этом обработанные сообщения также помещаются в один или несколько топиков.
- *API коннекторов* помогают подключать внешние источники данных (например, РБД) в качестве источников сообщений (так, возможен перехват событий изменения данных в базе) и в качестве приемников.

В Kafka взаимодействие между клиентами и кластером происходит по протоколу TCP, чему способствуют имеющиеся SDK для различных языков программирования, в том числе и .Net. Но базовые языки SDK — Java и Scala.

В кластере хранение потоков сообщений (в терминологии Kafka именуемых еще *записями*) происходит логически в объектах, называемых *топиками* (рис. 10.5). Каждая запись состоит из ключа, значения и временной метки. По сути, топик — это последовательность записей (сообщений), которые были опубликованы клиентами. Топики Kafka поддерживают от 0 до нескольких подписчиков. Каждый топик физически представлен в виде разделяемого журнала (partitioned log). Каждый раздел есть упорядоченная последовательность записей, к которой постоянно добавляются новые, поступающие на вход Kafka.

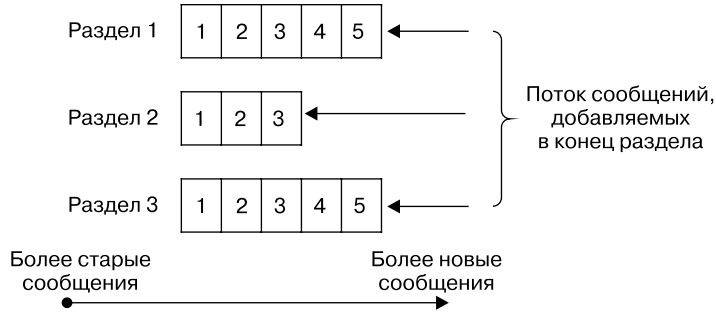


Рис. 10.5. Физическое представление топиков

Каждой записи в разделе соответствует номер в последовательности, называемый еще смещением, которое однозначно определяет данное сообщение в последовательности. В отличие от очереди Kafka удаляет сообщение не после обработки сервиса, а по истечении времени жизни сообщений. Это очень важное свойство, обеспечивающее возможность читать из одного топика разным потребителям. Притом с каждым потребителем ассоциировано смещение (рис. 10.6). И каждый акт чтения приводит лишь к увеличению значения для каждого клиента в отдельности и определяется именно клиентом.

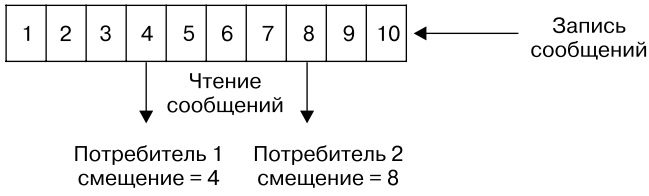


Рис. 10.6. Чтение сообщений несколькими клиентами

В нормальном случае это смещение увеличивается на единицу после успешного чтения одного сообщения из топика. Но при необходимости клиент может сдвинуть данное смещение и повторить операцию чтения.

Использование концепции разделов преследует следующие цели.

Во-первых, разделы обеспечивают возможность масштабировать топики в том случае, когда один топик не помещается в пределах одного узла. При этом каждый раздел имеет один ведущий (не путайте его с головным узлом всего кластера) узел и ноль или несколько узлов-последователей. За обработку операций чтения-записи отвечает головной узел, в то время как последователи являются его пассивными копиями. При отказе ведущего узла один из узлов-последователей автоматически станет головным. Каждый узел кластера — головной для некоторых разделов и последователь для других. Во-вторых, такая репликация увеличивает производительность чтения за счет возможности параллельных операций чтения.

Производитель может помещать сообщение в любой топик по его выбору явно или в режиме round robin неявно (то есть с равномерным заполнением). Потребители объединены в так называемые группы потребителей, и каждое сообщение, публикуемое в топике, доставляется одному клиенту в каждой группе потребителей. Клиенты в данном случае могут физически размещаться на одном или нескольких серверах/виртуальных машинах. Более детально доставка сообщений выглядит следующим образом. Для всех клиентов, относящихся к одной группе потребителей, сообщения могут быть распределены между клиентами с целью оптимизировать нагрузку. Если клиенты относятся к разным группам потребителей, то каждое сообщение будет разослано в каждую группу. Разделение сообщений из разделов по разным группам потребителей показано на рис. 10.7.

Теперь кратко опишу основные параметры доставки и хранения сообщений, гарантированные Kafka.

- Сообщения, посылаемые производителем в конкретный топик, будут добавлены строго в том порядке, в каком они были отосланы.
- Клиент видит тот порядок сообщений в топике, который был получен при сохранении сообщений. В итоге доставка сообщений от производителя к потребителю производится строго в порядке их поступления.
- N-кратная репликация топика обеспечивает устойчивость топика к отказу N – 1 узлов без потери работоспособности.

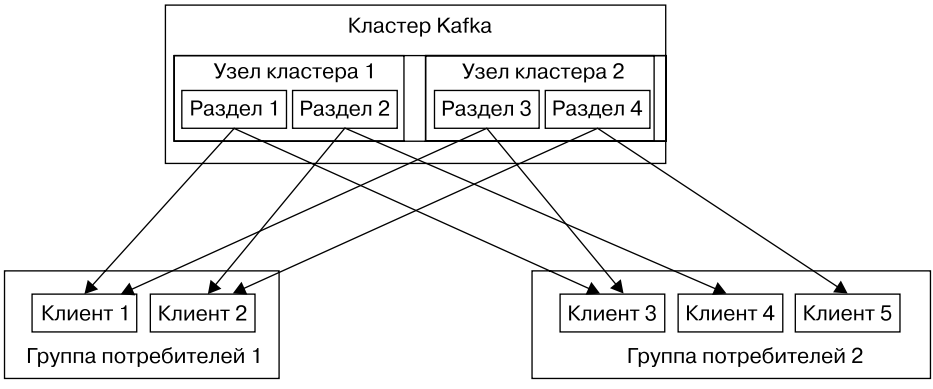


Рис. 10.7. Архитектура групп потребителей Kafka

Итак, сервис Apache Kafka может быть использован в следующих режимах.

- Сервис — брокер сообщений (очередь) или сервис публикации — подписки сообщений (топик). Действительно, в основе Kafka лежит группа топиков, которую можно преобразовать в очередь при наличии одного подписчика. (Следует помнить: в отличие от обычных сервисов брокеров сообщений, построенных по принципу очередей, в Kafka удаление сообщений происходит только по истечении его времени жизни, в то время как в брокерах реализован принцип Peek-Delete, то есть извлечения и удаления после успешной обработки.) Принцип групп потребителей обобщает две эти концепции, а возможность публикации сообщений во всех топиках с распределением round robin делает Kafka универсальным многорежимным брокером сообщений.
- Сервис потокового анализа сообщений. Это возможно благодаря включенным в Kafka API для потоковых процессоров, что позволяет строить сложные системы, созданные по принципу Event Driven, с сервисами, фильтрующими сообщения или реагирующими на них, а также с сервисами, агрегирующими сообщения.

Все указанные свойства позволяют использовать Kafka как ключевой компонент платформы, работающей с потоковыми данными и обладающей широкими возможностями для построения сложных систем обработки потоков сообщений. Но вместе с тем Kafka достаточно сложен в плане развертывания и настройки кластера из нескольких узлов, что требует существенных усилий административного плана. Но, с другой стороны, поскольку идеи, лежащие в основе Kafka, очень хорошо подходят для построения систем, потокового приема и обработки сообщений, то облачные провайдеры предоставляют PaaS-сервисы, реализующие эти идеи и скрывающие все сложности построения и администрирования кластера Kafka. Но так как указанные сервисы имеют ряд ограничений в плане кастомизации и расширения за пределами выделенных для сервисов лимитов, облачные провайдеры

предоставляют специальные IaaS/PaaS-сервисы для физического развертывания Kafka в кластере виртуальных машин. В этом случае пользователь обладает практически полной свободой конфигурирования и расширения. К таким сервисам относится Azure HDInsight. Он уже упоминался выше. Он создан для того, чтобы, с одной стороны, предоставить пользователю сервисы из экосистемы Hadoop сами по себе, без внешних оберток, а с другой — избавить от сложностей, возникающих при прямом инсталлировании, администрировании и конфигурировании IaaS. Несколько особняком стоит Docker-хостинг. Поскольку это чрезвычайно важная тема, то мы рассмотрим ее, но сначала познакомимся с PaaS-сервисами, реализованными с помощью основных концепций Kafka.

10.2. Azure Event Hub

Рассмотрим сервис концентратора сообщений Azure Event Hub. Он представляет собой сервис, построенный по модели PaaS. В качестве источников сообщений для Azure Event Hub могут выступать различные группы клиентов (рис. 10.8). Прежде всего, это очень большая группа облачных сервисов, чьи выходы или триггеры можно сконфигурировать для посылки сообщений напрямую в Event Hub. Это могут быть Stream Analytics Job, Event Grid и значительная группа сервисов, перенаправляющих события — логи в Event Hub (прежде всего, построенные с помощью AppService: Api App, Web App, Mobile App и Function App).

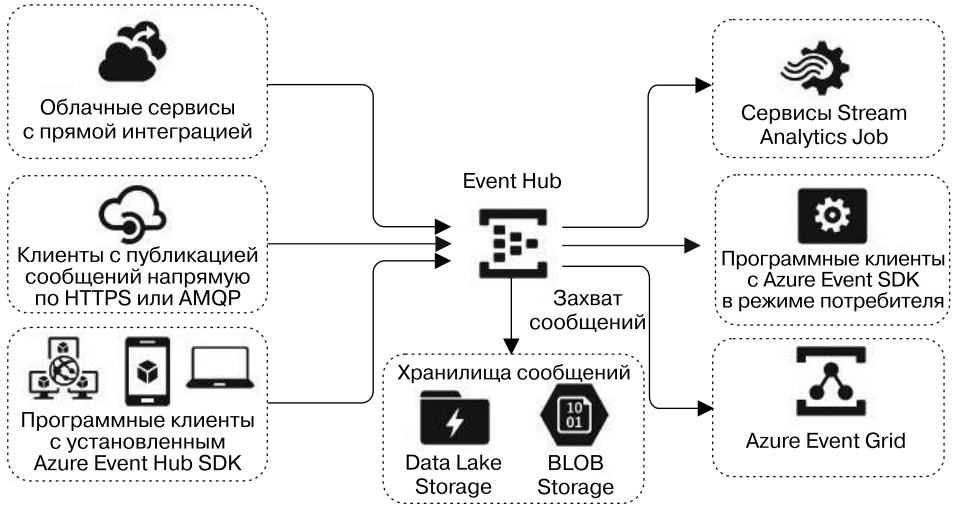


Рис. 10.8. Общая структура взаимодействия Event Hub со сторонними сервисами

Доставленные в концентратор сообщения могут быть напрямую захвачены (capture) и помещены в хранилище Blob Storage или Data Lake Store.

Следующая группа источников — внешние программные клиенты или устройства, для которых отсутствует Azure Event Hub SDK и которые не могут быть напрямую интегрированы с сервисами Azure. К таким клиентам относятся прежде всего устройства IoT. Они могут посылать сообщения во вход Event Hub с помощью протоколов HTTPS или AMQP. Рассмотрение способов подключения данных устройств выходит за рамки нашей книги.

И наконец, программные клиенты, которые генерируют сообщения и посылают их в Event Hub, задействуя Azure Event Hub SDK. Эта группа включает в себя Azure PowerShell и Azure CLI.

В качестве приемников сообщений (consumers — «потребители») из Event Hub могут выступать сервисы потоковой аналитики Stream Analytics Job или сервис интеграции Event Grid. Кроме того, возможен прием сообщений программными клиентами с помощью Azure Event Hub SDK. Потребители подключаются к Event Hub, используя протокол AMQP 1.0.

Рассмотрим основные концепции Azure Event Hub, необходимые для понимания путей его использования и конфигурирования. Любой источник (в документации называется еще издателем — publisher), который посылает сообщение в концентратор, должен применить протокол HTTPS или AMQP 1.0. Выбор того или иного протокола определяется типом клиента, сети коммуникации и требованиями к скорости передачи сообщений. Протоколу AMQP необходимо создание постоянного соединения между двумя двунаправленными TCP-сокетами. Оно защищается путем использования протокола шифрования транспортного уровня TLS или SSL/TLS. Все это означает, в частности, что для первоначального установления соединения протокол AMQP требует бо́льшего времени, чем HTTPS, но при постоянном потоке сообщений у первого производительность гораздо выше. При редких сообщениях более предпочтителен HTTPS.

Для того чтобы идентифицировать себя, источники могут использовать механизмы идентификации на основе механизма SAS (Shared Access Signature) tokens. В простейшем случае все издатели могут применить один общий SAS-токен на всех или задействовать гибкую политику распределения различных токенов SAS для разных издателей. Кроме разделения источников по различным SAS-токенам, используется разделение по ключу раздела (об этом ниже).

Каждый источник может отправлять сообщение размером не более 256 Кбайт. Это значит, что нужно разбивать длинное сообщение на несколько посылаемых последовательно или переходить к бинарному формату и использовать сжатие данных.

Теперь рассмотрим, как сообщение обрабатывается непосредственно в Event Hub. Как уже указывалось в предыдущей главе, в основе концепции концентраторов сообщений лежит группа топиков, имеющих точки входа и множественные выходы, к которым могут подключаться подписчики-потребители. В случае EventHub подобные топики именуются *разделами* (partitions). Концептуально разделы EventHub — это упорядоченная последовательность сообщений, организованная по принципу очереди «первым вошел — первым вышел» (FIFO) (рис. 10.9).

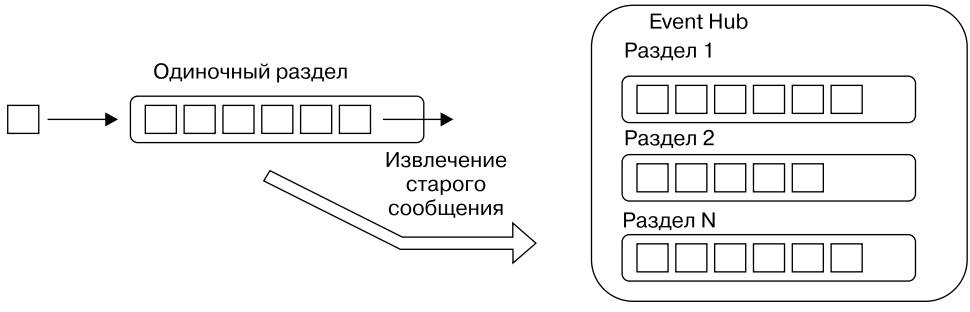


Рис. 10.9. Концепция разделов Event Hub

Каждый раздел — независимая последовательность сообщений в пределах всего Event Hub. Один Event Hub может включать от 2 до 32 разделов, и эта величина не может быть изменена после создания Event Hub. Очень важно понимать, что количество ОДНОВРЕМЕННО извлекаемых сообщений из концентратора равно количеству разделов.

Сообщения в разделе (а по сути в очереди) хранится до тех пор, пока его не извлечет потребитель (оно при этом не удаляется, а перестает быть доступным — см. ниже), или до истечения определенного времени хранения (retention period), которое может настраиваться. Здесь следует уточнить. В любом случае сообщение физически удаляется только по истечении времени хранения. Чтобы обеспечить возможность читать сообщения из раздела, в Azure Event Hub реализована концепция *смещения* (offset). Смещение в данном случае — это позиция в разделе, которая показывает текущую позицию для чтения, то есть, по сути, курсор с номером текущей читаемой позиции. После успешного извлечения сообщения потребитель сдвигает курсор на одну позицию. Azure Event Hub SDK позволяет установить произвольную позицию, с которой можно читать сообщения, но так поступать не рекомендуется. При этом за хранение смещения ответственен клиент-потребитель, и он сам должен хранить и обновлять данную величину.

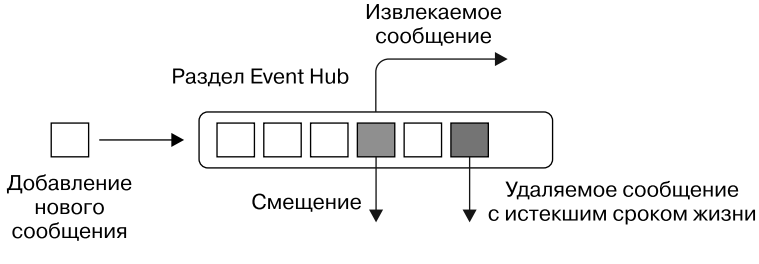


Рис. 10.10. Чтение и добавление сообщений в раздел

Таким образом, потребитель *может* прочитать одно и то же сообщение несколько раз, но только в том случае, если он каждый раз будет указывать смеще-

ние, соответствующее позиции этого сообщения. Однако стандартные Azure Event Hub SDK по умолчанию *исключают* такую возможность, поскольку обеспечивают механизм надежного хранения и обновления смещения. Как правило, для хранения смещения используется Storage Account. Сервисы Azure, являющиеся приемниками сообщений из Event Hub, берут на себя ответственность за хранение сообщений.

Каждый раздел в рамках Event Hub имеет собственный *ключ раздела* (partition key), который позволяет направить сообщения из конкретных источников в конкретный раздел. Цель такого действия — организация потоков сообщений. Например, концентратор содержит много разделов и источников разной производительности (количества сообщений в единицу времени) и требуется создать выделенный канал с публикацией и чтением сообщений. Если у источника явно не указан ключ раздела, то они распределяются между разделами равномерно (round robin).

Теперь рассмотрим вопрос извлечения сообщений из концентратора. Все потребители логически объединяются в группы, управляемые совместно и называемые *группами потребителей* (consumer group) (рис. 10.11). Группы позволяют организовать независимое чтение из одних и тех же разделов разными потребителями. При этом каждая группа дает каждому клиентскому приложению возможность иметь свое *представление* (view) (величины текущих смещений для каждого раздела) и, таким образом, независимо читать данные из разделов. Для каждой группы потребителей подобные представления полностью различны, что позволяет реализовать сложные сценарии обработки данных. Максимальное количество групп потребителей — 20, при этом в каждой группе может быть не более пяти собственно потребителей, а одновременно считывать сообщение из раздела может только один потребитель из группы.

Разделение концентратора на разделы обеспечивает параллельное чтение сообщений множеством потребителей. Кроме того, производительность концентратора с точки зрения приема заданного количества в единицу времени может быть настраиваемой. Для этого существует параметр, называемый *единицей пропускной способности* (throughput unit). Каждая единица пропускной способности включает в себя следующие величины:

- для входного потока — 1 Мбайт в секунду или 1000 сообщений в секунду (в зависимости от того, какой из этих лимитов будет достигнут первым);
- для выходного потока — 2 Мбайт в секунду.

Конкретная величина пропускной способности настраивается для каждого концентратора отдельно. Если источники превышают допустимую пропускную способность, то у них генерируется исключение. При этом приемник просто не может принимать сообщения быстрее заданной величины. Единицы пропускной способности также влияют на стоимость использования сервиса. Будьте внимательны! Следует знать, что существуют лимиты пропускной способности, выделяемые на пространство имен, а не на отдельные экземпляры сервиса Event Hub.

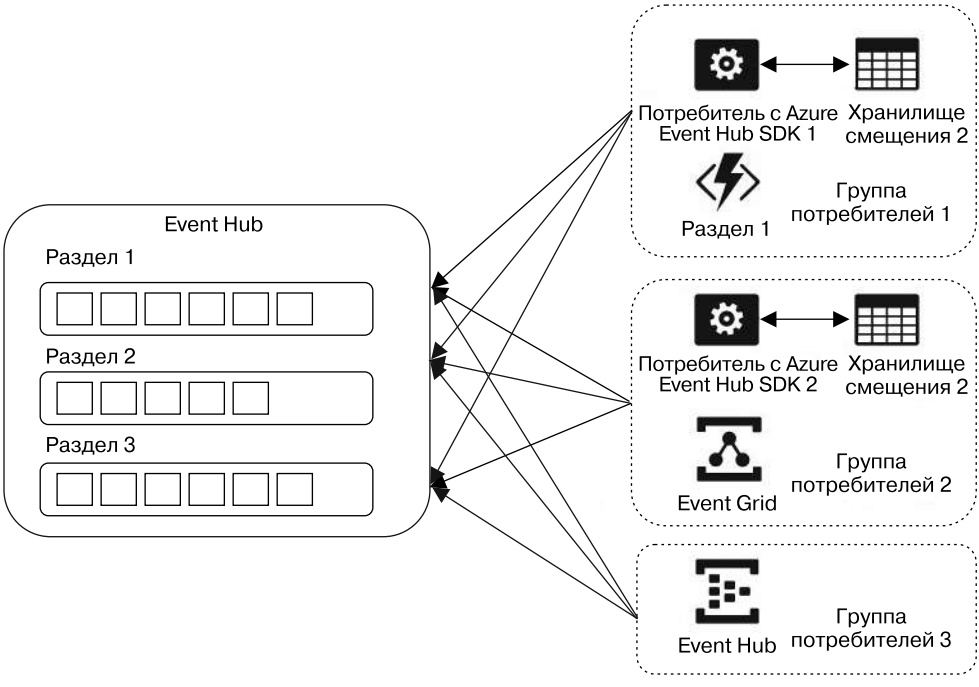


Рис. 10.11. Концепция групп потребителей

Концентраторы сообщений логически группируются в *пространства имен* (namespace) (рис. 10.12).

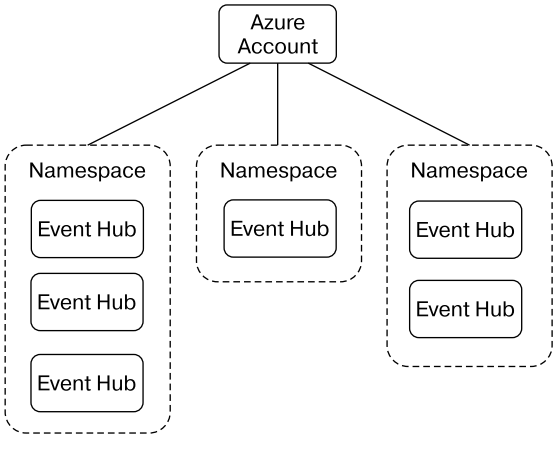


Рис. 10.12. Логическая структура сервиса Azure Event Hub

Теперь рассмотрим, как создавать и использовать Azure Event Hub. Воспользуемся порталом Azure. Прежде всего необходимо выбрать ссылку + New (Добавить) и в строке поиска набрать `Event Hubs` (рис. 10.13).

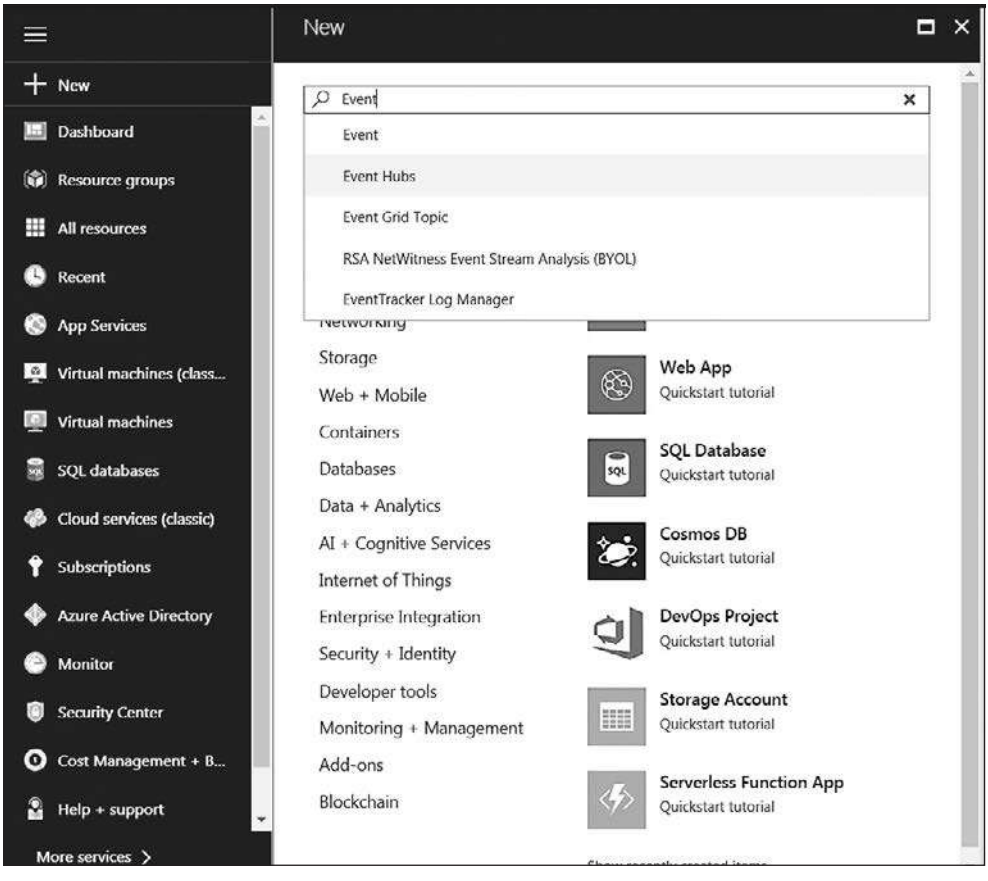


Рис. 10.13. Поиск сервиса Event Hub из списка доступных

После выбора Event Hubs и нажатия кнопки + New откроется информационная страница сервиса (рис. 10.14).

Далее следует заполнить первоначальную форму настройки Event Hub (рис. 10.15).

Прежде всего создаем пространство имен. В данном примере это `mainGameConcentrator`. Далее выбираем ценовой уровень (Pricing tier) — в настоящее время доступно два ценовых уровня: Basic и Standard. Они различаются количеством групп потребителей (consumer groups), временем хранения сообщений (retention period) и наличием/отсутствием специальных правил для источников сообщений (publisher policies). Для целей нашего примера следует выбрать уровень Standard,

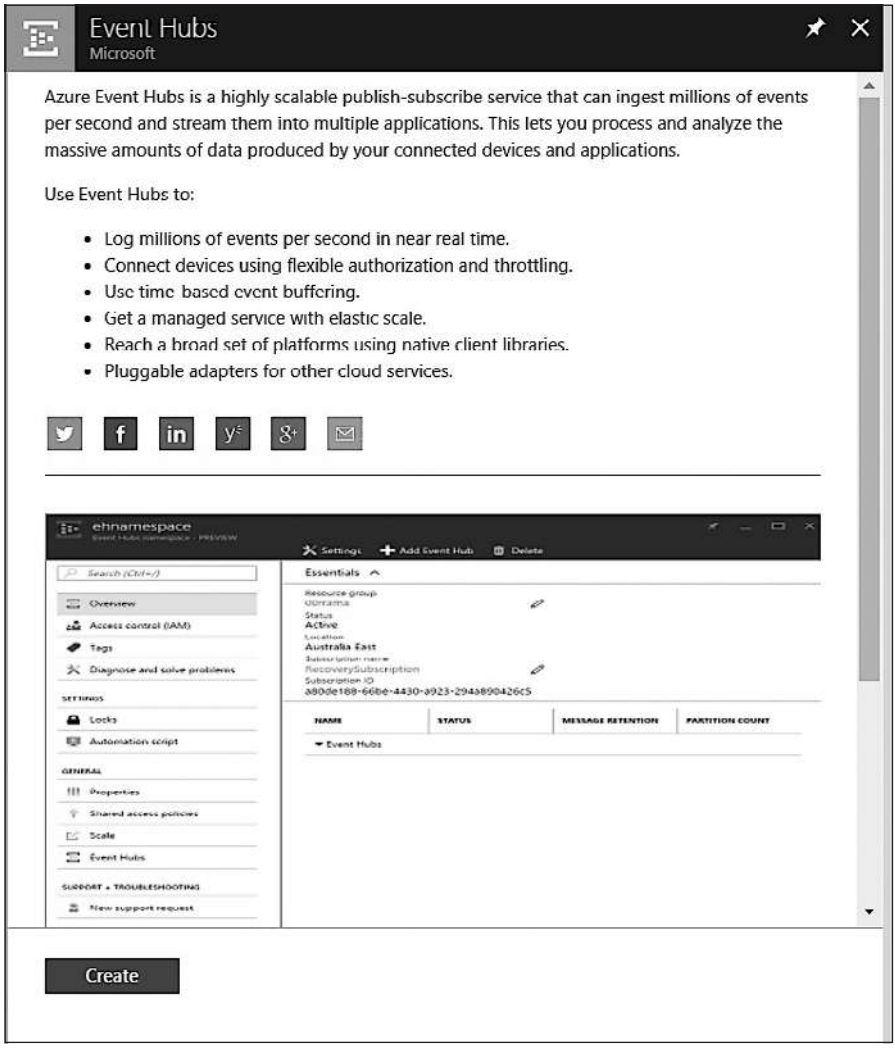


Рис. 10.14. Информационная страница сервиса Azure Event Hub

существующую группу ресурсов PockerRumExample, местоположение — Central US и одну единицу пропускной способности. Для целей примера снимите флажок в Enable auto-inflate. Эта опция позволяет автоматически увеличивать количество единиц пропускной способности в ответ на возросшую нагрузку. Верхний предел для увеличения определяется в поле Specify Upper Limit.

После создания пространства имен станет доступна страница, показанная на рис. 10.16. На ней прежде всего отображаются главные метрики для заданных временны́х интервалов.

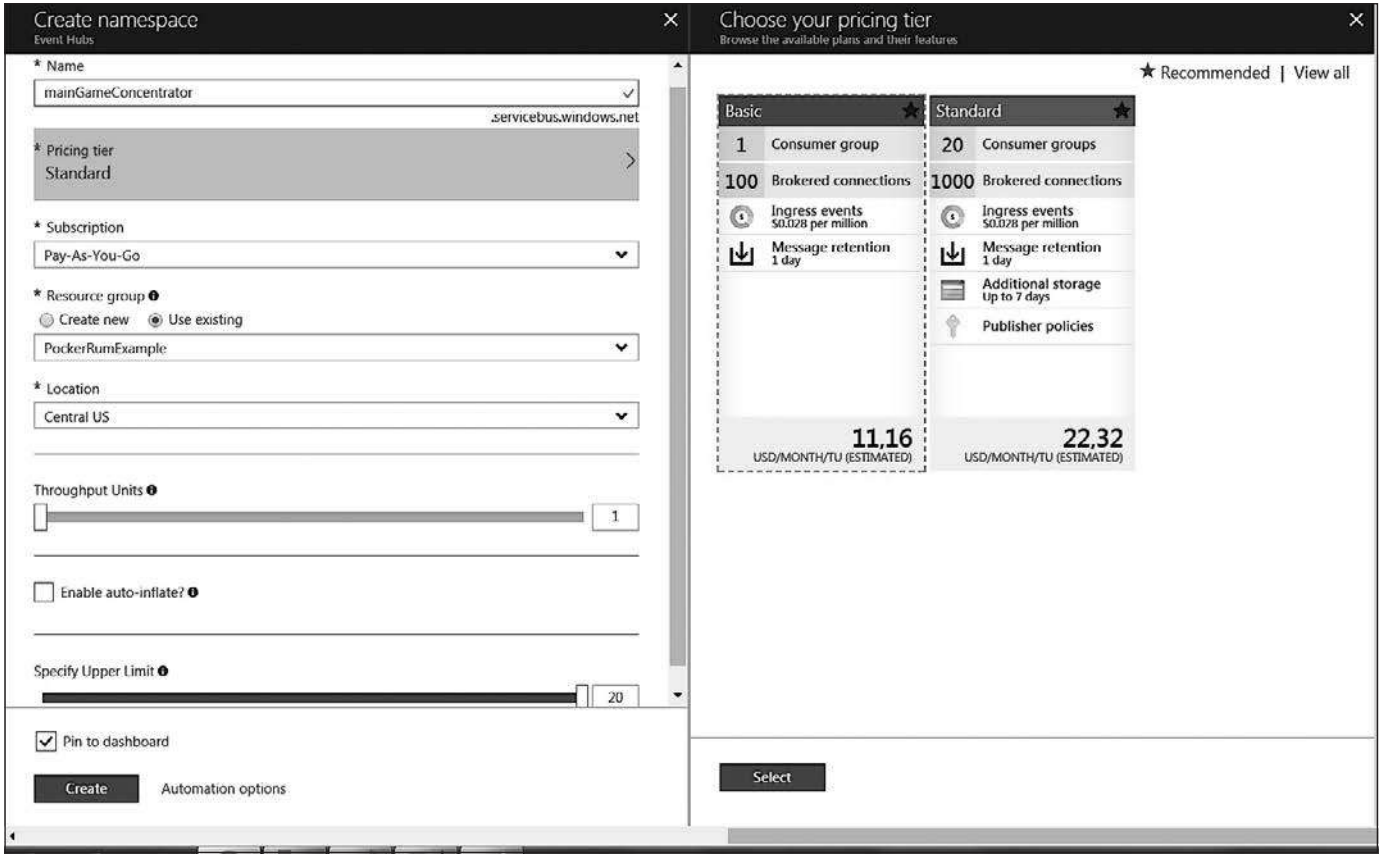


Рис. 10.15. Форма настройки Event Hub

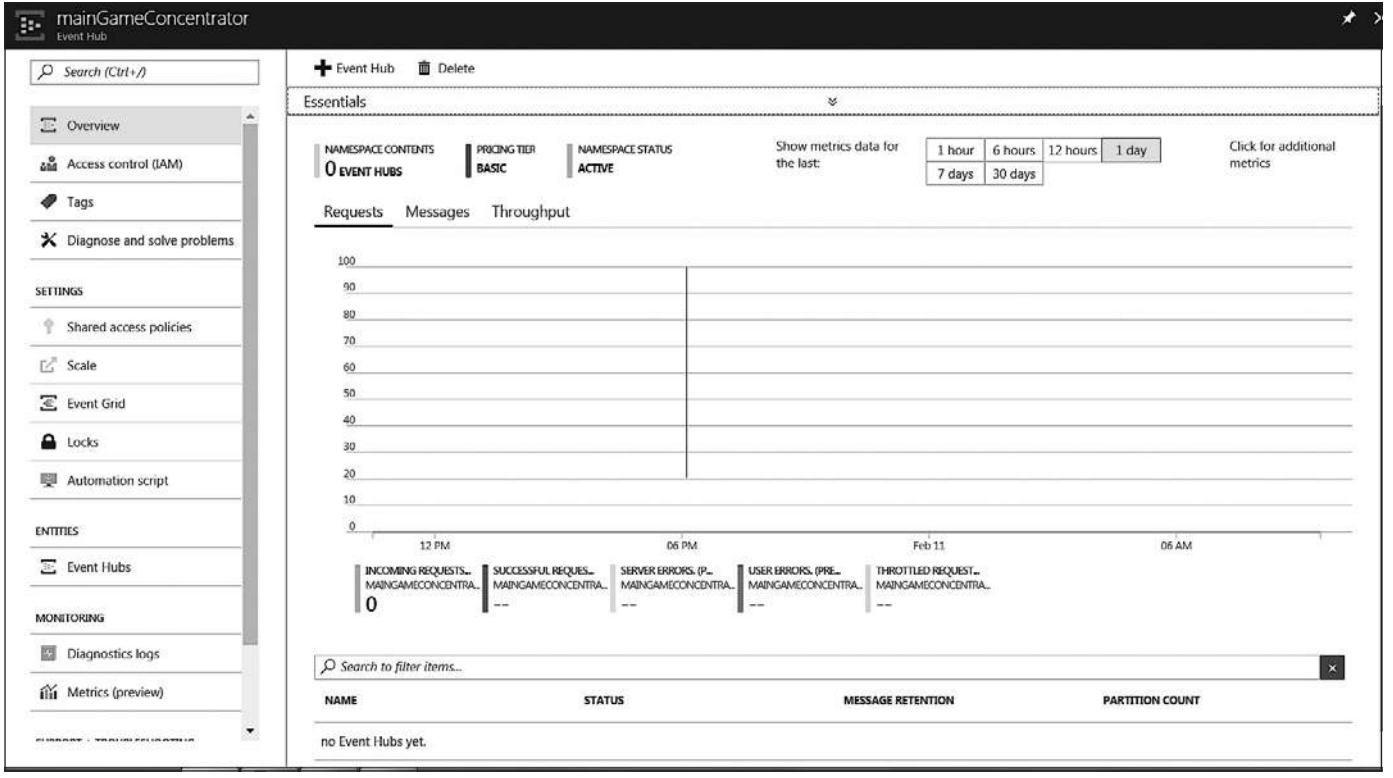


Рис. 10.16. Заглавная страница пространства имен Event Hub

Следующее, что нужно сделать, — это создать экземпляр Event Hub. Для этого необходимо нажать ссылку + Event Hub. На рис. 10.17 представлена форма конфигурирования создаваемого Event Hub.

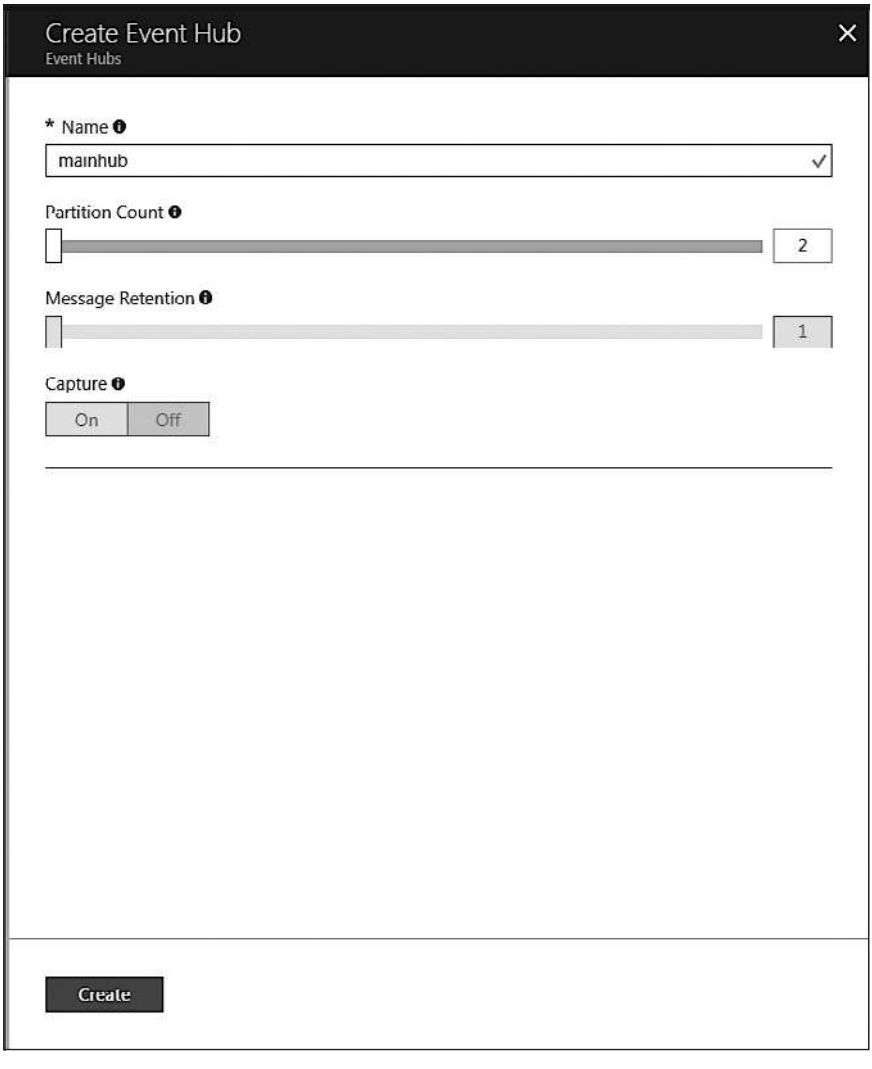


Рис. 10.17. Форма конфигурирования создаваемого Event Hub

Я дал имя `mainhub` вновь создаваемому концентратору. Количество разделов оставим минимальным — 2. Помните, что это неизменяемая величина: если в последующем понадобится добавить количество разделов, то придется пересоздавать концентратор. Поскольку мы выбрали ценовой уровень Basic, то настройка

времени жизни сообщений (retention period) и включение захвата сообщений (capture) недоступны. На рис. 10.18 можно увидеть добавленный в список концентратор.

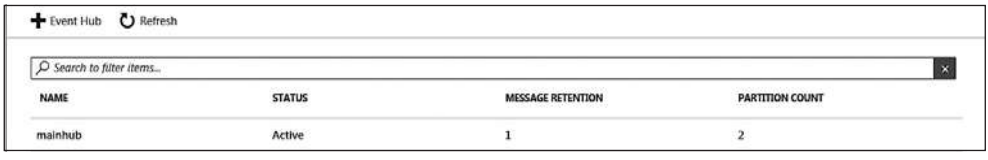


Рис. 10.18. Созданный концентратор, добавленный в список

Далее нужно сконфигурировать настройки политики совместного доступа. Для этого следует нажать ссылку Shared access policies (рис. 10.19).

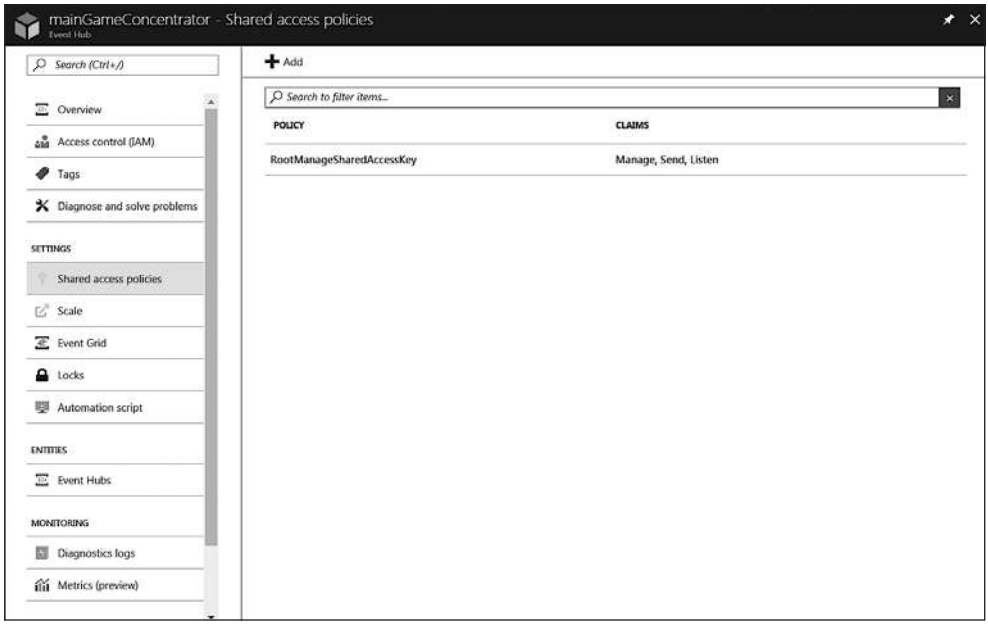


Рис. 10.19. Страница управления политиками

По умолчанию доступен только один тип политики SAS — RootManageSharedAccessKey. Он позволяет обладателю ключа доступа делать в Event Hub все: добавлять сообщения, читать их из концентратора и иметь доступ к API управления. Правилом хорошего тона считается создание политик с более ограниченными правами. Это можно сделать, нажав на ссылку +Add. После этого откроется форма, показанная на рис. 10.20. Дадим ей имя SendReadAccess, снимем флажки Send (права на посылку сообщений в концентратор) и Listen (права на чтение сообщений из концентратора). Чтобы завершить создание политики, после заполнения формы

следует нажать кнопку Create (Создать). Имейте в виду: эта политика задается на уровне пространства имен! Для каждого конкретного концентратора можно создавать свои политики с более гранулярным разграничением доступа.

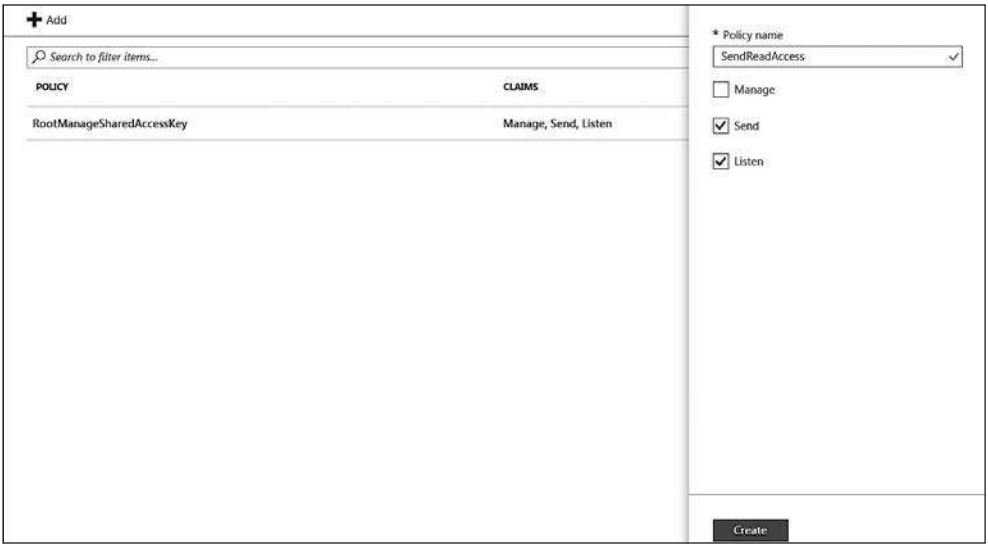


Рис. 10.20. Создание новой политики SAS

Теперь, после создания новой политики, можно получить доступ к ключам и строкам подключения (чтобы их увидеть, необходимо щелкнуть на строке с данной политикой) (рис. 10.21).

Эти ключи и строки подключения будут использоваться для программных клиентов и сервисов. Благодаря наличию множества политик подключений очень удобно управлять правами доступа большого количества клиентов. При этом можно установить отношения «один клиент — одна политика», что позволяет очень гибко управлять доступом к концентратору.

Теперь посмотрим, как именно программные клиенты могут взаимодействовать с концентратором. Как и в большинстве примеров кода этой книги, будет использовано консольное приложение .NET Core 2.0, созданное с помощью IDE Visual Studio 2017 Community Edition. Начнем с клиента-источника.

Для установки Azure EventHub SDK необходимо установить NuGet пакеты от Microsoft (рис. 10.22).

Кроме того, понадобится установить Newtonsoft.JSON, чтобы иметь возможность работать с программным кодом в JSON.

Для отсылки сообщений в EventHub нужно использовать класс `EventHubClient` (рис. 10.23). Чтобы инстанцировать экземпляр этого класса, необходима строка подключения, которую мы можем видеть на вкладке SAS политик (см. рис. 10.21), а именно Connecting string-primary key.

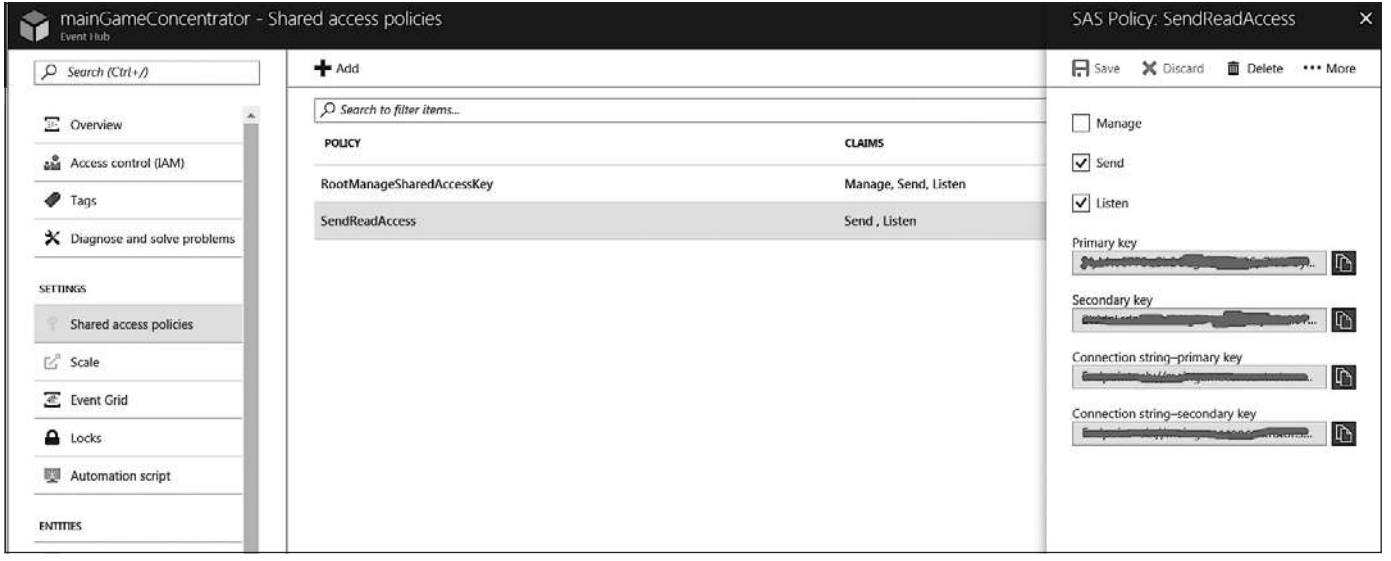


Рис. 10.21. Получение ключей и строк подключения к новой политике

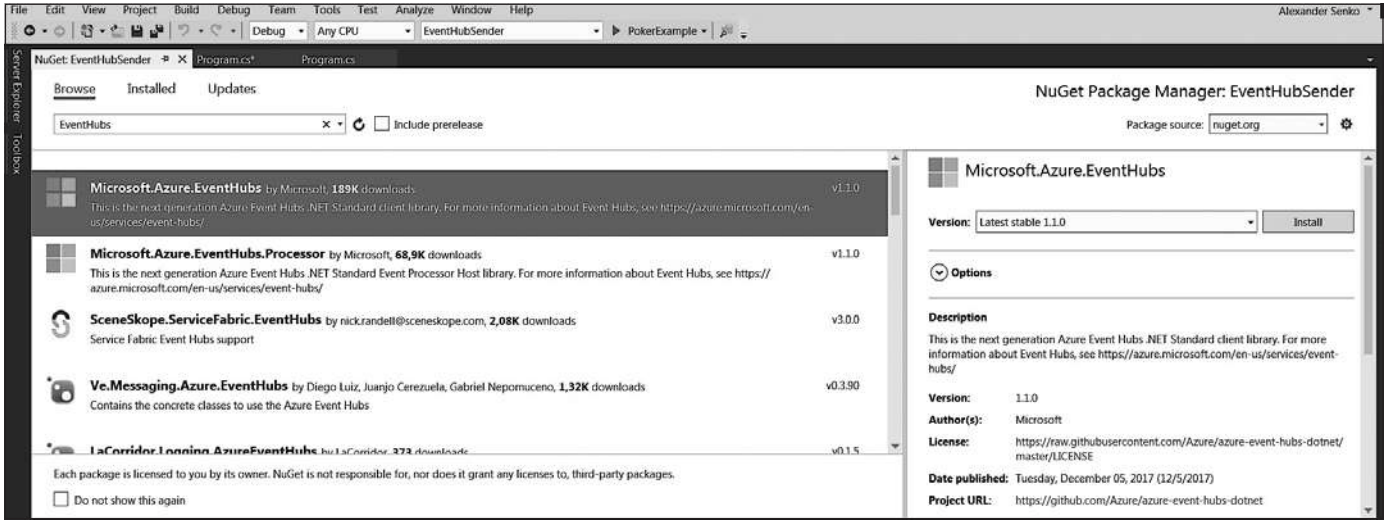


Рис. 10.22. Пакет NuGet для Azure Event Hub SDK

```
string eventHubCnnStr = @"Endpoint=sb://maingameconcentrator.servicebus.windows.net/;SharedAccessKeyName=SendReadAccess;SharedAccessKey=
var connectionStringBuilder = new EventHubsConnectionStringBuilder(eventHubCnnStr)
{
    EntityPath = "mainhub"
};

_eventHubClient = EventHubClient.CreateFromConnectionString(connectionStringBuilder.ToString());
```

Рис. 10.23. Инстанцирование экземпляра класса EventHubClient строкой соединения

Строка соединения состоит из URL конечной точки (в данном случае это `sb://maingameconcentrator...` — конечная точка пространства имен), имени политики SAS и токена SAS. Чтобы определить конкретный Event Hub, используется свойство `EntityPath` класса `EventHubsConnectionStringBuilder`.

Чтобы послать сообщение в Event Hub, необходимо выполнить код, показанный на рис. 10.24. В нем отсылка сообщения происходит в строке `_eventHubClient.SendAsync(...)`. Сериализованное в формате JSON сообщение кодируется далее в массив байтов и в виде экземпляра класса `EventData` передается методу `.SendAsync(...)`.

```
Random rnd = new Random();

string message = JsonConvert.SerializeObject(new
{
    PLAYERID = rnd.Next(1,10),
    TABLEID = 1,
    STEPID = Guid.NewGuid().ToString(),
    TIME = DateTime.Now,
    POT = rnd.Next(50, 500)
});
Console.WriteLine($"Sending message: {message}");
await _eventHubClient.SendAsync(new EventData(Encoding.UTF8.GetBytes(message)));
```

Рис. 10.24. Пример отсылки сообщений в Event Hub

Полный текст программы имеет следующий вид:

```
using Microsoft.Azure.EventHubs;
using Newtonsoft.Json;
using System;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace PokerExample
{
  class Program
  {
    private static EventHubClient _eventHubClient;

    static void Main(string[] args)
    {
      int messageCount = 3600;
      string key = "";
      string sasName = "";
      string eventHubCnnStr = $"Endpoint=sb://maingameconcentrator.servicebus.
        windows.net/; SharedAccessKeyName={sasName}; SharedAccessKey={key}";
      var connectionStringBuilder =
        new EventHubsConnectionStringBuilder(eventHubCnnStr)
      {
        EntityPath = "mainhub"
      };
```

```
      _eventHubClient = EventHubClient.
        CreateFromConnectionString(connectionStringBuilder.ToString());

      for (int m = 0; m < messageCount; m++)
      {
        MainAsync().Wait();
        Thread.Sleep(500);
      }
    }

    static async Task MainAsync()
    {
      Random rnd = new Random();

      string message = JsonConvert.SerializeObject(new
      {
        PLAYERID = rnd.Next(1,10),
        TABLEID = 1,
        STEPID = Guid.NewGuid().ToString(),
        TIME = DateTime.Now,
        POT = rnd.Next(50, 500)
      });
      Console.WriteLine($"Sending message: {message}");
      await _eventHubClient.SendAsync(new
        EventData(Encoding.UTF8.GetBytes(message)));

    }
  }
}
```

После запуска программы через некоторое время сообщения будут добавлены в концентратор mainhub. Примерный вид результата панели обзора с метриками, относящимися к пространству имен, представлен на рис. 10.25 (открыта вкладка, отвечающая за мониторинг сообщений).

В данном случае в пространство имен было послано 468 сообщений. На вкладках Request и Throughput можно отследить общее количество запросов, в том числе и неуспешных, и оценить достаточность величины полосы пропускания. Большое количество неуспешных запросов и высокое значение Throughput говорит о том, что необходимо увеличить количество единиц полосы пропускания через вкладку Scale (если вы умудрились выбрать лимит, то придется обращаться в службу поддержки Azure с просьбой увеличить его). При малом значении Throughput количество единиц полосы пропускания следует уменьшать. Этот процесс может быть автоматизирован (см. шаг, описанный на рис. 10.15, автоматизация доступна на уровне пространства имен).

Более детальный мониторинг на уровне концентратора доступен при выборе конкретного Event Hub из списка. Для концентратора mainhub этот результат имеет вид, показанный на рис. 10.26.

Теперь рассмотрим, как можно читать сообщения из концентратора программными клиентами (подключение сервисов Azure к концентратору будет описано в других главах). Для построения клиента-потребителя, помимо NuGet-пакетов, добавленных для источника сообщений, требуется еще EventHubs.Processor (рис. 10.27).

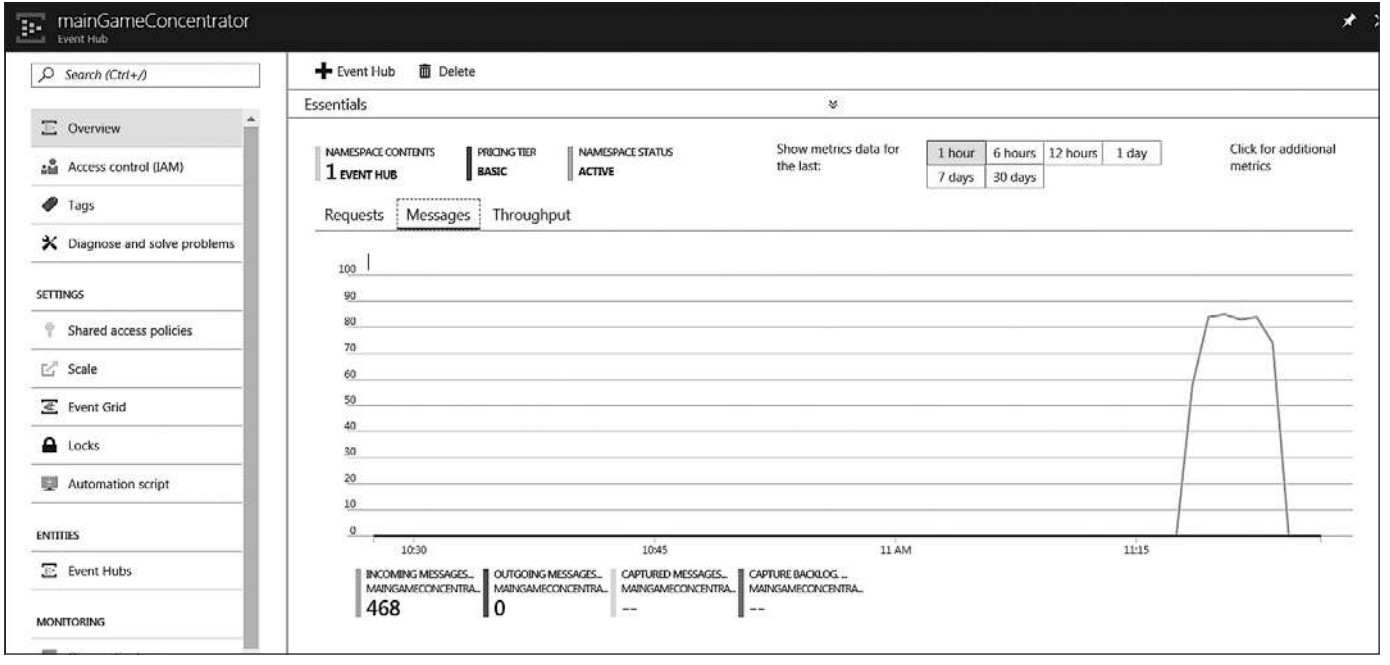


Рис. 10.25. Мониторинг работы Event Hub на уровне пространства имен на вкладке Messages

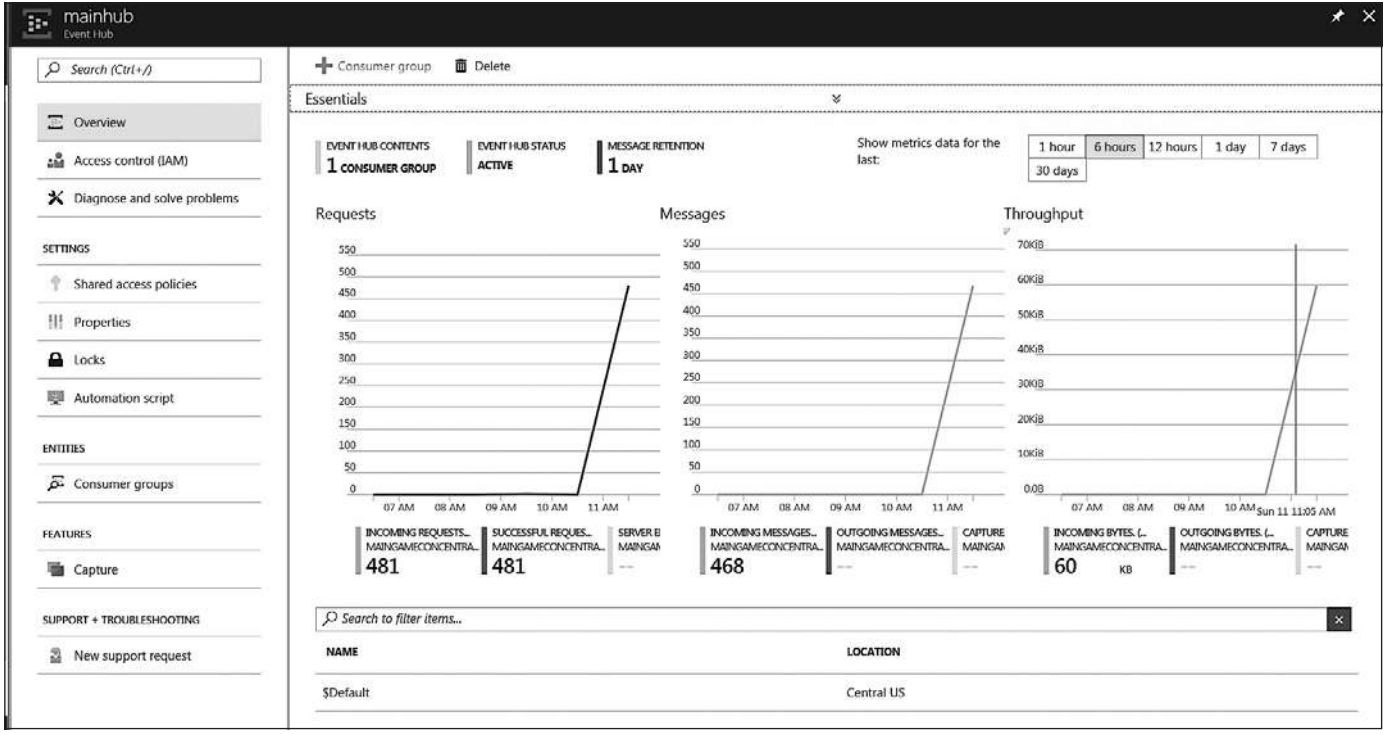


Рис. 10.26. Мониторинг работы концентратора mainhub

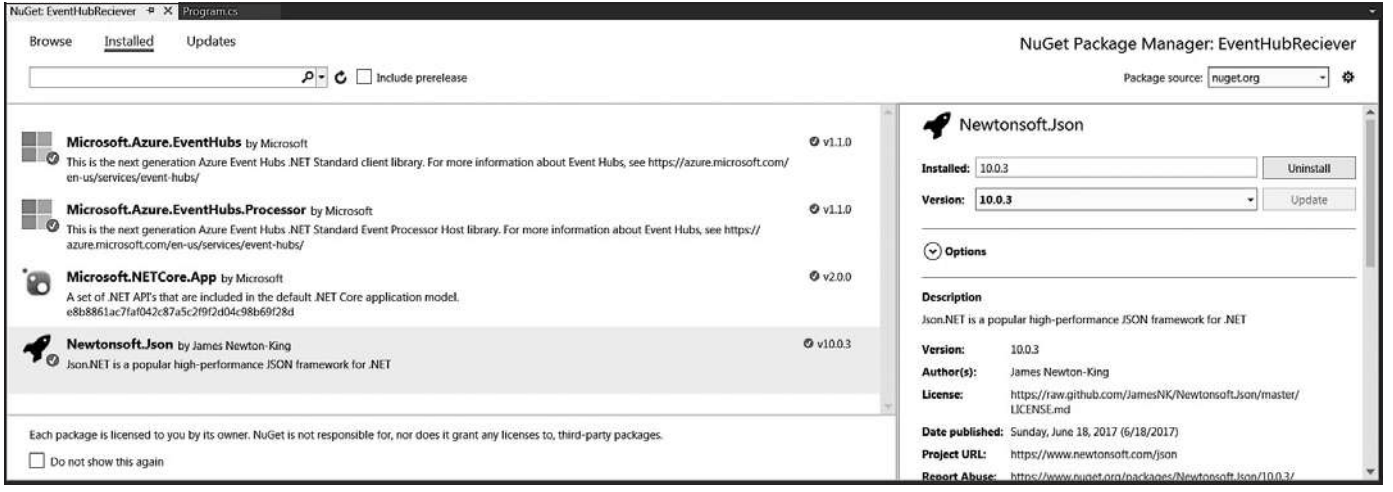


Рис. 10.27. Минимальный набор NuGet-пакетов, необходимых для построения клиента Event Hub

Итак, как уже было сказано выше, для успешной работы клиента-приемника сообщений необходим контейнер для хранения смещения. По умолчанию в качестве такого контейнера используется BLOB-контейнер из Azure BLOB Storage. Для того чтобы обрабатывать сообщения, в программе необходим класс, реализующий интерфейс `IEventProcessor`. Самый минималистичный способ реализации этого интерфейса имеет следующий вид (рис. 10.28).

```
public class PokerExampleEventProcessor : IEventProcessor
{
    public Task CloseAsync(PartitionContext context, CloseReason reason)
    {
        return Task.CompletedTask;
    }

    public Task OpenAsync(PartitionContext context)
    {
        return Task.CompletedTask;
    }

    public Task ProcessErrorAsync(PartitionContext context, Exception error)
    {
        return Task.CompletedTask;
    }

    public Task ProcessEventsAsync(PartitionContext context, IEnumerable<EventData> messages)
    {
        foreach (var eventData in messages)
        {
            var data = Encoding.UTF8.GetString(eventData.Body.Array, eventData.Body.Offset, eventData.Body.Count);
            Console.WriteLine($"Data from EventHub: '{context.PartitionId}', Data: '{data}'");
        }

        return context.CheckpointAsync();
    }
}
```

Рис. 10.28. Пример реализации интерфейса IEventProcessor

Обратите внимание, что это учебный пример! В реальном коде как минимум должен быть реализован метод `ProcessorErrorAsync`. Непосредственно обработка сообщений происходит в методе `ProcessEventAsync`. Сообщения из концентратора принимается в виде коллекции `IEnumerable<EventData> messages`. Каждый элемент этой коллекции должен быть декодирован из массива байтов в строку JSON, которая, собственно, и представляет собой полезную нагрузку сообщения.

Данный класс отвечает за обработку сообщений, но необходим также класс-«хост», который будет отвечать за подключение к концентратору и к хранилищу смещения. Вот как это выглядит в коде (рис. 10.29).

Итак, в качестве параметров конструктора класса `EventProcessorHost` выступают параметры, относящиеся к концентратору (`hubName` — имя концентратора, `$Default` — имя группы потребителей, `hubCnnStr` — строка подключения к концентратору, `storageConnStr` — строка подключения к хранилищу Azure BLOB Storage, `storageContainerName` — имя BLOB-контейнера в хранилище). Далее следует зарегистрировать класс, реализующий интерфейс `IEventProcessor`

```
var connectionStringBuilder = new EventHubsConnectionStringBuilder(hubCnnStr);

var eventProcessorHost = new EventProcessorHost(
    hubName,
    PartitionReceiver.DefaultConsumerGroupName,
    hubCnnStr,
    storageConnStr,
    storageContainerName);

await eventProcessorHost.RegisterEventProcessorAsync<PokerExampleEventProcessor>();
Console.ReadLine();
await eventProcessorHost.UnregisterEventProcessorAsync();
```

Рис. 10.29. Пример использования класса EventProcessorHost

(в данном случае это `PockerExampleEventProcessor`). Текстовый вывод на консоль подобной программы выглядит следующим образом (рис. 10.30).

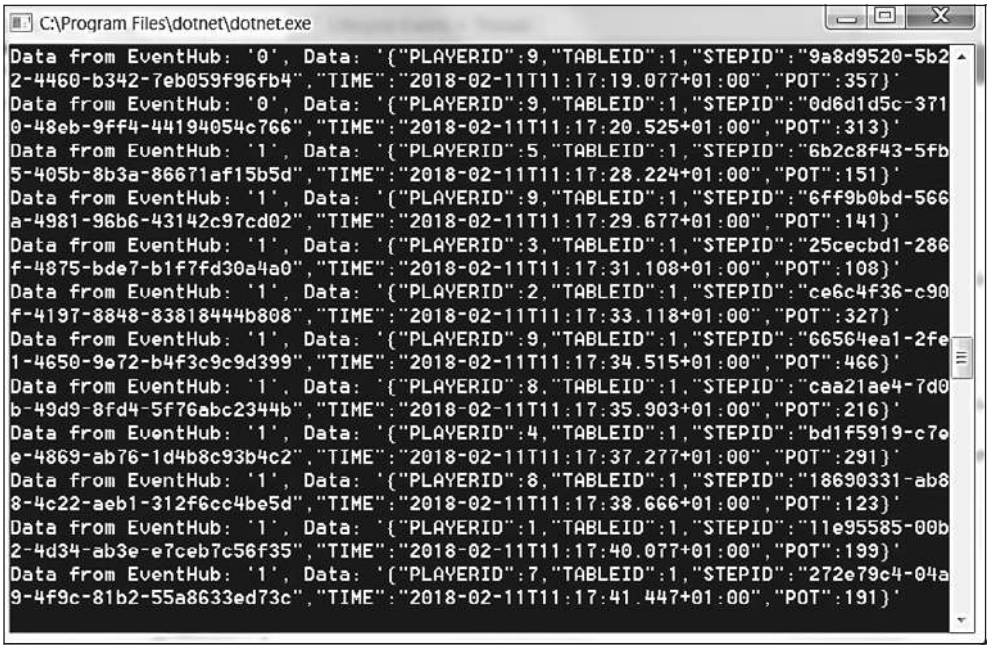


Рис. 10.30. Результат работы программы-потребителя

Обратите внимание: в примере использована группа потребителей по умолчанию — `$Default`. Для ценового уровня Basic это единственная доступная группа потребителей.

Полный код примера (файл `Program.cs`) выглядит так (рис. 10.31).

```
class Program
{
    private static async Task MainAsync()
    {
        string sasKey = "[redacted]=";
        string policyName = "SendReadAccess";
        string hubCnnStr = $"Endpoint=sb://maingameconcentrator.servicebus.windows.net/;SharedAccessKeyName={policyName};SharedAccessKey={sasKey}";
        string hubName = "mainhub";

        string storageContainerName = @"eventhuboffsets";
        string storageAccountName = @"[redacted]";
        string storageAccountKey = @"[redacted]==";

        string storageConnStr = $"DefaultEndpointsProtocol=https;AccountName={storageAccountName};AccountKey={storageAccountKey}";

        var connectionStringBuilder = new EventHubsConnectionStringBuilder(hubCnnStr);

        var eventProcessorHost = new EventProcessorHost(
            hubName,
            PartitionReceiver.DefaultConsumerGroupName,
            hubCnnStr,
            storageConnStr,
            storageContainerName);

        await eventProcessorHost.RegisterEventProcessorAsync<PokerExampleEventProcessor>();
        Console.ReadLine();
        await eventProcessorHost.UnregisterEventProcessorAsync();
    }

    static void Main(string[] args)
    {
        MainAsync().Wait();
    }
}
```

Рис. 10.31. Главный код программы клиента

10.3. AWS Kinesis Data Streams

AWS Kinesis Data Streams представляет собой сервис концентрации и приема потоковых данных от AWS. Он является частью общей платформы Kinesis, которая работает с потоковыми данными и включает в себя:

- *Kinesis Video Streams* — сервис для приема потоковых медиаданных, в качестве которых могут выступать видео- и звуковые потоки, потоки информации от систем ультразвукового обзора и радиолокации. Он позволяет подключать клиентские приложения, занимающиеся потоковой обработкой аудио- и видеоматериалов. Является новым сервисом группы медиасервисов AWS, его рассмотрение выходит за рамки нашей книги;
- *Kinesis Firehouse* — сервис для доставки потоковых данных в хранилища AWS S3, AWS RedShift, Amazon Elasticsearch и дополнительно в Splunk. (Splunk представляет собой не сервис AWS, а отдельную платформу для анализа логов и построения панелей мониторинга, которая разворачивается в экземплярах EC2.) Этот сервис позволяет в ряде случаев отказаться от использования клиентских приложений в целях доставки данных к указанным хранилищам;
- *Kinesis Data Analytics* — сервис потокового анализа данных, принятых AWS Kinesis Data Streams, позволяющий с помощью SQL-подобного синтаксиса строить задания потоковой обработки, а по сути онлайн-фильтрации, агрегирования, объединения из разных потоков и выборки подмножества полей из всех доступных в сообщении. Этот сервис будет подробно рассмотрен в части IV;
- *Kinesis Data Streams* — это сервис концентратора сообщений от AWS. Начнем его подробное рассмотрение.

Итак, AWS Kinesis Data Streams — один из сервисов платформы Kinesis, которые могут конфигурироваться друг с другом (рис. 10.32).

В целом основные идеи этого сервиса повторяют идеи, с которыми мы познакомились, изучая Kafka. Однако у AWS Kinesis Data Streams имеется и своя специфика, в частности термины и логические понятия. Изучим их.

Потоки Kinesis Data Streams — это упорядоченные последовательности записей данных. *Запись данных* — единица хранения информации в AWS Kinesis Data Streams. Каждая запись имеет порядковый номер, который генерируется и присваивается самим сервисом AWS Kinesis Data Streams, ключ раздела и собственно полезной нагрузки данных, представляющей собой неизменяемую последовательность байтов. Kinesis Data Streams никак не изменяет или никак не влияет на эти данные, которых может быть до 1 Мбайт. Но при этом имеет возможность шифрования записей с помощью сервиса AWS KMS.

Порядковый номер — уникальный, в пределах фрагмента, номер записи, который присваивается после добавления новой записи в Kinesis Data Streams. По мере добавления записей текущее максимальное значение этого номера постоянно увеличивается.

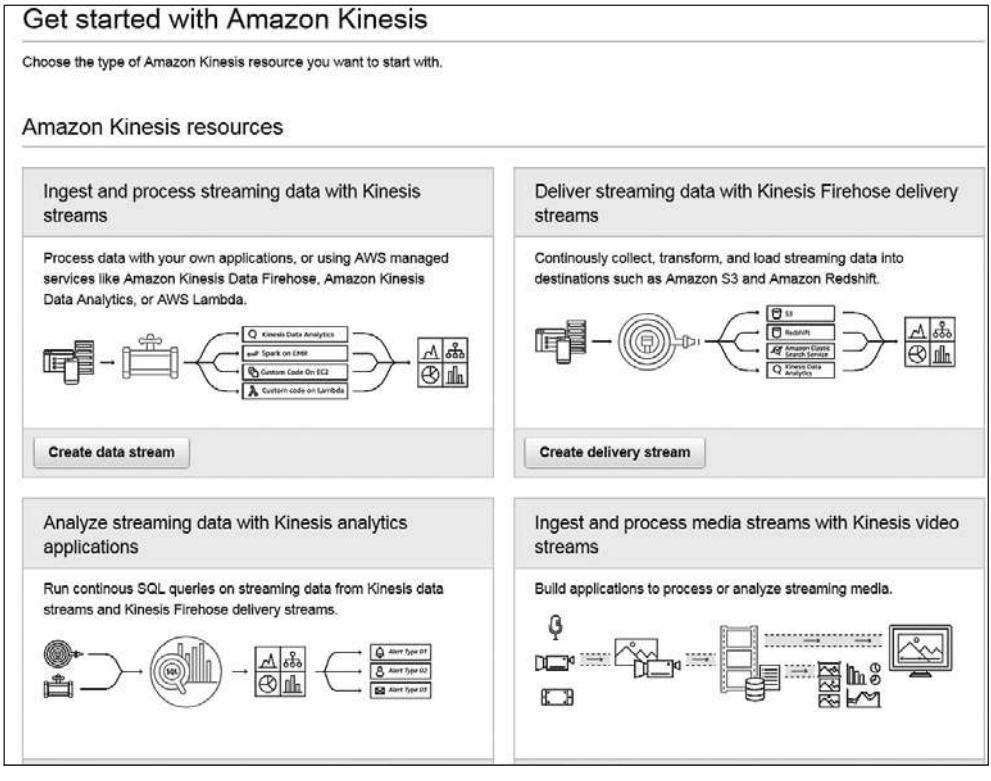


Рис. 10.32. Общий вид панели конфигурирования сервисов платформы Kinesis

Ключ раздела — ключ, используемый для разделения записей на фрагменты. Сам сервис Kinesis Data Streams распределяет записи среди многих разделов, а данный ключ, ассоциированный с каждой записью, определяет, к какому фрагменту относится конкретная запись. Для этого используется хеш-функция MD5, преобразующая ключ раздела в целое значение, являющееся номером фрагмента.

Итак, записи разделяются на *фрагменты* (shards). Они являются группами записей, на которые сервис выделяет фиксированный объем (unit of capacity). Эти группы имеют уникальные идентификаторы, и каждый поток состоит из одного или нескольких фрагментов. На каждый фрагмент, помимо емкости, выделяется фиксированная величина общей полосы пропускания. Конкретные цифры для одного фрагмента: пять транзакций в секунду, скорость чтения — до 2 Мбайт в секунду и запись — до 1 Мбайт в секунду. Общая емкость и полоса пропускания потока определяются суммированием емкостей и полос пропускания потоков.

Точно так же, как и в случае Kafka, единственно возможный способ удалить записи — удалить их по истечении заданного промежутка *времени обращения*

(retention period). По умолчанию этот период (он также является минимальным временем) равен 24 часам и может быть увеличен до семи дней.

Клиентские приложения могут взаимодействовать с Kinesis Data Streams следующими путями: как источник сообщений, то есть их производитель (producer), как приемник сообщений — потребитель (consumer) и как приложение для управления самим сервисом с использованием AWS SDK напрямую.

Производитель может применить специальную библиотеку Klient Producer Library (KPL) для высокоуровневого взаимодействия с Kinesis Data Streams и использовать напрямую API сервиса AWS Kinesis Data Streams. Кроме того, имеются специальные агенты, подходящие для приема логов от EC2 или физического сервера.

Потребитель может использовать библиотеку Kinesis Client Library (KCL). Она берет на себя все трудности взаимодействия с фрагментами, работу с номерами последовательности и текущих смещений. Для хранения величины смещения служит база данных DynamoDB. Кроме того, применение KCL — рекомендуемый путь доставки сообщений в DynamoDB для их сохранения. При этом в DynamoDB создается по одной таблице на каждый экземпляр KCL.

Обе библиотеки написаны на Java, а для других языков программирования существуют библиотеки-обертки.

Теперь рассмотрим подробнее, как создавать AWS Kinesis Data Streams и работать с ним. Для этого на общей панели создания сервисов Kinesis (см. рис. 10.32) следует нажать кнопку Create data stream (Создать поток данных). Откроется панель конфигурирования Kinesis Data Streams (рис. 10.33). Здесь указывается имя сервиса (Kinesis stream name). Кроме того, имеется калькулятор (Shard calculator), позволяющий оценить требуемое количество фрагментов, исходя из предполагаемого значения размера сообщений, количества сообщений в секунду и количества приложений, которые должны одновременно читать сообщения из потока. Это количество и указывается в соответствующем поле настройки. В дальнейшем данная величина может быть изменена, что выгодно отличает AWS Kinesis Data Streams от Azure Event Hub, в котором количество разделов — величина неизменяемая и задаваемая только в момент создания.

После создания нового сервиса он становится доступным на общей панели мониторинга Kinesis Data Streams (рис. 10.34).

Все рассмотренные прежде настройки доступны для последующей модификации (рис. 10.35).

Теперь рассмотрим второй компонент группы сервисов Kinesis — Firehouse. Он представляет собой сервис доставки сообщений, полученных из ряда источников потоковых данных, в различные сервисы хранения данных (рис. 10.36). В качестве источников могут выступать сервисы из группы Kinesis — Data Streams и Analytics, события сервисов логирования и мониторинга — CloudWatch, CloudTrail, а также события, поставляемые сервисами AWS IoT и клиентскими приложениями AWS SDK. Таким образом, этот сервис предоставляет возможность прямой поставки потоковых данных в облачные хранилища.

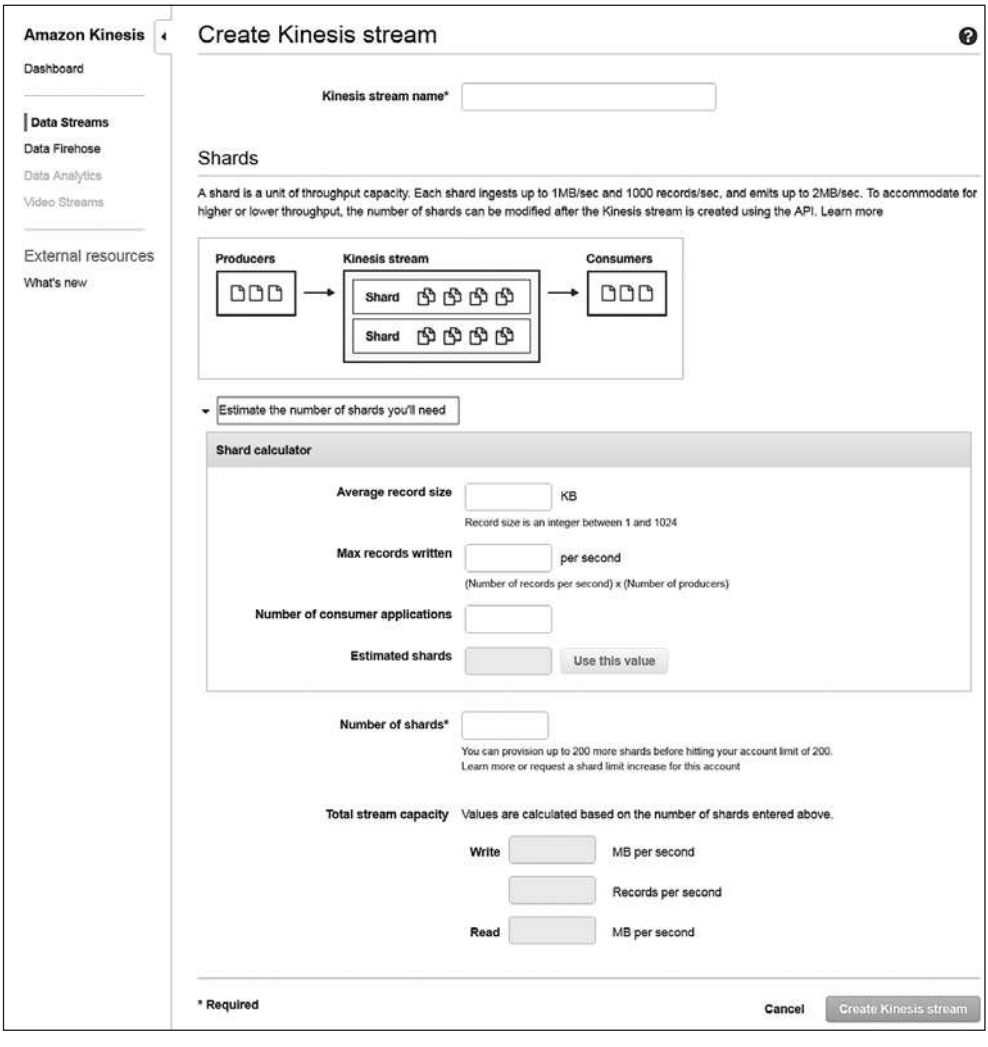


Рис. 10.33. Панель конфигурирования Kinesis Data Stream

Рассмотрим теперь подробнее, как создавать пример как создавать и конфигурировать сервис AWS Kinesis Firehouse. Для этого необходимо перейти на общую панель конфигурирования сервисов Kinesis (см. рис. 10.32) и нажать кнопку Create delivery stream (Создать поток). В результате откроется первая форма конфигурирования, показанная на рис. 10.37. Прежде всего следует указать источник данных — прямое помещение (Direct PUT or other sources) или подключение к экземпляру Kinesis Data Streams (в данном случае это экземпляр сервиса с именем `pocker_concentrator`).

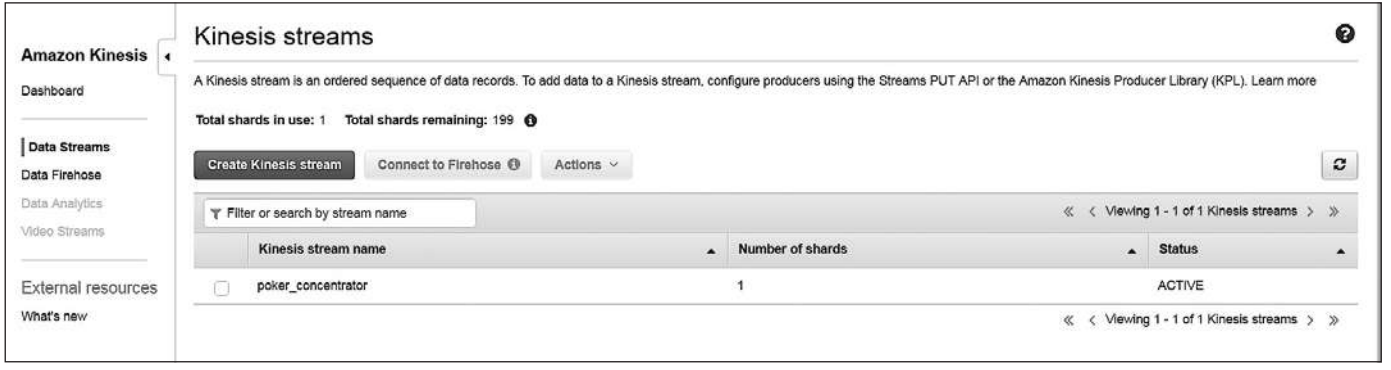


Рис. 10.34. Общий вид сервиса мониторинга потоков

Рис. 10.35. Панель конфигурирования созданного экземпляра

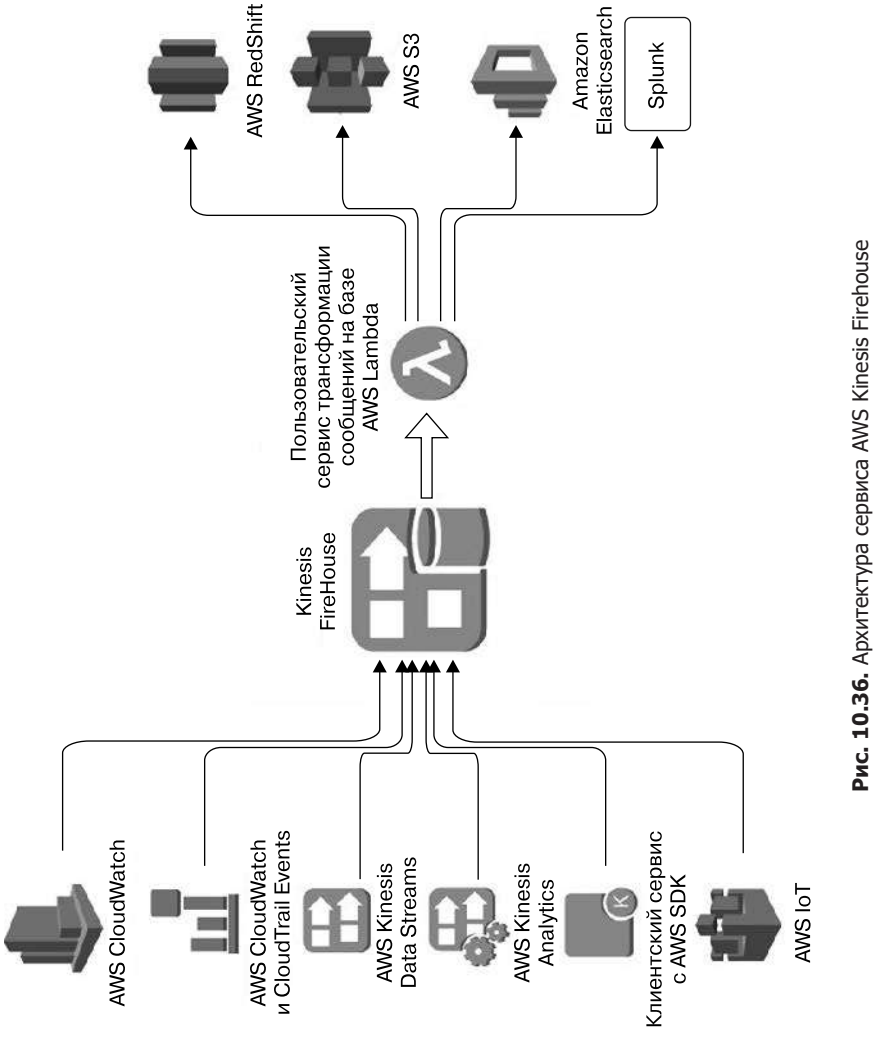


Рис. 10.36. Архитектура сервиса AWS Kinesis Firehouse

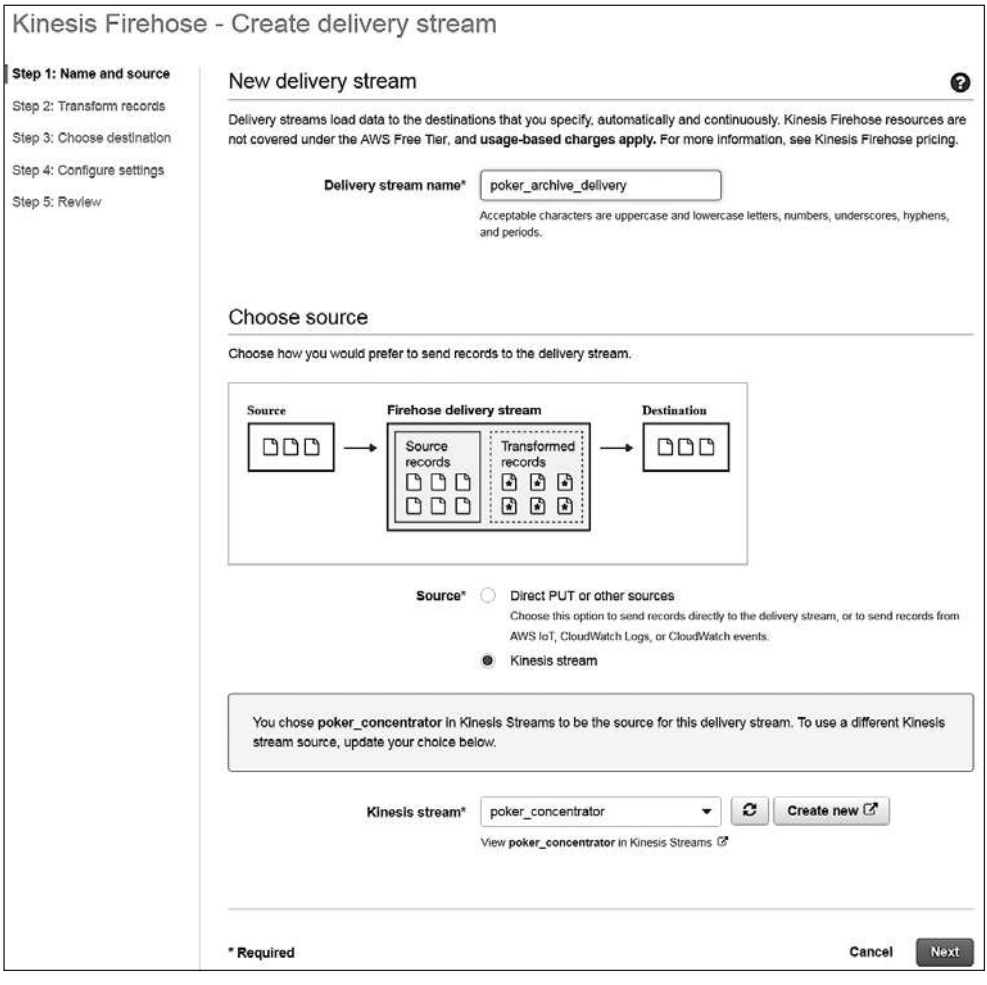


Рис. 10.37. Конфигурирование источников AWS Delivery Stream

Далее возможно конфигурирование сервиса трансформации, построенного на базе пользовательского бессервисного приложения на основе AWS Lambda (рис. 10.38).

Далее следует сконфигурировать хранилище-назначение (рис. 10.39). Для этого необходимо указать имя бакета (S3 bucket) и обозначить префикс (Prefix) для файлов, которые будут созданы автоматически. Если в аккаунте отсутствует требуемый бакет, то его можно создать с помощью кнопки Create new (Создать новый). На рис. 10.40 приведен пример конфигурирования AWS S3.

Последним шагом будет конечный обзор параметров создаваемого сервиса, которые при необходимости могут быть изменены (рис. 10.41).

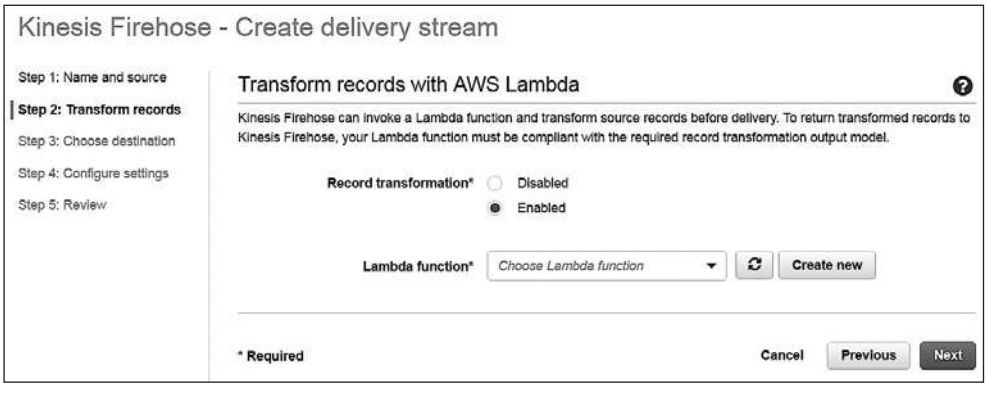


Рис. 10.38. Подключение пользовательского сервиса трансформации сообщений перед их добавлением в хранилище

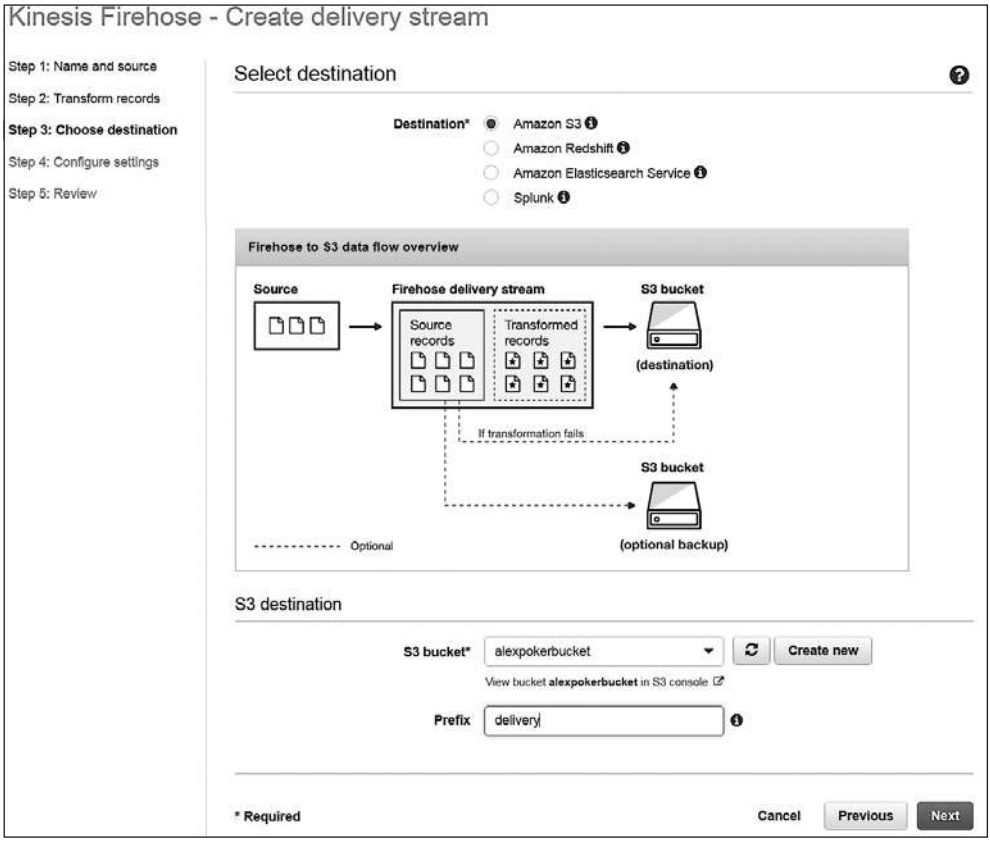


Рис. 10.39. Форма конфигурирования хранилища-назначения

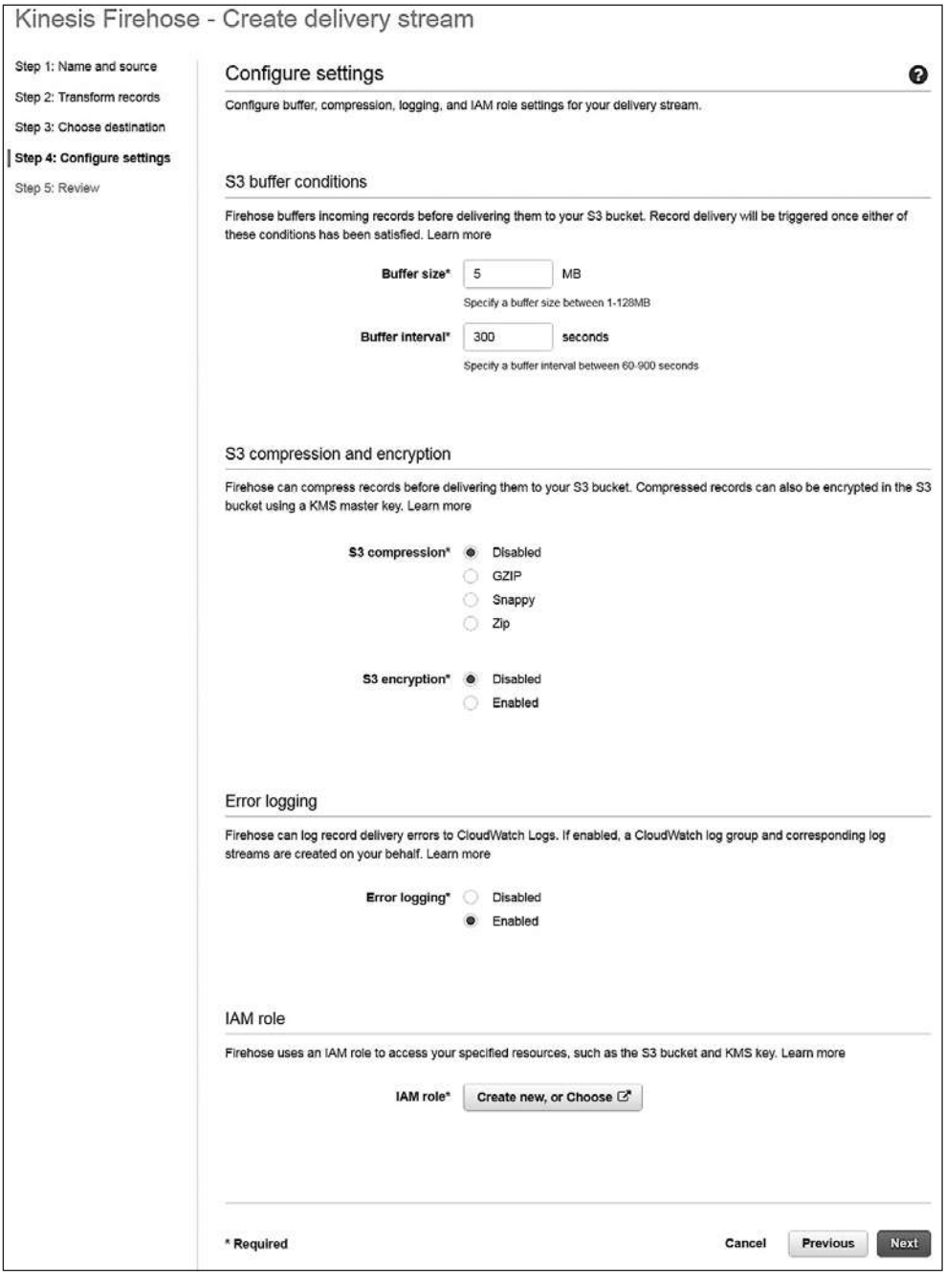


Рис. 10.40. Форма конфигурирования параметров AWS S3

Рис. 10.41. Панель финального обзора создаваемого сервиса

В итоге созданный сервис Kinesis Firehouse имеет следующий вид (рис. 10.42).

Параметры созданного экземпляра сервиса AWS Firehouse под именем pocker_archive_delivery можно посмотреть, щелкнув на соответствующей строке (рис. 10.43).

Теперь рассмотрим пример, который покажет, как связать все эти сервисы. Для этого нужно создать источник сообщений, посылающий их в сконфигурированный нами AWS Kinesis Data Stream с подключенным к нему сервисом AWS Kinesis Firehouse и в итоге в S3 (рис. 10.44).

В этот раз я не буду применять библиотеки KPL и строить консольное приложение. Я покажу, как использовать AWS SDK CLI напрямую из командной строки, а заодно и как задействовать и конфигурировать эту утилиту.

Рис. 10.42. Обзорная панель сервиса AWS Firehouse

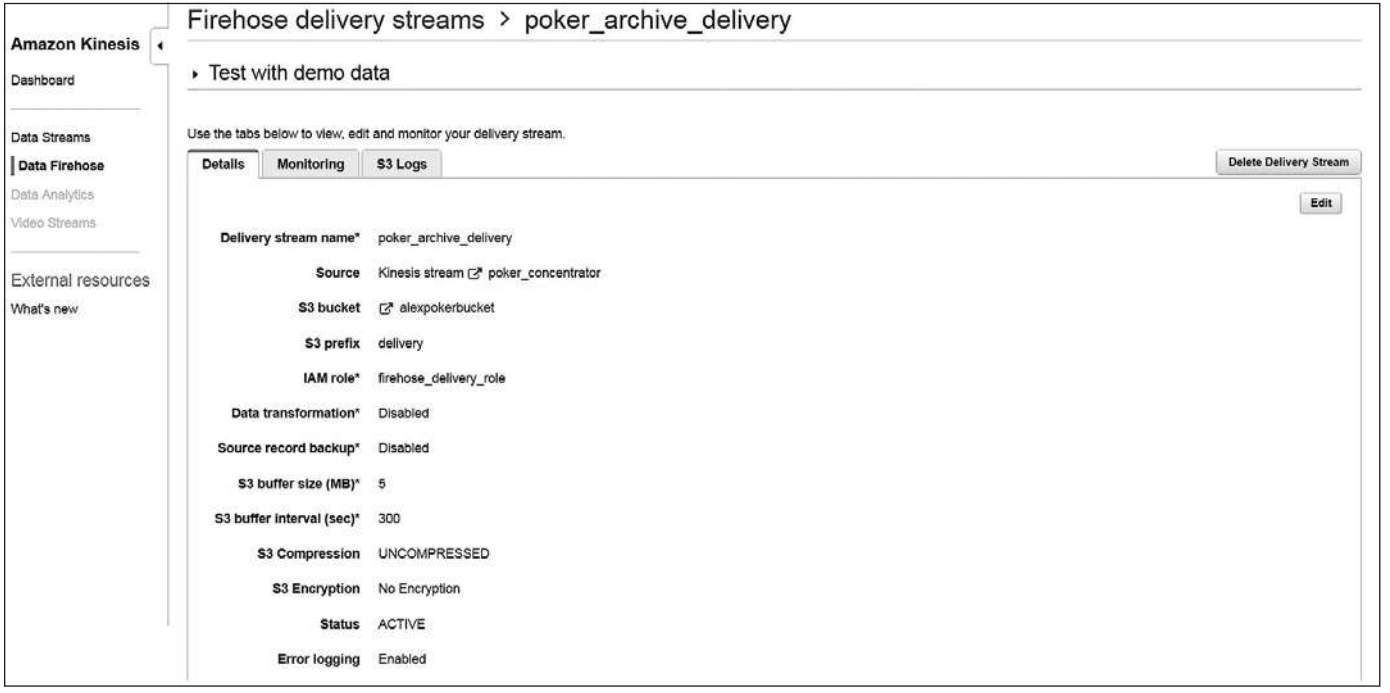


Рис. 10.43. Параметры созданного экземпляра сервиса AWS Firehouse

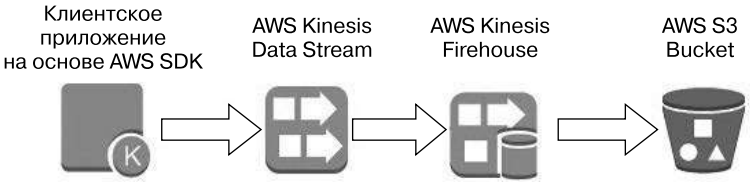


Рис. 10.44. Архитектура примера

Прежде всего необходимо установить AWS CLI на ваш компьютер или на виртуальную машину. Лично я предпочитаю использовать Ubuntu или Amazon AMI для работы с данной утилитой, хотя можно применять и Windows-версии. Итак, нужно установить AWS CLI, введя команду `pip install awscli` в командной оболочке. Далее следует сконфигурировать AWS CLI, для чего нужно создать нового пользователя с программным доступом. Это делается с помощью сервиса AWS IAM (Identity and Access Management). Внешний вид обзорной панели этого сервиса показан на рис. 10.45 (я закрасил чувствительную информацию, ведь аудитория предполагается интеллектуальная, технически грамотная, но потенциально может «пошалить»).

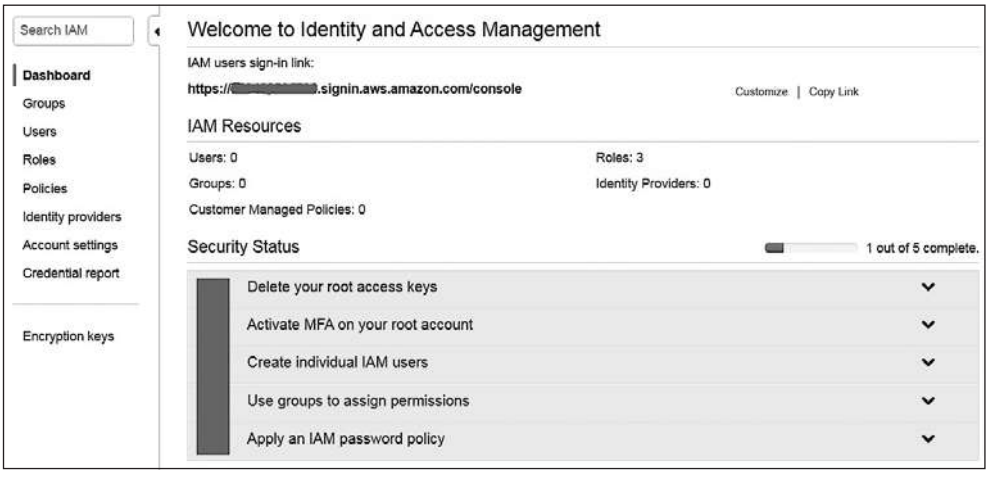


Рис. 10.45. Внешний вид общей панели IAM

Далее следует создать нового пользователя, для чего нужно перейти на вкладку Users и нажать ссылку Add user. Откроется следующая панель (рис. 10.46).

Здесь следует дать пользователю имя (в данном случае это `programmatical_user`) и обязательно установить флажок Programmatic access в поле Access type. Затем следует добавить права доступа в виде политики `KinesisFullAccess` (рис. 10.47).

На рис. 10.48 представлен обзор параметров создаваемого пользователя, а на рис. 10.49 — получение ключей программного доступа.

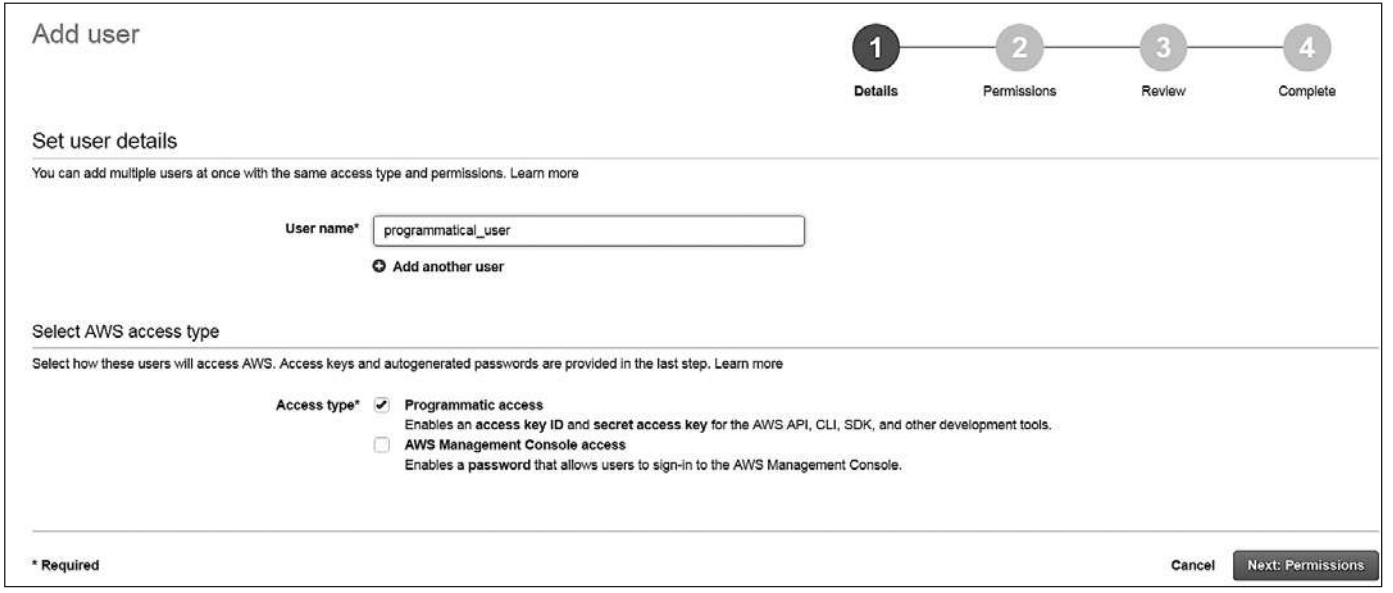


Рис. 10.46. Добавление пользователя с программным доступом

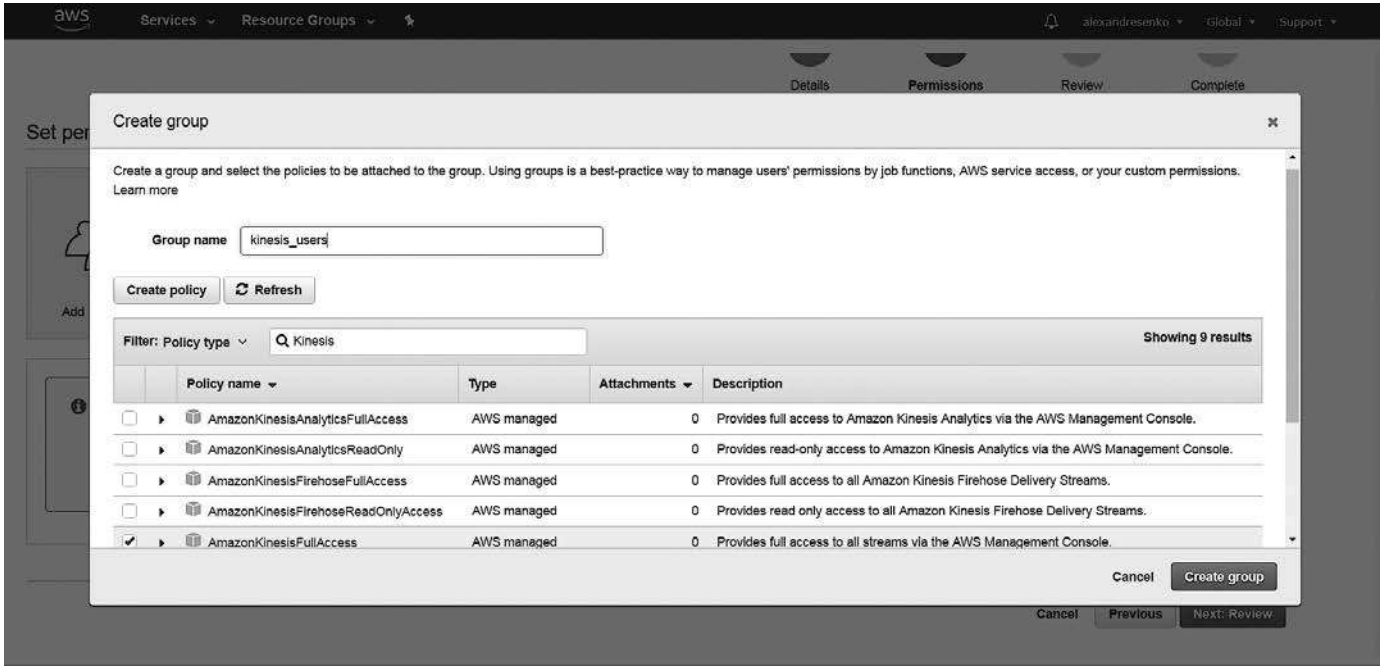


Рис. 10.47. Создание группы и добавление политики, предназначенной для работы с Kinesis

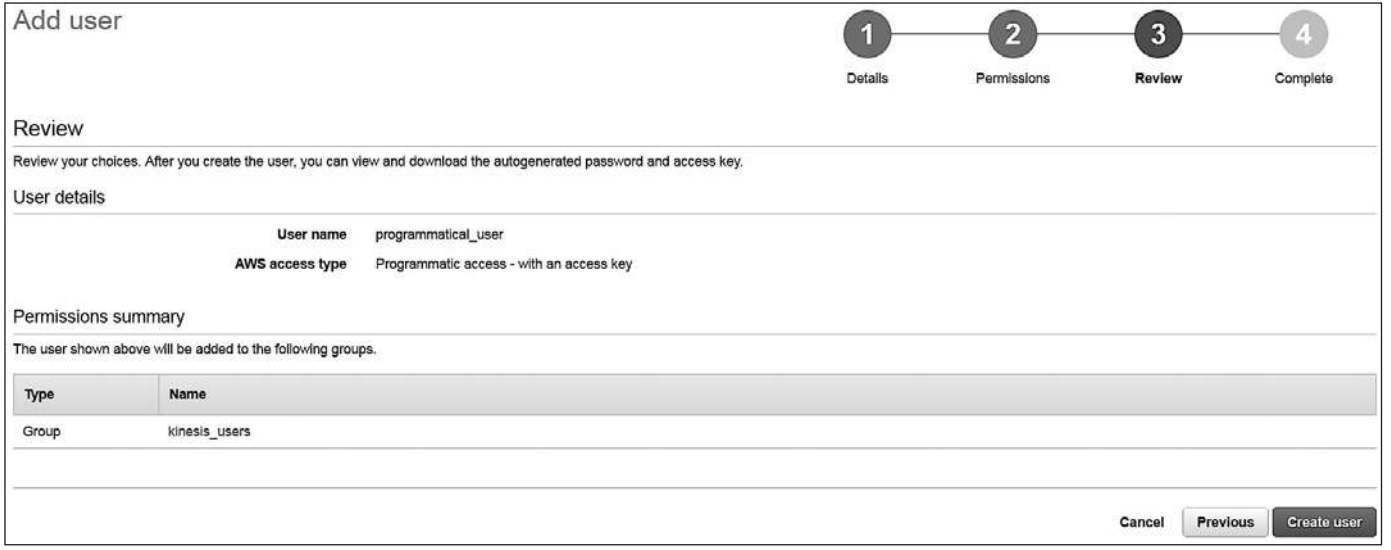


Рис. 10.48. Обзор параметров создаваемого пользователя

Рис. 10.49. Получение ключей программного доступа

Для конфигурирования AWS CLI нужны два ключа: Access key ID и Secret access key. Далее в терминале SSH следует ввести команду `aws configure` и последовательно добавить эти ключи и регион, где размещаются ваши сервисы Kinesis (рис. 10.50).

Рис. 10.50. Конфигурирование AWS CLI

Основная команда AWS CLI — `aws kinesis put-record`, которая помещает сообщение в Kinesis Data Streams. Сценарий (скрипт), отправляющий данные каждую секунду в Kinesis Data Streams, представлен на рис. 10.51.

```
#!/bin/bash

while true; do
    aws kinesis put-record --stream-name poker_concentrator --data '{ "PlayerId": "1", "TableId":"1","Data":"()"}' --partition-key 1
    sleep 1
done
```

Рис. 10.51. Bash-скрипт, отправляющий тестовые данные в Kinesis Data Streams

После создания этого сценария не забудьте выполнить команду `chmod +x <filename>`.

При работе программы можно наблюдать выход, представленный на рис. 10.52. Здесь отображается номер фрагмента для каждого раздела (`"ShardId"`) и номер последовательности (`"SequenceNumber"`).

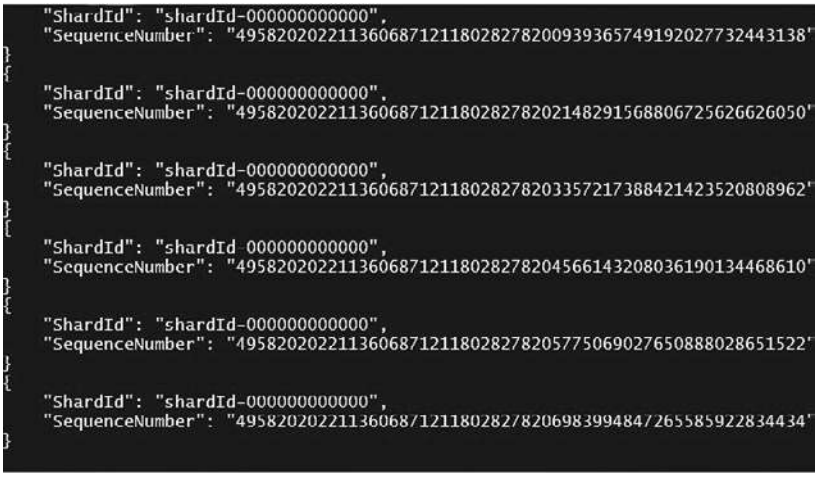


Рис. 10.52. Консольный выход от работающей программы

Открыв вкладку мониторинга работы Kinesis Data Streams и Kinesis Firehouse, можно заметить результаты работы программы (рис. 10.53, 10.54).

Далее можно проверить результаты работы всей цепочки, просмотреть S3-бакет, в который помещаются результаты работы программы (рис. 10.55).

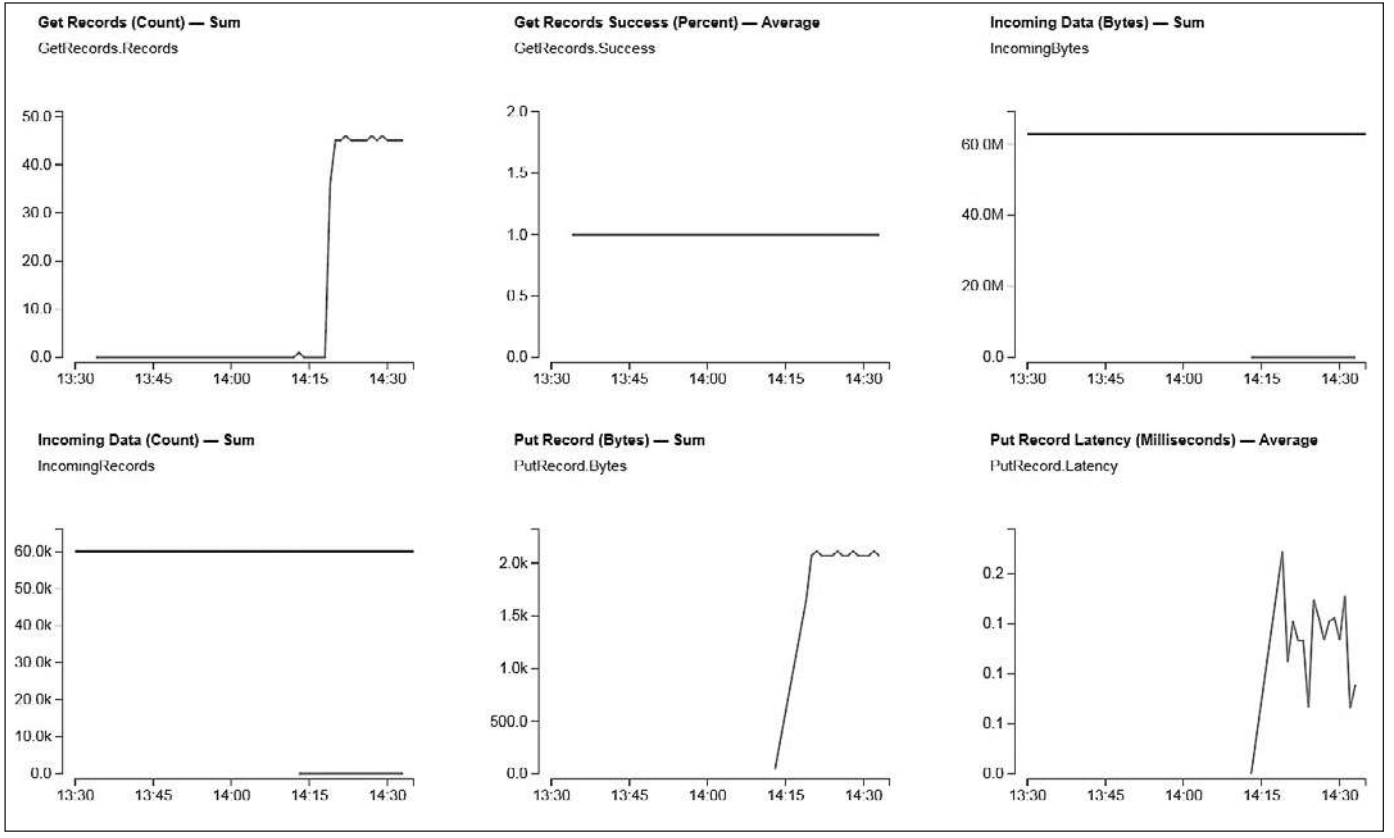


Рис. 10.53. Мониторинг работы сервиса Kinesis Data Streams

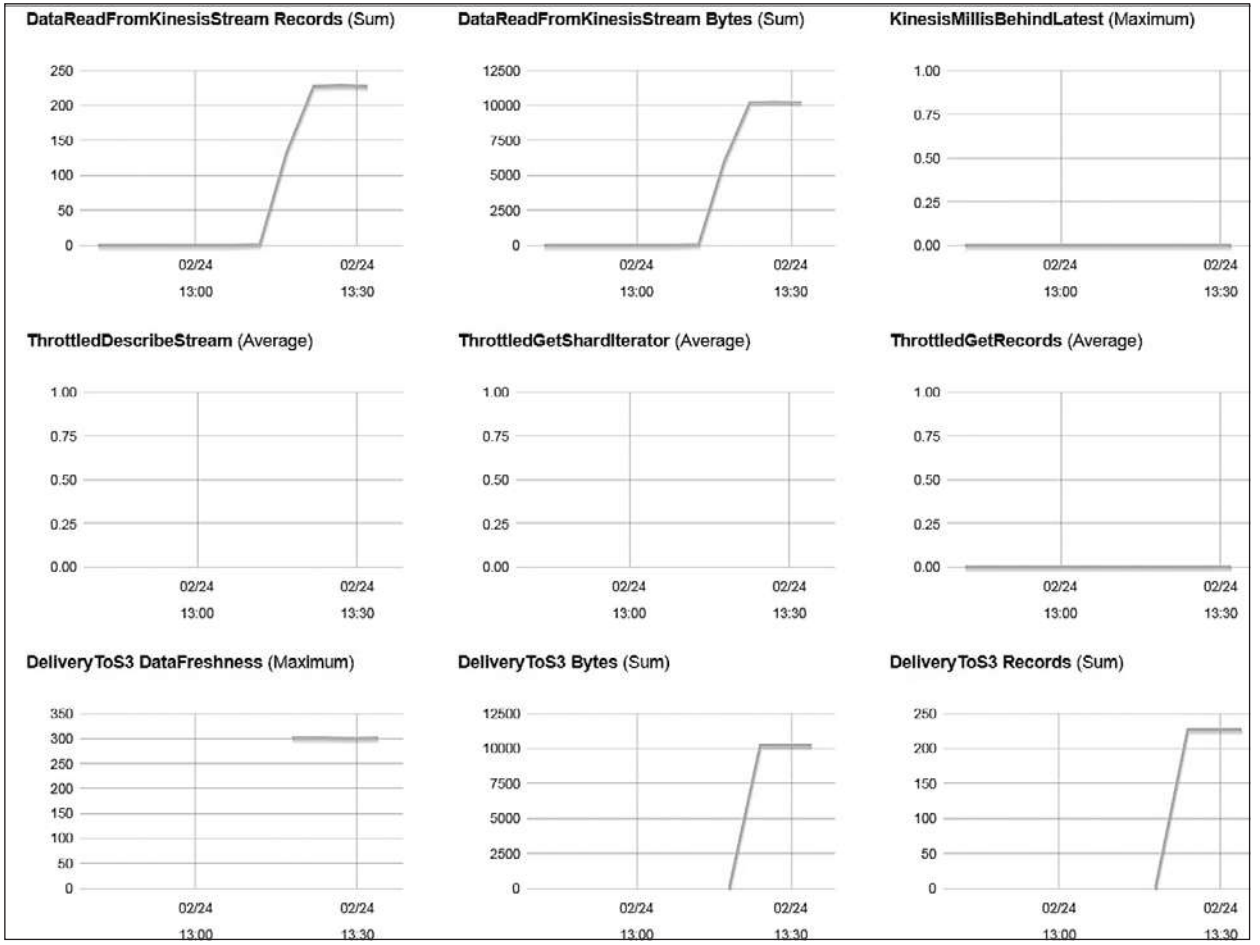


Рис. 10.54. Мониторинг работы сервиса Kinesis Firehouse

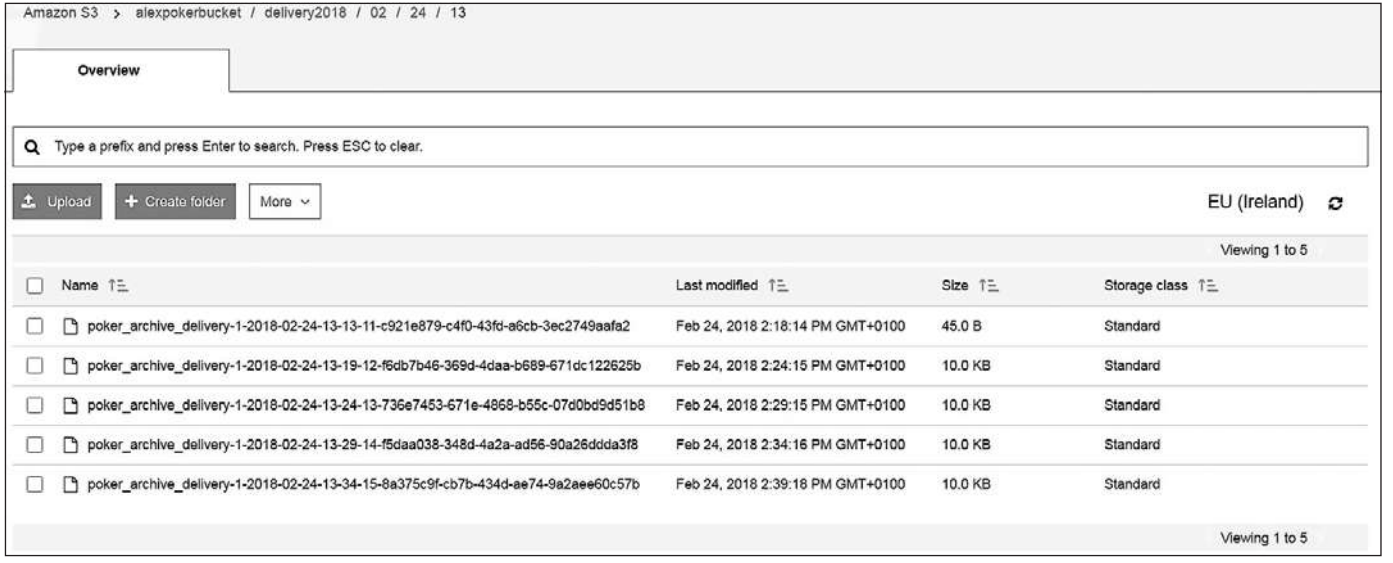


Рис. 10.55. Файлы, сгенерированные системой Kinesis, которые содержат расположенные в S3 тестовые данные

На рис. 10.56 показаны сохраненные в файле сообщения. Данные открыты с помощью программы Блокнот ++ и приведены к более удобному для восприятия виду (иначе они идут одной длинной строкой).

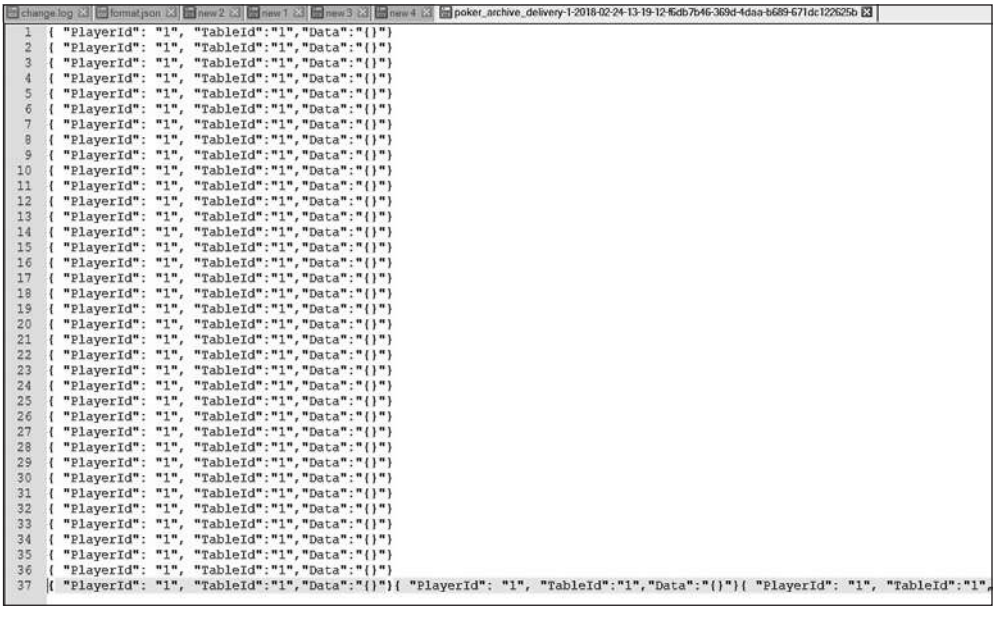


Рис. 10.56. Сохраненные в файле сообщения, посланные в AWS Kinesis Data Streams

10.4. Облачные сервисы развертывания кластерных систем

Итак, в предыдущих разделах текущей главы мы рассмотрели архитектуру сервисов приема потоковых данных — Kafka, Azure Event Hub и AWS Kinesis Data Stream. Из всех трех наибольшей гибкостью и неограниченной масштабируемостью обладает Kafka. Но в то же время развертывание этой системы связано с определенными объективными трудностями. В частности, я наблюдал реальную ситуацию, когда у одного заказчика, обладающего «флотом» подключенных IoT-устройств в количестве порядка 300 тысяч, возникла необходимость перейти с решения платформы концентрации сообщений на стандартную типа Kafka. И несмотря на полное соответствие Kafka всем требованиям по быстродействию, надежности, масштабируемости, сопрягаемости с иными внешними системами хранения и потоковой обработки данных, выбор был сделан в пользу AWS Kinesis.

Причиной такого выбора послужил тот факт, что развертывание этого облачного сервиса концентрации сообщений, характеризующегося огромной масштабируемостью, надежностью и минимальными затратами на администрирование, занимает меньше минуты (если вы проделали вслед за мной шаги, описанные в предыдущем разделе, то, наверное, уже это поняли). В то же время развертывание отказоустойчивого кластера Kafka в производственном режиме занимает месяцы. Это обусловлено тем, что требуется огромная дополнительная работа, связанная с администрированием самих физических серверов, связывающих их локальных сетей, с периодическими обновлениями ПО, деплоями, патчами, резервным копированием и т. д.

Используя облачный PaaS, счастливый читатель за минуту может развернуть мощную облачную систему приема потоковых данных, удовлетворяющую практически все его потребности. Вдобавок совместно с этой системой имеются другие сервисы, обеспечивающие как потоковый анализ его сообщений (AWS Kinesis Analytics, Azure Stream Analytics), так и их доставку непосредственно в хранилище (компонент Capture в Azure Event Hub и сервис AWD Kinesis Firehouse), причем в случае Azure Stream Analytics возможна интеграция с внешними сервисами машинного обучения.

Ну а как быть, если непременно необходима система, развернутая в кластере из виртуальных машин в облаке? Ведь существует очень много сервисов, в частности из мира Apache Hadoop, которые требуют кластера. Мы уже рассматривали DistCp, Sqoop, Kafka, Hadoop. Но ведь есть еще HBase, Kassandra, Spark, Storm и др. Как поступить в этом случае? Для того чтобы облегчить работу с подобными системами, оба облачных аккаунта имеют два специализированных сервиса, обеспечивающих простоту работы с кластерами. Во-первых, есть специализированные сервисы создания кластеров с предустановленными системами Apache. К таким сервисам относятся Azure HDInsight и AWS EMR. Во-вторых, можно использовать кластер контейнеров Docker. Мы рассмотрим создание специализированного кластера Kafka на примере HDInsight. Сервис AWS EMR будет рассмотрен в части IV вместе с технологией Apache Spark.

Сервис Azure HDInsight в настоящее время позволяет развертывать кластеры со следующими установленными системами: Hadoop, Spark, HBase, Interactive Query, Kafka, Storm и R Server. В этой главе мы будем изучать HDInsight на примере Kafka. Данный сервис предоставляет следующие возможности для Kafka:

- уровень доступности (uptime) — 99 % по соглашению об обслуживании (SLA);
- обеспечение отказоустойчивости за счет физического дублирования серверов на два экземпляра — домен обновления (Update Domain) и домен отказа (Fault Domain). Физическое разделение в этом случае происходит на уровне как сетевого интерфейса, так и сети питания. Репликация между серверами происходит автоматически и под контролем Azure;

- поддержка сервиса Managed Disks Azure, обеспечивающего масштабирование до 16 Тбайт;
- интеграция с системой мониторинга и диагностики Azure Log Analytics;
- масштабирование кластера «вверх» и «вниз» с помощью REST API.

Рассмотрим архитектуру Kafka на основе кластера HDInsight (рис. 10.57).

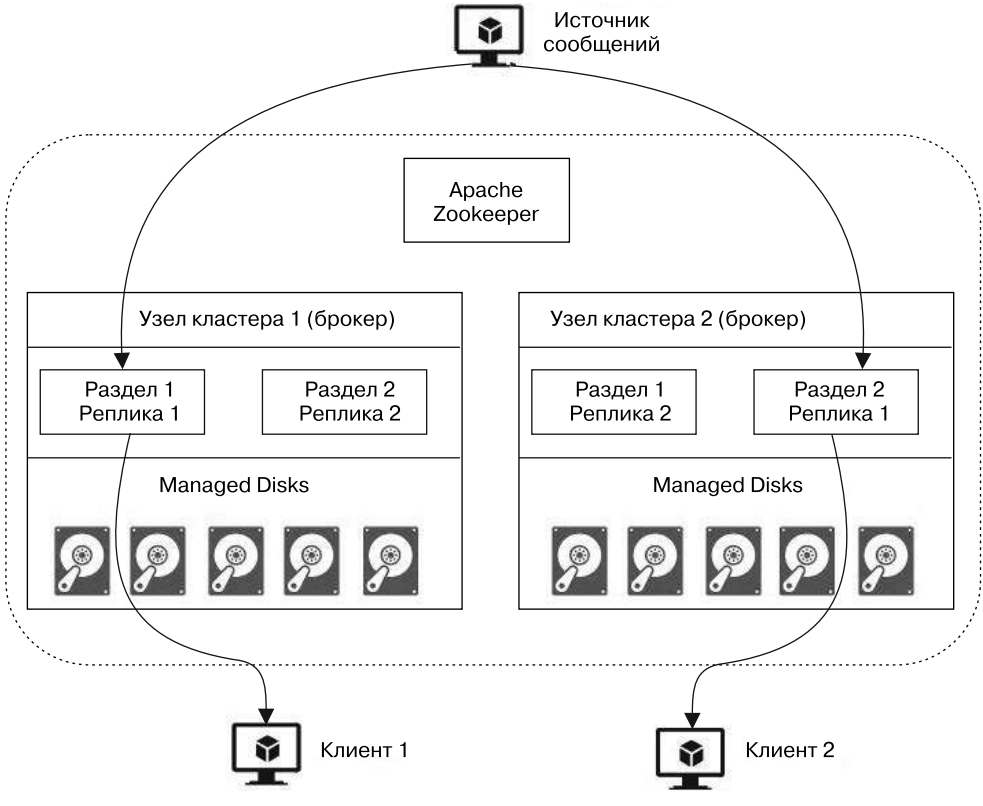


Рис. 10.57. Архитектура Apache Kafka

Головной узел строится на основе Apache Zookeeper. Он представляет собой централизованный сервис для конфигурирования, именования, распределенной синхронизации и управления группой сервисов в среде распределенных приложений, расположенных в кластере. Этот сервис автоматически устанавливается в головной узел кластера и подключается ко всем его физическим узлам (в документации они называются также брокерами (broker)).

Теперь посмотрим, как создавать кластер Kafka. Для этого в строке поиска ресурсов (Marketplace) выбираем HDInsight. Первая страница создания кластера выглядит следующим образом (рис. 10.58).

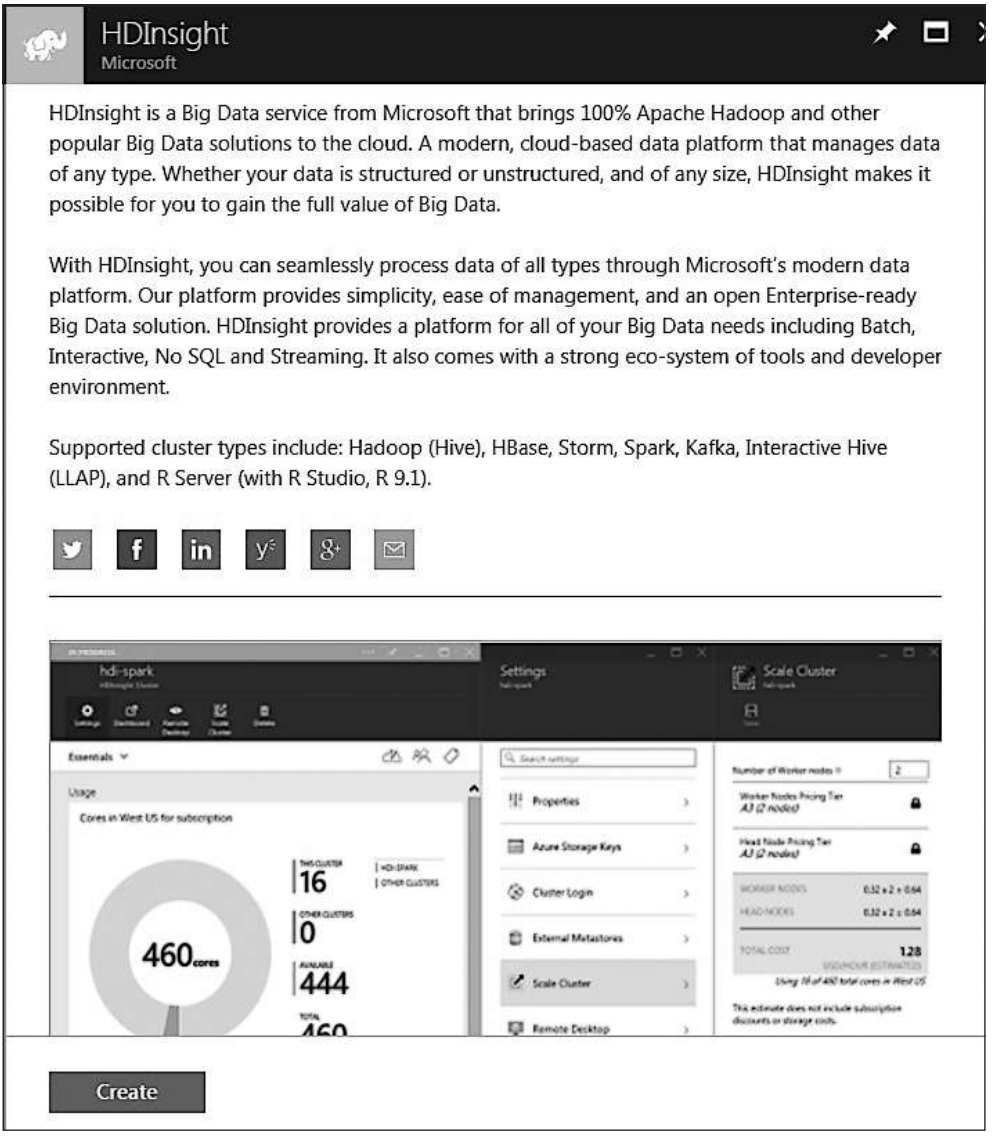


Рис. 10.58. Первая страница создания кластера

На рис. 10.59 представлена форма конфигурирования кластера.

В этой форме нужно указать имя кластера (Cluster name), выбрать тип кластера (Kafka), указать учетные данные и ресурсную группу. Далее следует сконфигурировать хранилище Storage Account, которое должно быть в том же регионе, что и кластер (рис. 10.60).

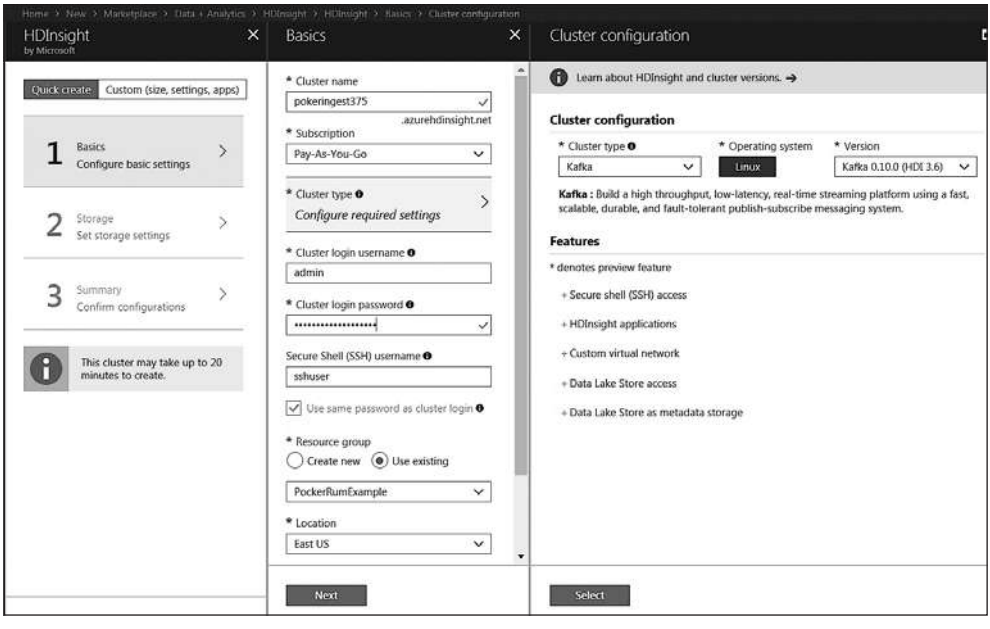


Рис. 10.59. Форма конфигурирования кластера

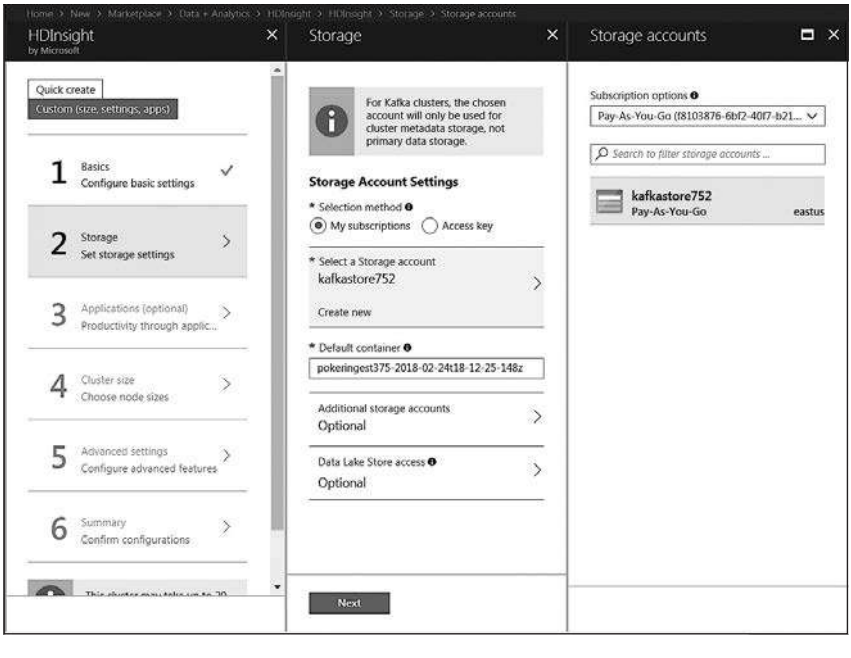


Рис. 10.60. Форма конфигурирования Storage Account для кластера HDInsight Kafka

Далее можно установить в кластер дополнительные приложения. Эти сервисы относятся к экосистеме Apache Hadoop, но не все они доступны для данного типа кластера. В нашем случае доступны два: WANdisco и StreamSets (рис. 10.61). Вероятно, к моменту выхода книги этот список будет расширен или сужен.

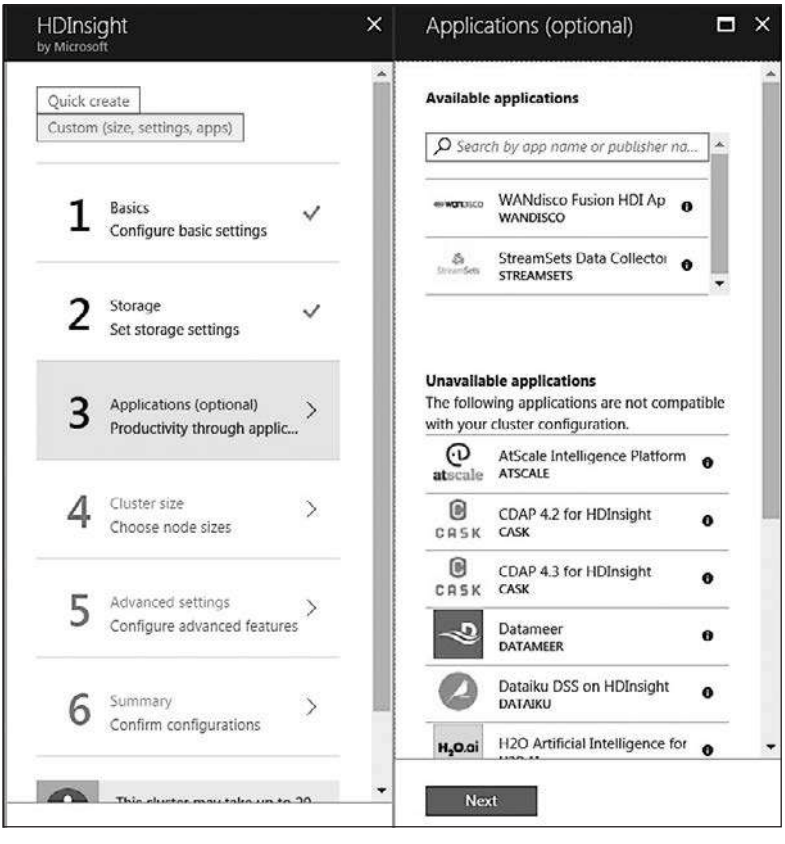


Рис. 10.61. Панель установки дополнительных компонентов в кластер Kafka

Далее следует сконфигурировать узлы кластера, выбрав количество узлов, а затем ценовой уровень каждого отдельного узла (рис. 10.62).

Часовая стоимость — весьма существенный параметр, на который стоит обратить внимание при создании кластера. Кроме того, стоит помнить, что у кластера нет кнопки «Стоп» и плата за его использование будет взиматься все время, пока он существует. Единственный вариант оптимизации затрат в данном случае — применить опцию масштабирования или (еще лучше) создать и удалить кластер с помощью сценария автоматизации Azure CLI или Azure PowerShell. Но при этом все конфигурации должны быть автоматизированы в виде сценария, запускаемого после его создания. Все перечисленное следует помнить при создании кластера.

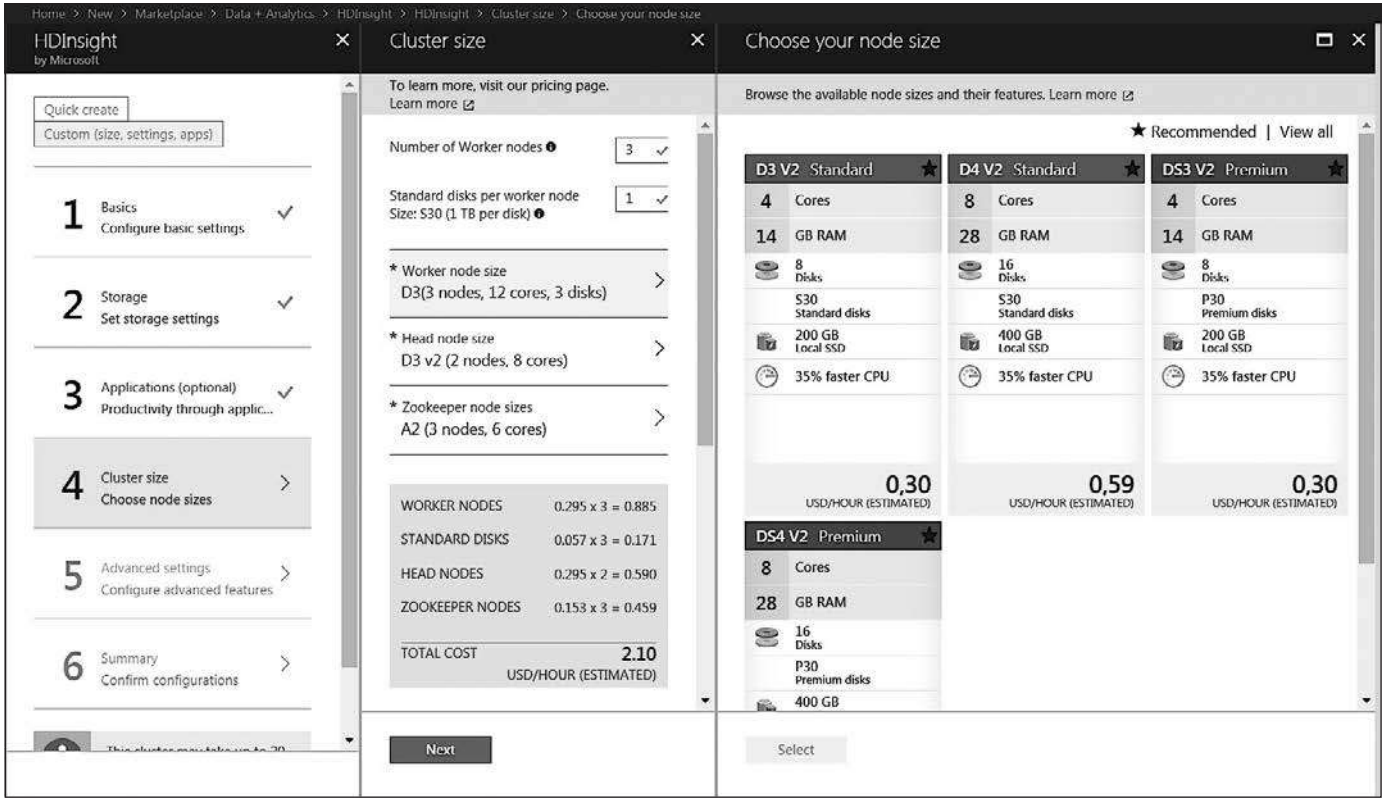


Рис. 10.62. Форма конфигурирования узлов кластера

После прохождения всех предыдущих шагов откроется форма с обзором создаваемого кластера (рис. 10.63).

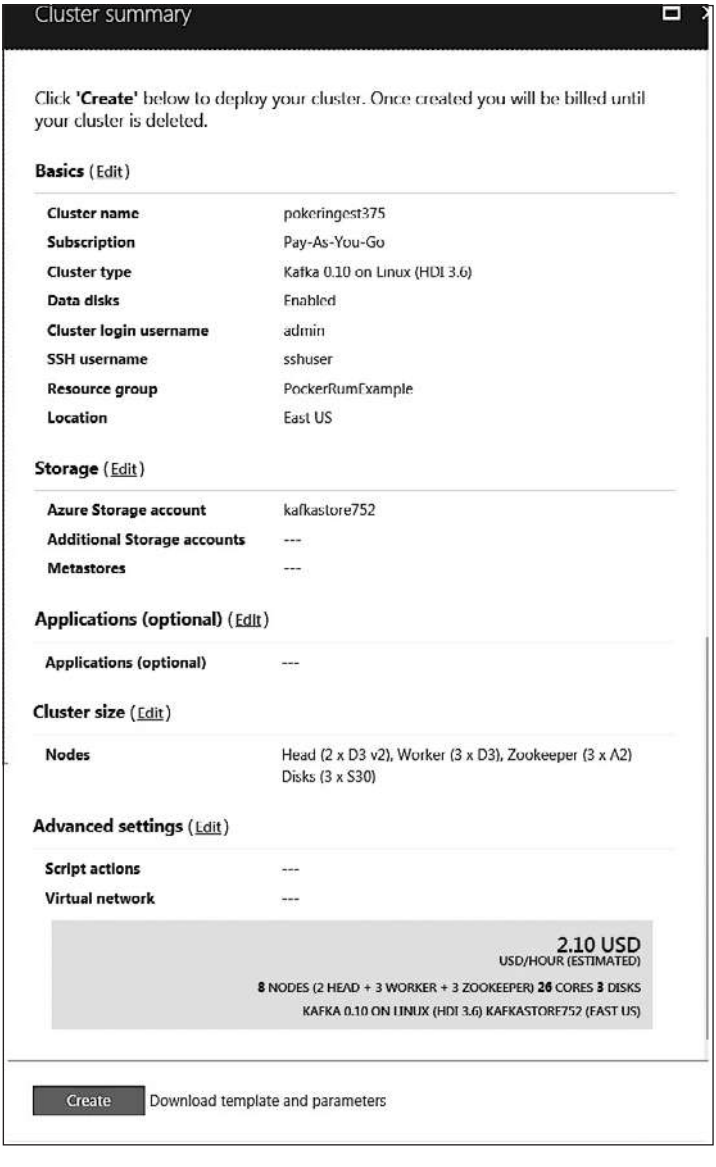


Рис. 10.63. Страница обзора создаваемого кластера

После создания кластера, а это может занять продолжительное время, доступна панель кластера HDInsight (рис. 10.64).

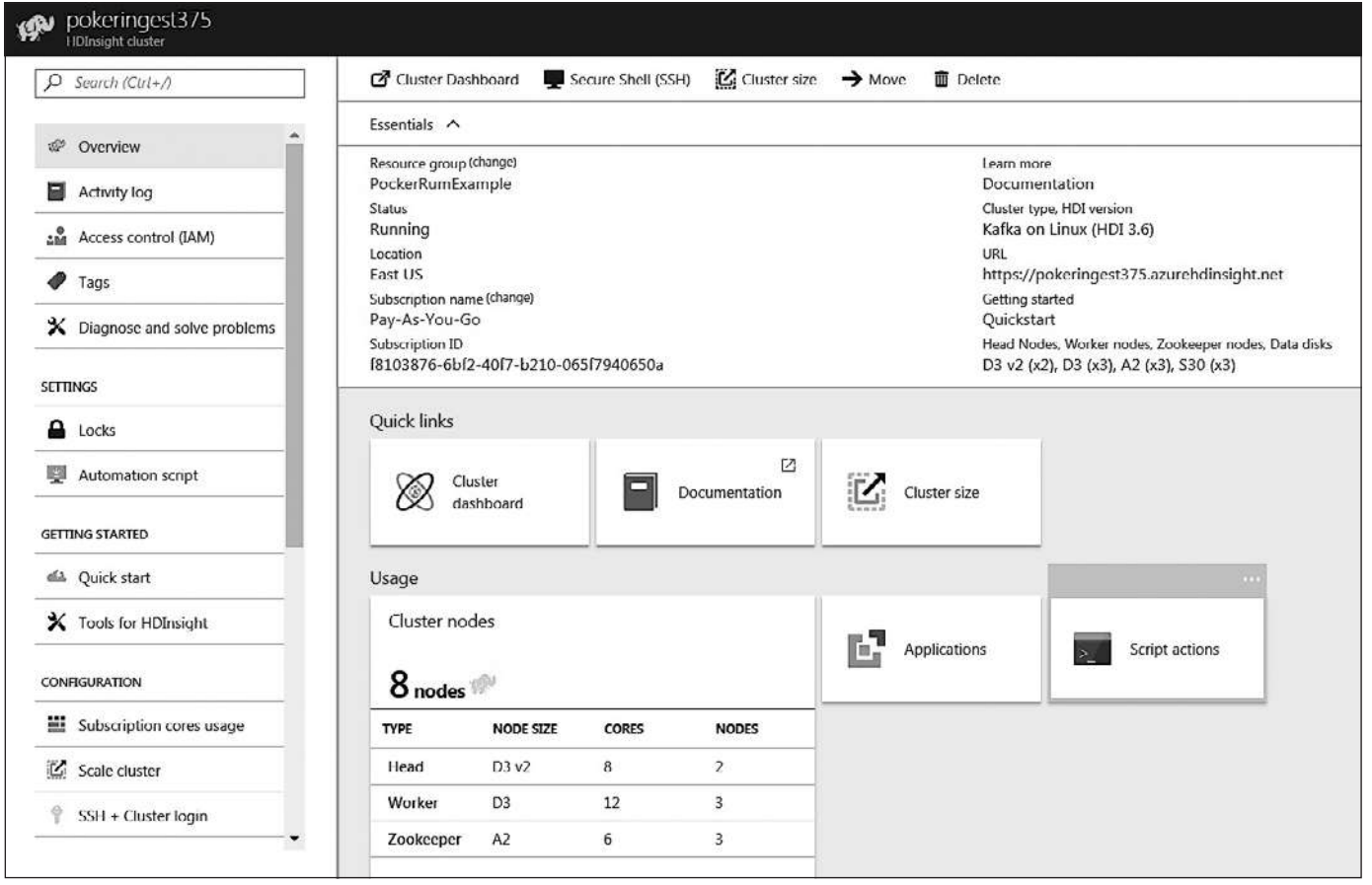


Рис. 10.64. Общая панель созданного кластера

Рассмотрим важные компоненты кластера. Компонент Script action позволяет выполнять bash-скрипты в кластерах. Компонент Applications дает возможность устанавливать дополнительные приложения, аналогичные указанным на рис. 10.61. Изменять размер кластера можно в компоненте Cluster size. Для перехода на главную страницу веб-интерфейса Zookeeper следует использовать вкладку Cluster dashboard или перейти напрямую по URL https://pockeringest375.azurehdinsight.hub. В появившейся форме ввода пароля нужно ввести учетные данные, которые были введены при создании кластера. В результате откроется стандартная страница Zookeeper (рис. 10.65).

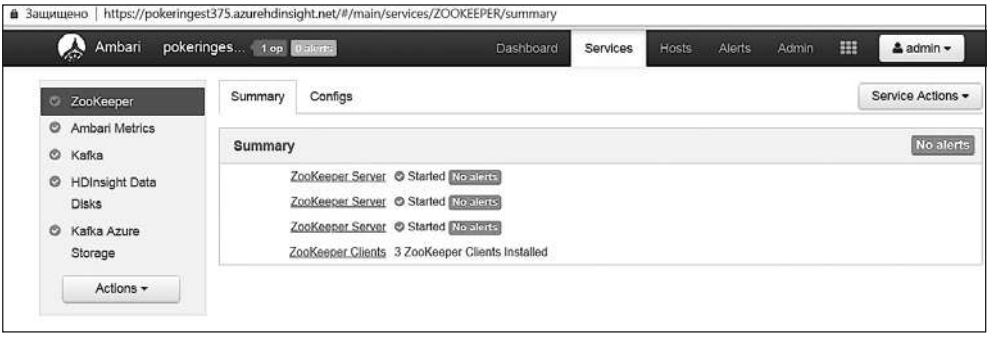


Рис. 10.65. Веб-интерфейс панели мониторинга кластера системы Zookeeper

Этот веб-интерфейс позволяет анализировать и конфигурировать кластер, а также выполнять в нем задания.

Ключевая особенность сервиса HDInsight — возможность получения доступа по SSH. Рассмотрим, как ее использовать. На общей панели кластера (см. рис. 10.64) следует нажать на ссылку Secure Shell (SSH). В результате откроется окно с данными, необходимыми для удаленного входа в систему (рис. 10.66).

После входа по SSH следует войти в терминал и продолжить конфигурирование. Конечно, это можно выполнить с помощью bash-скрипта, но мы проделаем все действия последовательно. Внешний вид панели входа имеет следующий вид (рис. 10.67).

После входа прежде всего нужно с помощью команды `sudo apt -y install jq` установить утилиту jq, которая используется с форматом JSON.

Затем следует задать переменные окружения:

- `CLUSTERNAME` — переменная, представляющая собой имя кластера. Задать ее можно с помощью команды `export CLUSTERNAME=pockeringest375`;
- `KAFKAZKHOST` — переменная, содержащая информацию для хоста Zookeeper. Задается с помощью такой команды:

```
export KAFKAZKHOSTS=`curl -sS -u admin -G \
https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/
services/ZOOKEEPER/components/ZOOKEEPER_SERVER | \
 jq -r '["\(.host_components[].HostRoles.host_name):2181"] | join(",")' |
cut -d',' -f1,2`
```

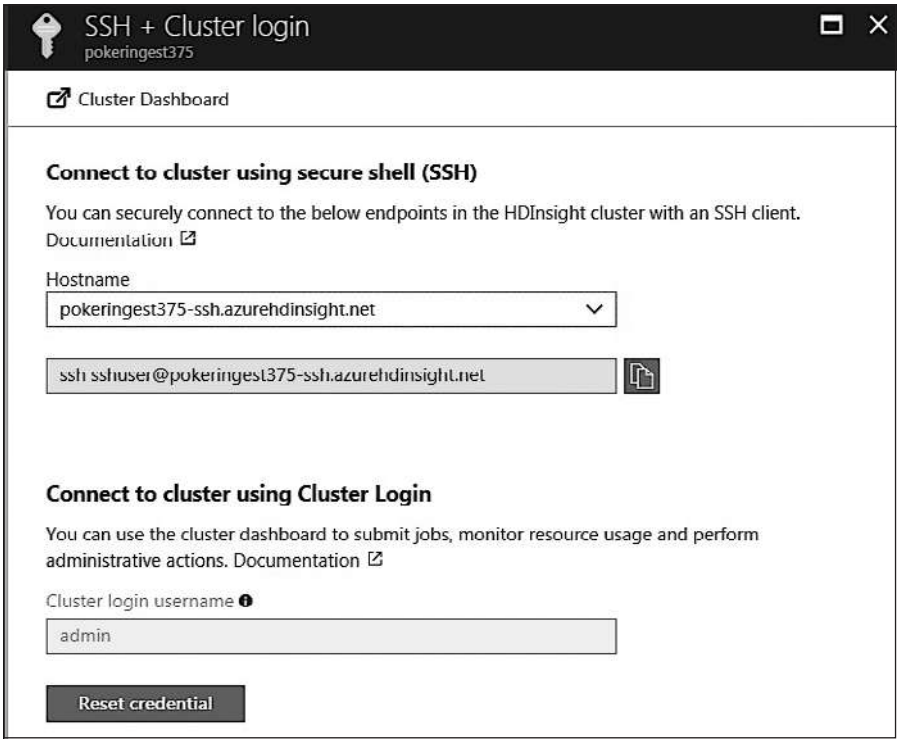


Рис. 10.66. Панель с инструкцией для входа в кластер по SSH

```
 * Support:        https://ubuntu.com/advantage

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

85 packages can be updated.
47 updates are security updates.



Welcome to Kafka on HDInsight.


The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

sshuser@hn0-pokeri:~$
```

Рис. 10.67. Внешний вид SSH-терминала кластера

После выполнения этой команды следует ввести пароль для пользователя admin, введенный ранее при конфигурировании кластера. Далее можно проверить правильность установки данной переменной, задействуя команду `echo '$KAFKAZKHOSTS='$KAFKAZKHOSTS`;

- KAFKABROKERS — переменная, содержащая информацию для хостов брокера. Задается с помощью следующей команды:

```
export KAFKABROKERS=`curl -sS -u admin -G \
https://$CLUSTERNAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/
services/KAFKA/components/KAFKA_BROKER | jq -r '["\(.host_components[].
HostRoles.host_name):9092"] | join(",")' | cut -d',' -f1,2`
```

Проверка новой переменной производится с использованием команды `echo '$KAFKABROKERS='$KAFKABROKERS`.

На рис. 10.68 представлены результаты выполнения этой команды, выведенные на терминал.

```
sshuser@hn0-pokeri:~$ read -p "Enter the HDInsight cluster name: " CLUSTERNAME
Enter the HDInsight cluster name:
sshuser@hn0-pokeri:~$ read -p "Enter the HDInsight cluster name: " CLUSTERNAME
Enter the HDInsight cluster name: pokeringest375
sshuser@hn0-pokeri:~$ export KAFKAZKHOSTS=`curl -sS -u admin -G https://$CLUSTER
NAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/services/ZOOKEEPER/componen
ts/ZOOKEEPER_SERVER | jq -r '["\(.host_components[].HostRoles.host_name):2181"]
| join(",")' | cut -d',' -f1,2`
Enter host password for user 'admin':
sshuser@hn0-pokeri:~$
sshuser@hn0-pokeri:~$ echo '$KAFKAZKHOSTS='$KAFKAZKHOSTS
$KAFKAZKHOSTS=zk0-pokeri.ef5qrlkmlixebkkb153wovp2vf.bx.internal.cloudapp.net:218
1,zk1-pokeri.ef5qrlkmlixebkkb153wovp2vf.bx.internal.cloudapp.net:2181
sshuser@hn0-pokeri:~$ export KAFKABROKERS=`curl -sS -u admin -G https://$CLUSTER
NAME.azurehdinsight.net/api/v1/clusters/$CLUSTERNAME/services/KAFKA/components/K
AFKA_BROKER | jq -r '["\(.host_components[].HostRoles.host_name):9092"] | join("
,")' | cut -d',' -f1,2`
Enter host password for user 'admin':
sshuser@hn0-pokeri:~$ echo '$KAFKABROKERS='$KAFKABROKERS
$KAFKABROKERS=wn0-pokeri.ef5qrlkmlixebkkb153wovp2vf.bx.internal.cloudapp.net:909
2,wn1-pokeri.ef5qrlkmlixebkkb153wovp2vf.bx.internal.cloudapp.net:9092
sshuser@hn0-pokeri:~$
```

Рис. 10.68. Внешний вид терминала с выводом на экран установленных переменных

Далее следует создать топик, используя команду, приведенную ниже:

```
/usr/hdp/current/kafka-broker/bin/kafka-topics.sh --create --replication-factor 3 \
 --partitions 8 --topic poker_consumer --zookeeper $KAFKAZKHOSTS
```

С помощью этой команды мы создаем топик с троекратным реплицированием, состоящий из восьми разделов с именем `poker_consumer`. Чтобы посмотреть список доступных топиков, следует выполнить команду `/usr/hdp/current/kafka-broker/bin/kafka-topics.sh --list --zookeeper $KAFKAZKHOSTS`.

Далее можно работать с этим топиком: посылать в него сообщения и читать их из него. Для уточнения деталей построения производителя и потребителя Kafka

с помощью различных программных средств я рекомендую обратиться к документации Microsoft Azure HDInsight и Apache Kafka.

Итак, сервис HDInsight позволяет автоматизировать создание кластеров с установленными средствами из экосистемы Apache Hadoop максимально автоматизированным способом. Но может случиться так, что и этого будет недостаточно. Например, требуется установить в кластере сервис из экосистемы Apache Hadoop, который отсутствует в HDInsight. Для этого можно применить напрямую кластер виртуальных машин, сконфигурированных вручную пользователем (что весьма непросто) или кластер, где развернуты сервисы Docker, в которые и помещены необходимые программы. В настоящее время Docker — технология, стремительно набирающая популярность, в том числе для хостинга приложений различного характера. Она одинаково хорошо подходит для построения как микросервисов, так и монолитных приложений. Физический кластер серверов позволяет реплицировать экземпляры сервисов среди узлов кластера и строить отказоустойчивые системы. Обратите внимание: архитектуры микросервисов на базе Docker и бессерверных приложений с использованием паттерна Event Driven принципиально отличаются, однако обсуждение этого вопроса лежит вне плоскости тематики книги.

Итак, Docker — платформа, разработанная для автоматизации процессов развертывания и управления приложениями в виртуализованной среде на уровне операционной системы. При этом само приложение вместе со всеми зависимостями и окружением можно перенести в любую Linux-систему (сейчас это возможно и для Windows) и работает совершенно единообразно в различных средах — будь то локальная машина разработчика или продакшен-сервер.

Итак, традиционно виртуальные машины работают поверх гипервизоров 1-го и 2-го типа. На рис. 10.69 слева показан гипервизор 1-го типа, поверх которого запускаются виртуальные машины, вмещающие операционные системы, внутри которых содержатся библиотеки и запускаются приложения. Гипервизор 2-го типа запускается поверх операционной системы, но виртуальные машины по-прежнему содержат гостевые операционные системы. Таким образом, переносимые образы виртуальных машин включают полноценные операционные системы и, как следствие, занимают довольно большой объем и требуют много времени для запуска.

Рис. 10.69. Сравнение концепций виртуальной машины и Docker

В отличие от этого контейнеры Docker запускаются с помощью платформы Docker, работающей поверх операционной системы. Но ключевое отличие таких контейнеров — они используют виртуализацию на уровне ядра операционной системы и не содержат собственных операционных систем. Таким образом, в отличие от других технологий виртуализации, контейнеры очень «легковесны», и процесс их переноса и запуска происходит очень быстро и просто.

Рассмотрим основные концепции данной технологии. Прежде всего, это *образ* (image) — предназначенный для исполнения пакет, включающий все необходимое для запуска приложения (среда исполнения, библиотеки, конфигурационные файлы, переменные окружения). Образы можно построить с помощью базовых образов Docker, загруженных из внешних репозиториев, приложения и специального файла, описывающего, как должен быть построен образ. Далее этот новый образ следует поместить в *репозиторий Docker* (Docker registry) локально на той машине, где собирается образ, или переместить во внешний репозиторий. Затем данный образ можно извлечь из репозитория и запустить на этом же или другом сервере с установленным Docker.

Таким образом, среда Docker предоставляет средства разработки и развертывания приложений. При разработке для построения образа (а это, по сути, шаблон, описание контейнера Docker) используются скомпилированные файлы, библиотеки и специальные файлы, относящиеся к Docker: `dockerfile`, `docker-compose.yml` и пр. На сервере разработки Docker, используя эти файлы, производит сборку (build) образа Docker. Из этого образа контейнер может быть запущен прямо на этом сервере или перемещен в репозиторий (рис. 10.70).

Вообще репозиторий представляет собой хранилище образов Docker, которое поддерживает команды Docker Push, Pull и др., относящиеся к образам. Как правило, репозитории бывают трех видов:

- локальные — размещенные на тех же серверах, что и Docker, и устанавливающиеся вместе с ним;
- частные — закрытые репозитории, требующие регистрации пользователей и нередко имеющие сетевые ограничения доступа; предназначены для закрытой разработки и развертывания Docker;
- публичные — широкодоступные и широкоизвестные репозитории, обеспечивающие анонимный доступ для всех; к таковым относится, например, https://hub.docker.com/.

Итак, для запуска приложения в контейнере Docker необходимо иметь сервер с установленным Docker. Далее следует найти и скачать требуемый образ из публично доступного или частного репозитория (команда `docker pull <`*`имя образа`*`>`) и запустить контейнер на сервере (команда `docker run <`*`имя образа`*`>`).

Теперь рассмотрим, как работает Docker в случае кластерных систем. В сложных кластерных системах, состоящих из многих элементов, каждые логически обособленные и однородные компоненты именуются *сервисами* (services). Например, в случае Kafka мы имеем два сервиса: сервис с головным узлом (сервис Zookeeper)

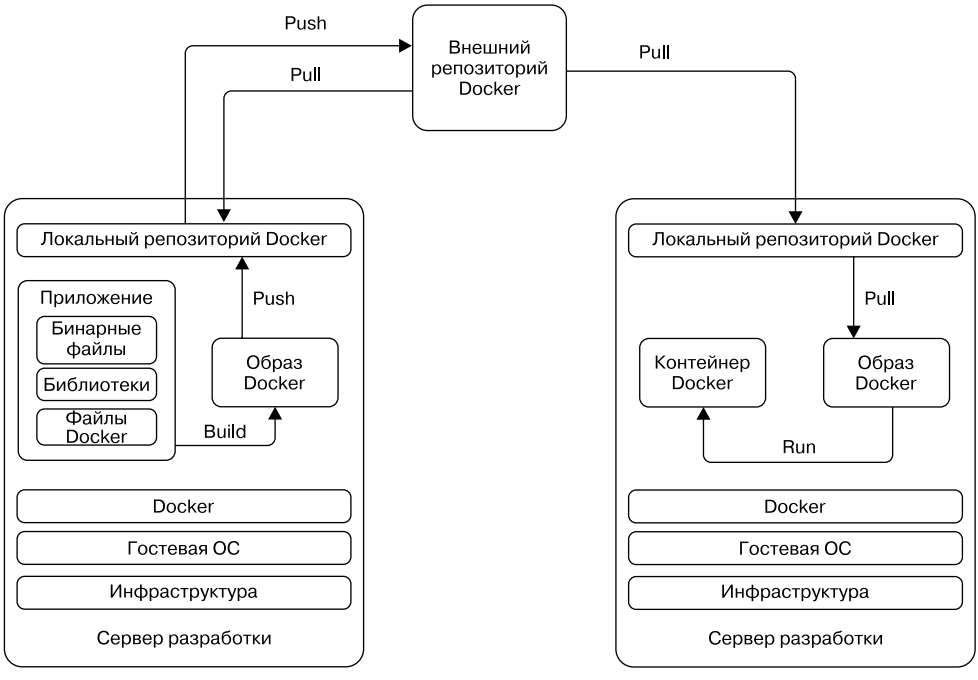


Рис. 10.70. Процесс разработки и развертывания приложений с использованием Docker

и сервис исполнительных узлов (сервис брокера). В традиционной многослойной архитектуре это сервисы пользовательского интерфейса, бизнес-логики и доступа данных. В качестве физической основы хостинга выступает группа серверов, в терминологии Docker именуемых *роем* (swarm), в которой будут запущены сервисы. Каждый сервер из данной группы должны иметь установленный экземпляр Docker и быть присоединенным к рою. Все вместе — рой серверов и группы сервисов, управляемые совместно, — именуются *стеком* (stack) (рис. 10.71).

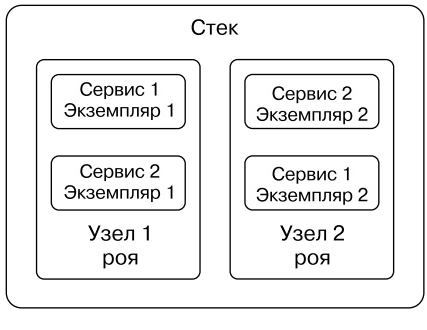


Рис. 10.71. Компоненты стека Docker

Вся эта архитектура описывается с помощью специального файла — `docker-compose.yml`. Он содержит описание сервисов, а именно количества запущенных реплик, балансировки нагрузки, распределения этих экземпляров по кластеру. Для создания образов и распределения их по стеку служат специальные приложения Docker, например Docker Compose. Файл `docker-compose.yml`, помимо описания структурных величин (скажем, количества реплик сервиса), содержит описание численных величин, относящихся к распределению физических ресурсов между экземплярами сервисов (допустим, количеству оперативной памяти). А вся система в целом позволяет автоматизировать процесс развертывания Docker в кластере. Масштабирование кластера при этом происходит за счет изменения файла `docker-compose.yml` и его передеплаивания.

Это очень мощный механизм, не лишенный тем не менее ряда недостатков: в нем нет удобного централизованного мониторинга потребления ресурсов экземплярами сервисов, нет средств автоматического масштабирования и перераспределения ресурсов кластера между экземплярами сервисов. Чтобы заполнить этот пробел, существуют системы управления контейнерами и кластерами как единым целым, значительно расширяющие базовую функциональность Docker Swarm. К таковым можно отнести Kubernetes — проект с открытым исходным кодом, разработанный первоначально в Google и поддерживаемый многими компаниями и облачными провайдерами. Теперь он входит в состав Azure (сервис AKS) и AWS EKS (Elastic Kubernetes Service). Помимо этого, AWS предоставляет отдельный сервис — AWS ECS (Elastic Container Service). Подобная интеграция облачного провайдера и системы управления контейнерами добавляет новые возможности в плане объединения кластера с облачными сервисами логирования, мониторинга, масштабирования, безопасности.

Когда тот или иной сервис недоступен на Docker, можно автоматически развернуть его на сконфигурированных как кластер виртуальных машинах, применяя шаблоны автоматизации: AWS CloudFormation или Azure ARM Template. Как правило, для популярных сервисов готовые шаблоны предоставляет сам облачный провайдер или их можно создать. В частности, в основе HDInsight лежит эта технология — применяются ARM-шаблоны. Но ее описание выходит далеко за рамки тематики книги.

10.5. Резюме

В данной главе были рассмотрены сервисы концентраторов сообщений различных вариантов, как PaaS-сервисы (Azure Event Hub и AWS Kinesis Data Streams), так и на базе решений IaaS/PaaS (HDInsight). Кроме того, кратко описана альтернативная технология построения кластерных систем на основе Docker. Обобщим области применения и преимущества/недостатки этих систем.

С точки зрения гибкости, почти неограниченной масштабируемости и расширяемости наиболее подходящий вариант — построение системы на основе прямой установки Kafka в IaaS-кластер. Однако построение системы таким образом может

быть весьма сложным в плане первоначального конфигурирования и настройки (в зависимости от размера кластера установка и конфигурирование с нуля может занимать от нескольких недель до нескольких месяцев). Упростить процедуру первоначальной настройки и конфигурирования позволяют сервисы IaaS/PaaS, такие как Azure HDInsight или AWS EMR. С их помощью можно относительно легко создавать и удалять кластеры с сервисами Apache Hadoop. Но и в данном случае окончательное конфигурирование системы может занимать дни.

Самый быстрый и простой способ создания сервисов этого типа — использование PaaS-сервисов AWS Kinesis Data Streams или Azure Event Hub. Их создание и конфигурирование занимает минуты. При этом в большинстве случаев возможностей PaaS-сервисов вполне достаточно. Таким образом, при необходимости быстро и «прямо сейчас» иметь мощную и масштабируемую систему приема сообщений AWS Kinesis Data Streams или Azure Event Hub — ваш выбор.

Теперь сравним эти два сервиса с точки зрения предоставляемых возможностей. Совместно с Kinesis Analytics и Firehouse концентратор AWS Kinesis Data Streams образует единую платформу обработки, анализа и доставки сообщений. У Azure таких сервисов два: Azure Event Hub и Azure Stream Analytics. Удобство AWS по сравнению Azure состоит в том, что сообщения из концентратора могут быть доставлены в бо́льшее количество источников: AWS RedShift, AWS S3, Amazon ES и Splunk. В случае Azure Event Hub такой сервис прямой доставки работает только с Azure Blob Storage и Azure Data Lake Store. Однако Azure Stream Analytics имеет много выходных каналов для помещения сообщений в различные источники (подробнее см. в следующей главе). Очень существенным различием этих сервисов является то, что количество разделов в Azure Event Hub неизменяемо после создания, в то время как в AWS Kinesis Data Streams это величина, настраиваемая после ее создания.

11 Облачные сервисы копирования и трансформации данных

Конвейер движется, и одна за другой с него сходят превосходные и дешевые машины. Они выезжают через широкие ворота в мир, в прерию, на свободу. Люди, которые их сделали, остаются в заключении. Это удивительная картина торжества техники и бедствий человека.

Ильф и Петров. Одноэтажная Америка

11.1. Общие понятия

В предыдущих главах и частях вы познакомились с сервисами хранения данных и рассмотрели различные способы доставки данных в облачные хранилища. Отмечалось, что каждое хранилище приспособлено для своих целей. Например, Azure Table Storage очень удобно для хранения данных телеметрии, метрики, записей логов и пр. В предыдущей части мы рассмотрели пример использования нереляционной документоориентированной базы данных для хранения игровых событий. Они представляют собой JSON-документы. В реальной информационной системе, помимо этого, существуют источники данных других типов: реляционные и графовые базы, CSV-файлы в хранилище файлов общего назначения и др.

Для того чтобы сервисы аналитики могли проанализировать все эти данные, их нужно поместить в специализированное централизованное хранилище, допускающее такой анализ. Традиционно таковым является реляционное хранилище DWH. Оно позволяет применять эффективные высокопроизводительные запросы к данным из разных источников, приведенным к единообразной форме таблицы

в DWH. Такой подход копирования данных из разных источников называется ETL (Extract Transform Load — «извлечение, преобразование, загрузка»). Данные надо извлечь из источника, преобразовать (отфильтровать, агрегировать, извлечь подмножество полей) и загрузить в центральное хранилище.

Для примера системы онлайн-покера следует извлечь данные из нереляционной БД игровых событий и базы игровых ситуаций. Поскольку данные в этом хранилище существенно денормализованные, то для помещения их в реляционное хранилище из этих данных нужно только их подмножество. Состав этого подмножества зависит от того, что необходимо предоставить в хранилище. Например, для анализа статистики выигравших и проигравших потребуется извлекать идентификаторы игроков, игр и суммы ставок.

Следующий пример — система мониторинга электроэнергии, в которой первичные данные (например, величины конкретных отсчетов) хранятся сначала в таблице «ключ — значение». Затем эту информацию требуется агрегировать (подсчитать энергопотребление для каждого потребителя за установленный временной промежуток) и поместить ее в реляционное хранилище. В то же время данные пользователей, данные, связанные с финансовыми аспектами (например, данные CRM), хранятся в реляционных БД. Чтобы провести анализ данных из нереляционных БД, связав их с конкретным пользователем или клиентом, необходимо скопировать в DWH также и эти данные.

Допустим, требуется провести более сложный интеллектуальный анализ данных, например связанный с анализом подозрительных шагов или действий в покер-руме. Для этого, помимо данных из хранилища игровых событий, реляционных данных о пользователях, необходимо использовать данные логов приложений. В подобном случае наиболее подходящий вариант — применение нереляционного хранилища данных, такого как Data Lake. В этом случае файлы логов просто копируются, как и данные из нереляционной и реляционной баз данных, а преобразование выполняется уже после копирования. Такой подход называется ELT (Extract Load Transform — «извлечение, загрузка, преобразование»). В подобной ситуации необходимо использовать Data Lake, поскольку в настоящее время самые мощные системы интеллектуального анализа данных представлены в основном сервисами из экосистемы Apache Hadoop, что требует HDFS-совместимого хранилища, коим и является Data Lake.

Таким образом, очевидно, что сервисы копирования и трансформации данных, которые переносят их из одного хранилище в другое, подвергая преобразованию, — очень важная составляющая информационной системы. Выше уже упоминались два сервиса из этой области — Apache Sqoop и DistCp. Но они требуют кластерных решений типа Azure HDInsight и AWS EMS. Эти сервисы предоставляют «ядро» или «движок» копирования и трансформации, однако конвейер данных может состоять из многих этапов копирования и преобразования информации. Управление этим конвейером берут на себя облачные сервисы копирования и трансформации данных: Azure Data Factory, AWS Data Pipeline и AWS Glue. Все эти сервисы будут рассмотрены в текущей главе.

11.2. Azure Data Factory

Azure Data Factory представляет собой сервис трансформации и копирования от Azure. В настоящее время реализована версия 2 данного сервиса, в то время как для заказчиков, применявших сервис ранее, доступна версия 1, отличающаяся от текущей. Но мы ограничимся рассмотрением версии 1, поскольку она является наиболее актуальной и единственно доступной для новых пользователей.

Итак, Azure Data Factory — сервис прежде всего для управления процессами копирования и трансформации данных. Как правило, этот сервис состоит из следующих компонентов (рис. 11.1).

- *Наборы данных* (dataset) — конкретные поименованные представления данных. В их качестве могут выступать файлы, таблицы и пр. Наборы данных могут быть как входными, называемыми *источниками* (source), так и выходными, называемыми *воронками* (sink).
- *Подключенные серверы* (linked servers) — наборы метаданных источников (адреса, учетные данные, протоколы и др.), на которых располагаются источники и приемники данных.
- *Наборы выполняемых заданий* (activity) — одно или несколько выполняемых заданий по копированию или трансформации данных.
- *Конвейер* (pipeline) — набор сервисов по обеспечению периодического выполнения, управления и мониторинга, логически соединенный с подключенными серверами, наборами данных и заданиями для выполнения конкретной задачи.

Рис. 11.1. Архитектура сервиса Azure Data Factory

Теперь рассмотрим, как все компоненты работают и взаимодействуют между собой. Прежде всего, сервис Azure Data Factory состоит из одного или нескольких *конвейеров*. Каждый конвейер реализует конкретную задачу ETL/ELT, для чего ему нужны источники и приемники данных, собственно список выполняемых

заданий, триггеры (то есть условия начала выполнения) и набор входных параметров, подаваемых на вход.

В настоящее время в качестве источников данных Azure Data Dactory выступают хранилища, как облачные (в том числе и в иных облачных средах, например AWS), так и локальные: реляционные и нереляционные БД, файлы (текстовые, CSV-, HTML-, JSON- и XML-таблицы), хранящиеся в различных файловых системах и доступные по разным протоколам (FTP, SFTP, HDFS, AWS S3), реляционные и нереляционные хранилища данных, сервисы REST и SOAP (в том числе поддерживающие протокол OData), SAP, Dynamic CRM, Office 365 и др.

В то же время количество приемников данных в значительной степени зависит от того, какой *исполняемый режим* (runtime) выбран для Azure Data Factory. По умолчанию конвейер управляется ресурсами Azure и для пользователя выглядит как бессерверный. Такой режим называется *встроенным исполняемым режимом* (integrated runtime, IR). В нем пользователю не требуется выполнять никаких действий по администрированию и мониторингу конвейера — все нужные вычислительные ресурсы Azure создает автоматически. Но в этом случае перенос информации ограничивается облачными источниками и приемниками данных, расположенными в публично доступных сетях.

Чтобы преодолеть это ограничение, задействуется вариант *режима внешнего хостинга* (self-hosted) — размещения исполняемого модуля Azure Data Factory на физическом сервере или виртуальной машине. В таком случае применимы все возможности по копированию и переносу данных между любыми источниками и приемниками из доступного набора. Однако пользователь отвечает за установку и администрирование хоста для исполняемого модуля.

Кроме того, существует также очень интересный промежуточный вариант — *интегрированный исполняемый режим SSIS* (SSIS IR). Он позволяет с помощью Azure Data Factory запускать пакеты SSIS, для чего могут служить как выделяемые машины с установленными Microsoft SQL, так и пользовательские SQL-серверы (обратите внимание: эта возможность сейчас находится в стадии Preview).

Вернемся к остальным компонентам Azure Data Factory. Важнейший и ключевой компонент — *выполняемое задание* (activity). Существуют три вида заданий: перенос данных, их трансформация и поток управления (control flow). Рассмотрим их по порядку.

Перенос данных служит для копирования данных из источника в приемник без выполнения каких бы то ни было трансформаций, агрегаций и пр. Это простое копирование из источника в приемник. Оба — источник и приемник — должны быть представлены как наборы данных (data set) в подключенных серверах (linked server). Как раз для подобной активности мы и рассмотрели различные исполняемые режимы (IR, SSIS IR, self-hosted).

Трансформация данных подразумевает применение вычислительных ресурсов кластера, выделяемого по требованию (Azure HDInsight), или ресурсов, выделяемых пользователем. В настоящее время преобразовывать данные позволяют сле-

дующие технологии: HDInsight — Hive, Pig, Spark, Hadoop Streaming, MapReduce; Machine Learning; хранимая процедура в базе данных SQL или хранилище данных; U-SQL-сценарий Azure Data Lake Analytics и пользовательская программа, написанная на .NET. При трансформации возможны два исполняемых режима: с выделением ресурсов по требованию (например, кластер Azure HDInsight) или на ресурсах, предоставляемых пользователем (на его виртуальных машинах). Как видим, преобразование в этом случае может применяться в сценариях ETL/ELT, для выполнения пакетных, а также периодических административных заданий (например, запуск хранимой процедуры в Azure DWH для пересчета индексов). И Azure Data Factory при этом играет роль дирижера (orchestrator).

И наконец, очень важным является *поток управления*, который служит для управления другими заданиями, используя условные, циклические и прочие управляющие задания-операторы. В настоящее время поддерживаются следующие типы заданий:

- задание условного логического оператора ветвления IF (IF condition activity); работает по тому же принципу, что и логический оператор if в языках программирования: если логическое условие является истинным (true), то выполняется соответствующее задание А, в противном случае выполняется задание Б;
- задание условного цикла с предусловием Until (Until condition activity); представляет собой реализацию оператора повторения выполнения задания до тех пор, пока условие, ассоциированное с этим циклом, истинно или не будет достигнут тайм-аут;
- задание по выполнению REST-запроса (web activity);
- циклическое задание с фиксированным количеством шагов (for each activity) — оператор повторения задания определенное число раз;
- задание ожидания (wait activity) обеспечивает задержку выполнения основного задания на определенное время;
- задание последовательного перебора (lookup activity) может служить для чтения и последовательного просмотра записей в таблицах и файлах из внешних источников;
- задание по получению метаданных (get metadata activity) используется для получения метаданных от подключенных источников;
- задание активации другого конвейера данных (execute pipeline activity).

Кроме того, Azure Data Factory содержит достаточно богатый доменно-специфичный «язык» из богатого набора операторов, переменных (определяемых пользователем), функций и возможность строить различные выражения из этих элементов, обеспечивая довольно гибкое управление заданиями. Все компоненты Azure Data Factory можно описать с помощью JSON, и для наиболее гибкого конфигурирования данного сервиса я рекомендую использовать именно JSON, но для

этого лучше обратиться к официальной документации, поскольку данное описание довольно сложное.

Теперь рассмотрим вопрос запуска конвейера данных. Собственно, запуск включает передачу на вход конвейера параметров и активацию соответствующей конечной точки, будь то REST или точка запуска, определяемая в триггере конвейера. При этом каждый исполняемый конвейер получает уникальный идентификатор и становится доступным для средств мониторинга. Запустить конвейер можно с помощью:

- команды Azure CLI или Azure PowerShell;
- прямого REST-запроса к конечной точке запуска;
- триггера, определяемого в JSON-файле описания конвейера.

Обратите внимание: в настоящий момент Azure Data Factory не поддерживает запуск по событию и *прямую* интеграцию, например, с Azure Function (по крайней мере на момент написания книги такой возможности нет), Azure Logic App и др., но можно построить архитектуру Event Driving *косвенно*, через активизацию конечной точки REST.

Единственно поддерживаемый вид запуска, управляемый напрямую Azure Data Factory, — это триггер, запускаемый по расписанию. Чтобы быть точным, триггер — внутренний сервис Azure Data Factory, который определяет момент времени и периодичность запуска конвейера. Один триггер может служить для запуска нескольких конвейеров. И наоборот, один конвейер может быть запущен несколькими различными триггерами.

Теперь рассмотрим, как создавать и конфигурировать сервис Azure Data Factory с помощью веб-портала Azure. В качестве примера возьмем задачу копирования данных из таблицы Azure Table Storage в реляционную БД Azure SQL. Концептуально это ничем не отличается от задачи копирования информации в реляционное хранилище DWH, но по финансовым затратам гораздо ниже.

Итак, сначала следует нажать ссылку добавления новых ресурсов веб-портала управления и ввести в строке поиска `Data Factory` (рис. 11.2).

После нажатия кнопки Create (Создать) откроется форма конфигурирования сервиса (рис. 11.3). Здесь указываются имя сервиса (Name), подписка Azure (Subscription), ресурсная группа (Resource Group), версия (Version) и местоположение (Location). Как уже отмечалось, доступны две версии, причем V2 находится в стадии Preview, но с наибольшей долей вероятности станет широкодоступной, поэтому мы ее и рассмотрим.

Завершив создание этого сервиса, можно перейти на главную панель мониторинга (рис. 11.4).

Но сама по себе эта панель интересна только с точки зрения мониторинга. Чтобы настроить Azure Data Factory, следует перейти по внешней ссылке Author & Monitor. В результате откроется внешняя страница конфигурирования вашего экземпляра Azure Data Factory (рис. 11.5).

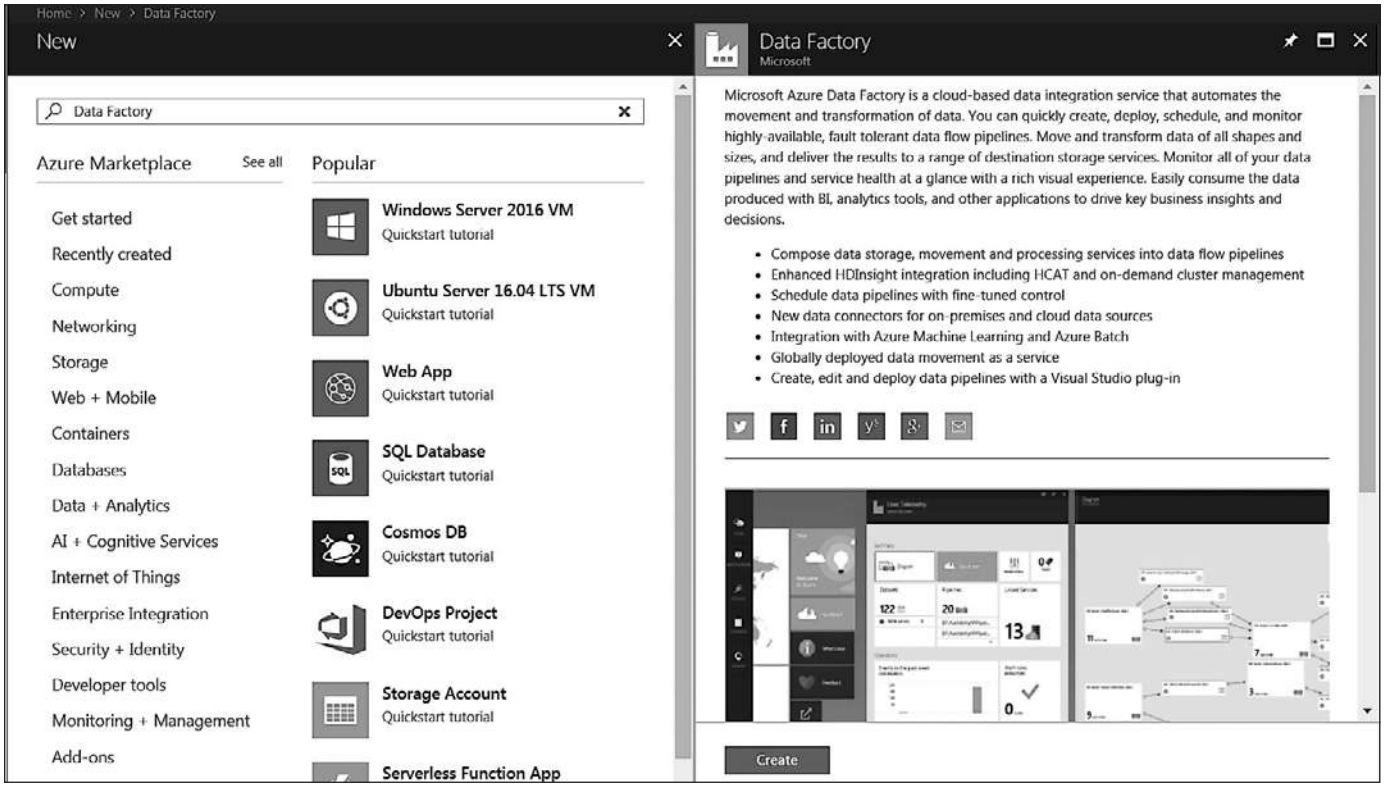


Рис. 11.2. Первоначальная страница создания сервиса Azure Data Factory

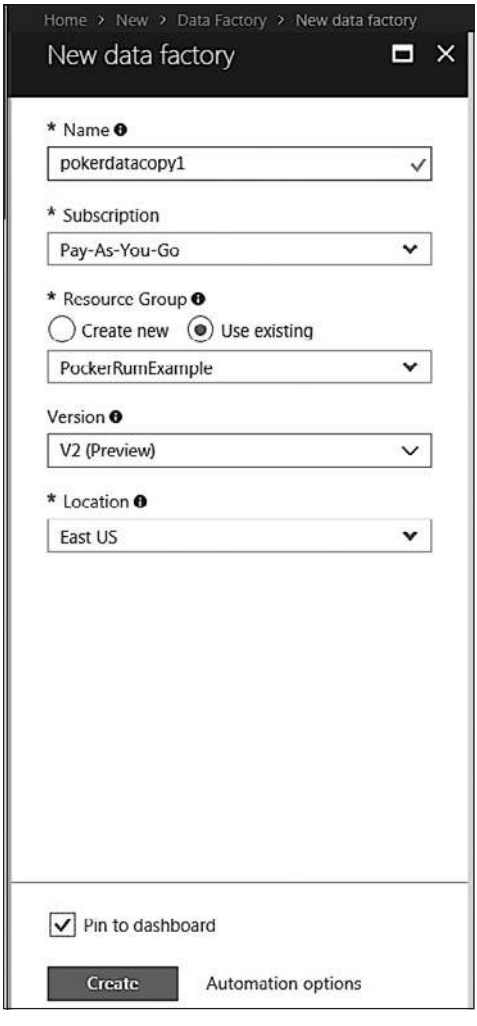


Рис. 11.3. Форма конфигурирования сервиса Azure Data Factory

Прямо на ней имеются ссылки на создание конвейера данных (Create pipeline), копирование данных (Copy Data), настройку интеграции с SSIS (Configure SSIS Integration Runtime) и настройку интеграции с Git (Configure Git Repository). Пойдем длинным путем и создадим конвейер данных, выбрав ссылку Create pipeline. В результате увидим такую страницу (рис. 11.6).

Здесь мы видим все основные компоненты конвейера данных, включающие в себя активности копирования (Data Flow — Copy), HDInsight, поток управления и пр. Но прежде всего необходимо настроить наборы данных. Для этого следует нажать на ссылку Connections в левом нижнем углу. Появится следующая форма (рис. 11.7).

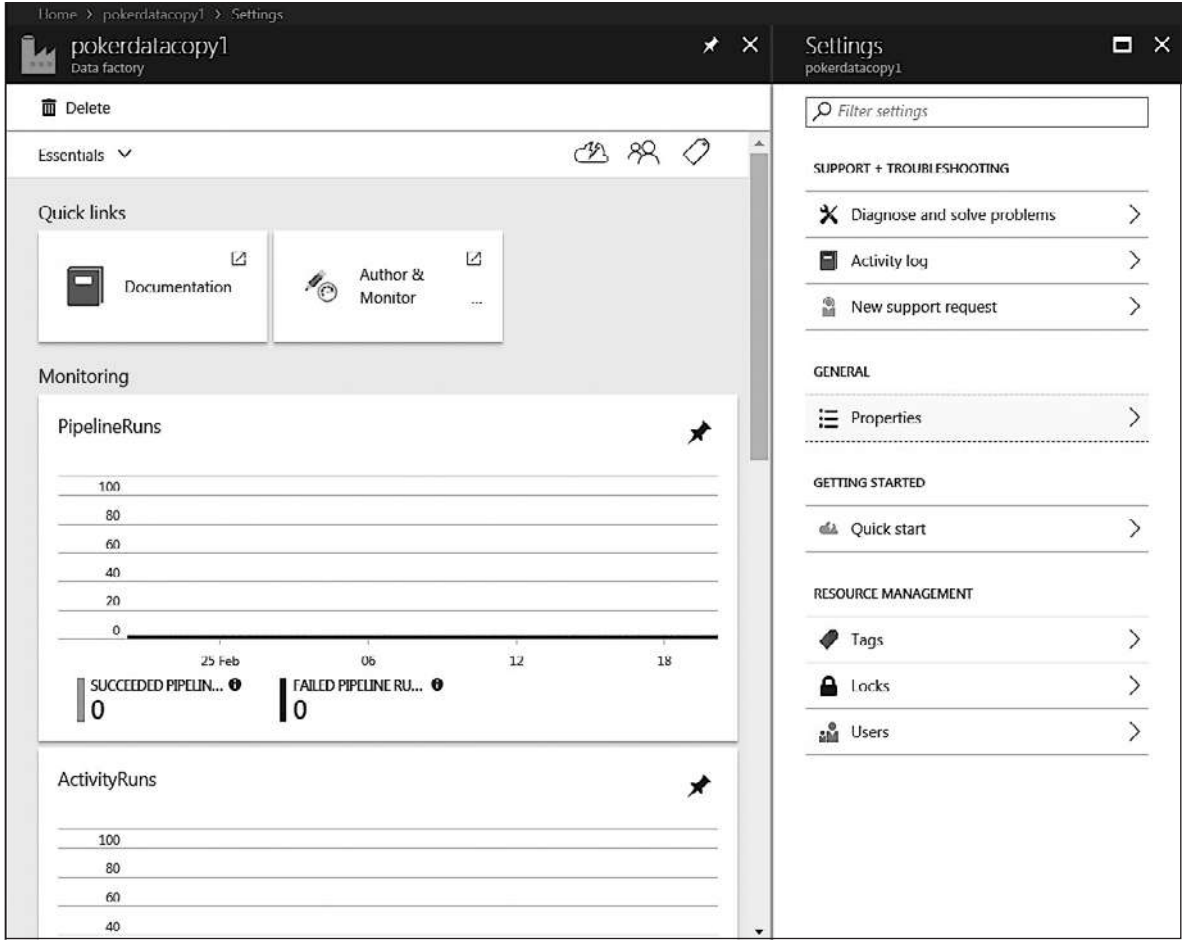


Рис. 11.4. Главная панель мониторинга сервиса Azure Data Factory

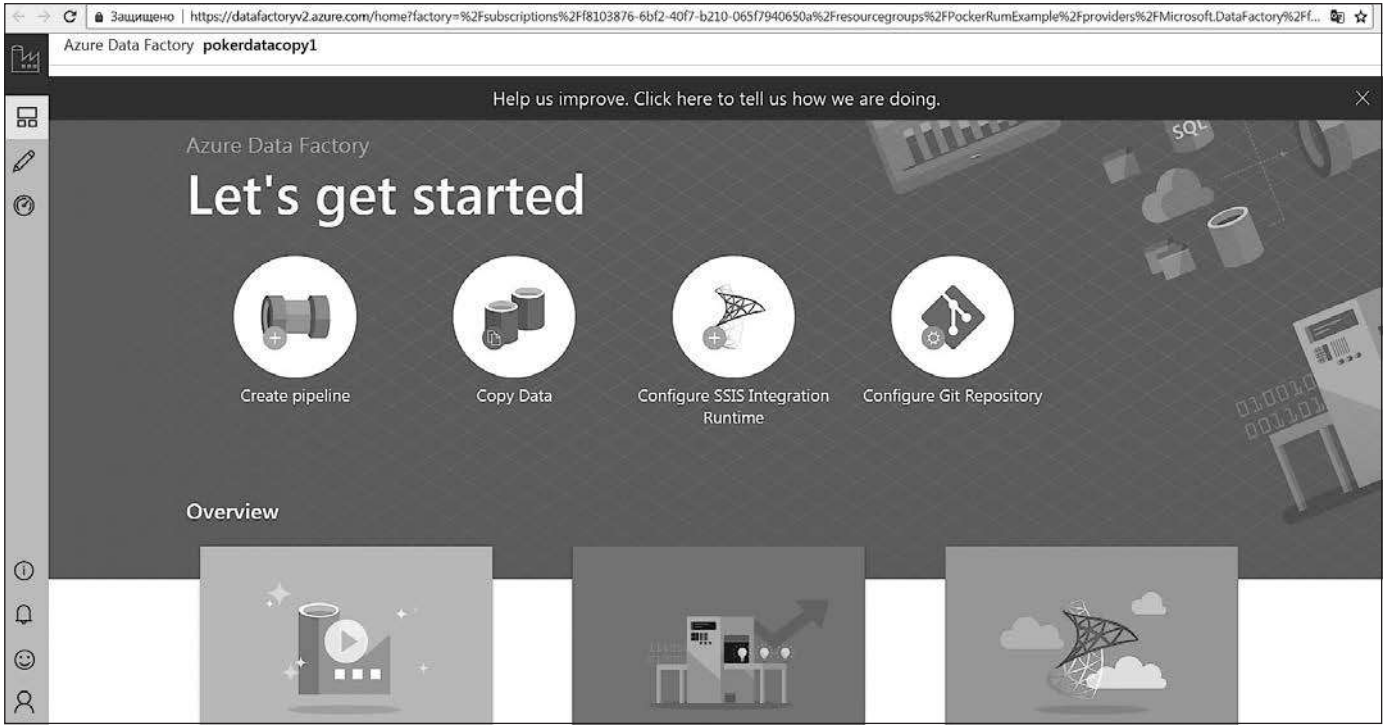


Рис. 11.5. Начальная страница конфигурирования сервиса Azure Data Factory

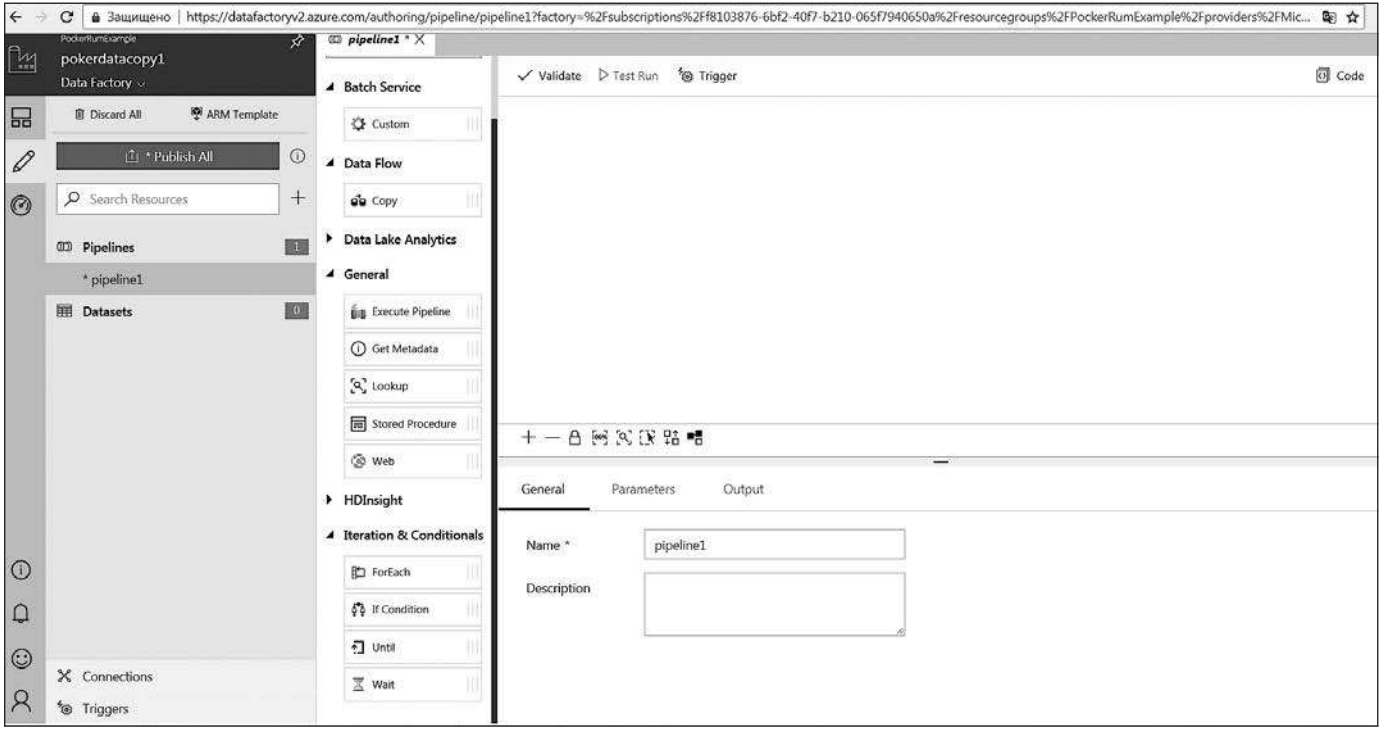


Рис. 11.6. Страница создания и конфигурирования конвейера данных

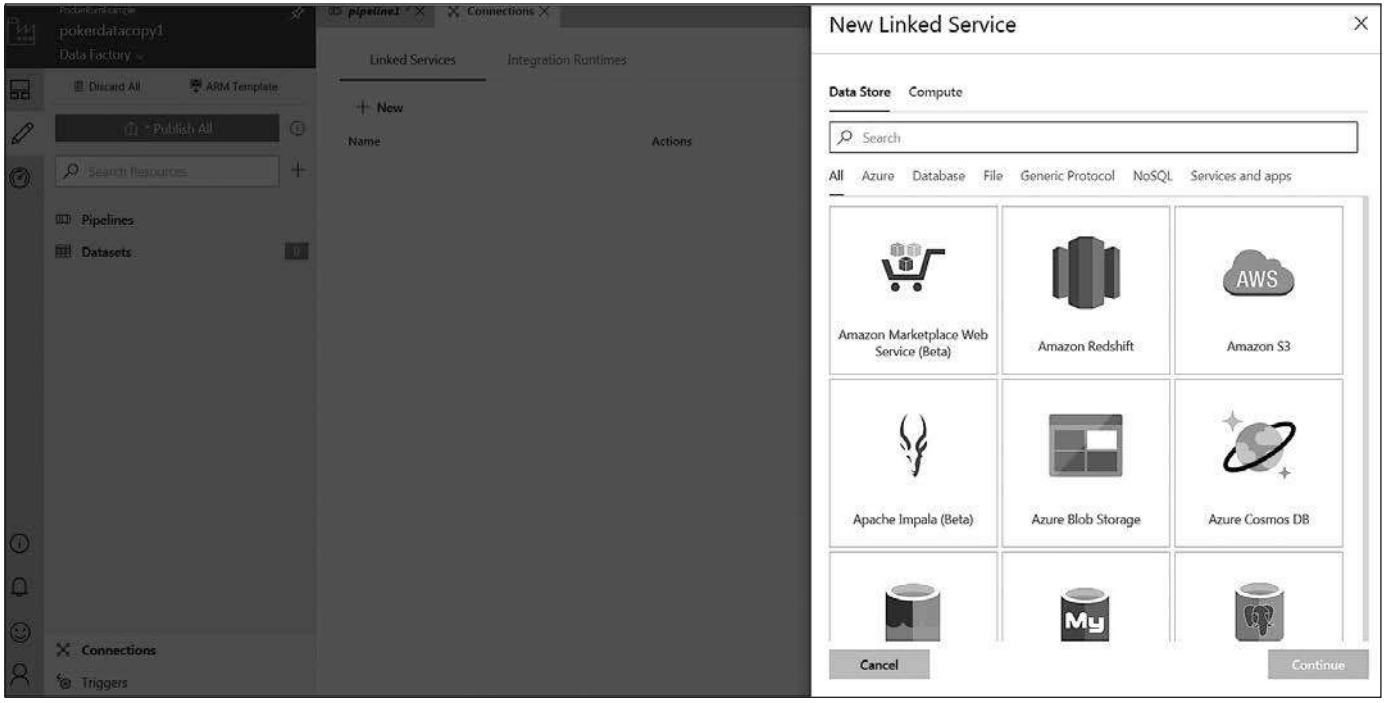


Рис. 11.7. Форма добавления подключенного сервера и настройки наборов данных

Нас интересует источник данных — таблица Azure Table Storage. Для этого в строке поиска следует написать `Azure Table Storage`, выбрать появившийся значок и нажать его. В результате появится форма конфигурирования (рис. 11.8).

Рис. 11.8. Форма конфигурирования подключенного сервера Azure Table Storage

Здесь указывается имя сервера (`PokerRowTelemetry`) и дается его описание (Description). Очень важный параметр — подключение через специальный исполняемый режим (Connect via integrated runtime). В данном случае следует выбрать режим по умолчанию (Default), поскольку этот подключенный сервис не требует настройки специализированного режима исполнения. Далее из подписки Azure (Azure subscription) нужно выбрать наш Azure Storage Account и проверить соединение с ним, нажав на ссылку Test connection. При успешной проверке соединения следует нажать кнопку Finish (Закончить).

Точно так же конфигурируется подключенный сервер Azure SQL (рис. 11.9).

Созданные подключенные серверы можно посмотреть на вкладке Connections (рис. 11.10).

Далее следует перейти на вкладку pipeline1 и вручную перетянуть графический компонент выполняемого задания DataFlow — Copy на общую панель конвейера (рис. 11.11). Нужно дать имя выполняемому заданию, установить тайм-аут (оставим

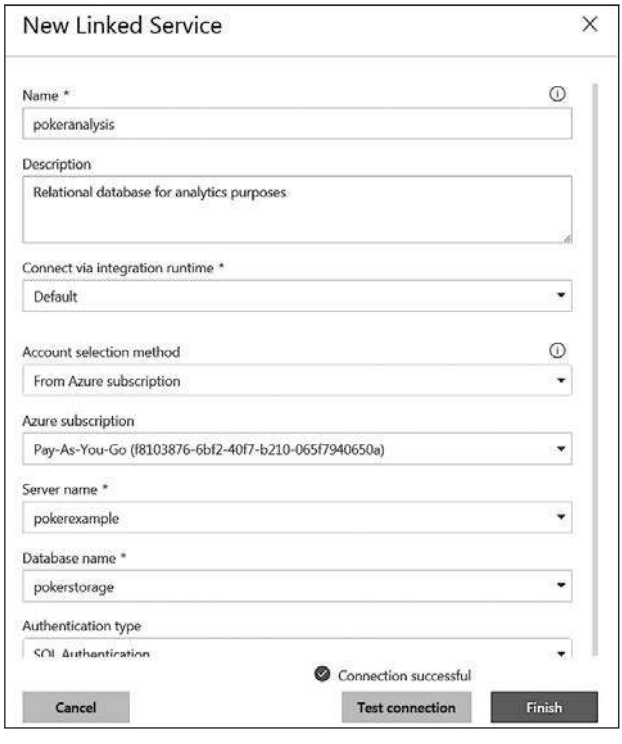


Рис. 11.9. Конфигурирование подключаемого сервера Azure SQL

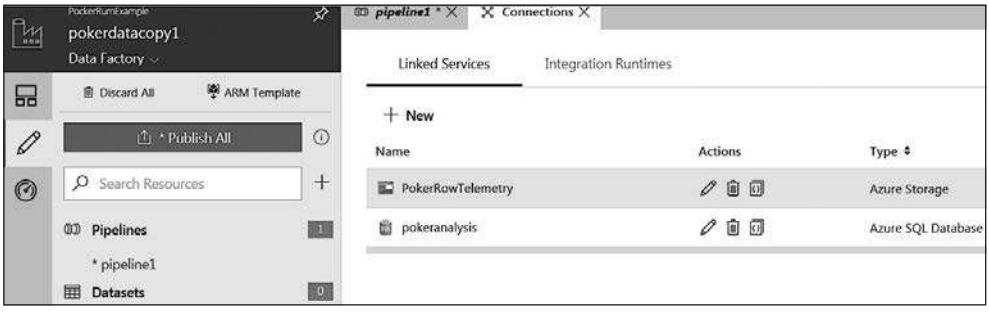


Рис. 11.10. Доступные подключенные серверы

его равным семи минутам), количество попыток повторений в случае отказа (Retry) — два и интервал между попытками — 30 секунд.

Далее необходимо сконфигурировать источник (Source) и приемник (Sink) данных. Перейдя на вкладку Source, следует нажать на ссылку + New, ввести в появившейся строке поиска SQL и сконфигурировать Azure Table Storage как источник данных (рис. 11.12).

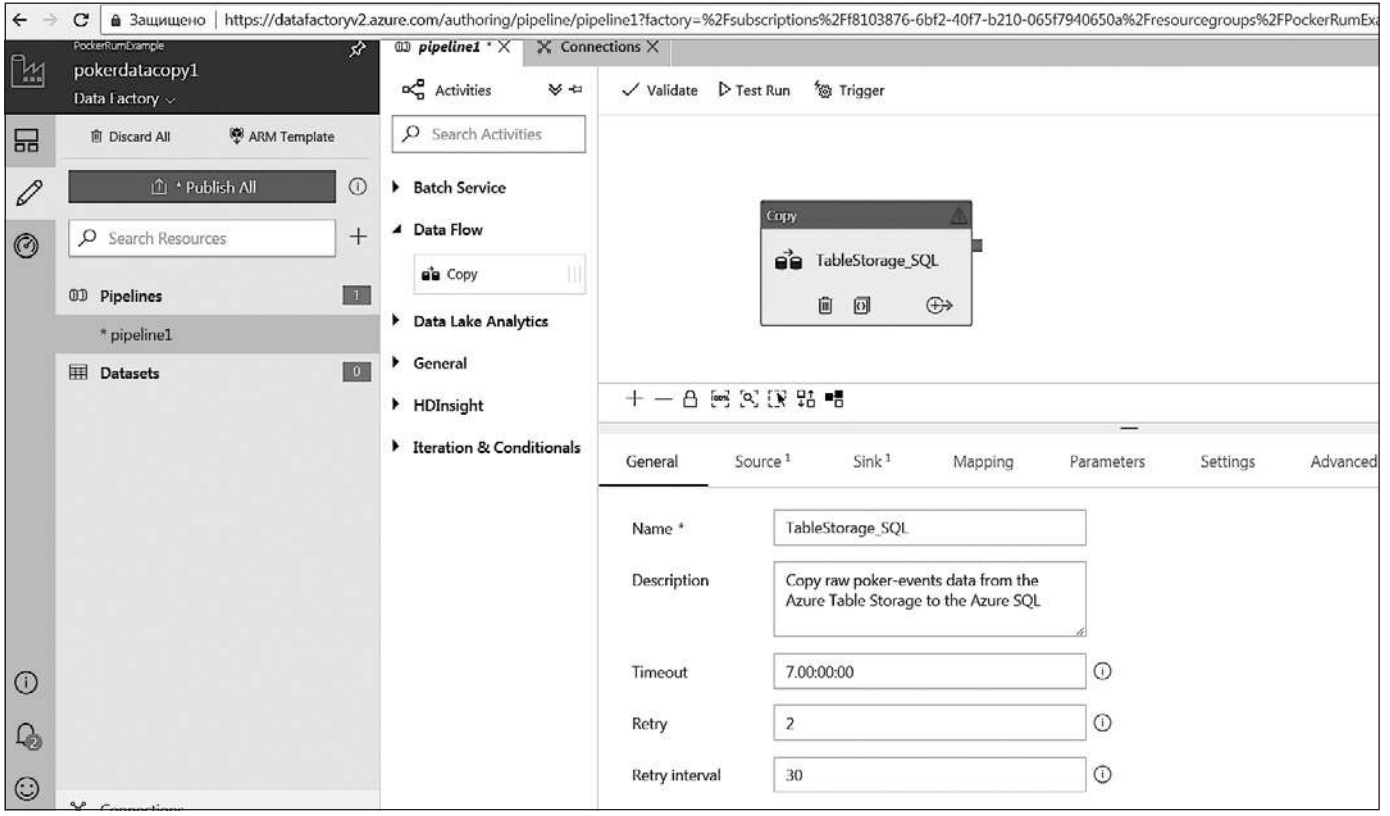


Рис. 11.11. Панель управления копирующей активности

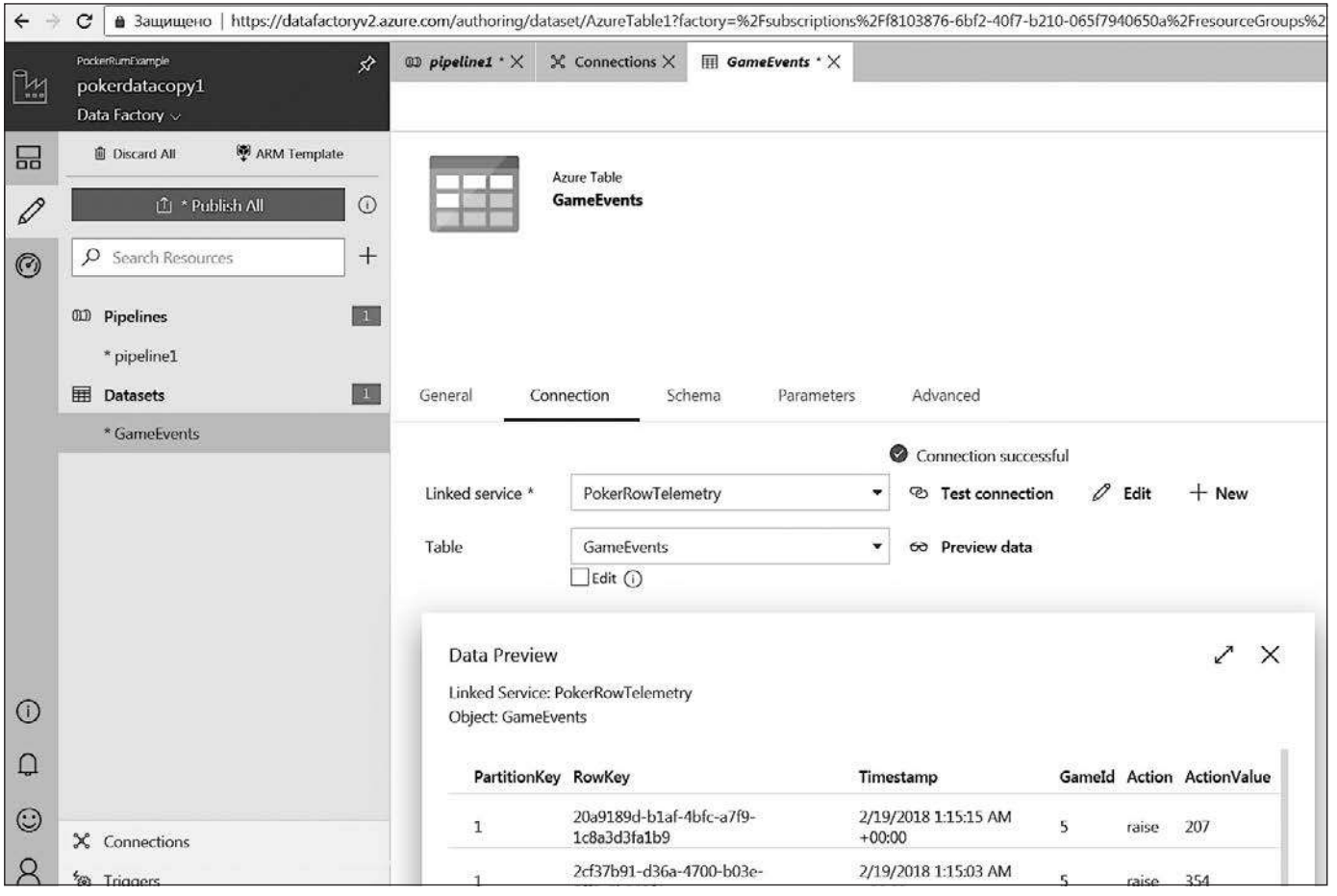


Рис. 11.12. Панель конфигурирования источника данных — Azure Table Storage

На этой панели нужно выбрать наш подключенный сервер (PokerRowTelemetry), имя таблицы (GameEvents). Далее можно проверить соединение (Test connection) и просмотреть данные (Preview data).

Далее в нашей базе данных SQL следует создать таблицу, которая будет хранить данные из Azure Table Storage, выполнив следующий сценарий в онлайн-редакторе или в стороннем приложении наподобие SQL Management Studio (рис. 11.13).

```
1 CREATE TABLE [GameEvents]
2 (
3     [PartitionKey] INT,
4     [GameId] INT,
5     [Action] VARCHAR(20),
6     [ActionValue] INT
7 )
8
9
```

Рис. 11.13. Создание таблицы приемника

Затем эту таблицу можно подключить как приемник (Sink) таким же образом, как подключали источник (рис. 11.14).

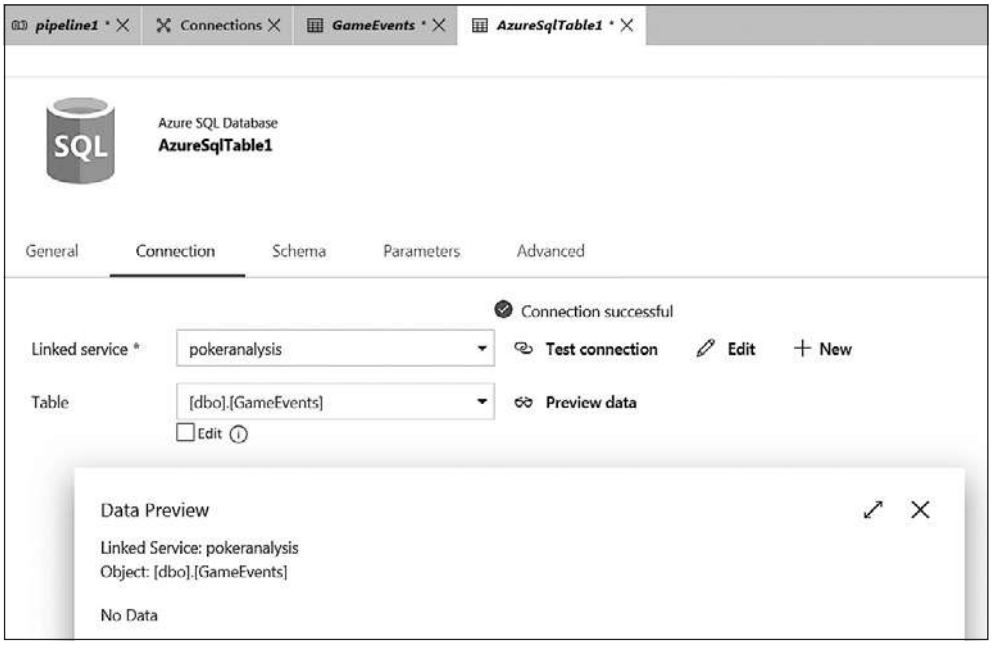


Рис. 11.14. Конфигурирование приемника данных — таблицы Azure SQL

После этого следует перейти на вкладку отображения схем источника и приемника выполняемого задания и при необходимости выполнить согласование схем, выбрав требуемые параметры и указав их типы.

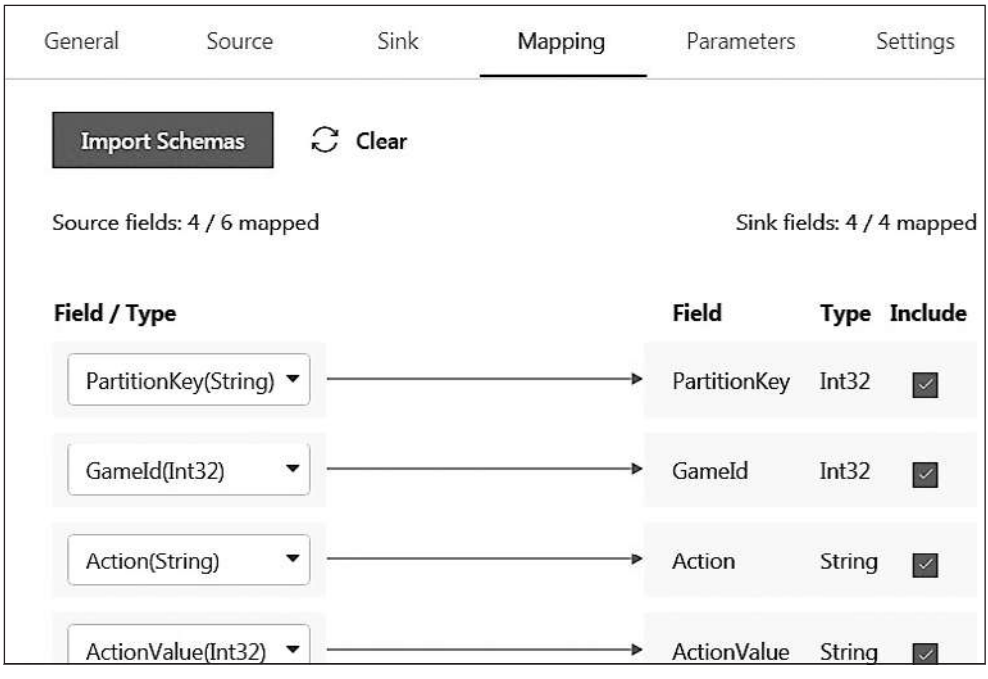


Рис. 11.15. Вкладка отображения схем источника и приемника

Затем, перейдя на вкладку конвейера, можно нажать на ссылку Test Run в верхней части вкладки. Если вы внимательно и правильно все сконфигурировали и настроили, то конвейер отработает успешно (рис. 11.16).

И в итоге можно выполнить запросы к скопированным данным. На рис. 11.17 показаны примеры запросов, выполненных прямо в онлайн-редакторе портала на вкладке Azure SQL. В левой половине рисунка подсчитывается суммарное количество ставок, сделанных каждым игроком. В правой изучается статистика типов шагов для заданного игрока. Более подробно вопросы выполнения аналитических запросов будут описаны в главе 12.

После создания и тестирования нужно задеплоить шаблон (по сути, весь этот портал служит графической оберткой для создания JSON-документа), нажав на ссылку Publish All, размещенную на общей панели конвейера pokerdatacopy1. Затем можно создать триггер, который будет запускать этот конвейер в заданные промежутки времени. Для этого следует нажать на ссылку Trigger в верхней части панели конвейера (рис. 11.18).

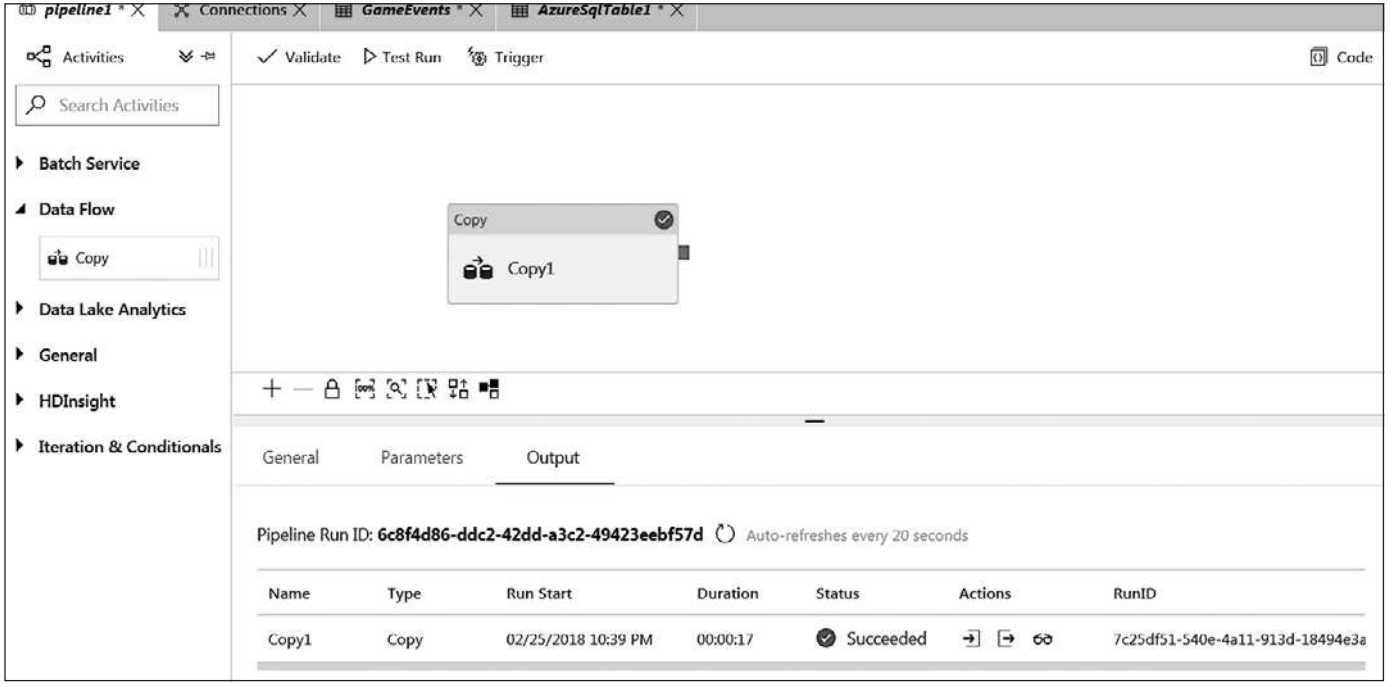


Рис. 11.16. Результат успешного выполнения копирующего задания Azure Data Factory

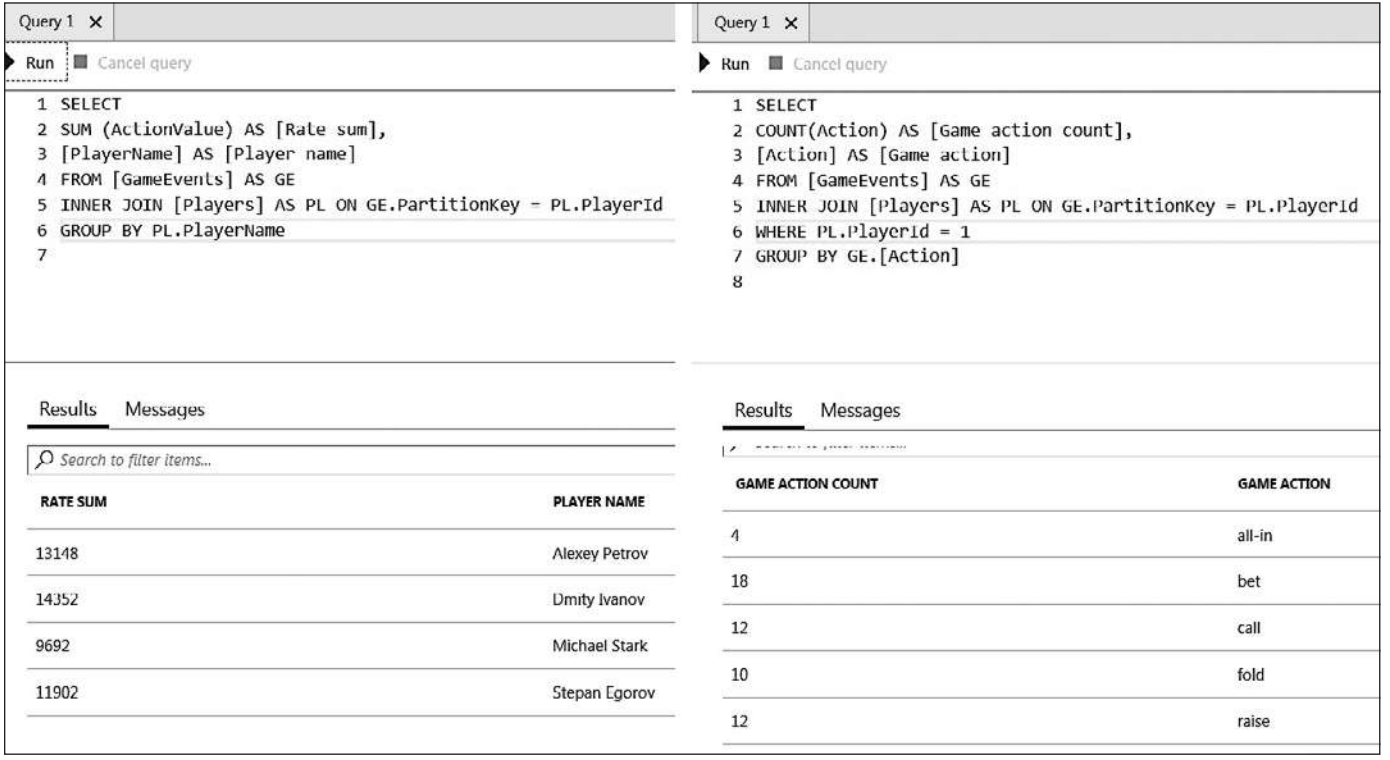


Рис. 11.17. Пример аналитических запросов к полученным данным

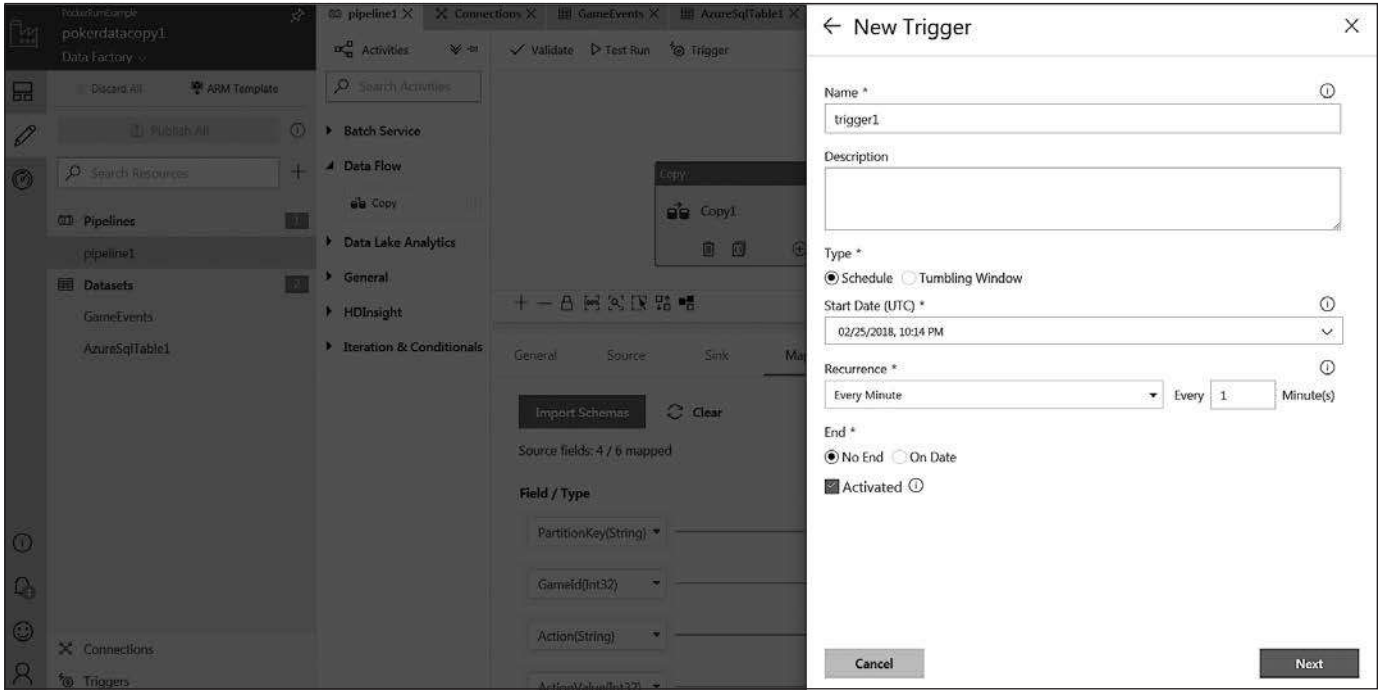


Рис. 11.18. Добавление триггера запуска конвейера данных

11.3. AWS Data Pipeline

Сервис AWS Data Pipeline предназначен для автоматизации копирования и трансформации данных в среде AWS. Он позволяет создавать управляемые конвейеры данных, работа которых будет зависеть от результата выполнения отдельных ступеней этого конвейера. AWS Data Pipeline содержит следующие компоненты.

- *Описание конвейера* (pipeline definition) — JSON-файл с описанием логики переноса данных. Включает в себя:
  - имена, местоположение и форматы источников данных;
  - описание заданий (activities), которые будут преобразовывать данные;
  - планировщики этих заданий;
  - вычислительные ресурсы, которые станут выполнять эти задания;
  - предусловия, которые должны быть удовлетворены перед выполнением заданий;
  - каналы оповещения об изменении статуса выполнения заданий (например, при успешном выполнении задания или сбое).
- Собственно *конвейер* (pipeline), который планирует и запускает задания. Перед запуском необходимо загрузить его описание. Для запущенного конвейера можно отредактировать описание, загрузить его и снова перезапустить конвейер, чтобы «подхватились» изменения. С конвейером ассоциированы следующие элементы.
  - Компоненты конвейера (pipeline components) представляют собой его бизнес-логику, описанную в разных секциях файла описания. Если конкретнее, то содержат описание источников данных, заданий, параметров планировщика заданий и предусловий для запуска заданий.
  - Экземпляры (instances) — это исполняемые задания, являющиеся компонентами потока исполнения конвейера.
  - Попытки (attempts) — элемент, отслеживающий повторные попытки запуска задания при отказе предыдущей. Количество попыток ограничено параметром, приведенном в файле описания конвейера, однако для разных видов отказов допустимы различные попытки, что и отслеживает этот элемент. При повторном запуске задания используются те же ресурсы (EC2, EMR), что и при предыдущем.
- *Исполнитель заданий* (task runner) загружает задания из конвейера, затем исполняет их. Например, копирует файлы в S3 EMRFS, а затем запускает EMR для их анализа. Исполнитель заданий автоматически устанавливается и запускается на ресурсах (EC2-экземплярах), приведенных в описании конвейера, или же может быть установлен и сконфигурирован на существующих экземпля-

рах. В качестве исполнителя может выступать приложение, созданное пользователем, или автоматически устанавливаемое приложение, поставляемое AWS Data Pipeline. Исполнитель активно взаимодействует с конвейером данных, забирая исполняемое задание, отсылая статус процесса исполнения и сигнализируя об успешном или неуспешном исполнении (рис. 11.19).

- *Узлы данных* (data nodes) определяют местоположение и тип данных, которые используются как вход и выход конвейера. В настоящее время в качестве узлов выступают:
  - узел DynamoDB (`DynamodDBDataNode`);
  - узел SQL (`SqlDataNode`);
  - узел RedShift (`RedshiftDataNode`);
  - узел S3 (`S3DataNode`).

  Список поддерживаемых баз данных включает в себя AWS RDS, AWS RedShift и базы, поддерживающие стандарт JDBC.
- Задания (activities) — компонент конвейера, определяющий действия, которые необходимо произвести над данными. AWS Data Pipeline содержит несколько готовых к исполнению заданий, представляющих типовые сценарии копирования данных из одного источника в другой, запуска сценария Hive, сценария пользователя (поддерживается Python и Scala) и др. В настоящее время поддерживаются следующие типы заданий:
  - *копирующее задание* (`CopyActivity`) — представляет собой задание копирования данных из одного источника в другой;
  - *задание EMR* (`EMRActivity`) — задание, выполняемое в кластере EMR;
  - *задание Hive* (`HiveActivity`) — задание в виде сценария Hive, выполняемое в кластере EMR;
  - *копирующее задание Hive* (`HiveCopeActivity`) — задание в виде сценария Hive, выполняемое в кластере EMR, обеспечивающее расширенную поддержку для передачи информации между узлами данных S3 и DynamoDB;
  - *задание Pig* (`PigAvtivity`) — задание в виде сценария Pig, выполняемое в кластере EMR;
  - *копирующее задание, исполняемое на RedShift* (`RedshiftCopyActivity`), — задание, осуществляющее перенос данных между таблицами AWS Redshift;
  - *задание выполнения сценария командной оболочки* (`ShellCommandActivity`) — задание в виде пользовательского сценария, выполняемого в командной оболочке Linux/Unix;
  - *задание SQL* (`SQLActivity`) — задание в виде сценария SQL, выполняемое в базе данных.

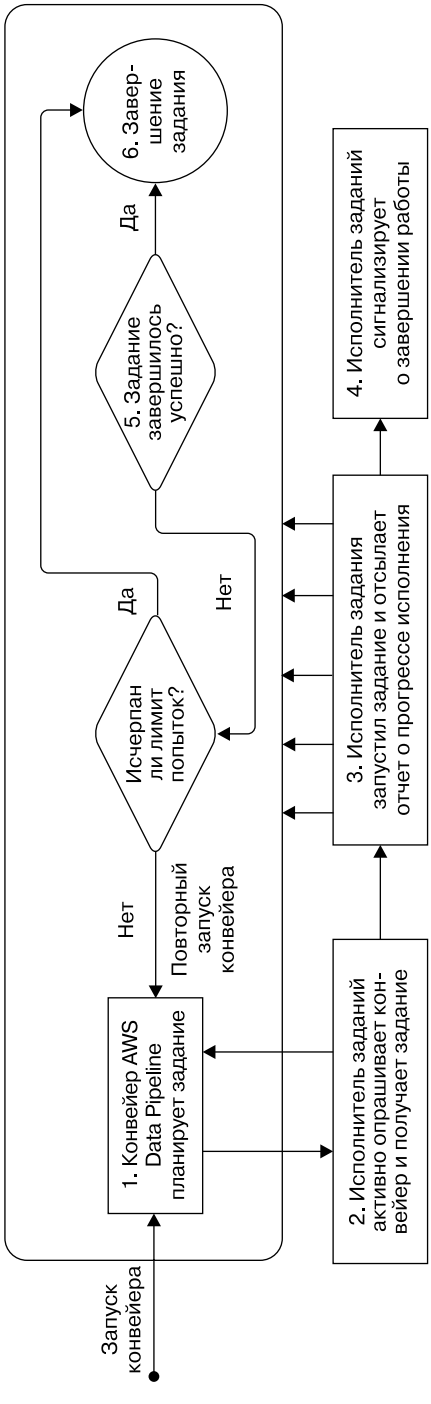


Рис. 11.19. Алгоритм взаимодействия исполнителя заданий с конвейером

- *Предусловия* (preconditions) — это компоненты конвейера данных, представляющие собой условия, которые должны быть выполнены перед запуском задания. Все они могут быть разделены на две группы: управляемые системой и управляемые пользователем. Первые не требуют дополнительных компьютерных ресурсов для своего функционирования и контролируются самим сервисом AWS Data Pipeline. К таким предусловиям относятся следующие:
  - `DynamoDBDataExists` — проверка наличия данных в определенной таблице DynamoDB;
  - `DynamoDBTableExists` — проверка существования таблицы DynamoDB;
  - `S3KeyExists` — проверка существования ключа файла в AWS S3;
  - `S3PrefixNotEmpty` — проверка того, что заданный префикс файла существует (то есть не является пустым).

  Предусловия, управляемые пользователем, выполняются только на вычислительных ресурсах (EC2-экземплярах), которые пользователь должен явно указать. К этой группе относятся следующие предусловия:
  - `Exists` — проверка существования заданного узла данных;
  - `ShellCommandPrecondition` — предусловие в виде команды оболочки Linux/Unix.
- *Ресурсы* (resources) — вычислительные ресурсы, требуемые для выполнения заданий конвейера. В настоящее время доступны ресурсы в виде EC2-экземпляров и EMR-кластера.
- *Действия* (actions) — шаги конвейера данных, выполняемые его компонентами при наступлении определенного условия, такого как успех, отказ и пр. Сами действия могут состоять в отсылке сообщения в SNS (SnsAlarm) или удалении ресурсов (terminate).

Познакомившись с основными компонентами сервиса AWS Data Pipeline, на конкретном примере рассмотрим, как все это работает в целом.

Прежде всего на панели списка конвейеров следует нажать кнопку Create new pipeline (Создать новый конвейер) (рис. 11.20).

Рис. 11.20. Список конвейеров данных

После этого откроется форма конфигурирования конвейера (рис. 11.21). В ней следует указать источники файла описания конвейера.

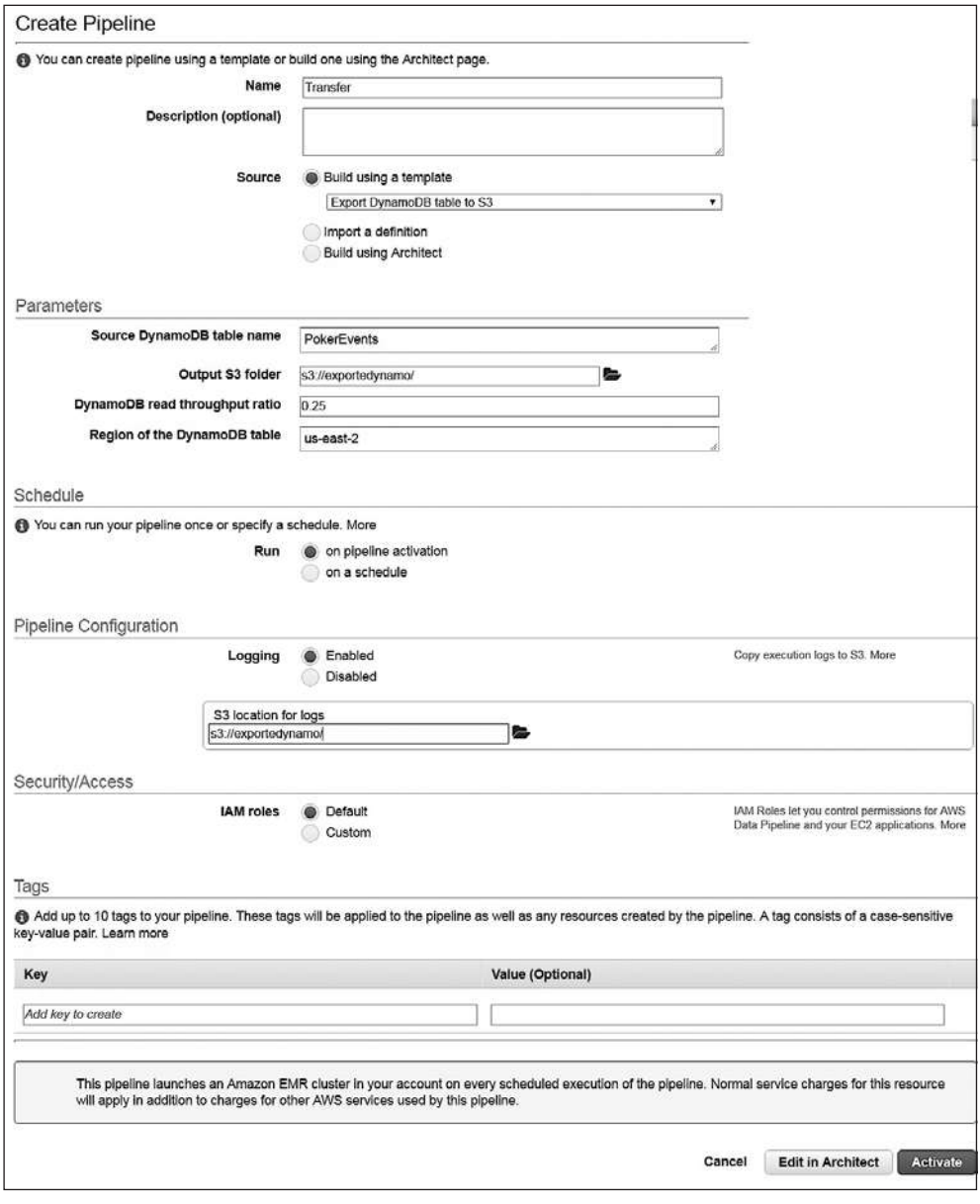


Рис. 11.21. Форма конфигурирования конвейера данных

Имеется всего три источника:

- можно выбрать из шаблона (Build using a template) (вариант выбора показан на рис. 11.22);

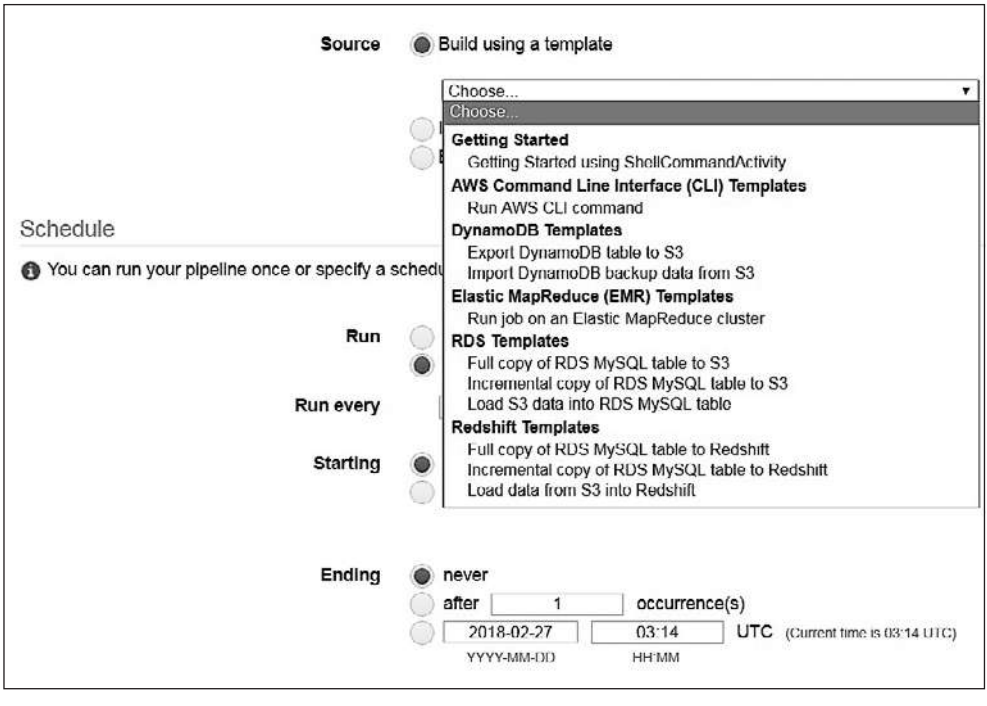


Рис. 11.22. Готовые шаблоны трансформации и копирования

- прямой импорт JSON-файла описания конвейера (Import a definition);
- создание шаблона с использованием специальной панели Build using Architect, которая показана на рис. 11.23.

Для примера на рис. 11.21 показан вариант конфигурирования Data Factory для копирования данных из таблицы PokerEvents DynamoDB (Source DynamoDB table name) в папку S3 (Output S3 folder), указывается регион, где расположена таблица us-east-2 (Region of the DynamoDB table). В поле Schedule можно указать тип запуска: моментальный (on pipeline activation) или по расписанию (on a schedule).

Данная панель позволяет более подробно настроить отдельные компоненты конвейера.

Результаты работы конвейера можно наблюдать на панели списка конвейеров (List Pipelines), показанной на рис. 11.24.

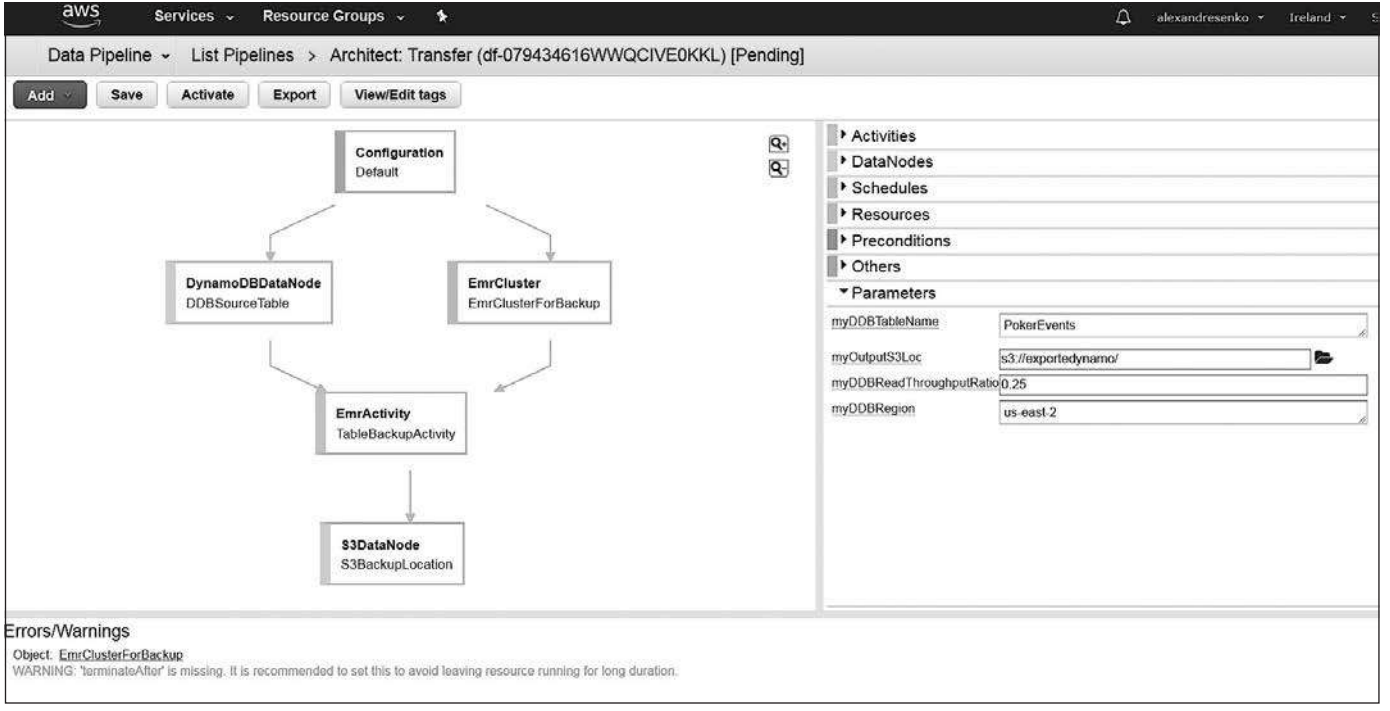


Рис. 11.23. Панель графического конфигурирования файла описания конвейера

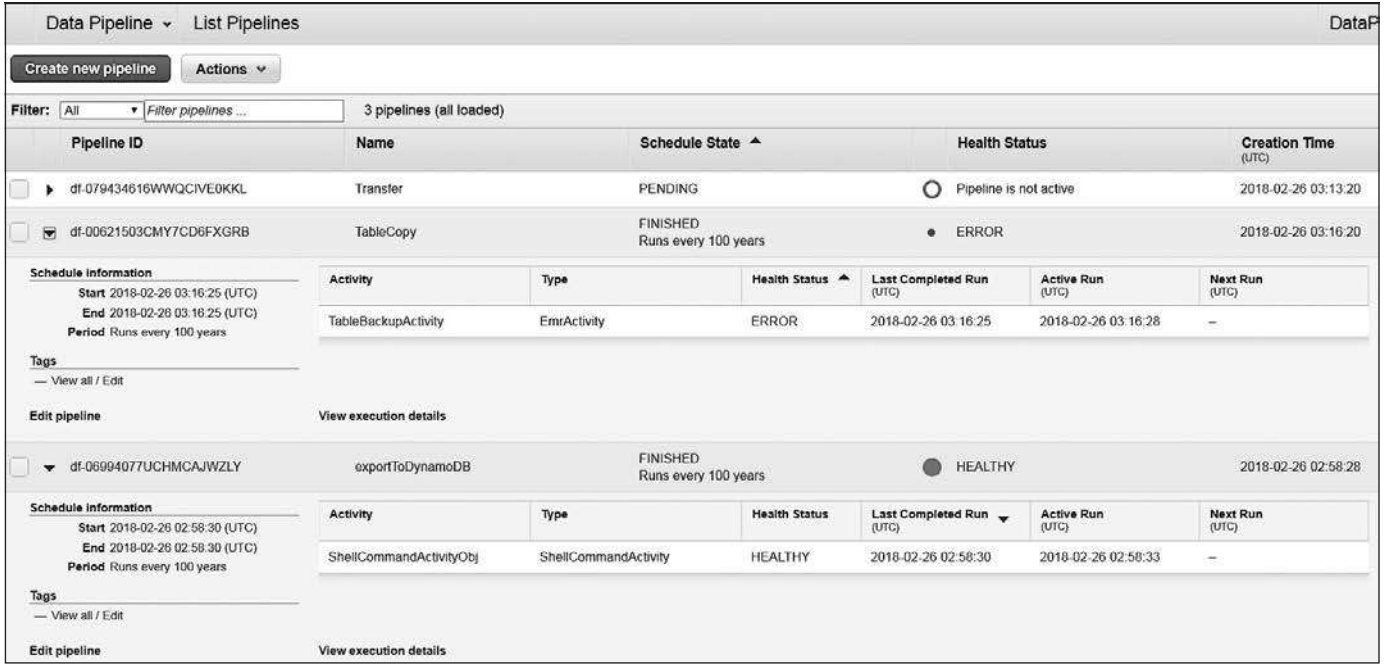


Рис. 11.24. Результат работы конвейера Data Pipeline

11.4. AWS Glue

AWS Glue — бессерверный сервис, предназначенный для категоризации данных в так называемый каталог данных и выполнения задач ETL. Включает в себя:

- централизованное хранилище метаданных хранилищ данных (AWS Data Catalog);
- подсистему ETL, автоматически генерирующую код на Python или Scala, который описывает трансформацию данных;
- планировщик заданий;
- подсистему мониторинга и перезапуска.

Основной сценарий использования этого сервиса — построение системы доставки данных из различных источников в централизованное хранилище данных. Ключевым компонентом этого сервиса, отличающим его от Data Pipeline, является каталог данных — Data Catalog. Рассмотрим подробнее концепцию каталога данных.

В предыдущих главах мы рассмотрели различные сервисы хранения информации в облачных средах: в файлах в специальных файловых хранилищах и хранилищах общего назначения, в базах и хранилищах данных. В частности, отмечалось, что для целей последующего анализа наиболее удобным является централизованное хранилище, в котором информация может быть обработана единообразно. Для того чтобы наполнить это хранилище, необходимо знать обо всех источниках данных. Теперь рассмотрим конкретную ситуацию из моей реальной практики. В некой системе, состоящей из набора сервисов, имеется большое количество БД, каждая из которых относится к своему сервису. В этих базах содержится операционная информация, которую можно использовать для отыскания скрытых бизнес-закономерностей, относящихся ко всей системе. Чтобы сделать это практически, владельцы данной информационной системы приглашают консультанта по анализу данных. Первое, что он должен сделать, — разобраться с текущим строением системы, адресами и структурой источников данных. У консультанта есть несколько возможностей: например, физически обойти все команды, ответственные за сервисы, и расспросить их или разослать всем письма с просьбой откликнуться — мол, у кого какие источники данных и что в них содержится. В обоих случаях такое выяснение занимает достаточно продолжительное время — возможно, даже месяцы, в зависимости от организационной структуры и количества сервисов. Но тут консультанта может ждать ряд существенных трудностей. Например, команда, поддерживающая сервисы, может и не до конца разбираться, что содержится в старых, мало используемых базах данных. Но ведь в «старых неиспользуемых» хранилищах информации могут содержаться весьма важные бизнес-данные. А теперь предположим, что к этим данным необходимо обратиться не одному консультанту, а нескольким, относящимся к разным группам. Например, из финансового отдела, отдела BI и пр. Конечно, можно объединить их в один отдел, но проблема передачи сведений о существующей структуре данных для произвольного члена команды все еще существует. Совершенно логично, чтобы эта информация — метаданные об источниках данных — хранилась централизованно. При этом нужно обеспечить

к ним доступ с разделяемой ответственностью (не всем членам команды следует предоставлять все данные), регулярно обновлять эту информацию и поддерживать актуальность документации. Для отдельной группы, которая этим занимается, последний пункт выполнить очень трудно: этой группе необходимо устанавливать связи со всеми иными группами, ответственными за сервисы. А при достаточно большом количестве сервисов и групп это становится крайне проблематичным, и даже может наступить ситуация, когда, по сути, «никто ничего не знает».

Все указанные проблемы достаточно серьезные и могут решиться с помощью сервисов каталогов данных (data catalog). Этот тип сервиса представляет собой централизованное хранилище метаданных — сведений об источниках данных, включая адрес, формат, документацию, сведения об актуальности, о ссылках на команды, ответственных за них. Информация об источниках данных может быть добавлена непосредственно командами, отвечающими за их создание и поддержание. То есть каждая команда ответственна за поддержание актуальности только своего источника. После добавления такие сведения становятся доступными для поиска на специальном портале. Если в качестве метаданных используются теги, то возможен простой поиск по ключевым словам. Например, информационная система относится к удаленному мониторингу автомобилей, и ключевыми словами могут быть «топливо» (все источники данных, относящиеся к топливу), «масло», «одометр» и пр. Эти ключевые слова можно представить в виде словаря, чтобы упростить поиск. Указанные метаданные пригодны для построения систем ETL напрямую (как в случае с AWS Glue) или косвенно (Azure Data Catalog).

Архитектура AWS Glue включает в себя следующие компоненты (рис. 11.25).

- *Выполняемые задания* (jobs) — непосредственно рабочие процессы, производящие требуемое извлечение, трансформацию и загрузку данных из источников в назначения.
- *Сборщик*, или *краулер* (crawler), — программа, подключаемая к хранилищам данных, которая используется для наполнения каталога данных метаданными источников. В каталоге сборщику указывается хранилище данных, и он, определив схему источника, создает в каталоге соответствующий объект, называемый таблицей, поскольку все источники данных так или иначе могут быть описаны как таблицы (таблицы SQL, файлы в S3). Вдобавок к описанию таблицы сборщик извлекает дополнительные метаданные, необходимые для построения сценария выполняемого задания ETL.
- *Классификатор* (classifier) — часть сборщика, определяющая схему данных источника. Поддерживает источники типа JSON, CSV, AVRO и XML. Кроме того, имеет возможность работать с реляционными базами данных, поддерживающими протокол JDBC. Пользователь может написать собственный классификатор с помощью специального синтаксиса для описания схемы, например grok.
- *Сценарий трансформации данных* — сценарий, выполняющий задачу трансформации и копирования, генерируемый AWS Glue или предоставляемый пользователем. Для описания задач поддерживаются языки Python, PySpark и Scala.

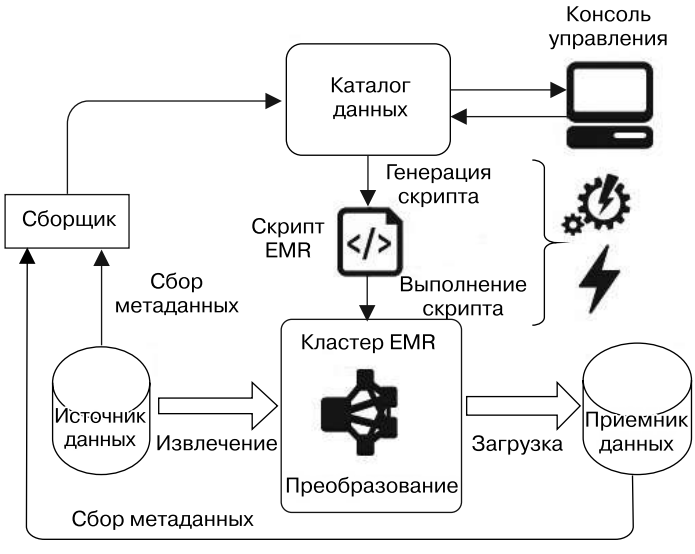


Рис. 11.25. Архитектура сервиса AWS Glue

- *Триггер* (trigger) — встроенный механизм запуска задания, работающий по расписанию или на основе перехваченного события. При срабатывании триггера происходит основной процесс ETL: на внутренних вычислительных ресурсах, содержащих фреймворк Apache Spark, выполняется сценарий трансформации данных, сгенерированный AWS Glue или предоставляемый пользователем.
- *Сервер Notebook* — поскольку в основе движка AWS Glue лежит Apache Spark, то имеется возможность напрямую взаимодействовать с ним через сервер веб-приложения Notebook (я предпочитаю не переводить, но по сути это блокнот), через который пользователь может интерактивно взаимодействовать с кластером и писать запросы с помощью PySpark.
- *Каталог данных* (data catalog) — централизованное хранилище данных, которое содержит описания таблиц, сгруппированные в базы данных, выполняемых заданий и пр.

Важно четко понимать: таблицы и базы данных в AWS Glue — это объекты хранения метаданных. Они не имеют в своем составе непосредственно данных, а лишь содержат сведения о них (название, описание, ключевые слова, схему и пр.).

Теперь посмотрим на конкретном примере, как работать с сервисом AWS Glue. Поставим задачу скопировать в AWS Redshift данные в виде файла JSON, хранящегося в AWS S3. Этот файл мы получили, экспериментируя с AWS Kinesis Firehouse, доставляя сообщения от работающего bash-скрипта в AWS S3. Консоль управления AWS Glue имеет вид, показанный на рис. 11.26.

Рис. 11.26. Консоль управления AWS Glue

На этом рисунке приведены таблицы с метаданными источников. Каждая таблица отнесена к своей БД. В данном примере обе таблицы описывают источники данных, расположенные в AWS S3 и имеющие форматы JSON и CSV. Для добавления таблицы необходимо нажать на ссылку Add tables. Далее следует выбрать режим добавления метаданных: вручную (ссылка Add table manually) или с помощью сборщика (ссылка Add tables using a crawler). Выберем сборщик, в результате чего откроется следующая форма (рис. 11.27).

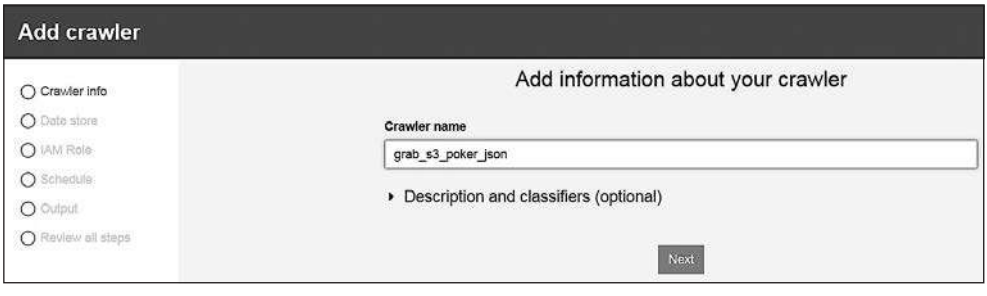


Рис. 11.27. Панель добавления сборщика

Кроме того, можно подключать классификатор (мы его не выбрали и потому не будем конфигурировать). Поскольку данные в файле организованы в виде последовательности JSON-объектов без помещения в корневой объект (например, массив), нет необходимости дополнительно настраивать классификатор.

Следующий шаг — подключение физического хранилища. В качестве такового может выступать AWS S3 или реляционная база данных, поддерживающая JDBC-соединение. Подключение БД мы рассмотрим позже. А для подключения AWS S3 необходимо выбрать источник S3, выбрать путь к файлу или указать общий паттерн, которому должно соответствовать имя файла (рис. 11.28).

Далее для сборщика настраиваются права доступа и IAM-роль (рис. 11.29).

Далее настраивается планировщик заданий (рис. 11.30). Они могут быть различного типа в зависимости от периодичности их запуска. Для демонстрационных целей я сконфигурирую запуск планировщика по требованию (run on demand).

Затем нужно создать базу данных — логическую группировку таблиц. При этом можно настроить связь таблицы в каталоге данных и физического источника данных — появится возможность удалить таблицу из каталога при удалении источника (рис. 11.31).

Далее можно просмотреть страницу с обзором созданного сборщика (рис. 11.32).

Созданный сборщик в консоли управления имеет статус Ready, не имеет ассоциированных с ним логов и еще не создал ни одной таблицы (рис. 11.33). Чтобы запустить сборщик, следует выделить строку и нажать на ссылку Run crawler.

На рис. 11.34 показан отработавший сборщик, добавивший таблицу в каталог данных.

Рис. 11.28. Конфигурирование хранилища для сборщика

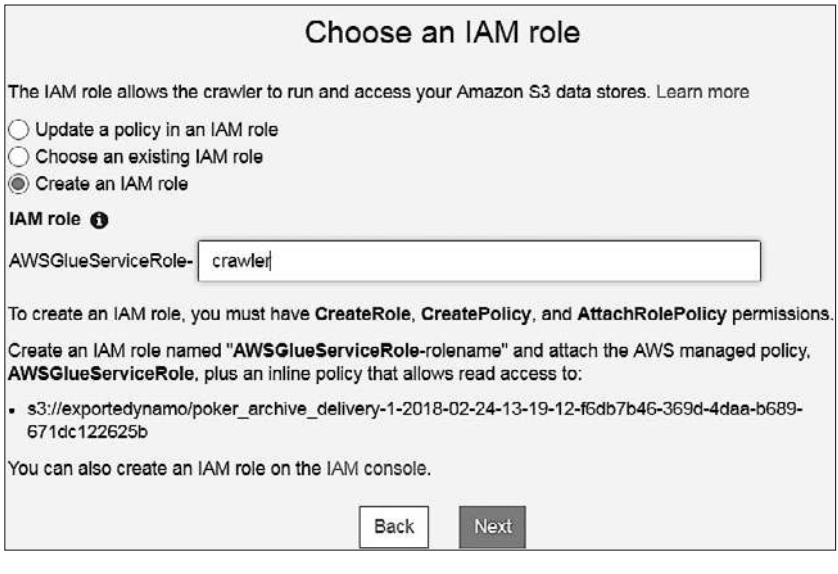


Рис. 11.29. Настройка IAM-роли

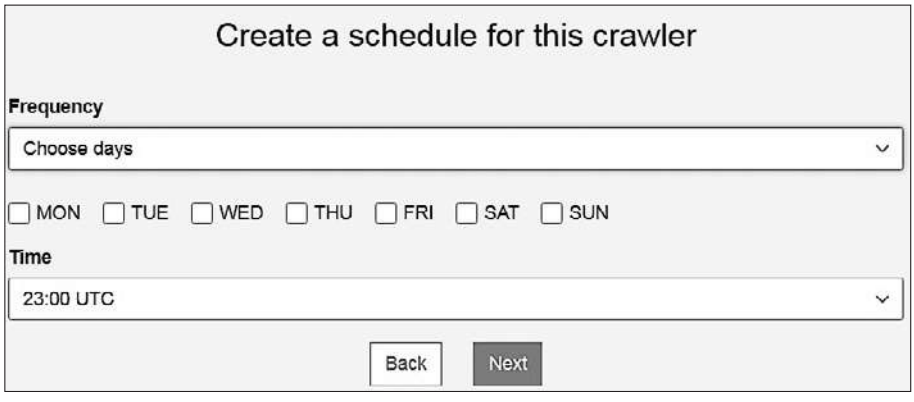


Рис. 11.30. Настройка планировщика заданий

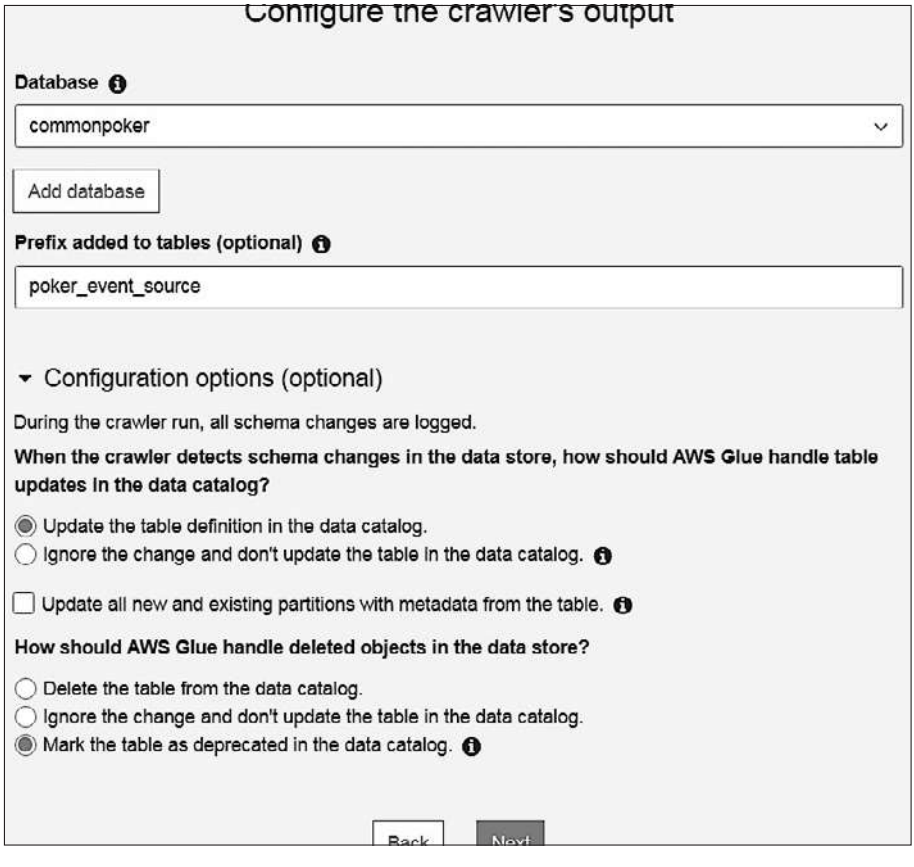


Рис. 11.31. Конфигурирование логической группы таблиц метаданных — базы данных

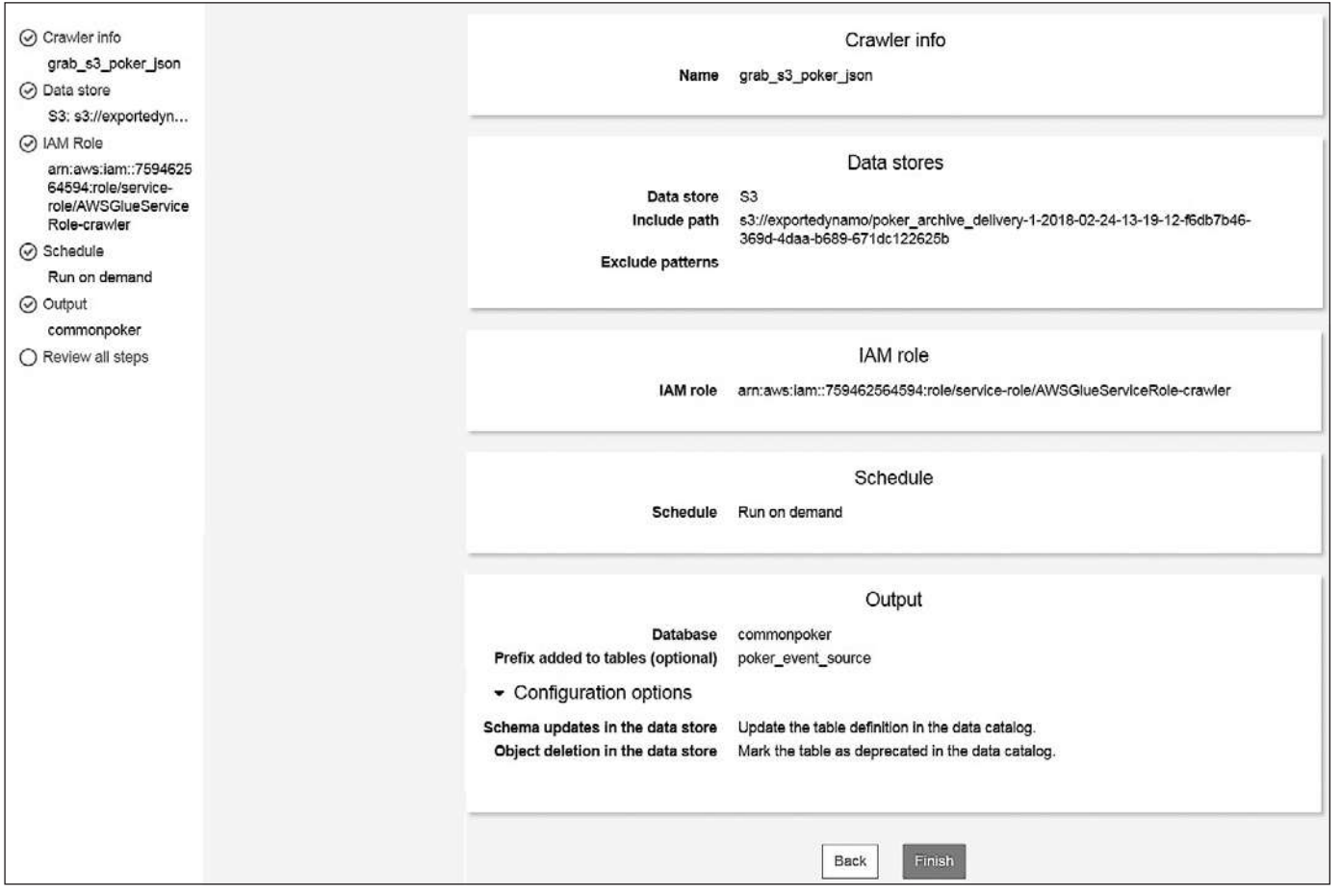


Рис. 11.32. Обзор созданного сборщика

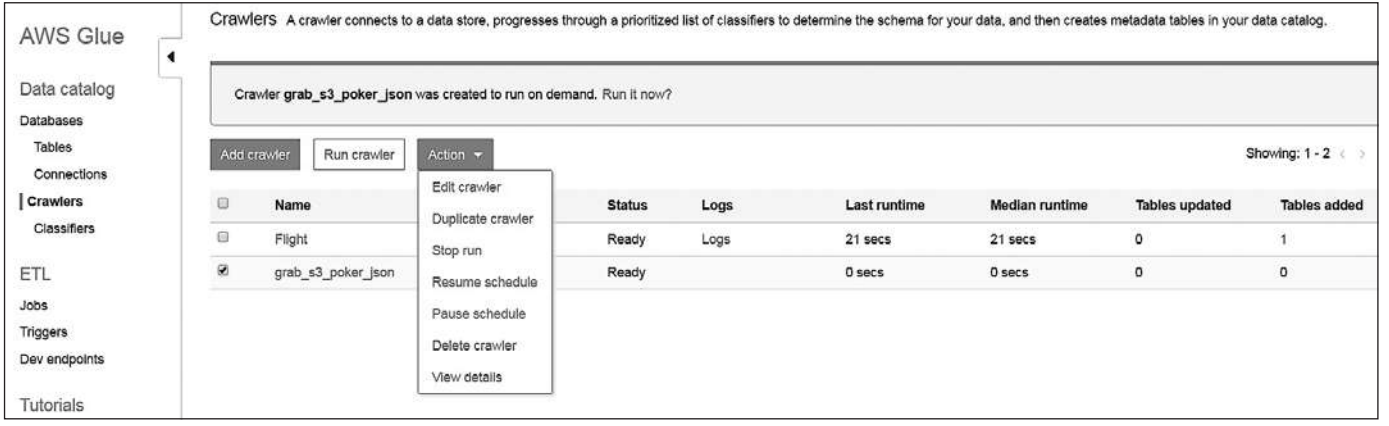


Рис. 11.33. Консоль управления сервисом AWS Data Glue — вкладка сборщиков

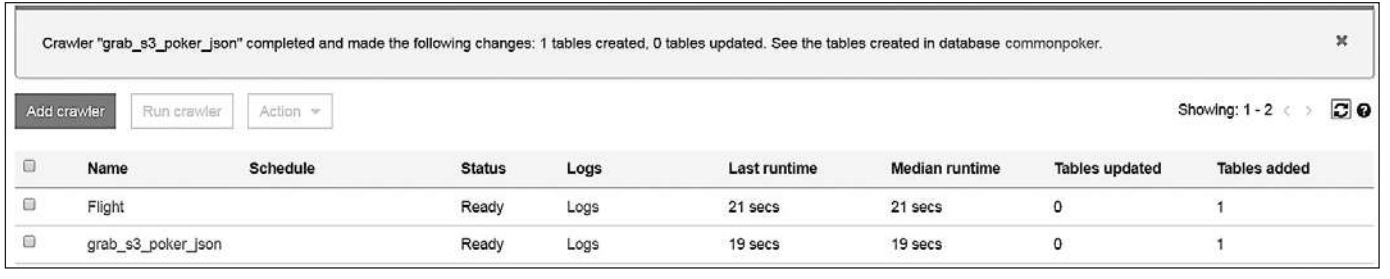


Рис. 11.34. Отработавший сборщик, добавивший таблицу в каталог данных

Чтобы посмотреть добавленную таблицу, следует перейти к вкладке Tables (рис. 11.35).

На этой вкладке я умышленно навел указатель на строку поиска, чтобы вы имели возможность посмотреть, как может происходить фильтрация и поиск таблиц. Если щелкнуть на ссылке с именем таблицы, то можно увидеть, что представляют собой метаданные таблицы (рис. 11.36).

Поскольку мы собираемся копировать данные из этого источника в таблицу — назначение реляционной базы данных, то рассмотрим способ извлечения метаданных из базы данных RDS. Для этого надо перейти на вкладку Databases/Connections и нажать ссылку добавления соединителя (рис. 11.37).

В данном случае добавлена реляционная база данных MS SQL из сервиса AWS RDS, для которой нужно настроить подключение. Для этого следует указать название экземпляра (`analyser-mk`), базы данных (`poker_collection`), ввести учетные данные пользователя, который будет иметь права на запись в таблицу базы данных, и нажать кнопку Next (Далее) (рис. 11.38).

Обратите внимание: сетевая группа безопасности (NSG) экземпляра RDS должна обеспечивать доступ из подсети AWS Glue. Далее в сборщике настраиваются параметры использования требуемой таблицы в базе данных.

Теперь посмотрим, как запускать задание ETL на основе метаданных, сохраненных в AWS Glue. Для этого следует перейти на вкладку jobs и нажать на ссылку Add job (рис. 11.39).

На этой вкладке следует дать имя выполняемому заданию (в данном случае `transform_analytics`), выбрать IAM-роль, указать S3-бакет, в котором будут храниться сценарий и размещаться временные данные (рис. 11.40).

Далее следует добавить таблицу как источник данных (рис. 11.41), точно так же и таблицу-приемник данных и сконфигурировать маппинг колонок таблицы (рис. 11.42).

После успешного прохождения всех предыдущих шагов откроется вкладка конфигурирования задания ETL (рис. 11.43). Здесь необходимо сделать пояснение. Дело в том, что задание ETL будет выполняться с помощью сервиса AWS EMR, в кластере которого будет развернут Apache Spark. Одной из возможностей работы с этим сервисом является использование языка Python. Сценарий на этом языке и отображается в главном окне. Пользователь может его отредактировать и посмотреть на диаграмму потока исполнения *(слева)*, соответствующую этому сценарию.

Чтобы запустить выполнение, следует нажать на ссылку Run job. После этого сценарий начнет выполняться. Не волнуйтесь из-за длительности выполнения, ведь должен создаться кластер AWS EMR, а затем удалиться. После успешного или неуспешного завершения работы сценария результаты будут представлены в логах.

Рис. 11.35. Вкладка с добавленными таблицами

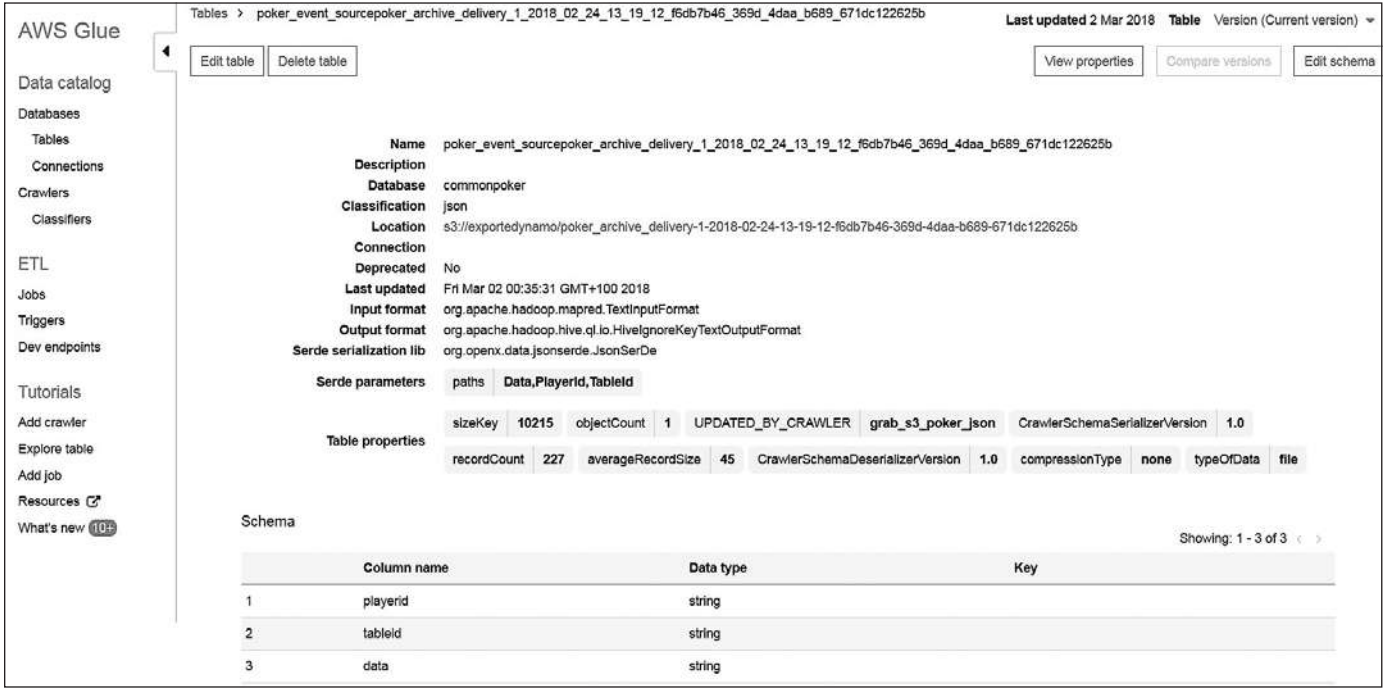


Рис. 11.36. Метаданные JSON-файла, извлеченные сборщиком

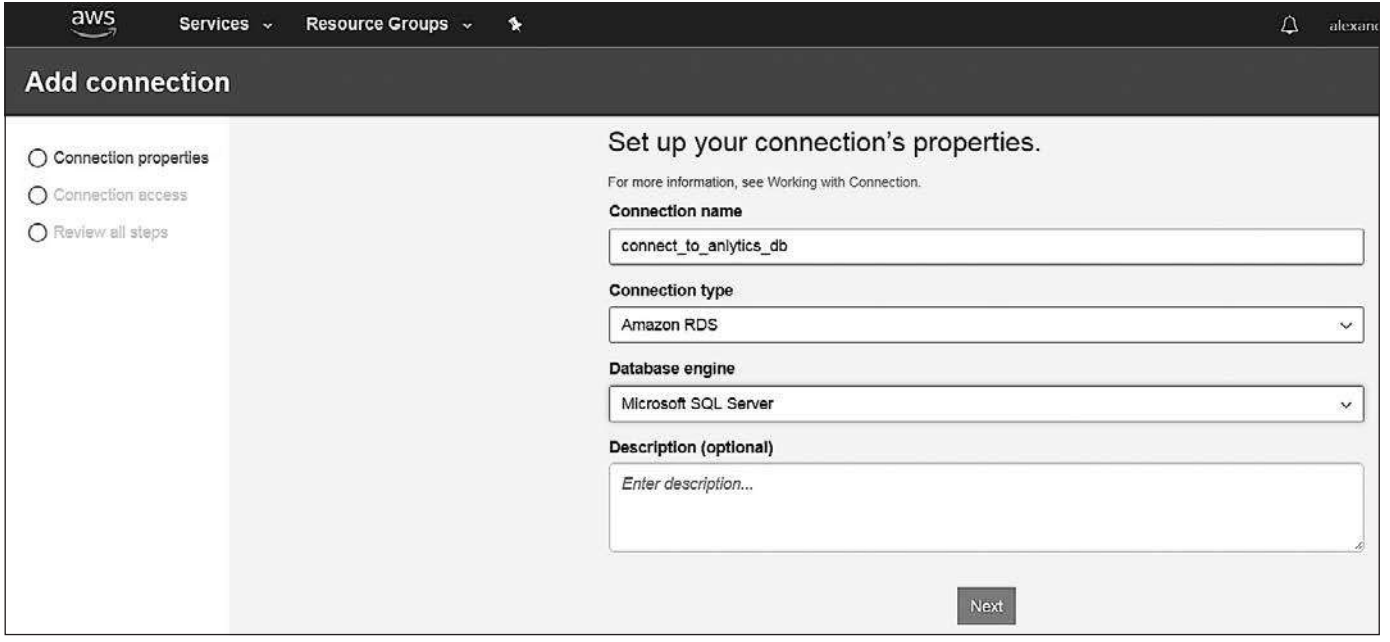


Рис. 11.37. Добавление соединения с базой данных с помощью модуля JDBC

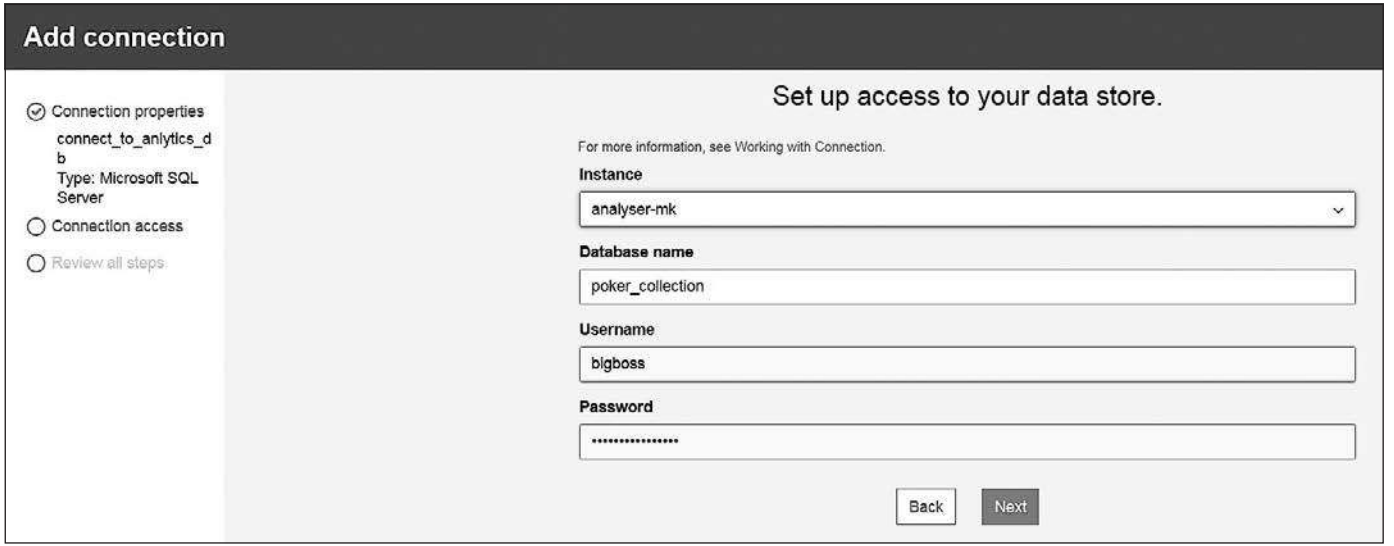


Рис. 11.38. Конфигурирование подключения к базе данных

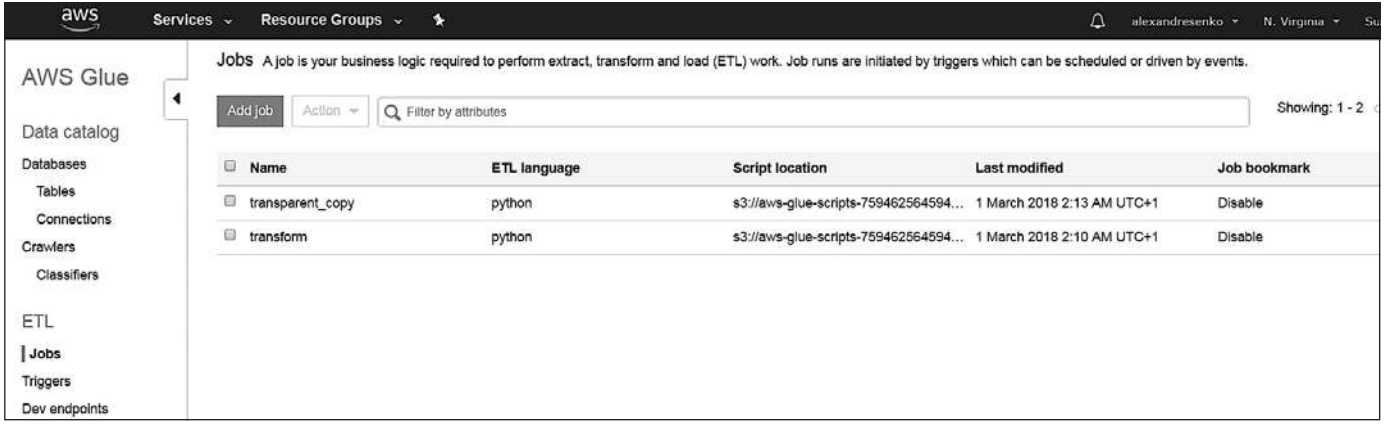


Рис. 11.39. Вкладка списка заданий ETL

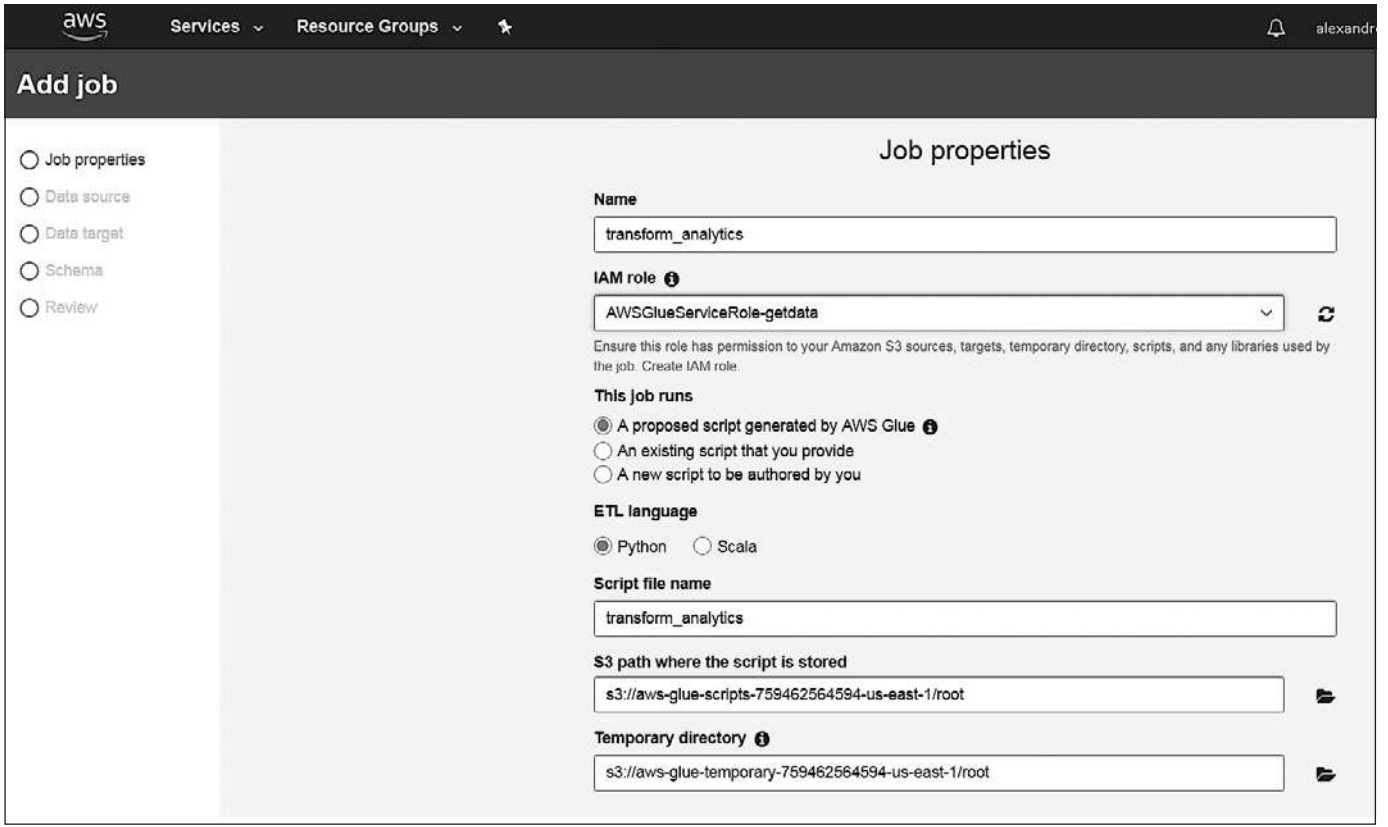


Рис. 11.40. Настройка ETL-задания

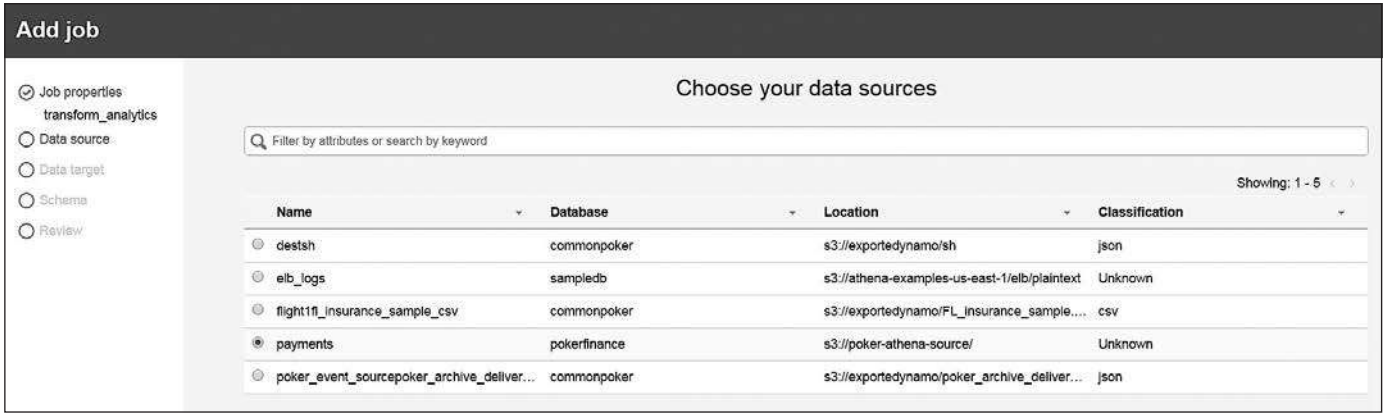


Рис. 11.41. Добавление источника данных

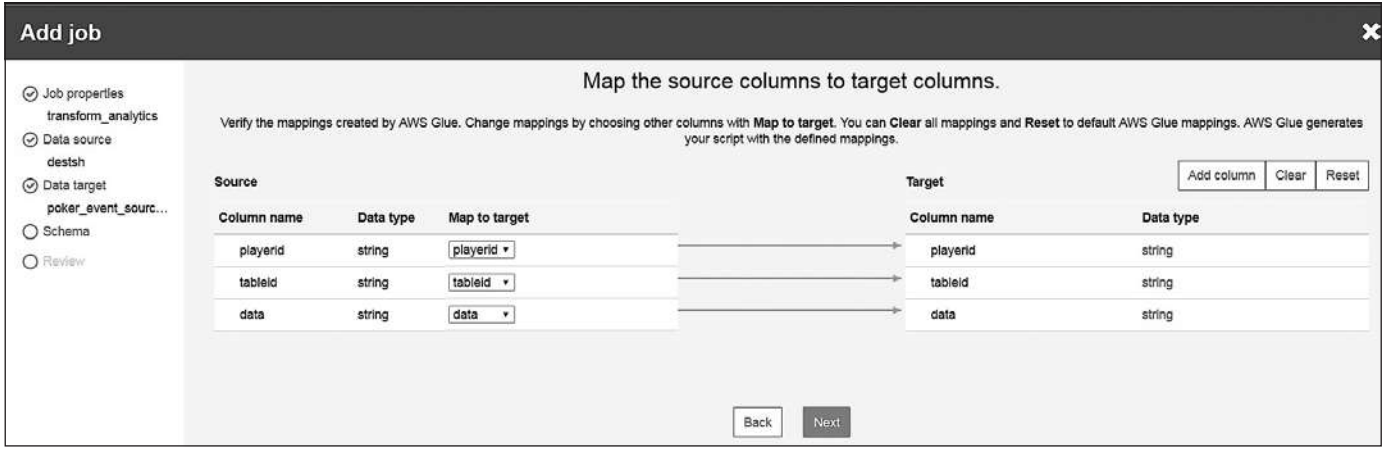


Рис. 11.42. Настройка маппинга колонок

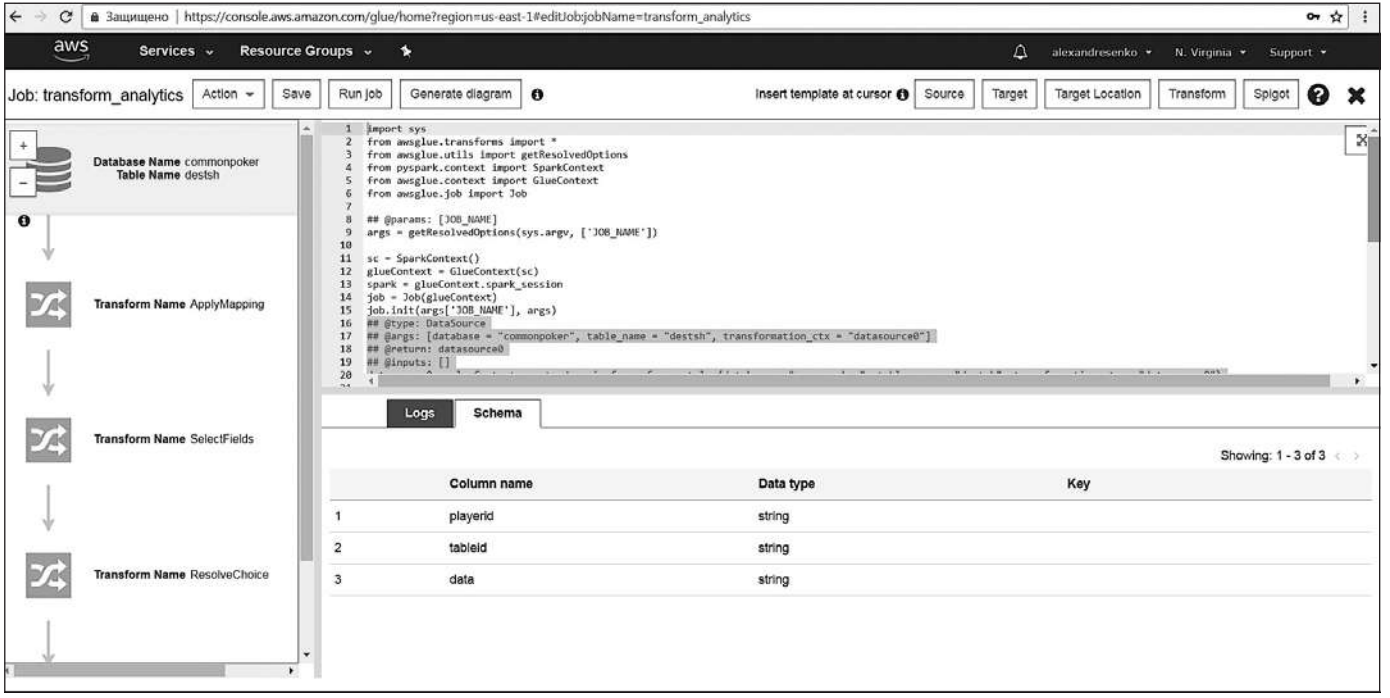


Рис. 11.43. Страница конфигурирования ETL-задания

11.5. Резюме

В данной главе мы рассмотрели сервисы копирования и трансформации данных Azure Data Factory, AWS Data Pipeline и AWS Glue. Характерная особенность Azure Data Factory — широчайший охват источников и приемников. Все сервисы хранения данных от Azure, а также много внешних источников можно подключить к Data Factory. Кроме того, имеется возможность исполнять пакеты SSIS, что в ряде случаев очень удобно. Azure Data Factory в этом плане пока проигрывает.

Сервис AWS Glue представляет собой очень удобное средство каталогизации, трансформации и копирования данных, что позволяет одновременно создавать каталоги данных и использовать их для ETL. AWS Glue содержит общее с AWS Athena хранилище метаданных. Ближайший аналог этого сервиса — Azure Data Catalog, который, однако, не относится к сервисам трансформации и копирования, являясь лишь хранилищем метаданных, а потому в этой книге обойден вниманием.

Вообще сервисы трансформации и копирования играют важнейшую роль в построении информационных систем анализа больших данных. Действительно, требования к сервисам хранения и анализа противоречивы и не могут быть удовлетворены в рамках одного хранилища, особенно когда данных много. И чтобы удовлетворить эти требования, необходимо использовать хранилища разных типов и сервис, который связывает их информационно. Потому-то один из таких сервисов назван Glue — «клей».

Часть IV
Анализ BigData в облаке

> Но что есть знание? Знание — это ожидание определенного события после того, как произошли некоторые другие события. Кто не знает ничего, может ожидать всего. Кто знает что-то, тот считает, что может произойти не все, а лишь некоторые явления, иные же не произойдут. Следовательно, знание — это ограничение разнообразия, и оно тем больше, чем меньше неуверенность ожидающего.
>
> *Станислав Лем. Сумма технологии*

И вот наконец мы подобрались, наверное, к самой интересной части всей книги — к анализу больших данных. Собственно говоря, ради него и создается вся остальная инфраструктура доставки, преобразования, хранения и др. А в чем заключается анализ данных? Как отмечалось в главе 1, это применение к данным специальных алгоритмов фильтрации, сортировки, агрегирования, выполнения арифметических и статистических операций в целях выявления закономерностей, скрытых тенденций и пр. Таким образом, анализ данных состоит в их выборке, агрегировании или ином преобразовании, делающих информацию удобной для изучения. Действительно, сырые данные, будь то большие данные или просто данные из файлов логов, человеку трудно прочесть. Они представляют собой, по сути, таблицы или структурированные файлы формата JSON/XML, и, просто просматривая их, можно извлечь чрезвычайно мало информации в силу психофизиологических особенностей человека.

Анализ данных может быть следующих видов:

- *потоковый* — применяется для анализа потоков данных в реальном режиме или с небольшой задержкой;
- *интерактивный* — служит для анализа данных, находящихся в хранилище, при активном участии пользователя; последний при этом вводит запрос на вебпортале или в терминале и ожидает ответ в течение минимального времени;
- *пакетный* — используется для выполнения периодических задач анализа в хранилище данных, например агрегирования, фильтрации и др.;
- *интеллектуальный* — это анализ данных с применением сложных алгоритмов машинного обучения. Я задействовал термин «сложных», поскольку во всех предыдущих случаях алгоритм анализа пользователь пишет сам. В случае же интеллектуального анализа этот алгоритм, а вернее, его параметры определяются путем процедуры обучения непосредственно на самих данных. Интеллектуальным этот анализ называется ввиду того, что в нем применяются элементы искусственного интеллекта, например машинное обучение с помощью алгоритмов на основе нейронных сетей. Сервисы интеллектуального анализа данных — это темы отдельной книги и здесь подробно рассматриваться не будут.

Как правило, указанные виды сервисов применяются в той или иной комбинации. Очень интересный пример их комбинации — система анализа финансовых онлайн-транзакций банковских карточек на предмет обнаружения подозрительных действий (выдача большой суммы наличных в другой стране или выдача малых сумм наличных в различных банкоматах в малый промежуток времени и пр.). Для выявления таких действий необходимо анализировать поток сообщений, выделить из него сообщения, относящиеся к одной карточке и потоковому анализу. И здесь возможны несколько вариантов. Например, предварительное накопление сведений о транзакциях в хранилище и применение сервисов интерактивного анализа для предварительного изучения данной информации. Затем на основании этого изучения строится система пакетного анализа, которая выделяет существенные признаки, характеризующие типовое поведение держателя карточки. Они служат для дальнейшей корректировки алгоритмов (а точнее, численных индексов в алгоритмах) и потокового анализа.

Главная идея такой архитектуры — изучение закономерностей поведения конкретных держателей карточек с целью определить их финансовые паттерны поведения. Эти паттерны выражаются в конечном итоге в числовых или исчислимых показателях (величина дневной суммы, снимаемой с карточки; страны, в которых снимается сумма), используемых для корректировки соответствующих параметров у систем потокового анализа. Но тут очевиден существенный недостаток: если паттерны будут найдены и жестко закодированы в виде алгоритма на компилируемом языке, то адаптировать его для конкретного специфичного пользователя не представляется возможным.

Нивелировать этот недостаток поможет применение машинного обучения, которое в общем делает то же самое — «обучается» на выборке данных, строит модель, отражающую в себе поведение пользователя (например, это могут быть «обученная» нейронная сеть или весовые коэффициенты связей персептрона и пр.). Потоковый анализ в данном случае будет состоять в подаче на вход такой модели входного потока. Ее выход имеет два состояния: «транзакция правильная» и «транзакция подозрительная». Модель можно периодически обновлять — уточнять ее на новом наборе данных. Но это может повлечь значительные трудности, связанные с возможностью переобучения, то есть построения неверной модели на переизбытке данных. Вообще, тема интеллектуального анализа данных и облачных сервисов, предоставляющих инструменты для его проведения, очень обширна и требует отдельной книги, а потому далее рассматриваться не будет. Я надеюсь, вы не обидитесь на меня за это. Во введении было указано, что машинного обучения в книге практически не будет. Если читатели проявят интерес к данной теме — такая книга, безусловно, появится.

Сами алгоритмы анализа в различных системах можно описать тремя разными способами:

- в виде сценария SQL-подобного синтаксиса;
- с помощью специализированного синтаксиса запросов, отличного от SQL;

- через программу на компилируемом или интерпретируемом языке программирования общего назначения, которая использует тот или иной фреймворк обработки данных.

Рассмотрим эти способы более подробно.

Использование SQL для обработки данных различной природы получило распространение совсем недавно. Традиционно для этой цели служило «складирование» данных в одно общее реляционное хранилище (DWH), ввиду чего они извлекались из своих источников, подвергались трансформации и помещались в таблицы SQL DWH. Затем к этим данным можно применить запросы SQL и сделать доступными для традиционных средств BI. В последнее время появились сервисы, позволяющие применять SQL-подобный синтаксис к данным, хранящимся в HDFS (SparkSQL, HiveQL, AWS Athena SQL, AWS Redshift Spectrum, Azure Data Lake Analytics), и к потоковым данным (AWS Kinesis Analytics, Azure Stream Analytics). Такой подход очень удобен, поскольку запросы, используемые при интерактивном анализе данных, можно задействовать напрямую для построения задач пакетного и потокового анализа. Кроме того, BigData-разработчик, работающий со всеми этими аспектами обработки данных, должен хорошо знать SQL, а не три совершенно не связанных между собой синтаксиса или фреймворка языка программирования общего назначения.

Следующий способ описания алгоритмов обработки данных — специализированные языки манипулирования данными. Некоторые из них имеют достаточно давнюю историю и применялись для обработки данных до того, как появился сам термин «большие данные», хотя, безусловно, физически большие объемы данных, требующие обработки, существуют достаточно давно. В качестве примера можно назвать такие языки (а в ряде случаев скорее целые экосистемы), как PigLatin, R, MatLab, SPSS, Elasticsearch Query, PowerBI query language и др. Некоторые стали де-факто стандартом обработки данных (например, R) и представлены в облачных средах в виде готовых сервисов (Azure HDInsight R-Server, Power BI) или образов виртуальной машины. Подобные языки используются для интерактивного анализа данных в системах аналитики или BI. Сам анализ производится в специализированных серверах аналитики, к которым подключены источники данных. В отличие от этих языков PigLatin в облачных системах служит в качестве альтернативы программной реализации паттерна MapReduce для целей пакетной обработки данных (чаще всего это процедуры ETL/ELT) в среде Apache Pig.

Особняком среди этих языков стоит язык запросов Elasticsearch. В настоящее время он очень широко используются в стеке программных продуктов *ELK* (рис. IV.1), состоящем из следующих компонентов: *Elasticsearch* — кластер системы хранения и индексирования информации; *Logstash* — набор коннекторов, обеспечивающих поступление информации в Elasticsearch; *Kibana* — система отображения информации в виде интерактивных графиков.

Информация добавляется в Elasticsearch путем REST-запросов, которые могут быть посланы с помощью сервисов Logstash, пользовательскими программами или сценариями командной оболочки. Непосредственно Logstash представляет собой

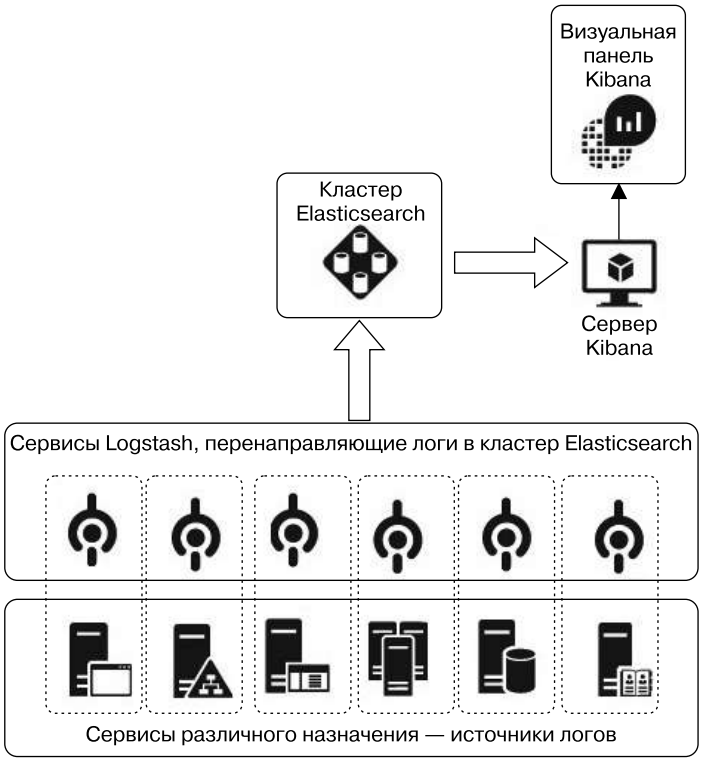


Рис. IV.1. Общая архитектура стека ELK

специализированный сервис стека ELK, устанавливаемый на серверы-источники логов или специализированные серверы с доступом к логам источников. Logstash отвечает за захват логов из источников, их фильтрацию и преобразование и перенаправление в приемник — кластер Elasticsearch. Для обеспечения парсинга логов источника служит специальный синтаксис и расширение — Grok.

После добавления информации в кластер Elasticsearch и ее индексирования (то есть упорядочения с помощью структуры данных, обеспечивающей высокую скорость поиска) она становится доступна для поиска через запросы. В основе Elasticsearch лежит Apache Lucene — поисковый движок, отвечающий за построение массивно-параллельной системы хранения и поиска информации. Помимо поиска, информация в Elasticsearch доступна для анализа путем языка запросов и отображена с помощью Kibana.

Стек ELK в облачных средах в настоящее время используется в качестве системы централизованного хранилища и анализа логов, к которой с помощью Logstash можно подключить различные хранилища. ELK настолько популярен, что, например, в облачном провайдере AWS представлен как сервис AWS Elasticsearch

Service. Кроме того, возможна установка отдельных компонентов в виде контейнеров Docker в сервисах кластеров, предоставляемых облачными провайдерами.

И наконец, для данных можно применить языки программирования общего назначения, которые вместе со специальными библиотеками компилируются (или транслируются, если это интерпретируемый язык) в исполняемые программы, запускаемые в соответствующем сервисе. Чаще всего в качестве таких языков выступают Java, Scala и Python. По сравнению с SQL и доменно-специфичными языками они имеют преимущества, связанные с очень большой гибкостью и разнообразием прикладных библиотек. Java, Scala и Python позволяют (по крайней мере потенциально) реализовать любой сколь угодно сложный алгоритм анализа данных. В то же время реализация типовых конструкций анализа (фильтрация, агрегирование, соединение и пр.) требует достаточно большого количества строк кода. Кроме того, люди, не имеющие опыта разработки и отладки программ на языках программирования общего назначения, могут испытывать сложности, применяя сервисы анализа, основанные на этих языках.

Ряд сервисов анализа данных (например, система Spark) предоставляет внешний веб-интерфейс, в котором можно применять Python в интерактивном режиме. Мы уже встречались с этим фактом, рассматривая выполнение ETL с помощью AWS Glue. Другие сервисы, такие как Hadoop MapReduce, требуют создания программ на компилируемом языке Java. Более подробно эти системы и нюансы их использования будут рассмотрены ниже.

Как уже отмечалось ранее, цель всего анализа данных — не получение новой информации, а поиск и отображение существующей, скрытой в большом объеме данных. Фильтрация, группировка, отображение, с одной стороны, и соединение с объединением — с другой, — это лишь способы предоставления существующей информации в виде, самом удобном и наглядном для интерпретации. Зачастую такой подход называется data mining («добывание информации») — по аналогии с добычей полезного минерала из пустой породы, тоже состоящей из минералов, но не представляющей пока интереса для анализа.

Особенно хорошо этот подход изучать на примере аналогии — добычи золота. Прежде чем начать добычу на том или ином месторождении, геологоразведчик анализирует образцы из различных проб, чтобы оценить концентрацию золота (аналог *интерактивного анализа*). Как только определено, что золото в пробах есть, следует оценить экономическую целесообразность его добычи. Для этого необходимо, чтобы концентрация золота на месторождении была достаточно высокой. (Пороговое значение зависит от транспортной и общей освоенности территории вокруг предполагаемого месторождения: чем более глухое и труднодоступное место, тем выше должна быть концентрация золота, чтобы его было выгодно осваивать.) После того как установлена целесообразность добычи, начинается собственно процесс разработки месторождения (аналог *пакетного анализа*). При этом чаще всего происходит многостадийный процесс обогащения и очищения, для чего порой золотоносная порода после первоначальной обработки транспортируется за пределы своего первоначального источника (аналог *ETL*) или (как, например, в случае технологии

кучного выщелачивания) транспортируется без предварительного обогащения к месту обработки и уже там подвергается действию цианидных реагентов (аналог *ELT*). Кроме того, золото можно добыть (хотя бы потенциально) в качестве попутного компонента в других технологических процессах: рафинирования меди, из вулканических вод или непосредственно из морской воды (аналог *потокового анализа*). Но в ряде случаев происходит стихийная добыча старателями, которые мечтают найти свой огромный самородок. Часто кластерные системы анализа данных используются именно в таком режиме: проанализировать логи за месяц в поисках 500 ошибок, определить причину странного поведения системы (падения производительности, увеличения времени отклика и т. д.) и пр.

Теперь рассмотрим облачные сервисы анализа данных, предоставляемые Azure и AWS.

12 Интерактивный анализ данных

— Копайте, копайте, ребята, — сказал Сильвер с холодной насмешкой. — Авось выкопаете два-три земляных ореха. Их так любят свиньи.

Роберт Стивенсон. Остров сокровищ

12.1. Общие сведения об интерактивном анализе данных

Интерактивный анализ предполагает активное участие специалиста по обработке данных (data scientist). Этот человек выдвигает гипотезы, выбирает программные инструменты анализа, определяет источники данных, применяет к ним запросы на языке, который поддерживают эти инструменты. Сервисы интерактивного анализа предоставляют средства отображения информации в виде таблиц или графиков. Такой анализ применяется, когда надо изучить данные в ответ на однократный запрос бизнеса. Например, установить тенденцию в бизнес-данных, не отображаемую в периодических отчетах. Трудность интерактивного анализа заключается в том, что требуется создать систему, которая позволяет выполнять запросы пользователя к большому объему данных (речь идет не о петабайтах, а «всего лишь» о многих гигабайтах) за достаточно малое время, исчисляемое минутами. Это необходимо для того, чтобы работа специалиста по обработке данных была максимально эффективной и цепочка «выдвижение гипотез — написание запроса — выполнение запроса — анализ результата — корректировка гипотез...» занимала наименьшее время. Очевидно, что для построения такой системы подойдут ресурсы или сервисы, создаваемые по требованию. Как правило, они состоят из хранилища данных, допускающего быстрый анализ; кластера, предоставляющего вычислительные ресурсы для построения запроса и выполнения параллельных вычислений; сервисов

копирования и трансформации данных, служащих для перемещения информации из сторонних источников в это хранилище.

Я могу привести реальный пример применения интерактивного анализа данных, принесший ощутимую материальную пользу владельцу информационной системы. Заказчик предоставлял своим клиентам сервис дистанционного мониторинга и управления подключенными приборами здания: отопления, вентиляции, кондиционирования и пр. Данные телеметрии после цепочки IoT-шлюзов и промежуточных облачных сервисов помещались в облачное хранилище Azure Table Storage, где в конечном итоге составили порядка нескольких терабайт. Возникла проблема: в одном из зданий периодически ломался контроллер. Замены ее не решали, поломки продолжались, значит, имелась какая-то внешняя причина, вызывающая некие события в электросети, поломки и дорогостоящие ремонты. Прямая проверка электросети никакого результата не давала, а приносила одни затраты. В итоге заказчик принял решение проанализировать данные, задействовав сервисы Azure BigData. Для этого с помощью сервиса Azure Data Factory данные копировались в хранилище Azure Data Lake и затем с использованием кластера Azure HDinsight Spark подверглись анализу. После этого выяснилось, что причиной непрекращающихся дорогостоящих поломок послужила неисправная газоразрядная лампа, дававшая серьезные помехи и пульсации сетевого напряжения. Проблема была устранена, кластер и хранилище Azure Data Lake удалены. Затраты на облачные ресурсы анализа данных и рабочее время специалиста окупились за пару месяцев.

12.2. Анализ реляционных данных

Наиболее распространенная модель хранения данных в различных информационных системах — реляционная, в которой информация логически представлена в виде связанных таблиц. Для выборки и обновления информации в этих таблицах используется специальный стандартный язык запросов — SQL (Structured Query Language). Его поддерживают очень многие базы данных, хранилища и средства аналитики, потому рассмотрим его подробнее. Следует заметить, что, несмотря на единообразие и широкое распространение SQL как стандарта, существует множество различных синтаксисов этого языка, являющихся его конкретными реализациями (например, T-SQL от Microsoft, PL/SQL от Oracle, PostgreSQL и пр.). Разница между ними проявляется в различных поддерживаемых типах данных, применении встроенных функций, расширяемости пользовательских типов данных, процедурах и функциях, а также в различной поддержке стандартных синтаксических конструкций. Тем не менее, зная базисные элементы SQL и принципы построения запросов, вы относительно легко можете освоить любой синтаксис SQL от любого производителя.

Для взаимодействия с реляционной БД или реляционным хранилищем требуется внешняя программа с графическим интерфейсом редактора, например SQL

Management Studio для Microsoft SQL Server или SQL Workbench для PostgreSQL. Облачные среды порой предоставляют онлайн-интерфейсы редакторов запросов (с рядом из них вы познакомились в главе 2).

Все операторы языка SQL можно разделить на две группы: DDL (Data Definition Language — язык описания данных) и DML (Data Manipulation Language — язык манипулирования данными).

DDL — это группа операторов, предназначенная для описания схемы данных, то есть таблиц и связей между ними. Содержит операторы создания (`CREATE`), изменения (`ALTER`) и удаления (`DROP`) объектов хранения (таблиц) и манипулирования (пользовательские функции, хранимые процедуры и т. п.) данными. Кроме того, DDL включает в себя операторы указания типа данных (числовые, текстовые, символьные, XML и пр.), организации и связи данных (первичный и внешний ключи, индекс, автоинкрементное поле и др.). Теперь посмотрим, как с помощью этих операторов создавать таблицы. Для простоты я сделаю это для Azure SQL, поскольку данный сервис содержит встроенный онлайн-редактор запросов, который не нужно инсталлировать и подключать.

Продолжим рассматривать наш пример с покер-румом. Предположим, нужно создать базу данных клиентов рума, где будут храниться сведения о клиентах, их адреса, контактная информация, клиентский счет, а также информация о платежных транзакциях по счету.

Анализ предметной области привел к структуре базы данных, состоящей из пяти таблиц (рис. 12.1).

- `Clients` — таблица клиентов, содержащая общие сведения о клиенте: его тип (имеется связь по внешнему ключу с таблицей типов клиентов), имя, фамилия, описание, дата рождения, имя пользователя в системе и дата создания клиента.
- `ClientTypes` — таблица типа клиентов. Таковыми могут быть клиенты, играющие с разными лимитами, премиум, стандарт, ВИП и др. Таблица содержит номер типа, имя, описание и дату создания. Отношение между клиентами и типами клиентов является «один к одному», то есть каждый клиент должен иметь один тип.
- `Contacts` — таблица контактов клиента. Каждый клиент может иметь один или несколько контактов. Запись этой таблицы состоит из сведений о стране, городе, адреса в пределах города, телефона, адреса электронной почты и описания, куда можно включить особые случаи.
- `ClientAccounts` — таблица игровых счетов клиентов. Каждый клиент может иметь только один счет (отношение «один к одному»). В этой таблице хранятся сведения о банковском номере клиента (или его идентификатор в реальной внешней платежной системе), сумме на счету, дате создания и полях описания.

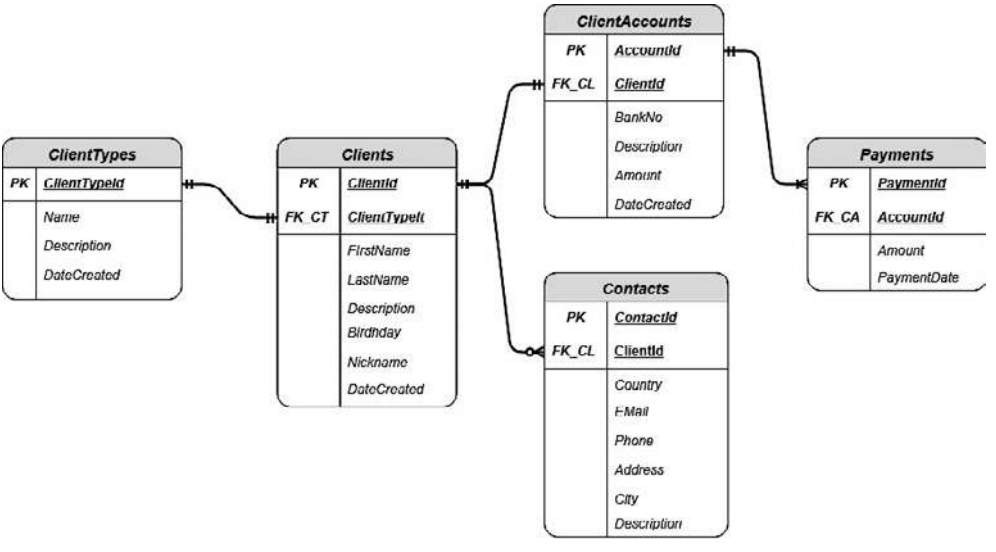


Рис. 12.1. Схема реляционной базы данных клиентов

- `Payments` — таблица клиентских финансовых транзакций на счету клиента. К таковым относятся поступление на счет и снятие с него. В данной БД это самая «быстрорастущая» таблица, и по-хорошему ее следует поместить в реляционное хранилище, поскольку при наличии множества длительно играющих клиентов она будет занимать большой объем. Но для целей демонстрации мы оставим ее. Кроме того, при выполнении финансовой транзакции происходит обновления баланса на клиентском счету, а сами транзакции лишь описывают акты изменения этой величины. Таблица содержит две величины: объем, вносимый на счет, и дату транзакции. Если со счета снимаются средства, то величина вносимых средств идет со знаком «минус».

Начнем создание базы с таблицы типов клиентов. Сценарий DDL создания таблицы `ClientTypes` имеет следующий вид (напомню: в этом примере используется синтаксис T-SQL для Azure SQL) (рис. 12.2).

```
CREATE TABLE ClientTypes
(
    [ClientTypeId] INT NOT NULL PRIMARY KEY IDENTITY(1,1),
    [Name] NVARCHAR(20),
    [Description] NVARCHAR(1000),
    [DateCreated] DATETIME
)
```

Рис. 12.2. Сценарий создания таблицы ClientTypes

В этом сценарии создается таблица с полями.

- `ClientTypeId` — первичный ключ (`PRIMARY KEY`) таблицы, имеющей тип `INT` (целое 32-разрядное число), которое не допускает пустых значений (`NOT NULL`) и автоматически увеличивает свое значение на единицу с минимальным значением, равным 1 (`IDENTITY(1,1)`). Въедливый читатель может упрекнуть меня в применении избыточного типа данных — четырехбайтного `INT` вместо двухбайтного `SMALLINT` или даже однобайтного `TINYINT`. Я рекомендую таким читателям поупражняться с языком DDL, используя операторы `ALTER` или в крайнем случае `DROP/CREATE`.
- Поля `Name` и `Description` представляют собой текстовые типы данных, оперирующие набором символов Unicode. Поле `Name` ограничено 20 символами (`NVARCHAR(20)`), а поле `Description` — 1000 символами (`NVARCHAR(1000)`).
- Поле `DateCreated` имеет временно́й тип — `DATETIME`.

Далее создадим таблицу `Clients` (рис. 12.3).

```
CREATE TABLE Clients
(
    [ClientId] INT NOT NULL PRIMARY KEY IDENTITY(1,1),
    [ClientTypeId] INT,
    [FirstName] NVARCHAR(20),
    [LastName] NVARCHAR(20),
    [NickName] NVARCHAR(20),
    [Description] NVARCHAR(1000),
    [Birthday] DATETIME,
    [DateCreated] DATETIME,
    CONSTRAINT FK_ClientTypes_ClientTypeId FOREIGN KEY (ClientTypeId)
    REFERENCES ClientTypes(ClientTypeId)
)
```

Рис. 12.3. Сценарий создания таблицы Clients

Помимо уже известных типов данных, здесь приведен пример создания ограничения внешнего ключа. В нашем случае это строка `CONSTRAINT FK_ClientTypes_ClientTypeId FOREIGN KEY (ClientTypeId)`. Имя ограничения может быть любое, но я дал составное, включающее префикс `FK_`, имя таблицы, на которую нужно ссылаться (`ClientTypes_`), и имя поля ключа в ней (`Client TypeId`). Ограничение с таким именем можно удалять и изменять. Это ограничение связывается с ключом `ClientTypeId` таблицы `ClientTypes` с помощью синтаксической конструкции `REFERENCES ClientTypes(ClientTypeId)`.

Другие таблицы (`Contacts`, `ClinentAccounts` и `Payments`) создаются точно так же (рис. 12.4).

По поводу этих сценариев я хочу сделать следующие комментарии. Во-первых, для хранения денежных величин применен тип данных `DECIMAL` — десятичное

```
CREATE TABLE [dbo].[ClientAccounts]
(
    [AccountId]   INT PRIMARY KEY IDENTITY (1, 1) NOT NULL,
    [ClientId]    INT,
    [BankNo]      NVARCHAR (100),
    [Amount]      DECIMAL,
    [Description] NVARCHAR (1000),
    [DateCreated] DATETIME,
    CONSTRAINT [FK_Acc_Clients_ClientId] FOREIGN KEY ([ClientId]) REFERENCES [dbo].[Clients] ([ClientId])
);
CREATE TABLE [dbo].[Payments]
(
    [PaymentId]   INT PRIMARY KEY IDENTITY (1, 1) NOT NULL,
    [AccountId]   INT,
    [Amount]      DECIMAL,
    [PaymentDate] DATETIME,
    CONSTRAINT [FK_ClientAccounts_AccountId] FOREIGN KEY ([AccountId]) REFERENCES [dbo].[ClientAccounts] ([AccountId])
);
CREATE TABLE Contacts
(
    [ContactId]   INT PRIMARY KEY IDENTITY (1, 1) NOT NULL,
    [ClientId]    INT,
    [Country]     NVARCHAR (50),
    [City]        NVARCHAR (50),
    [Address]     NVARCHAR (50),
    [Email]       NVARCHAR (50),
    [Description] NVARCHAR (1000),
    [Phone]       NVARCHAR(50),
    [DateCreated] DATETIME,
    CONSTRAINT [FK_Clients_ClientId] FOREIGN KEY ([ClientId]) REFERENCES [dbo].[Clients] ([ClientId])
);
```

Рис. 12.4. Сценарии создания таблиц Contacts, ClinentAccounts и Payments

число с фиксированной точностью. При именовании таблиц можно использовать как прямое имя без пробелов — `Contacts`, так и полное имя с указанием названия схемы — `[dbo].[ClientAccounts]`. В данном случае схема — это логическая группа объектов (таблиц, хранимых процедур и др.). Схема по умолчанию — `dbo` (data base owner), и все объекты, которые созданы без указания ее, автоматически помещаются в данную схему.

Итак, мы изучили основы синтаксиса SQL DDL на примере простой базы данных Azure SQL. Теперь займемся другой группой SQL-операторов — DML (Data Manipulation Language — язык манипулирования данными). Эти операторы служат для выполнения операций с данными в созданной базе: добавления, удаления, извлечения и изменения. Рассмотрим операторы добавления данных в таблицы, заполнив нашу БД конкретными данными. Начнем с заполнения таблицы типов клиентов (рис. 12.5).

```
DECLARE @CurrentDate DATETIME
SET @CurrentDate = GETDATE()

INSERT INTO ClientTypes ([Name], [Description], [DateCreated] )
VALUES ('Free', N'Игрок без введения денег на счета и вывода выигрыша. Играет бесплатно', @CurrentDate),
('Ordinary', N'Игрок с базовыми возможностями ввода денег и вывода их со счета. Имеет наименьшие лимиты вывода', @CurrentDate),
('Premium', N'Игрок без ограничения лимитов ввода и вывода средств', @CurrentDate)
```

Рис. 12.5. Сценарий заполнения таблицы типов клиентов

Здесь прежде всего объявляется переменная `@CurrentDate` типа `DATETIME`, которая будет содержать текущие дату и время. Значение ей присваивается с помощью встроенной функции `GETDATE()`, возвращающей текущие значения даты и времени. Далее используется оператор DML — `INSERT INTO`. Затем указывается имя таблицы (в нашем случае это `ClientTypes`) и список колонок, в которые будут вставляться данные. В списке отсутствует первичный ключ, поскольку он был создан автоинкрементным. Непосредственно вставка строк производится после ключевого слова `VALUES`. Далее следует список вставляемых строк. Вообще операторы вставки строк очень разнообразны, но все мы рассматривать не будем.

Сценарий для добавления клиентов в таблицу клиентов выглядит следующим образом (рис. 12.6).

```
DECLARE @CurrentDate DATETIME
SET @CurrentDate = GETDATE()

INSERT INTO dbo.Clients ([ClientTypeId], [FirstName], [LastName], [NickName], [Description], [Birthday], [DateCreated])
VALUES  ( 1, 'Alexey', 'Petrov', 'AP', N'Леша из Москвы', N'12-29-1985',  @CurrentDate),
        ( 2, 'Dmitry', 'Semenov', 'AP', N'Дима из Харькова', N'10-11-1980',  @CurrentDate),
        ( 2, 'Andrei', 'Nikodimov', 'AP', N'', N'09-12-1972',  @CurrentDate),
        ( 3, 'Michael', 'Stark', 'AP', N'Mike from Washington DC', N'09-20-1950',  @CurrentDate),
        ( 1, 'Sven', 'Ericsson', 'AP', N'', N'07-29-1961',  @CurrentDate)
```

Рис. 12.6. Сценарий заполнения таблицы клиентов

Здесь следует отметить одну синтаксическую особенность — вставку строкового типа, преобразуемого к типу `DATETIME` в поле `Birthday`. Далее нужно заполнить таблицы контактов и банковских счетов клиентов (рис. 12.7, 12.8).

```
DECLARE @CurrentDate DATETIME
SET @CurrentDate = GETDATE()

INSERT INTO Contacts ([ClientId], [Country], [City], [Address], [Email], [Description], [Phone], [DateCreated])
VALUES
   ( 2, 'Russia', 'Moscow', 'Petrovka 30', 'Petrov@contoso.com', '', '+123456789', @CurrentDate ),
   ( 3, 'Ukraine', 'Kharkov', 'Tarasa Shevchenko 5', 'Semenov@example.com', '', '+987654123', @CurrentDate ),
   ( 4, 'Belarus', 'Minsk', 'Jacuba Kolasa 7', 'Nikodimov@cont.com','', '+654978123',  @CurrentDate ),
   ( 5, 'USA', 'Washington DC', 'Capitoly hill', 'stark@michael.com', '', '+312945687', @CurrentDate ),
   ( 6, 'Sweden', 'Stockholm', 'Eriksbersgatan 21', 'ericsson@els.com', '', '+951736248', @CurrentDate  )
```

Рис. 12.7. Сценарий добавления данных в таблицу контактов

```
DECLARE @CurrentDate DATETIME
SET @CurrentDate = GETDATE()

INSERT ClientAccounts ( [ClientId], [BankNo], [Amount], [Description], [DateCreated] )
VALUES ( 2, '987456321', 0, '', @CurrentDate ),
( 3, '321456987', 70, '', @CurrentDate  ),
( 4, '654321789', 500, '', @CurrentDate ),
( 5, '789456123', 250, '', @CurrentDate ),
( 6, '852369741', 0, '', @CurrentDate )
```

Рис. 12.8. Сценарий добавления данных в таблицу клиентских счетов

Таблицу банковских транзакций мы заполним чуть позже.

И вот наконец мы подошли к вопросу построения запросов на выборку данных. Любой запрос на любом SQL-подобном языке включает следующие компоненты.

- Набор данных в виде таблицы или источника данных, которые могут быть представлены в виде таблицы. Как уже указывалось выше, этот набор состоит из столбцов, каждый из которых имеет имя. Чтобы получить доступ к данным в столбце, необходимо в запросе обратиться по имени этого столбца. В дальнейшем наборы данных будем называть таблицами.
- Список столбцов (project — проекция в терминологии доменно-специфических запросов), данные из которых подлежат отображению.
- Предикат, или выражение-фильтр, применяемое ко всем строкам результата запроса и для тех строк, которые делают данный предикат равным истине (true). Это позволяет выделить из всего набора строк только нужные.
- Оператор сортировки с указанием имени столбца и направления (возрастание или убывание), по которому должен быть отсортирован результат выборки.

Набор данных может быть как одиночной таблицей, так и их *соединением* (JOIN). Это позволяет получать выборку информации, хранящейся в связанных таблицах. Соединение может быть «внутренним» (тип связи между таблицами — «один к одному»), «внешним правым», «внешним левым» (тип связи между таблицами — «один ко многим»).

Соединение таблиц выполняется наиболее быстро по ключевым полям, то есть по первичному ключу одной таблицы и внешнему ключу второй. Если на эти ключи имеются индексы, то производительность будет максимальной. В ряде случаев, когда необходимо связать несколько таблиц, для построения высокопроизводительного запроса очень важен порядок, в котором выполняется соединение. Но такая оптимизация запросов должна рассматриваться отдельно для каждого SQL-движка, будь то реляционная база данных того или иного типа либо нереляционное хранилище с возможностью выполнения SQL-запросов.

Для примера зададимся целью определить имена клиентов, суммы на их счетах и страны, где клиенты проживают при условии, что их банковский счет превышает 10 (рис. 12.9). В этом примере итоговое поле `[Имя клиента]` получается путем конкатенации трех строк: имени, фамилии и пробела между ними. В T-SQL оператором конкатенации строк является +, что, как правило, не совпадает с таковым в других диалектах SQL. Таблицам в выборке даются укороченные имена — *псевдонимы* (alias). В запросе это `CA` для таблицы `ClientAccount`, `CL` — для `Clients` и `CN` — для `Contacts`. Все они объединяются внутренним соединением, поскольку тип связи между ними — «один к одному».

Прежде чем рассматривать другие виды выборки, необходимо заполнить таблицу транзакций. Это может быть достаточно утомительной процедурой, а потому покажу, как ее автоматизировать, а заодно изучим специальные приемы,

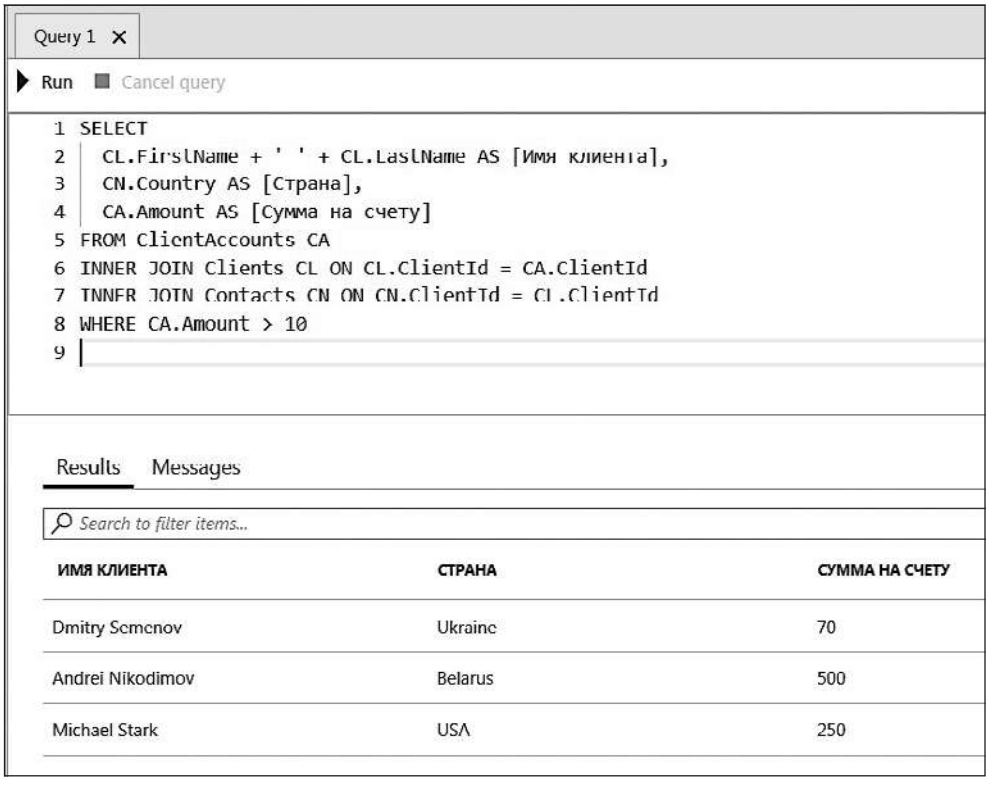


Рис. 12.9. Простая выборка данных

позволяющие инкапсулировать часто используемые запросы в одну сущность — *хранимые процедуры*. Такая процедура представляет собой объект базы данных, включающий скомпилированный набор операторов SQL.

Чтобы понять, как это работает, поставим себе такую методическую задачу — заполнить таблицу транзакций (`Payments`) случайными данными, причем заполняться должны строки только для игроков, не относящихся к типу `Free`. Одним из решением может быть хранимая процедура, сценарий которой приведен на рис. 12.10. Она включает в себя набор входных параметров, которым даны значения по умолчанию: `@MinClientId`, `@MaxClientId`, `@MinAmount` и `@MaxAmount`. Для минимального и максимального значений приведены значения `ClientId`, которые есть в нашей таблице `Clients`. Далее в хранимой процедуре декларируются переменные, хранящие текущую дату (`@CurrentDate`), и переменные, содержащие случайные значения номера клиента (`@RandomClient`) и случайное значение денежной величины, вносимой на счет (`@RandomAmount`). При этом используется встроенная функция `RAND`, возвращающая случайную величину типа `REAL` в диапазоне 0–1, и функция `ROUND`, округляющая аргумент до заданной точности.

```
CREATE PROCEDURE RandomFillPaymentTransactions
   @MinClientId INT = 2, @MaxClientId INT = 6, @MinAmount DECIMAL = - 50, @MaxAmount DECIMAL = 50
AS
BEGIN
   DECLARE @CurrentDate DATETIME = GETDATE()
   DECLARE @RandomClient INT = ROUND(((@MaxClientId - @MinClientId -1) * RAND() + @MinClientId), 0)
   DECLARE @RandomAmount DECIMAL = ROUND(((@MaxAmount - @MinAmount -1) * RAND() + @MinAmount), 0)
   DECLARE @AccountId INT =    (
       SELECT CL.ClientId
       FROM ClientAccounts CA
       INNER JOIN Clients CL ON CA.ClientId = CL.ClientId
       INNER JOIN ClientTypes CT ON CT.ClientTypeId = CL.ClientTypeId
       WHERE CL.ClientId = @RandomClient AND CT.ClientTypeId != 1  )
   IF @AccountId IS NOT NULL
   BEGIN
   INSERT INTO Payments ([AccountId], [Amount], [PaymentDate])
   VALUES (@AccountId, @RandomAmount, @CurrentDate)
   END
END
```

Рис. 12.10. Пример хранимой процедуры для заполнения таблицы Payments случайными данными

Далее с помощью известного нам запроса выбирается номер клиентского счета (`@AccountId`). В запросе на его выборку приведено условие — тип клиента не должен быть равен `1`, поскольку он не предполагает клиента, не играющего с реальными деньгами. Таким образом, если случайно переменная `@RandomClient` укажет на такого клиента, то значение `@AccountId` станет равным `NULL`. Далее следует условный оператор, который проверяет, есть ли численное значение у этой переменной, и если да, то в таблицу `Payments` вставляется новая строка.

Чтобы автоматизировать заполнение таблицы, можно как вариант применить следующий цикл (рис. 12.11).

```
DECLARE @Counter INT = 0, @MaxRecords INT = 1000

WHILE @Counter < @MaxRecords
BEGIN
   SET @Counter = @Counter + 1;
   EXEC RandomFillPaymentTransactions
END;
```

Рис. 12.11. Использование хранимой процедуры в цикле для заполнения таблицы Payments

Количество вставленных строк в таблицу может быть заметно меньше, чем величина `@MaxRecords`, поскольку добавляться будут не все записи, а только те, которые не соответствуют клиентам с типом 1.

Очень важный синтаксический элемент языка запросов — *группировка* (также называемая агрегированием) по какому-либо столбцу. Группировка по данным из столбца представляет собой следующее. В группируемом столбце есть значения,

которым соответствуют несколько строк с разными значениями других столбцов. При группировке все строки разделяются на группы строк, для которых значение группируемого столбца одинаково.

Рассмотрим группировку на таком примере: необходимо подсчитать сумму всех величин транзакций для каждого игрока и среднюю величину транзакции (рис. 12.12).

```
SELECT
    AVG(PA.Amount) [Средняя величина транзакций],
    SUM(PA.Amount) [Полная сумма транзакций],
    CL.FirstName + ' ' + CL.LastName AS [Имя клиента]
FROM Payments PA
INNER JOIN ClientAccounts CA ON CA.AccountId = PA.AccountId
INNER JOIN Clients CL ON CL.ClientId = CA.ClientId
GROUP BY CL.FirstName, CL.LastName
```

Рис. 12.12. Сценарий для определения общей суммы транзакций и средней величины транзакции

В этом сценарии применен оператор группировки `GROUP BY`, который выполняется для двух полей — `Clients.FirstName` и `Clients.LastName`. Для строк, соответствующих каждой группе, вычисляется сумма (с помощью функции `SUM`) и среднее значение (функция `AVG`). Если необходимо отфильтровать результаты группировки по агрегированным величинам (скажем, определить тех игроков, чей суммарный баланс выше заданной величины), то в конце сценария следует добавить оператор `HAVING SUM(PA.Amount) > @Amount`, где `@Amount` — искомый порог.

Мы изучили ряд встроенных функций, теперь посмотрим, как создавать пользовательские функции, на примере решения следующей задачи. Предположим, возникло рассогласование между величиной на счете игрока в таблице `ClientAccount` и в таблице транзакций и его необходимо устранить, обновив таблицу `ClientAccount` сгруппированными данными из таблицы `Payments`. Но проблема в том, что наши игроки вносили и забирали средства в различной валюте и надо пересчитать итоговую сумму для каждого игрока в валюте общего счета. Для этого создадим следующую функцию (рис. 12.13).

```
CREATE FUNCTION ConvertCurrency(@Rate DECIMAL, @InputValue DECIMAL)
RETURNS DECIMAL
AS
BEGIN
    DECLARE @Result DECIMAL = @Rate * @InputValue;
    RETURN  @Result
END
```

Рис. 12.13. Пример скалярной пользовательской функции

Если требуется скомбинировать данные из нескольких таблиц, не связанных ключами, то необходимо *объединить* (`UNION`) результаты запросов из различных таблиц. В отличие от соединения объединение есть пересечение множеств.

Выполнять сортировку результатов можно и с помощью оператора `ORDER BY`, за которым следует имя столбца, по которому происходит сортировка.

Кроме этих основных операторов и опций языка SQL, его диалекты включают множество других возможностей: оконные и табличные функции, определяемые пользователем типы и др. Ограниченный объем книги не позволяет рассмотреть все эти возможности. Остановимся на ряде очень важных понятий, которые будут играть ключевую роль при построении высокопроизводительных запросов к большим анализам данных. В нашей БД большое количество данных в таблицах может возникнуть, если в таблицах будет присутствовать множество игроков и сопутствующих им атрибутов (транзакций, адресов). Или, например, вы переборщите с вставкой новых строк с помощью цикла с рис. 12.11, выбрав в качестве величины `@MaxRecord` число, близкое к миллиону. (Конечно, это гипотетическая ситуация и такая вставка займет много времени, но... я в ранней юности умудрился сжечь электротехнический стенд для проведения лабораторных работ по теме «Короткое замыкание источников ЭДС», выполняя дословно указания методички, но ошибившись в одном параметре.) Ранее уже неоднократно упоминались такие объекты, как *индексы*. В ряде случаев они добавляются автоматически, здесь же кластерный индекс добавляется на первичный ключ. Это можно заметить, если посмотреть схему таблицы в Microsoft SQL Management Studio или Visual Studio. Так, для таблицы `Payments` сценарий с явным указанием кластерного индекса имеет следующий вид (рис. 12.14).

```
CREATE TABLE [dbo].[Payments] (
    [PaymentId]   INT          IDENTITY (1, 1) NOT NULL,
    [AccountId]   INT          NULL,
    [Amount]      DECIMAL (18) NULL,
    [PaymentDate] DATETIME     NULL,
    PRIMARY KEY CLUSTERED ([PaymentId] ASC),
    CONSTRAINT [FK_ClientAccounts_AccountId] FOREIGN KEY ([AccountId]) REFERENCES [dbo].[ClientAccounts] ([AccountId])
);
```

Рис. 12.14. Схема таблицы Payments с явным указанием кластерного индекса

Особенность Microsoft SQL и, как следствие, Azure SQL и Azure SQL DWH состоит в том, что кластерный индекс автоматически добавляется на первичный ключ и может быть только один. Это требует тщательного продумывания схемы реальной базы данных, особенно если предполагается, что размер ее таблиц станет очень большим. Почему кластерный индекс может быть только один? Да потому, что он физически упорядочивает строки таблицы в структуру типа сбалансированного дерева, благодаря чему существенно ускоряется операция поиска строки. Вот почему операции соединения, выполненные по ключам с кластеризованными индексами, выполняются гораздо быстрее, чем для полей без таких индексов.

Для более глубокого понимания факта, почему упорядочение ускоряет поиск, а не только голого знания об этом предлагаю такую аналогию с кластерным индексом: можете взять эту книгу, запомнить номер страницы и закрыть ее без закладки (загибов страниц). А теперь попытайтесь отыскать эту страницу. Первый способ — долгий, когда вы перелистываете книгу страница за страницей. А теперь примените другой алгоритм — откройте книгу примерно на середине и посмотрите номер страницы. Он больше того номера, который вы запомнили? Если да, то откройте середину левой половины страниц, если нет, то правой. И повторите алгоритм до тех пор, пока не найдете нужную. А теперь представьте гипотетическую ситуацию, когда пронумерованные страницы расположены в хаотичном порядке и вам необходимо найти каждую страницу и поставить в соответствие каждой из них страницы другой книги. А если у обоих книг страницы перепутаны?

Конечно, поиск страницы в книге объемом пару сотен страниц выглядит не так наглядно. А вот при наличии книги в несколько тысяч страниц (например, собрание сочинений Иммануила Канта в одном томе) скорость поиска ощутимо возрастет.

Кластерный индекс может быть назначен на составной ключ или на два поля, если такой ключ не задан. Книжной аналогией такого подхода является концепция «том-страница» для многотомного издания. И если у вас есть такое издание (например Большая советская энциклопедия), то можете повторить опыт поиска тома и страницы в упорядоченном случае (тома выставлены в порядке возрастания номеров) и неупорядоченном (тома упорядочены рандомно).

Неупорядоченные строки таблиц реляционных баз данных называется кучей. В ряде случаев, помимо упорядочения по кластерному индексу, может потребоваться упорядочение по специальному полю, на которое создается *некластеризованный индекс*. Это требуется для обеспечения быстрого поиска, упорядочения или соединения по полям, отличным от первичного ключа или от относящихся к кластерному индексу. Но в данном случае строки не упорядочиваются, а создается специальный упорядоченный указатель, что позволяет создавать индексы на все столбцы таблицы. Как это работает? Опять-таки вернемся к аналогии с книгой. Допустим, вас интересует не номер страницы сам по себе, а конкретная тема в книге. Кластеризованный индекс книги — номер страницы. Таким образом, вам нужен указатель, отображающий тему на номер страницы, — очевидно, это оглавление. Следующий вариант некластеризованного индекса — список ключевых слов с указанием первой страницы, где они появляются. В ряде случаев удобнее кластеризация не по темам, а по первой букве названия темы — так организована информация в больших массивах энциклопедий.

Здесь возникает очевидный вопрос: а почему бы не создать индексы на все столбцы? Делать это не стоит по нескольким причинам. Во-первых, существенно возрастет объем, занимаемый базой данных, поскольку на некластеризованные

индексы уходит много места. Во-вторых, может значительно упасть производительность операций добавления и обновления строк таблицы. Вот здесь мы встречаемся с существенным противоречием между операциями вставки/удаления/изменения (CRUD — CReate, Update, Delete) и выборки. При необходимости обеспечить высокую производительность CRUD очевидно, что количество некластеризованных индексов должно быть минимальным. В то же время, если нужна высокая производительность аналитических запросов, желательно иметь столько индексов, сколько требуется для повышения производительности данного запроса. Это особенно важно и заметно в случае больших таблиц.

Таким образом, четко видно различие между двумя видами нагрузок для реляционных баз данных: OLTP (онлайн-транзакции) и OLAP (онлайн-аналитики). В случае OLTP индексы пересчитываются после каждой транзакции, а в случае OLAP — только после изменения строк в таблице, например при добавлении новых записей. Вообще же существуют две стратегии наполнения таблиц OLAP.

- Стратегия полного обновления, когда при каждом ETL обновляются все строки таблицы в базе данных OLAP. Это требует передачи больших массивов и высокой производительности операции вставки, но является практически единственным выходом в случае, когда в источниках данных для OLAP могут потенциально изменяться любые строки и поиск измененных по тем или иным причинам затруднен или достаточно длителен.
- Стратегия частичного обновления предполагает добавление только тех строк, которые были изменены со времени предыдущего акта синхронизации. Эта стратегия предполагает, что определение измененных строк в источниках данных будет выполняться достаточно просто и быстро.

В любом случае после наполнения OLAP (а чаще всего это DWH) данными следует пересчитать индексы. Но, как я уже отмечал, для OLAP это производится однократно после выполнения ETL, а не после каждого изменения/добавления строк, как в случае OLTP.

Вообще же количество и разнообразие индексов в реальных базах и хранилищах данных достаточно большое, но я акцентировал внимание на наиболее важных и типичных.

Теперь ознакомимся подробнее с особенностями синтаксиса DDL и DML для DWH, важными для понимания принципов построения высокопроизводительных запросов к процедуре ETL. Начнем с Azure SQL DWH.

Прежде всего, концептуально (вспомним описанные ранее схемы «звезда» и «снежинка») все таблицы бывают трех видов.

- *Таблица фактов* содержит количественные величины, созданные в базе данных OLTP, отражающие некоторые фактические операции. В нашем примере это таблица `Payments`.

- *Таблица измерений* содержит данные отдельных атрибутов, отраженных в таблице фактов. Эти атрибуты изменяются существенно реже, чем факты. В нашем примере это таблица `ClientAccounts`, отражающая клиентский счет, который относится к конкретному факту.
- *Интеграционная таблица* — таблица для промежуточного (поэтому иногда называется staging) или временного хранения информации в цепочке ETL в целях последующей модификации или копирования в основную таблицу фактов.

По типу хранения таблицы могут быть следующими.

- *Обычная таблица* — основной вид таблиц, используемый в DWH для хранения данных. Физически размещается в Azure Storage.
- *Временная таблица* — таблица, существующая во время сеанса (выполнения запроса, функции или хранимой процедуры и пр.). Такие таблицы хранятся в локальных хранилищах вычислительных узлов, а потому скорость доступа к данным в них выше. По завершении сеанса таблица очищается от данных.
- *Внешняя таблица* — таблица, указывающая на данные, хранящиеся вне DWH в виде файлов в хранилищах Azure Blob Storage или Azure Storage Account. При этом возможен прямой импорт данных из этих файлов в DWH с помощью технологии PolyBase, которая будет описана далее.

Как отмечалось в части II при описании Azure SQL DWH, каждая таблица в DWH может иметь один из трех видов распределений: распределение с помощью хеш-функции (hash); последовательное (оно же циклическое) распределение (round robin); репликация (replicate).

Прежде чем показать, как создавать новую таблицу, я расскажу об одном виде индекса, который отсутствовал в базе данных Azure SQL, — индекс `Columnstore`. В этом случае строки таблицы хранятся в колоночном хранилище, что обеспечивает существенное повышение производительности операции чтения. Мы уже рассматривали этот тип хранения в главе 7, посвященной реляционному хранилищу DWH. Согласно документации Microsoft, этот тип обеспечивает десятикратное увеличение производительности и компрессии данных по сравнению с неиндексированными данными. Этот вид индексирования в Azure SQL DWH применен по умолчанию. Чтобы создать таблицу с явно указанным кластеризованным колоночным индексом, нужно выполнить следующий сценарий, показанный на примере создания таблицы `Payments` (рис. 12.15).

```
CREATE TABLE [dbo].[Payments] (
    [PaymentId]   INT NOT NULL,
    [AccountId]   INT,
    [Amount]      DECIMAL,
    [PaymentDate] DATETIME
)
WITH ( CLUSTERED COLUMNSTORE INDEX );
```

Рис. 12.15. Создание таблицы с явным указанием индекса Columnstore

Несмотря на огромные преимущества данного типа индексов в плане производительности и экономии пространства, есть ряд ограничений, когда такие индексы применять не следует.

- Наличие в таблице типов данных VARCHAR(MAX), NVARCHAR(MAX) и VARBINARY(MAX). В этом случае нужно использовать обычные кластеризованные и некластеризованные индексы. Вообще же применение подобных типов данных должно быть четко обосновано как в OLAP, так и в базах данных OLTP и подходит только для небольших таблиц.
- Временные таблицы. В этом случае надо применять именно сеансово-временные таблицы или таблицы, размещенные в куче.
- Небольшие таблицы. Термин «небольшие» в мире DWH означает таблицы с количеством строк до десятков — первой сотни миллионов строк.

Создание индекса требует времени. Поэтому для интеграционных таблиц, а также таблиц с небольшим сроком службы или для часто обновляемых таблиц требуется исключение, ограничивающее применение индексов и размещение таблицы в куче с помощью следующего синтаксиса (рис. 12.16).

```
CREATE TABLE [dbo].[Payments_Stage] (
    [PaymentId]   INT NOT NULL,
    [AccountId]   INT,
    [Amount]      DECIMAL,
    [PaymentDate] DATETIME
)
WITH ( HEAP );
```

Рис. 12.16. Создание таблицы с размещением в куче

Такие таблицы будут иметь наибольшую производительность при операциях добавления строк. Для промежуточного случая, когда требуется создать рабочую таблицу с количеством строк до сотни миллионов, следует применять первичные кластеризованные и вторичные некластеризованные индексы. Первичные очень эффективны для фильтрации и соединения по одному конкретному столбцу. При необходимости включить возможность эффективной фильтрации/поиска для других столбцов добавление вторичного индекса производится с помощью синтаксиса `CREATE INDEX Index_Payments_AccountId ON t1 (AccountId)`.

```
CREATE TABLE [dbo].[Payments_Stage] (
    [PaymentId]   INT NOT NULL,
    [AccountId]   INT,
    [Amount]      DECIMAL,
    [PaymentDate] DATETIME
)
WITH ( CLUSTERED INDEX (PaymentId) );
```

Рис. 12.17. Создание таблицы с первичным кластеризованным индексом на столбец PaymentId

Теперь рассмотрим, как создавать таблицы с разными видами распределений. Вспомним и уточним основные понятия. Распределение таблицы с помощью хеш-функции состоит в построчном разделении таблицы между вычислительными узлами по закону, определяемому хеш-функцией, применяемому к ключу распределения. Синтаксис создания подобной таблицы имеет следующий вид (рис. 12.18).

```
CREATE TABLE [dbo].[Payments_Stage] (
    [PaymentId]   INT NOT NULL,
    [AccountId]   INT,
    [Amount]      DECIMAL,
    [PaymentDate] DATETIME
)
WITH ( CLUSTERED COLUMNSTORE INDEX,  DISTRIBUTION = HASH([PaymentId]) );
```

Рис. 12.18. Создание таблицы с индексом Columnstore и хеш-распределением по ключу PaymentId

В данном случае требуется тщательно выбрать ключ распределения. Зачастую оптимальное значение можно установить с помощью эксперимента, выбрав из нескольких альтернатив. Общие рекомендации таковы:

- столбец должен включать много уникальных значений;
- не содержит или содержит минимальное количество значений `NULL`;
- не имеет типа `DATETIME`;
- не используется в условных операторах `WHERE`;
- служит как ключ соединения и в операциях группировки.

Очевидно, что для нашей таблицы наиболее подходит столбец `PaymentId`, поскольку отвечает первым четырем требованиям.

Хеш-распределение весьма уместно в следующих случаях:

- размер таблицы на диске превышает 2 Гбайт или она содержит очень много строк (десятки и сотни миллионов). Концептуально это крупная таблица фактов;
- предполагаются частые операции обновления (ETL);
- предполагается применение операций соединения.

Таблицы с циклическим перебором создаются по умолчанию, если после ключевого слова `WITH` не указано иное. Их можно задействовать, когда:

- отсутствует или неочевиден ключ распределения; это может быть тип для «первого приближения» структуры DWH;
- не предполагается использование операций соединения с другими таблицами;
- таблица является временной или интеграционной.

Следующий возможный тип распределения таблицы — репликация, которая представляет собой копирование таблицы между вычислительными узлами. Этот тип

эффективен, когда размер таблицы меньше 2 Гбайт. Наличие копий таблицы во всех вычислительных узлах исключает потери времени на передачу информации по сети связи между узлами в случае выполнения операций соединения таблиц.

Синтаксис создания внешних таблиц был описан в главе 5. Рассматривались внешние таблицы, источниками данных в которых являются таблицы других баз данных Azure SQL или файлы в хранилище Azure BLOB Storage. Но можно указать внешний источник в виде файла в Azure Storage Account, Azure Data Lake Store или иного хранилища HDFS. Это позволяет сделать технология PolyBase, которая дает возможность строить запросы к данным, хранящимся в HDFS-совместимых хранилищах или Azure Blob Storage, точно таким же образом, как и для обычной таблицы. Архитектура этого сервиса выглядит следующим образом (рис. 12.19).

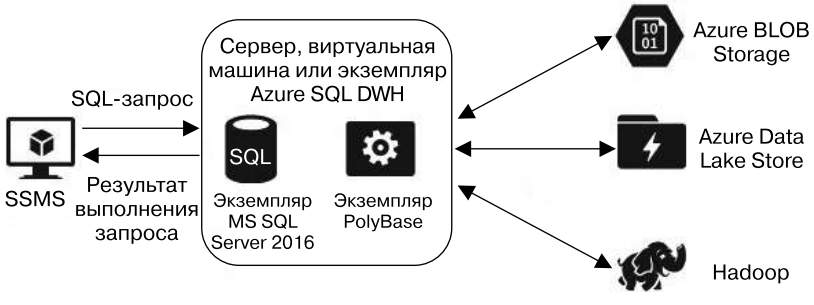


Рис. 12.19. Архитектура PolyBase

PolyBase — программа, устанавливаемая на сервер, на котором установлен MS SQL Server. Для AWS SWD этот компонент уже установлен. Далее пользователь, подключившись к этому MS SQL Server через SQL Management Studio или аналогичную программу — графический редактор запроса, может создать внешние таблицы с данными в Azure Blob Storage, Azure Data Lake Store или в другом источнике Hadoop. Доступность этих источников зависит от того, где расположена PolyBase: на локальном сервере с MS SQL, на виртуальной машине в облаке или на AWS SQL DWH. В любом случае эта технология позволяет строить общие SQL-запросы, комбинируя данные из физически разных источников.

Если файлы для внешних таблиц очень большие, то можно копировать (импортировать) данные в таблицу Azure SQL DWH, применив технологию CTAS T-SQL. Это технология создания таблицы в Azure SQL DWH с помощью оператора `SELECT`. CTAS позволяет выполнять параллельную загрузку данных из одной таблицы (например, внешней) в другую благодаря синтаксису, показанному на рис. 12.20. Здесь таблица `Payments` будет иметь такую же схему, как и схема `Payments_Stage`. Это в ряде случаев упрощает задачу создания таблицы, поскольку не требуется выполнять сначала сценарий DDL, а затем сложный `SELECT` — в данном случае оба этих шага совмещены.

```
CREATE TABLE [dbo].[Payments]
WITH
(
    DISTRIBUTION = ROUND_ROBIN,
    CLUSTERED COLUMNSTORE INDEX
)
AS
SELECT  *
FROM    [dbo].[Payments_Stage];
```

Рис. 12.20. Создание таблицы Payments из таблицы Payments_Stage с помощью технологии CTAS

Итак, мы рассмотрели основные возможности SQL-синтаксиса реализации T-SQL для Azure DWH SQL. Кроме рассмотренных, данный сервис предоставляет много других возможностей в плане как синтаксиса (например Materialized View), так и оптимизации производительности путем учета статистики выполнения запросов. Но, зная основы синтаксиса, описанного выше, вы легко самостоятельно освоите эти темы.

Синтаксис AWS Redshift, построенный на основе PostgreSQL, также включает в себя операторы DDL и DML. Ряд особенностей T-SQL не поддерживается (хранимые процедуры, внешние таблицы), в то же время есть аналогичные, например, поддерживаются определяемые пользователем функции и возможность прямого копирования данных из внешних источников (S3 и DynamoDB) с помощью синтаксиса COPY. Последняя возможность часто применяется не напрямую, а опосредованно, с использованием AWS Data Pipeline и AWS Glue.

Чтобы обеспечить возможность выполнять запросы к данным, хранящимся в файлах, расположенных в AWS S3, сервис реляционного хранилища AWS Redshift предоставляет встроенный сервис AWS RedShift Spectrum. Последний позволяет выполнять запросы с развитым параллелизмом на языке PostgreSQL без предварительного копирования данных в таблицы AWS Redshift. Более того, AWS Redshift Spectrum предоставляется в независимом кластере и не затрагивает основной кластер Redshift. Этот сервис концептуально близок к сервису Azure Data Lake Analytics в том плане, что может создавать таблицы AWS Redshift Spectrum, являющиеся хранилищем метаданных файлов, и зарегистрировать эти таблицы во внешних хранилищах метаданных: AWS Glue, AWS Athena, Apache Hive и др. Более того, можно создать внутри этого сервиса таблицы с внешними источниками данных и комбинированное хранилище данных, являющееся аналогом Azure DWH + PolyBase. При этом все таблицы с внешними источниками данных могут быть распределены между узлами кластера, что увеличит производительность запросов с этими таблицами. Далее, все таблицы AWS Redshift Spectrum можно использовать в совместных запросах вместе с таблицами AWS Redshift.

У технологии AWS Redshift Spectrum есть ряд ограничений, а именно: кластер AWS Redshift Spectrum и файлы в бакете AWS S3 должны находиться в одном регионе. Кроме того, внешние таблицы доступны только для чтения.

12.3. Azure Data Lake Analytics

Сервис нереляционного хранилища данных Azure Data Lake предоставляет встроенный сервис построения SQL-запросов к данным, расположенным в нем. Синтаксис запросов, реализованный в этом сервисе, называется U-SQL. Сами запросы выполняются как задания (jobs), и начисление платы в этом случае идет за транзакции чтения и записи, то есть за фактическое выполнение, а не хостинг. Подробнее рассмотрим этот язык.

U-SQL представляет собой комбинацию языков SQL и C#, оптимизированную для выполнения запросов к данным, хранящимся в Azure Data Lake Store. Такая комбинация позволяет работать со слабоструктурированными данными с отсутствием общей схемы или с гетерогенной схемой. Запросы U-SQL можно выполнять двумя способами: *локальное исполнение на вашем компьютере* (U-SQL Local Execution) и *выполнение в сервисе Azure Data Lake или исполнение в облаке* (U-SQL Cloud Execution). В практическом примере мы рассмотрим оба варианта, пока же я акцентирую внимание на синтаксисе и прежде всего перечислю его ключевые компоненты.

- *Переменная набора строк* (rowset variable) — переменная, хранящая результат выполнения запроса, содержащая строки. Каждая такая переменная начинается с символа `@`, за которым следует ее имя: `@variableName`. В качестве источников данных для переменной набора строк выступает оператор извлечения или другая переменная набора строк.
- *Оператор извлечения* (`EXTRACT`) предназначен для чтения данных из файлов и парсинга их схемы данных. По умолчанию позволяет читать текстовые данные с разделением табуляцией. Можно кастомизировать этот оператор, создав свой экстрактор.
- *Оператор вывода* (`OUTPUT`) отвечает за запись данных из переменной набора строк в файл. Создает файлы, в которых колонки в строках разделены запятыми. Этот оператор тоже можно кастомизировать.
- *Пути файлов* — оба оператора, извлечения и вывода, оперируют с файлами, расположенными в Azure Data Lake Store. Эти файлы расположены в иерархической структуре каталогов. Различие между путями файлов и локальными путями в том, что в качестве корневого каталога выступает аккаунт Azure Data Lake `adl://pokerexample17.azureData Lake store.net/`, относительно которого задается файловая структура.
- *Скалярные переменные* — переменные, содержащие скалярную величину определенного типа (`INT`, `VARCHAR`, `DATETIME`). Подчиняются тем же синтаксическим правилам, что и скалярные переменные T-SQL.
- *Операторы преобразования переменных набора строк, агрегации и фильтрации* — основные операторы выполнения запросов.

Теперь рассмотрим конкретный пример. Прежде всего изучим, как создавать файл, заполнив его с помощью запроса, выполняемого в облаке. Создадим экземпляр

Azure Data Lake Analytics, для чего в строке поиска Marketplace введем `Data Lake Analytics`. В результате появится форма, в которой следует нажать кнопку Create (Создать) (рис. 12.21).

После этого откроется форма конфигурирования сервиса (рис. 12.22).

В этой форме нужно указать имя аккаунта, которое будет префиксом адреса аккаунта (на рисунке это pokerexampleanalytics), выбрать существующую ресурсную группу (PokerRum), указать регион размещения сервиса (East US 2). После этого стоит добавить аккаунт Azure Data Lake Store. В данном случае добавлен существующий аккаунт pokerexample17. После завершения конфигурирования следует нажать кнопку Create (Создать).

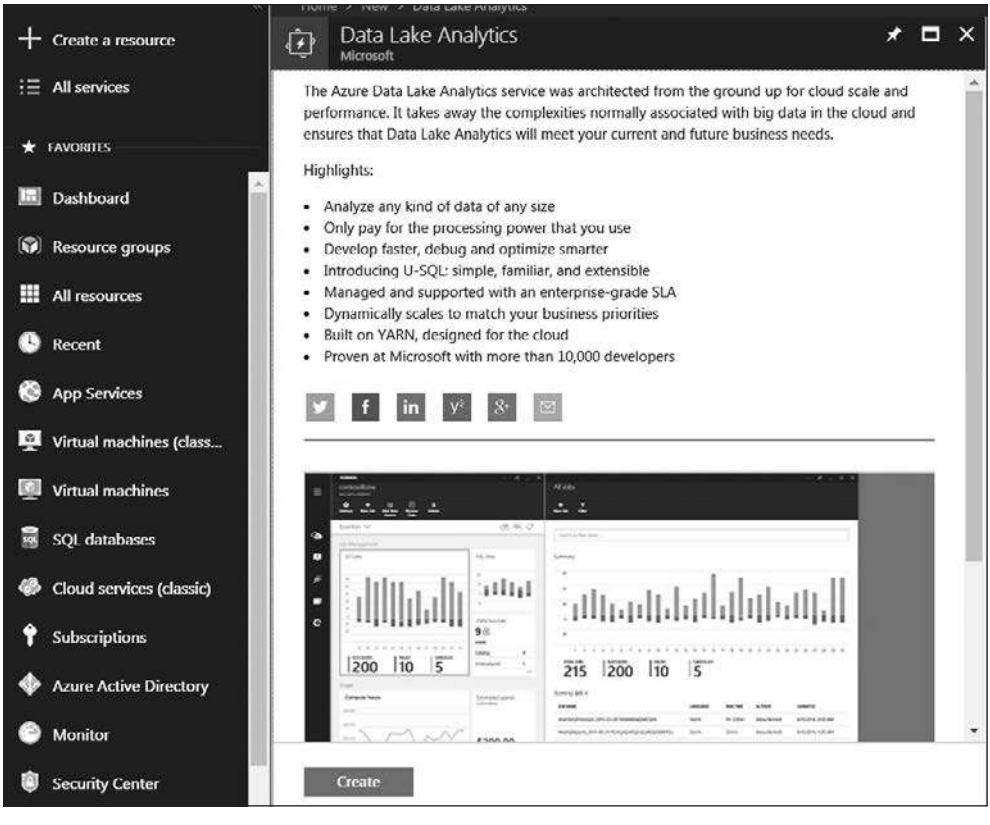


Рис. 12.21. Первая страница создания Data Lake Analytics

После успешного выполнения описанных действий можно открыть общую панель сервиса (рис. 12.23). Характерной особенностью этого сервиса является то, что доступ к нему контролируется с помощью Azure Active Directory и можно добавлять пользователей прямо через мастер добавления, вызываемый нажатием на ссылку Add user wizard, после чего настраиваются их права.

Теперь посмотрим, как использовать Azure Data Lake Analytics. Этот сервис работает с файлами в Azure Data Lake Store и Azure Blob Storage, которые должны быть туда помещены. Затем результат работы исполняемых заданий Azure Data Lake Analytics также сохраняется в новый файл в этом хранилище. Данный подход именуется ELT (Extract Load Transform — «извлечение, загрузка, преобразование»). Он принципиально отличается от традиционного подхода ETL, характерного для реляционного хранилища данных. Итак, результатами выполнения запросов являются новые файлы. С помощью встроенных операторов можно работать с форматами CSV и TSV. Последний представляет собой текстовый формат, при котором колонки разделяются символами табуляции, а строки — символами перевода строки.

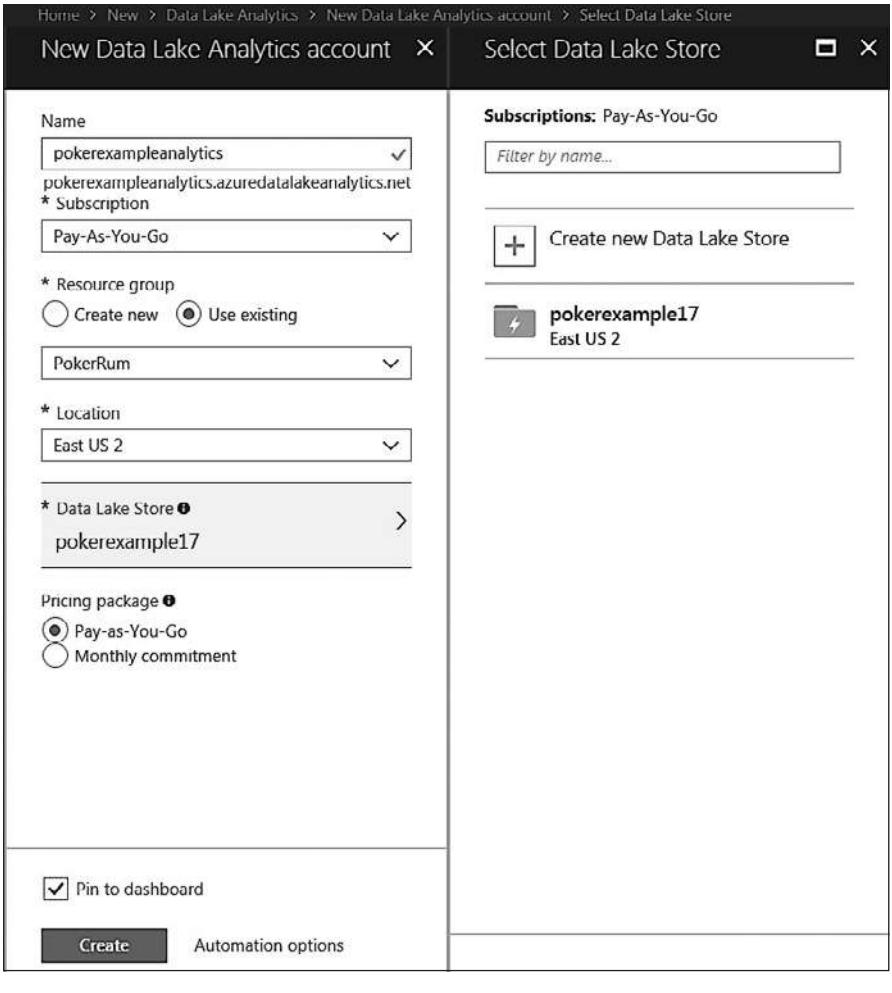


Рис. 12.22. Форма конфигурирования аккаунта Data Lake Analytics

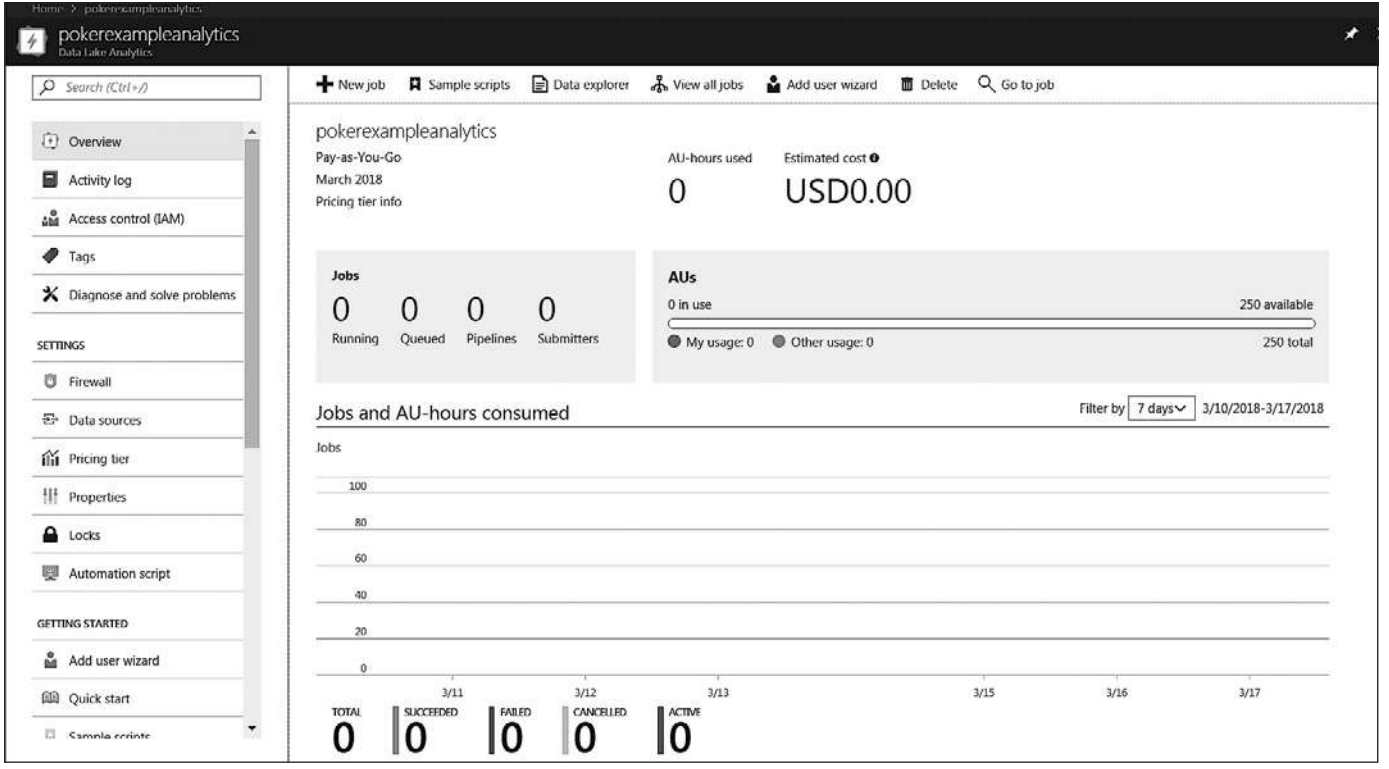


Рис. 12.23. Общая панель аккаунта Azure Data Lake Analytics

Создать файл проще всего с помощью выполняемого сценария. Чтобы создать выполняемое задание, следует нажать на ссылку + New job, после чего откроется форма онлайн-редактора запросов. Редактор со сценарием, создающим новый файл и заполняющий его информацией, имеет вид, показанный на рис. 12.24. В данном редакторе указывается имя выполняемого задания (Job name), определяется положение ползунка выбора уровня производительности AUs (analytics units) — обобщенных характеристик вычислительных ресурсов, выделяемых на выполнение сценария. Важный параметр выполнения — приблизительная стоимость задания, поскольку очень большие объемы данных занимают длительное время выполнения сценария и выдвигают высокие требования к ресурсам, что соответственно влечет затраты денежных средств. Сам сценарий можно сохранить на диск (ссылка Save as). Можно добавить файл с диска (Open file). Чтобы запустить выполняемое задание, следует нажать кнопку Submit (Отправить).

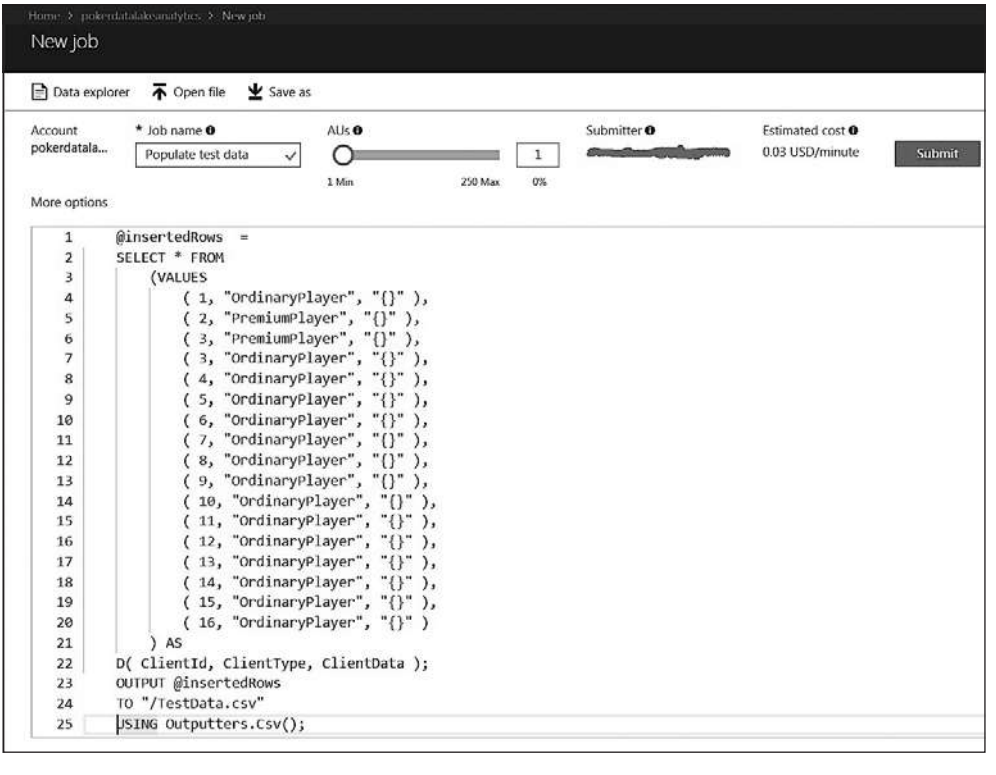


Рис. 12.24. Внешний вид редактора со сценарием создания и заполнения CSV-файла

Данный сценарий включает ряд компонентов. Переменная `@insertedRows` после выполнения сценария станет содержать набор строк, которые будут вставляться в файл. Синтаксис заполнения переменной набора строк включает в себя знакомые

нам операторы создания новых строк и вставки этих строк в переменную `@insertedRows`. Далее данные из этой переменной с помощью оператора `OUTPUT` помещаются в выходной канал. В этом случае выходным каналом является CSV-файл `"/TestData.csv"`, который создается с помощью оператора `Outputters.Csv()`.

Когда задание начнет выполняться, в портале откроется страница, на которой показаны этапы исполнения сценария в виде графа, а также приведены основные метрики: использование AU, приблизительная потраченная сумма, время исполнения, эффективность, приоритет и др. (рис. 12.25). Результат исполнения в виде данных, добавленных в файл, можно посмотреть на вкладке Data (рис. 12.26).

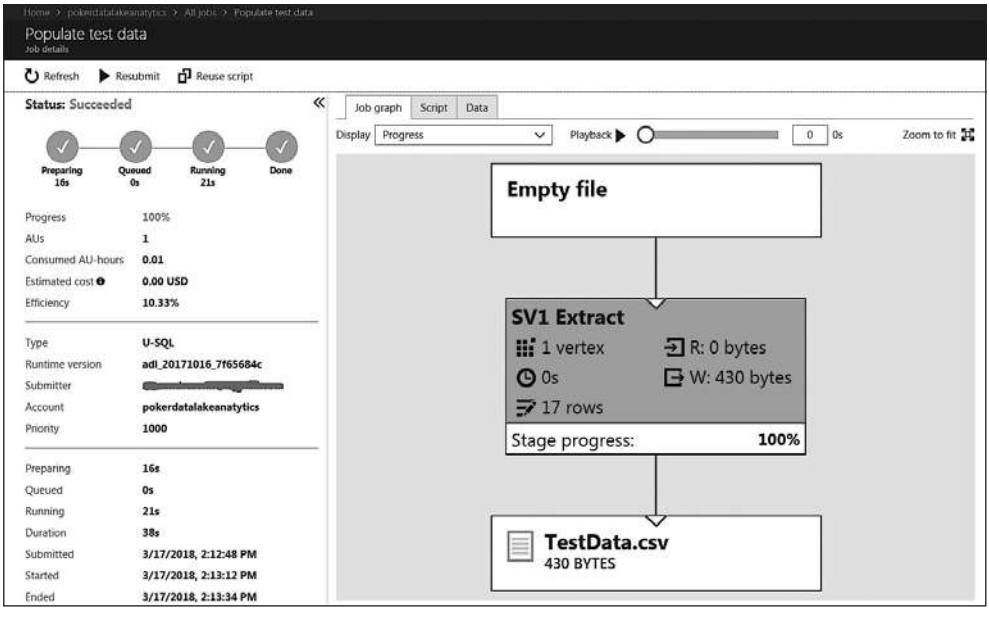


Рис. 12.25. Панель мониторинга выполняемого задания

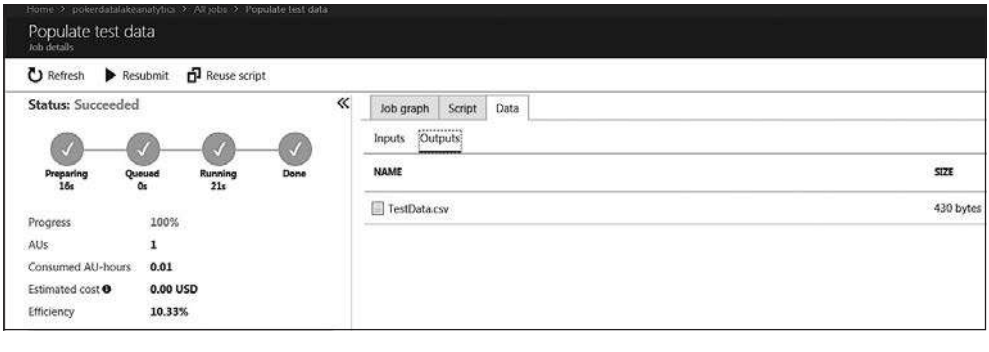
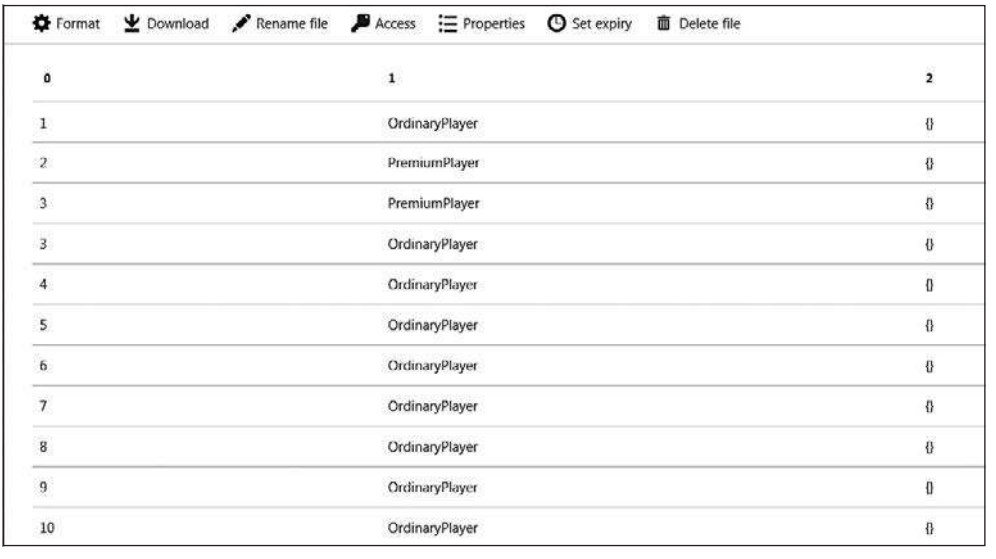


Рис. 12.26. Вкладка Data, где доступны входные и выходные источники

На рис. 12.27 представлен общий вид созданного файла.



0	1	2
1	OrdinaryPlayer	{}
2	PremiumPlayer	{}
3	PremiumPlayer	{}
3	OrdinaryPlayer	{}
4	OrdinaryPlayer	{}
5	OrdinaryPlayer	{}
6	OrdinaryPlayer	{}
7	OrdinaryPlayer	{}
8	OrdinaryPlayer	{}
9	OrdinaryPlayer	{}
10	OrdinaryPlayer	{}

Рис. 12.27. Обзор созданного файла

Однако описанный выше способ создания файла — скорее методический, чем прикладной. В реальных проектах основным способом наполнения Azure Data Lake Store является использование Azure Data Factory. Для дальнейшего объяснения примера я загрузил файл `Payments.tsv` с помощью Azure Data Factory. Источником была база данных Azure SQL, а файл-приемник создавался с разделением колонок символами табуляции.

Простейший сценарий, извлекающий данные из файла `Payments.tsv` и помещающий их без преобразования в файл `Payments_copy.tsv`, представлен на рис. 12.28.

В этом сценарии объявлены две переменные, содержащие пути файлов: `@InputFile` (путь ко входному файлу) и `@OutputFile` (путь к выходному). Обратите внимание, что выходной файл может остутствовать на момент выполнения сценария, поскольку он вместе с вмещающей его папкой будет создан при успешном выполнении сценария. Тип данных — `string`, соответствующий типу `System.string` платформы .Net. Граф выполнения файла имеет вид, показанный на рис. 12.29.

Для выполнения полезной трансформации данных необходимо использовать операторы преобразования переменных наборов строк. К таковым может относиться фильтрация строк, агрегирование, выборка определенных колонок и др. Но в любом случае выходным источником будет новый файл.

Начнем рассмотрение операторов трансформации с фильтрации и выбора заданного набора колонок. Пример подобного сценария показан на рис. 12.30.

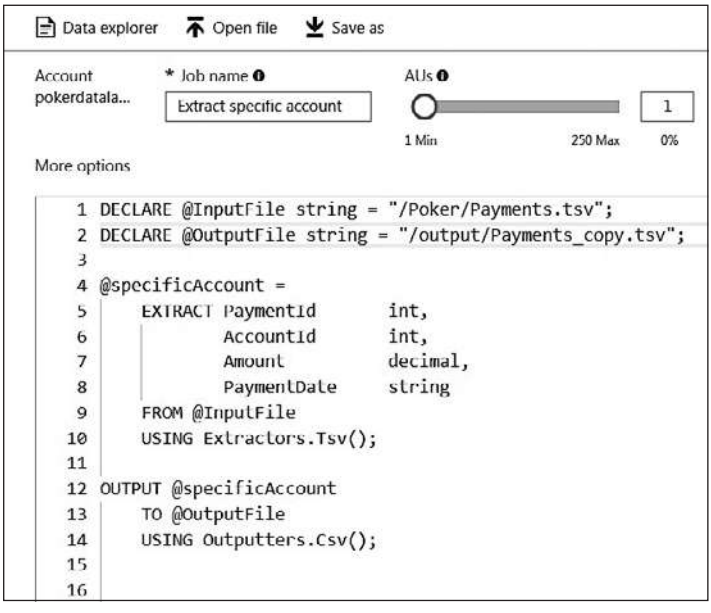


Рис. 12.28. Сценарий прямого копирования информации из файла Payments.tsv в Payments_copy.tsv

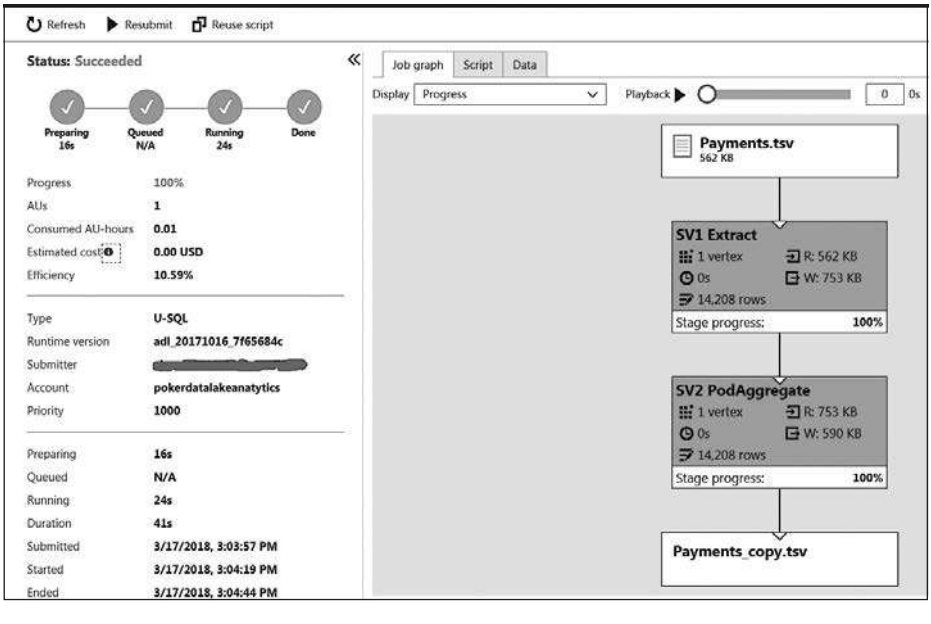


Рис. 12.29. Граф выполнения сценария копирования данных из файла-источника в файл-приемник

```
DECLARE @InputFile string = "/Poker/Payments.tsv";
DECLARE @OutputFile string = "/output/Payments_filtered.tsv";

@specificAccount =
    EXTRACT PaymentId       int,
            AccountId       int,
            Amount          decimal,
            PaymentDate     string
    FROM @InputFile
    USING Extractors.Tsv();

@result =
    SELECT AccountId, PaymentId, Amount
    FROM @specificAccount
WHERE AccountId == 1;

OUTPUT @result
    TO @OutputFile
    USING Outputters.Csv();
```

Рис. 12.30. Сценарий фильтрации данных

Помимо известных нам операторов, мы видим операторы, выбирающие из входного набора строк только те, которые соответствуют `AccountId`, равному `1`. Обратите внимание на синтаксис: для сравнения используется оператор `==`.

Результат выполнения сценария трансформации переменной набора строк помещается в новую переменную `@result`, которая, в свою очередь, с помощью оператора `OUTPUT` сохраняется в выходном файле

Тут следует дать пояснение относительно имен полей данных. Дело в том, что данные сами по себе хранятся в файлах CSV или TSV без указания имен полей. Эти имена необходимы для использования переменных набора строк и создаются при работе извлекателя (Extractor). Далее в сценарии к ним можно получить доступ, обращаясь именно к переменной, а не к файлу. Но в выходном файле строки будут сохраняться опять без добавления имен полей.

Теперь рассмотрим применение операторов группировки на примере сценария, суммирующего платежи для каждого аккаунта и выбирающего среди них те, для которых было выполнено не менее 1000 транзакций. Сценарий группировки показан на рис. 12.31.

Возможности сервиса Azure Storage Analytics не ограничиваются сценарием «файл-источник — сценарий преобразования набора строк — файл-приемник», но предоставляет средство группировки всех объектов в нереляционное хранилище со структурой, концептуально повторяющей DWH. Это средство называется

```
DECLARE @InputFile string = "/Poker/Payments.tsv";
DECLARE @OutputFile string = "/output/Payments_aggregated.tsv";
DECLARE @TransactionCount int = 1000;

@specificAccount =
    EXTRACT PaymentId       int,
            AccountId       int,
            Amount          decimal,
            PaymentDate     string
    FROM @InputFile
    USING Extractors.Tsv();

@result =
    SELECT AccountId,
    SUM(Amount) AS SumAmount
    FROM @specificAccount
GROUP BY AccountId
HAVING COUNT(Amount) > @TransactionCount;

OUTPUT @result
    TO @OutputFile
    USING Outputters.Csv();
```

Рис. 12.31. Сценарий агрегирования значений, извлеченных из входного файла

каталогом U-SQL (U-SQL Catalog). Каждый аккаунт Azure Data Lake Analytics содержит ровно один каталог. Этот каталог состоит из одной или нескольких БД, вмещающих данные, организованные в таблицы, и код в виде пользовательских функций, объектов учетных данных и пр. Структура каталога показана на рис. 12.32.

База данных — логическая группа пользовательских объектов. Каталог содержит как минимум одну базу — `master`. Пользовательские БД с именем `PokerDatabase` создаются с помощью команды `CREATE DATABASE IF NOT EXISTS PokerDatabase`, а удаляются командой `DROP DATABASE IF EXISTS PokerDatabase`. Таблицы представляют собой способы организации данных. Их можно создать пустыми и заполнить позже или создать при выполнении запроса `SELECT`.

Прежде чем создавать таблицу, посмотрим, как оборачивать сценарий извлечения строк из файла в *пользовательскую табличную функцию* (table-valued function, TVF). Расположим ее в базе данных `master`. Сценарий создания такой функции выглядит следующим образом (рис. 12.33).

Теперь создадим базу данных `PokerDatabase`, таблицу `Payments`, расположенную в этой базе, причем посмотрим, как создавать ее во время выполнения оператора

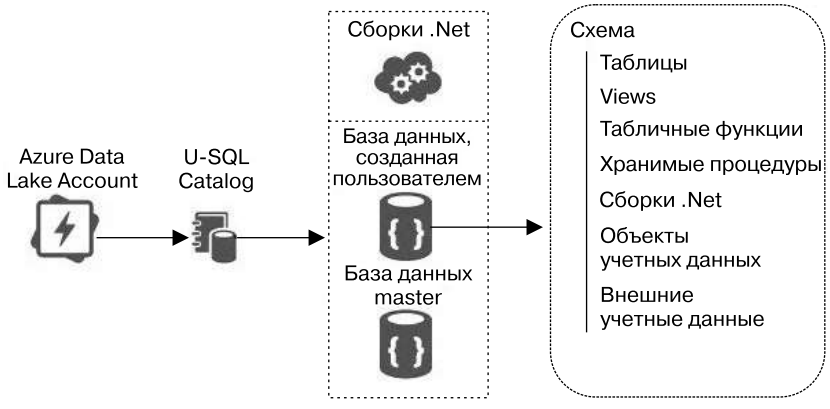


Рис. 12.32. Структура каталога U-SQL

```
1  DROP FUNCTION IF EXISTS ExtractInformation;
2
3  CREATE FUNCTION ExtractInformation()
4  RETURNS @result TABLE
5  (
6              PaymentId       int,
7              AccountId       int,
8              Amount          decimal,
9              PaymentDate     string
10 )
11 AS BEGIN
12 @result =
13     EXTRACT PaymentId       int,
14             AccountId       int,
15             Amount          decimal,
16             PaymentDate     string
17     FROM "/Poker/Payments.tsv"
18 USING Extractors.Tsv();
19 RETURN;
20 END;
```

Рис. 12.33. Пример создания пользовательской функции, извлекающей информацию из файла Payments.tsv

SELECT, вызываемого в пользовательской функции ExtractInformation. Второй вариант — создание таблицы DDL и заполнение ее с помощью созданной нами функции. Подобный сценарий имеет вид, представленный на рис. 12.34.

```
1 DROP DATABASE IF EXISTS PokerDatabase ;
2 CREATE DATABASE PokerDatabase ;
3 USE DATABASE PokerDatabase ;
4
5 DROP TABLE IF EXISTS Payments;
6
7 CREATE TABLE Payments(
8     INDEX sl_idx CLUSTERED (PaymentId ASC)
9             DISTRIBUTED BY HASH (PaymentId)
10 ) AS SELECT * FROM master.dbo.ExtractInformation() AS S;
11
12
```

Рис. 12.34. Создание базы данных PokerDatabase и таблицы Payments в ней

В этом сценарии удаляется предыдущая версия базы данных `PokerDatabase`, если такая существовала, далее создается новая с помощью операторов `CREATE DATABASE`. Затем оператор `USR DATABASE PokerDatabase` переключает контекст на применение базы данных `PokerDatabase`. Такое переключение позволяет использовать все объекты этой БД без указания полностью квалифицированного имени объекта, состоящего из имени базы и схемы. Далее следуют операторы `DDL`, удаляющие предыдущую таблицу `Payments` и создающую ее опять с применением оператора `SELECT`. Обратите внимание: таблица создается с кластерным индексом по имени `sl_idx`, а распределение строк по кластеру обеспечивается с помощью хеш-функции. Нет необходимости явно указывать названия и типы полей, поскольку это происходит автоматически при выполнении оператора `SELECT`. Саму пользовательскую функцию `ExtractInformation()` нужно указать с полностью определенным именем: базой данных по умолчанию `master` и схемой по умолчанию `dbo`. Граф выполнения этого запроса имеет вид, показанный на рис. 12.35. После создания таблица доступна для выполнения прямых запросов точно таким же образом, как и в случае обычных таблиц в РБД. Но главное отличие от реляционных баз данных и DWH состоит в том, что результат выполнения запроса помещается в переменную набора строк, которая, в свою очередь, сохраняется в файл.

Созданные таблицы позволяют строить практически такие же сложные запросы, как и в случае запросов в Azure SQL DWH, поскольку U-SQL поддерживает многие возможности подмножества DML T-SQL, а именно, соединение (`JOIN`), объединение (`UNION`), оконные функции (`WINDOW FUNCTIONS`), View и пр. С другой стороны, таблицы дают возможность создавать пользовательские объекты и кастомизировать встроенные с помощью .Net библиотеки, а также задействовать преобразования набора строк благодаря расширениям для языков Python и R.

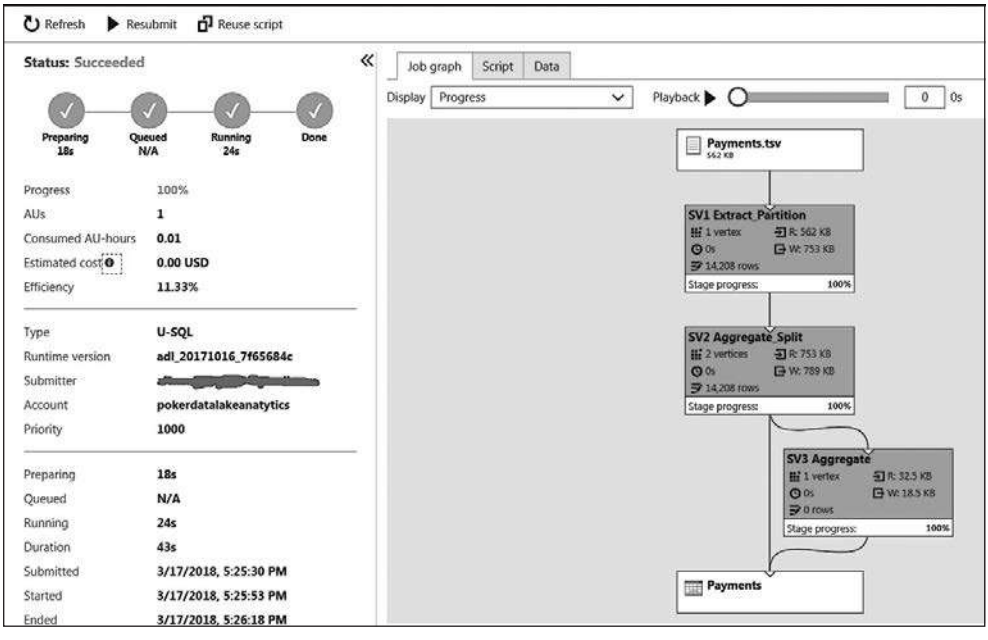


Рис. 12.35. Граф выполнения сценария создания таблицы через выполнение табличной функции

Особняком стоят когнитивные расширения языка U-SQL (cognitive extensions), обеспечивающие обработку файлов изображений в целях распознавания приведенных на них объектов: текстовых и цифровых символов, лиц, эмоций, других объектов. Кроме того, возможен интеллектуальный анализ текста в файле в виде извлечения ключевых фраз и выполнения анализа общей тональности.

С помощью SDK для .Net и Visual Studio созданную базу данных можно экспортировать из другого аккаунта Azure Data Lake Analytics и импортировать в подобный аккаунт. Как видим, связка Azure Data Lake Store/Analytics обеспечивает построение нереляционного хранилища, позволяющего выполнять как интерактивный, так и пакетный анализ данных. Существует несколько возможных путей автоматизации запуска выполняемых заданий Azure Data Lake Analytics. Первый вариант: использовать библиотеку Azure Data Lake Analytics — ADAL для .Net, которая позволяет создавать и управлять заданиями. Второй вариант: использовать сервис Azure Data Factory, который можно сконфигурировать на периодический запуск сценариев U-SQL. Использовать синтаксис, близкий к SQL, очень удобно для разработки, отладки и обновления, что ускоряет процесс освоения этой системы. Как уже отмечалось, существенным отличием концепции Data Lake от DWH является то, что она поддерживает поток ELT. С одной стороны, это упрощает процедуру копирования, а с другой — несколько усложняет отладку, поскольку

результат будет помещаться в файл. Таким образом, для визуализации информации, которая была извлечена с помощью сервиса Azure Data Lake Analytics, требуется сторонний специализированный сервис BI; Azure Data Lake Analytics в этом случае представляет собой движок преобразования файлов.

Как уже было сказано выше, помимо Azure Data Lake Store, в качестве хранилища данных можно применить Azure BLOB Storage, что позволяет максимально гибко использовать сервис Azure Data Lake Analytics. Однако наибольшая производительность, гибкость и возможность интеграции с сервисами HDInsight для обеспечения наивысшей производительности достигается в случае применения Azure Data Lake Store.

12.4. AWS Athena

AWS Athena представляет собой бессерверный сервис, позволяющий выполнять запросы на языке SQL для анализа данных файлов, хранящихся в AWS S3. Масштабирование сервиса при этом происходит автоматически. С помощью AWS Athena можно анализировать структурированные, частично структурированные и неструктурированные файлы, хранящиеся в AWS S3, включая CSV, JSON, Apache Parquet или Apache ORC. Сервис может быть интегрирован с AWS Glue Data Catalog. Для визуализации результатов запросов можно применить сервисы BI AWS QuickSight и сервисы, подключаемые благодаря протоколам JDBC и ODBC.

AWS Athena вмещает логические компоненты, содержащие метаданные о файлах с данными: базы и таблицы. *Таблицы* включают метаданные конкретного файла: адрес его размещения, формат хранения данных, имена колонок данных и пр. *Базы* представляют собой логическую группировку таблиц и содержат метаданные о схемах таблиц. При этом все таблицы регистрируются в общем внутреннем каталоге данных AWS Athena, который можно интегрировать с AWS Glue Data Catalog. Для того чтобы можно было выполнять запросы к файлам, находящимся в S3, каждый из файлов должен быть представлен в виде таблицы в каталоге данных, имеющемся в составе в AWS Athena.

Итак, первичная единица хранения метаданных — таблица AWS Athena. Существуют следующие способы создания таблиц.

- *Автоматическое создание* — для этого можно использовать сборщик AWS Glue напрямую из Athena. Такая опция доступна благодаря взаимной интеграции AWS Athena и AWS Glue Data Catalog. Таким образом, таблица сначала создается в AWS Glue, а затем регистрируется во внутреннем каталоге Athena. Существенное ограничение этого способа заключается в том, что регион размещения AWS Glue Data Catalog должен совпадать с регионом AWS Athena.
- *Ручное создание в мастере создания таблиц*, предоставляемом веб-консолью AWS Athena.

- *Ручное создание в онлайн-редакторе запросов*, для чего следует использовать операторы DDL языка Apache HiveQL, являющегося диалектом языка SQL.
- *Создание из кода или из сценария с помощью API или CLI*, которые удаленно запускают сценарии DDL.
- *Создание из внешних редакторов запросов*, подключаемых через JDBC.

AWS Athena отличается тем, что в качестве языка запросов используется технология Apache Presto. Изначально разработанная в Facebook, эта технология позволяет применять SQL-подобный синтаксис, выполняемый в кластере, к которому могут быть подключены многочисленные источники данных, в том числе HDFS, AWS S3, MySQL, PostgreSQL, Apache Kafka, Apache Cassandra и др. Кластеризация обеспечивает массивно-параллельное исполнение и, как следствие, масштабируемость и высокую производительность, а возможность подключения многих внешних источников обеспечивает потенциальность централизованного анализа данных без применения операций ETL/ELT. Целиком Apache Presto представлена как одно из приложений сервиса AWS EMR. AWS Athena предоставляет часть этой системы, а именно выполнение запросов на языке SQL к файлам AWS S3.

Следующая особенность AWS Athena — использование операторов языка HiveQL DDL для создания таблиц и баз данных. Apache Hive — система управления БД, построенная на основе Hadoop. Эта система позволяет выполнять запросы к данным в ней на языке HiveQL, практически полностью реализующем стандарте SQL-92, и поддерживает подключение по протоколу JDBC. Сама по себе система Hive представляет собой нереляционное хранилище данных, которое построенно поверх Hadoop и может использовать Apache Tez, Apache Spark или Map Reduce для выполнения запросов. Напрямую эта система не задействована в основе AWS Athena, поскольку философия данного сервиса состоит в предоставлении не хранилища данных, но автоматически масштабируемой системы запросов, которая не потребует применения ETL/ELT-операций. В итоге в AWS Athena применена комбинация Apache HiveQL DDL + Apache Presto DML.

Рассмотрим, как конкретно работать с этим сервисом, используя онлайн-редактор запросов. Стартовая страница сервиса имеет вид, показанный на рис. 12.36.

После нажатия кнопки Get Started (Приступить к работе) вы пройдете через интерактивный обучающий процесс, который покажет, как создавать, подключать и строить запросы к источникам данных. Мы рассмотрим этот процесс подробнее, но пока предположим, что вы его пропустили и были переброшены напрямую на базовую страницу сервиса AWS Athena, показанную на рис. 12.37. Если вы не удалили сервисы AWS Glue, с которыми мы экспериментировали в последней главе предыдущей части, и не поменяли регион, то обратите внимание вот на что: на новой странице доступны те же базы данных и те же таблицы, что были созданы сборщиками в AWS Glue. Именно это и видно на рис. 12.38.

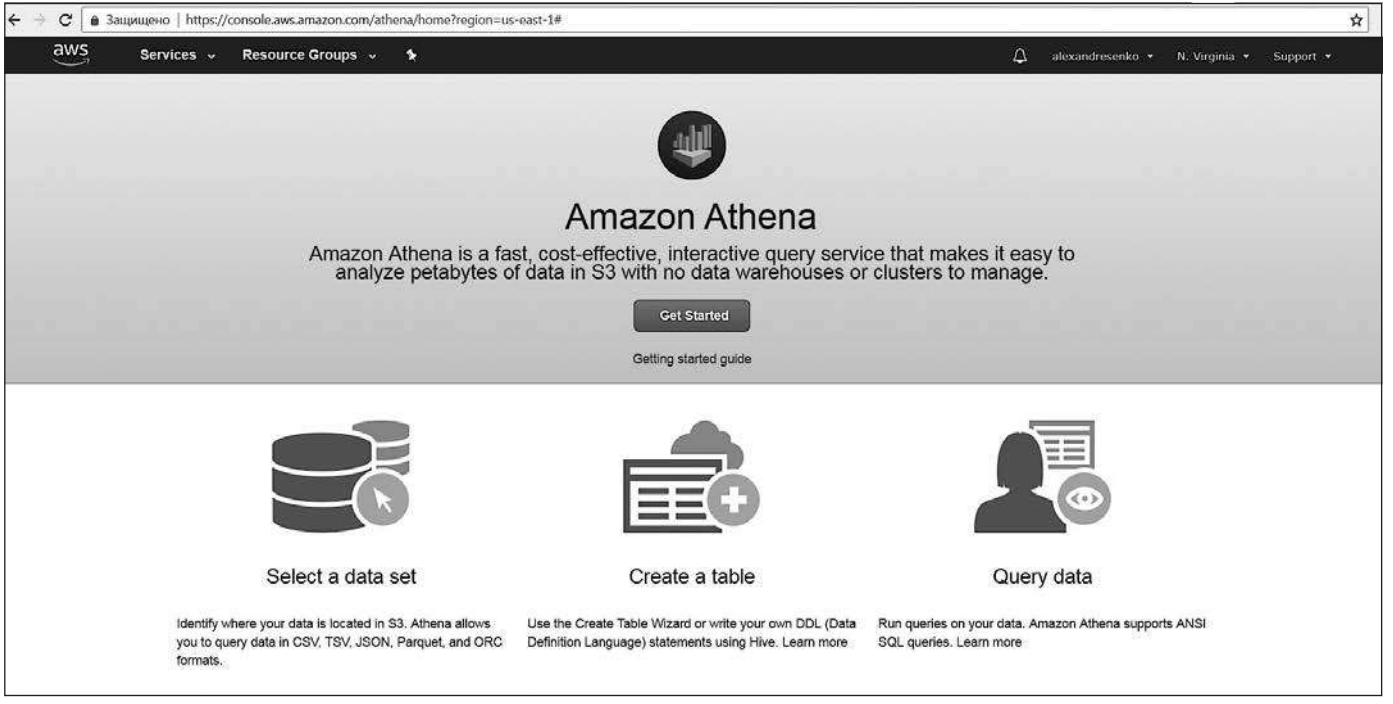


Рис. 12.36. Внешний вид стартовой страницы сервиса

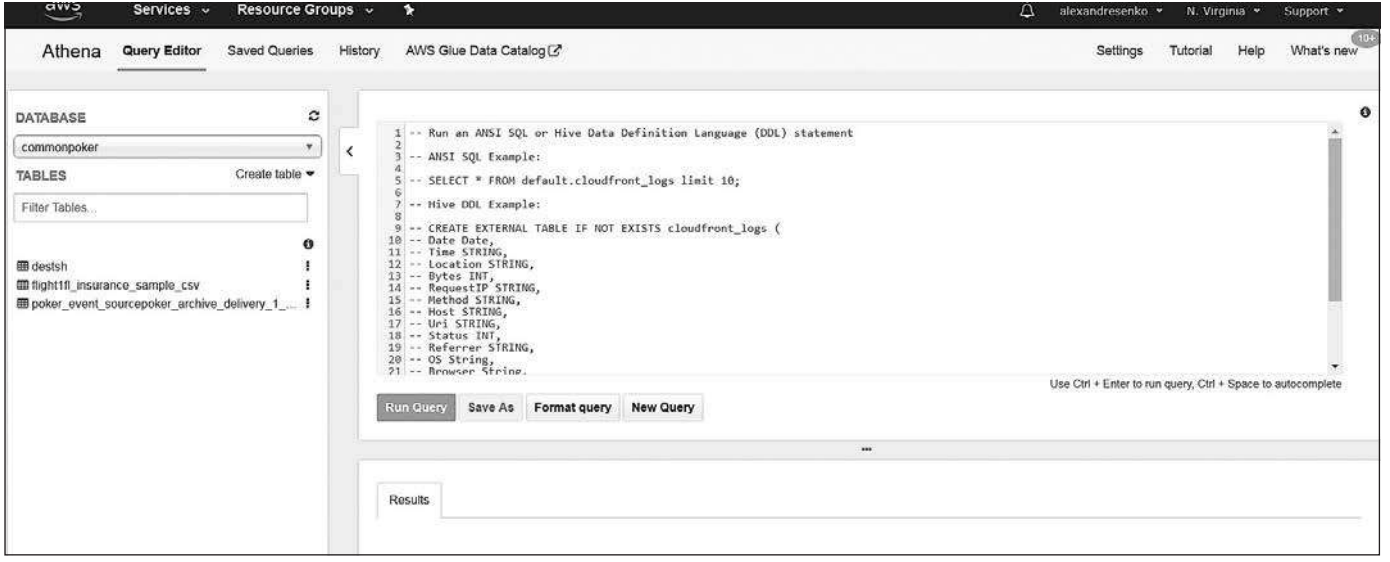


Рис. 12.37. Базовая страница сервиса AWS Athena

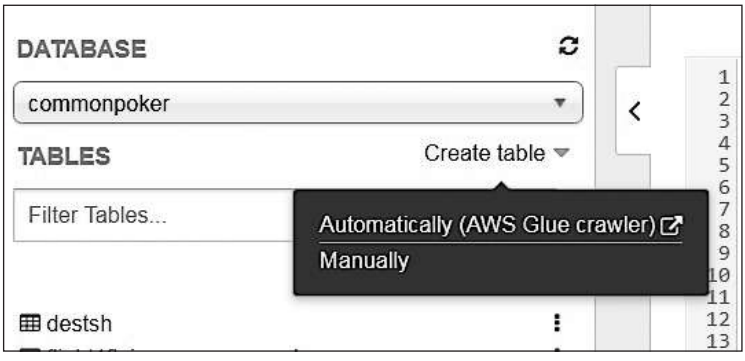


Рис. 12.38. Выбор способа создания таблицы

В данном случае выбрана база данных commonpoker и показан список таблиц. У вас список и база могут отсутствовать или называться иначе. Зададимся следующей задачей — проанализируем с помощью Athena CSV-файлы, полученные в предыдущем разделе при выполнении запросов Azure Data Lake Analytics. Эти файлы должны быть загружены в бакет AWS S3. Как только мы выполнили данное действие, нажимаем ссылку Create table, расположенную на левой вкладке страницы.

Можно использовать сборщик AWS Glue, но мы пойдем путем ручного создания, для чего следует нажать на ссылку Manually. Далее откроется вкладка конфигурирования базы данных и таблицы (рис. 12.39). Здесь нужно выбрать существующую базу или создать новую. Мы выберем второе и присвоим базе имя PokerFinance; таблице дадим имя `payments`. Далее следует указать местоположение папки в S3-бакете, в которой будут располагаться интересующие нас файлы. Обратите внимание: нужно указывать путь с начальным префиксом `s3://` и путь именно к папке, а не к конкретному файлу! Мы не конфигурировали шифрование и потому не устанавливаем флажок Encrypted data set.

Далее следует выбрать формат файла, в котором будут построены таблицы. Список доступных форматов приведен на рис. 12.40. Поскольку у нас файл типа CSV, то выберем этот формат.

Далее нужно указать имена и типы колонок с данными. Добавлять колонки можно двумя способами: последовательно выбирая их на базовой вкладке (Add a column) или группой (Bulk add columns) (рис. 12.41, 12.42).

Типы и имена колонок выбираются такие, которые соответствуют типам и предполагаемым именам колонок исходного источника данных. Поскольку мы используем файл с данными платежей `Payments_copy.csv`, полученными в предыдущем разделе, то укажем колонки и типы, соответствующие исходному источнику данных.

На следующем шаге можно настроить секционирование (разделение файлов) (рис. 12.43). Это имеет смысл сделать для файлов очень больших размеров, чтобы повысить производительность выполнения запросов. Поскольку у нас небольшой файл, то оставим этот пункт без изменений.

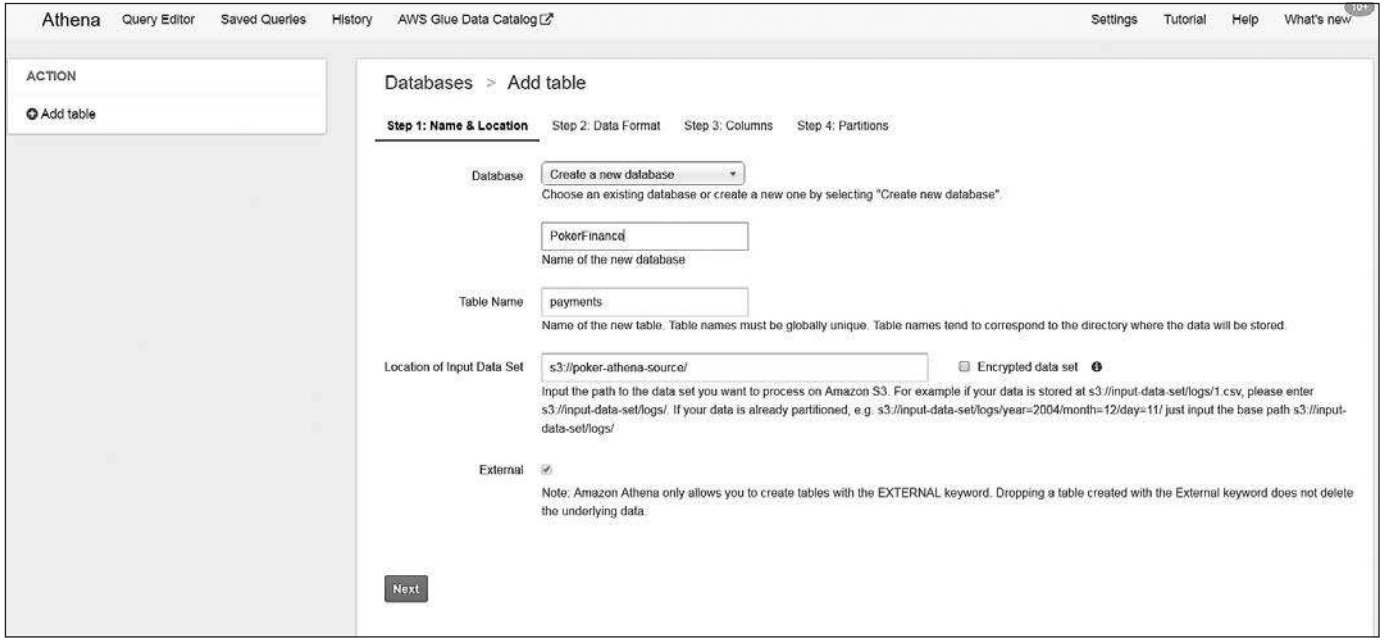


Рис. 12.39. Вкладка конфигурирования базы данных и таблицы

Рис. 12.40. Выбор формата файла S3

Рис. 12.41. Страница конфигурирования колонок таблицы метаданных

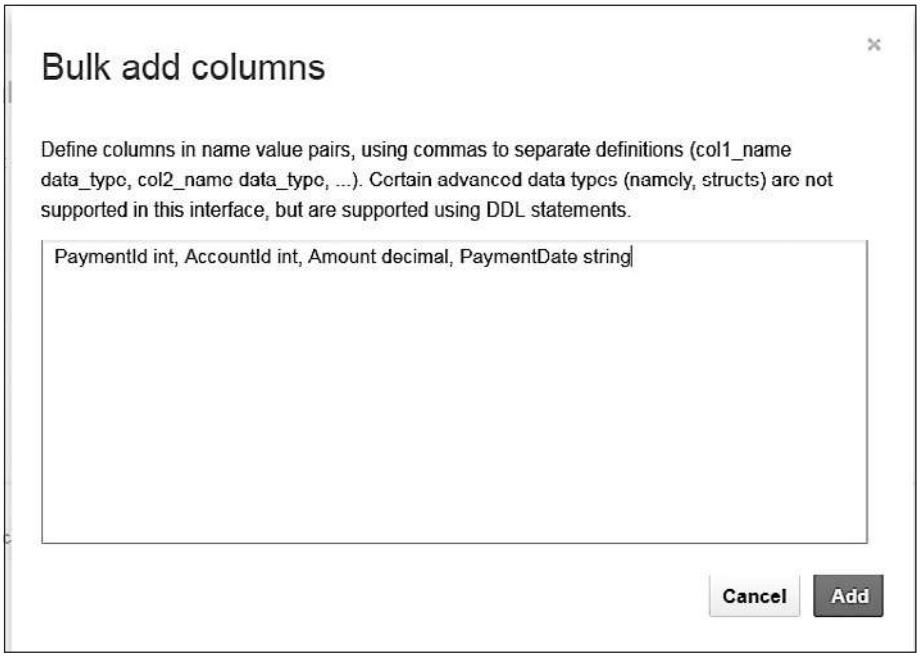


Рис. 12.42. Групповое добавление колонок

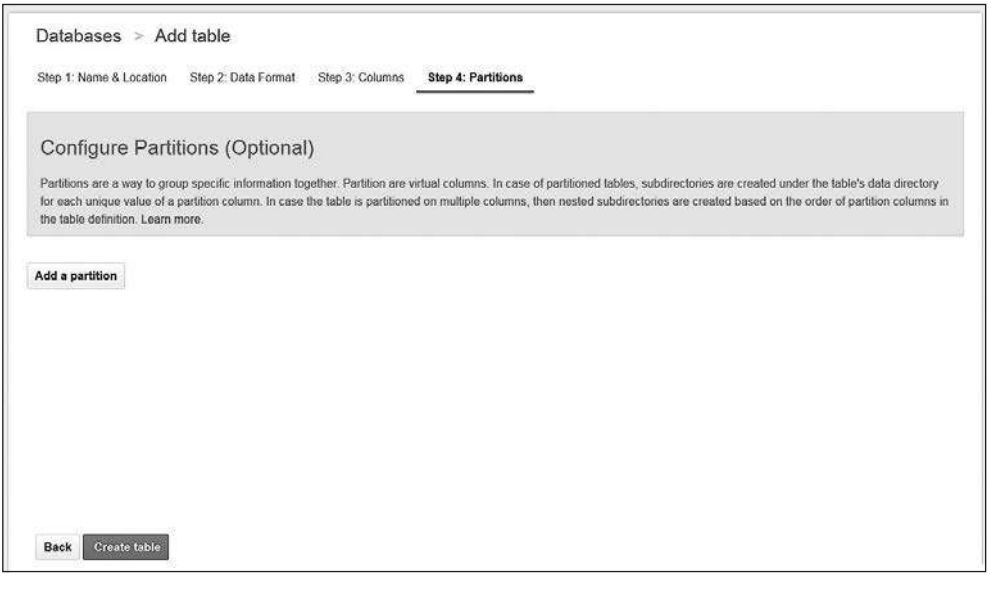


Рис. 12.43. Страница настройки секционирования

После этого будет сконфигурирован и выполнен сценарий HiveQL, создающий таблицу (рис. 12.44). Синтаксис создания таблиц в целом известен. В данном случае сценарий создает в базе данных PokerFinance внешнюю таблицу с именем `payments`. Указываются формат сериализации и символ — разделитель столбцов, местоположение файла, а также опции шифрования. Пользователь может сам конфигурировать формат ожидаемого входного файла: символы разделителей колонок, сериализации и др. Теперь эта таблица доступна для выполнения запросов и интеграции с сервисом AWS Glue.

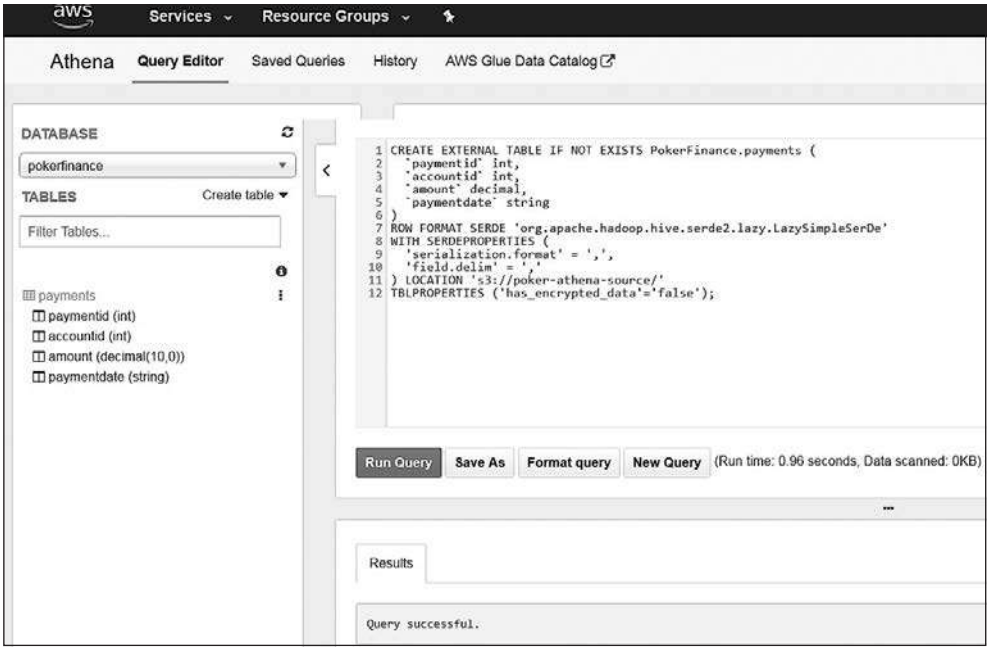


Рис. 12.44. Созданный и выполненный сценарий DDL HiveQL

Ну а дальше можно применять эти таблицы в запросах. Один пример показан на рис. 12.45, содержащий основные операторы выбора данных. Результат выполнения представлен в табличном виде, но его можно экспортировать в формат CSV, для чего следует щелкнуть на значке экспорта на вкладке результатов.

Помимо основных операторов выполнения запросов, включая объединение (`UNION`), соединение (`JOIN`), синтаксис AWS Athena позволяет работать со сложными (составными) типами: массивами, JSON, XML и географическими данными (широта, долгота, линии, полигоны). Кроме того, имеются готовые встроенные шаблоны для разбора логов сервисов CloudTrail, CloudFront, Load Balancer logs и VPC Flow Logs.

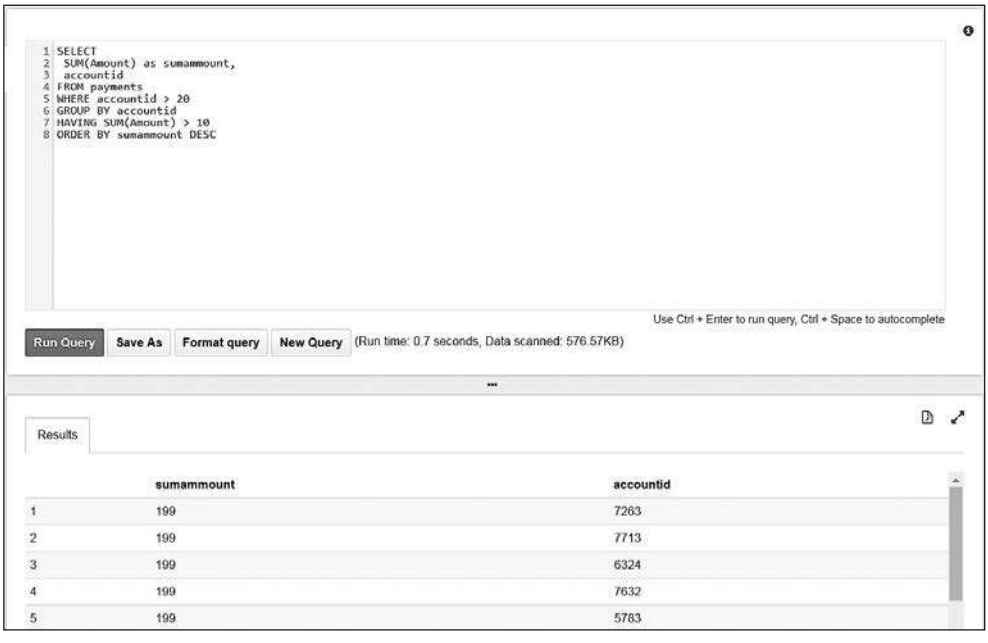


Рис. 12.45. Пример выполнения запроса на языке SQL к данным в таблицах

12.5. Apache Spark

В предыдущих главах рассматривались возможности анализа данных, предоставляемые сервисами реляционных и нереляционных хранилищ. Можно было обратить внимание, что основным языком построения запросов является тот или иной диалект SQL. Ключевое требование к системам интерактивного анализа больших данных — высокая отзывчивость, то есть, с одной стороны, высокая производительность и масштабируемость, а с другой — возможность написания запросов на интерпретируемых языках без необходимости предварительной компиляции, развертывания в кластер и др. В основе очень многих облачных сервисов, предоставляющих интерактивный анализ, лежит Apache Spark. Кроме того, этот фреймворк широко используется для построения стадии Transform в конвейерах ETL и ELT. Мы уже встречались с этим сервисом, рассматривая AWS Glue ETL и AWS Athena. Кроме того, сервисы копирования и трансформации данных AWS Data Pipeline и Azure Data Factory автоматически создают управляемые кластеры Spark. Этот сервис настолько популярен, что оба облачных провайдера предоставляют его в виде доступного для установки сервиса на управляемых кластерах Azure HDInsight и AWS EMR. С управляемым кластером HDInsight вы уже познакомились в главе 10,

а в этой главе я для разнообразия расскажу о AWS EMR — аналоге Azure HDInsight и об Apache Spark.

Apache Spark представляет собой программный продукт с открытым исходным кодом из экосистемы Hadoop, предназначенный для распределенной обработки структурированных, слабо структурированных и неструктурированных данных. Ключевым отличием его от других подобных фреймворков из экосистемы Hadoop является то, что обработка и хранение информации происходит в оперативной памяти без применения дисковых операций и, в частности, без применения паттерна MapReduce (описан более подробно в главе 14). Apache Spark написан на языках программирования Java и Scala, а потому предоставляет развитые интерфейсы для программ, написанных на этих языках. Дополнительно включает программные интерфейсы для Python и R.

Поскольку обрабатываемые наборы данных расположены в оперативной памяти, то алгоритмы машинного обучения могут обращаться к ним многократно. Если объем данных превышает объем оперативной памяти, то для их размещения используется дисковая память.

Apache Spark может работать поверх кластера Hadoop под управлением менеджеров кластера YARN или Mesos, но это не обязательно. Альтернативой является режим Standalone, который требует наличия кластера с установленной Java в каждом узле. Кроме того, Apache Spark можно установить в управляемые кластеры AWS EMR и Azure HDInsight. И наконец, доступна ручная установка на виртуальные машины Azure VM и AWS EC2. Кроме того, Spark может быть установлен и настроен на локальном компьютере.

Что касается масштабируемости, то, согласно документации Spark, наибольший размер среди существующих кластеров включает в себя 8000 вычислительных узлов.

Apache Spark включает ряд компонентов (рис. 12.46):

- *Spark SQL* — компонент Spark, позволяющий выполнять запросы к данным на языке SQL;
- *Spark Streaming* — компонент Spark, позволяющий обрабатывать потоковые данные;
- *Spark MLib* — встроенный набор библиотек машинного обучения;
- *GraphX* — компонент Spark, позволяющий обрабатывать данные, организованные в граф.

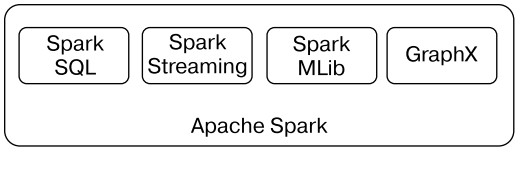


Рис. 12.46. Структура Apache Spark

Основное внимание мы уделим общим понятиям и компонентам Spark, описание GraphX и Spark MLib приведено не будет.

Теперь познакомимся с AWS EMR и посмотрим на примере его использования, как работать с базовыми программными интерфейсами Spark, а также с его компонентами.

AWS EMR представляет собой сервис управляемого кластера, предназначенный для работы фреймворков, служащих для обработки больших данных, таких как Apache Hadoop, Spark, Hive, Pig и др. AWS EMR используется совместно с сервисами копирования и трансформации данных AWS Glue ETL и AWS Data Pipeline.

Итак, в основе AWS EMR лежит *кластер*, который представляет собой коллекцию *узлов* — EC2-экземпляров виртуальных машин. Каждый узел есть обычный экземпляр EC2, который играет IAM-роль, требуемую для работы в составе кластера. Кроме того, в каждом узле установлен софт того или иного фреймворка. Каждый узел принадлежит одному из трех типов.

- *Головной узел* (master node) — узел, отвечающий за управление и координацию работы кластера и распределение выполняемых заданий среди узлов. Отслеживает выполнение задания, его статус и состояние узла.
- *Основной узел* (core node) — ведомый узел с программными компонентами, которые выполняют задания и сохраняют данные в распределенную файловую систему (HDFS) в кластере.
- *Узел-исполнитель* (task node) — ведомый узел с программными компонентами, которые только выполняют задания. Этот тип узла может отсутствовать.

EMR может работать в следующих режимах:

- временный кластер, который запускается для выполнения конкретного задания трансформации (например, MapReduce) или копирования (Apache Sqoop) и удаляется после окончания задания;
- постоянный кластер, предоставляющий консольный доступ (по SSH), позволяющий использовать AWS EMR API или AWS CLI для запуска заданий в кластере;
- постоянный кластер, в котором устанавливаются приложения (например, Apache Hive или Apache Pig), предоставляющие веб-интерфейс для выполнения интерактивных запросов;
- постоянный кластер для запуска выполнения заданий, в котором используется Hadoop API.

Среди этих режимов можно выделить два способа запуска заданий на обработку данных: прямой запуск задания с помощью программных интерфейсов или веб-интерфейсов фреймворка либо в качестве отдельного *шага* (step), выполняемого на временном кластере. Шаг представляет собой единицу работы, содержащую инструкции для выполнения программ, установленных в узлах

кластера. Весь процесс обработки данных на EMR разбивается на такие отдельные шаги, каждый из которых выполняет некую функцию и может находиться в определенном состоянии.

В качестве примеров шагов различного функционального назначения можно привести извлечение данных из внешнего хранилища и добавление их в EMR, выполнение трансформации данных программой или сценарием (например, Pig или Hive) и, наконец, запись результатов в выходное хранилище. В качестве входных и выходных источников данных могут выступать HDFS и бакеты AWS S3.

Может быть следующее состояние шагов:

- *выполнение* (`RUNNING`) — программное обеспечение кластера выполняет работу, описанную в данном шаге;
- *ожидание* (`PENDING`) — ПО кластера ожидает завершения выполнения предыдущих шагов, которые могут находиться в состоянии выполнения, ожидания или в завершенном состоянии;
- *завершенный* (`COMPLETED`) — программное обеспечение кластера успешно завершило выполнение данного шага и перешло к следующему;
- *отказавший* или *завершенный с ошибками* (`FAILED` или `TERMINATED_WITH_ERRORS`) — шаг переходит в такое состояние, если во время его выполнения возникли ошибки;
- *отмененный* (`CANCELED`) — в такое состояние переходят шаги, следующие за отказавшим.

Сам EMR-кластер может находиться в одном из следующих состояний:

- *запуск* (`STARTING`) — состояние кластера, при котором запускаются виртуальные машины с установленным на них программным обеспечением того или иного фреймворка;
- *первоначальное конфигурирование* (`BOOTSTRAPPING`) — это процесс конфигурирования созданного кластера с помощью пользовательских сценариев или команд установки дополнительного софта; следует по завершении запуска кластера;
- *выполнение заданий* (`RUNNING`) — последовательное выполнение цепочки шагов задания; следует после успешного конфигурирования первоначального кластера;
- *ожидание* (`WAITING`) — состояние постоянного кластера, в котором он пребывает после завершения выполнения заданий в ожидании новых заданий;
- *отключение* (`SHUTTING_DOWN`) — состояние кластера, наступающее после инициирования пользователем удаления постоянного кластера;
- *удаленный* (`TERMINATED`) — стадия кластера, следующая после отключения;
- *завершенный успешно* (`COMPLETED`) — стадия временного кластера, наступающая после успешного выполнения задания и удаления всех EC2-узлов кластера;
- *завершенный с ошибками* (`TERMINATED_WITH_ERRORS`) — стадия временного кластера при наличии одной или нескольких ошибок во время его выполнения.

Архитектура AWS EMR состоит из нескольких уровней, каждый из которых покрывает определенный аспект работы кластера.

Самым базовым является *уровень хранения данных*, включающий различные файловые системы, используемые в кластере. В качестве таковых могут выступать HDFS-распределенная файловая система Hadoop (мы ее уже рассматривали в главе, посвященной нереляционным хранилищам данных), файловая система EMR (EMRFS) — расширение HDFS, которое задействует хранилище AWS S3, или локальная файловая система каждого конкретного узла.

Следующий уровень — это *система управления ресурсами кластера* (cluster resource management). По умолчанию EMR использует YARN (Yet Another Resource Negotiator), централизованно управляющий ресурсами кластера и совместимый с большинством фреймворков обработки данных. Кроме того, каждый узел EMR содержит специальный агент, который отвечает за мониторинг состояния кластера и коммуникацию EC2-экземпляра с AWS EMR.

Уровень *фреймворка обработки данных* (data processing frameworks) включает непосредственно тот фреймворк (движок), который используется для выполнения заданий в кластере. Как правило, для его работы не требуется YARN. Главные фреймворки EMR — Hadoop MapReduce и Apache Spark.

И наконец, последним идет уровень дополнительных *приложений и программ* (applications and programs) — надстройка поверх фреймворка обработки данных. На этом уровне доступны такие компоненты, как Hive, Pig (поверх Apache Hadoop), Spark Streaming, MLib, GraphX, Spark SQL (поверх Apache Spark).

Для начала рассмотрим сам фреймворк Apache Spark, запущенный в долговременном кластере EMR.

Начнем с конфигурирования кластера EMR. Если вы в своем аккаунте экспериментировали с ETL-сервисами AWS Glue или AWS Data Pipeline и не поменяли регион, то при выборе EMR в строке поиска сервисов откроется страница EMR (рис. 12.47) со списком кластеров, которые будут иметь автоматически сгенерированные имена и находиться в стадии `TERMINATED` (или `TERMINATED_WITH_ERRORS`). Эти кластеры создавались автоматически для выполнения копирования и трансформации данных.

Чтобы создать свой кластер, следует нажать кнопку Create cluster (Создать кластер). После этого откроется страница конфигурирования кластера (на рис. 12.48 показана ее верхняя часть). Обратите внимание: здесь представлена страница «быстрого конфигурирования». Ссылка Go to advanced options позволяет перейти к более детальному конфигурированию кластера, в частности к указанию дополнительных программных компонентов, которые нужно установить в дополнение к основным.

Итак, прежде всего дадим имя кластеру — в данном случае это неоригинальное `Spark processor`. Далее выберем опцию логирования в S3 (флажок Logging) и укажем местоположение папки в S3-бакете, куда будут помещены логи. Из списка доступных выбираем последний релиз (на момент написания книги это emr-5.12.0), в качестве приложения указываем Spark (совместно с ним будут установлены Hadoop, YARN, распределенная система мониторинга Ganglia и онлайн-редактор Zeppelin). Затем устанавливаем флажок Use AWS Glue Data Catalog for table metadata, который подключает каталог данных AWS Glue в качестве хранилища метаданных таблиц.

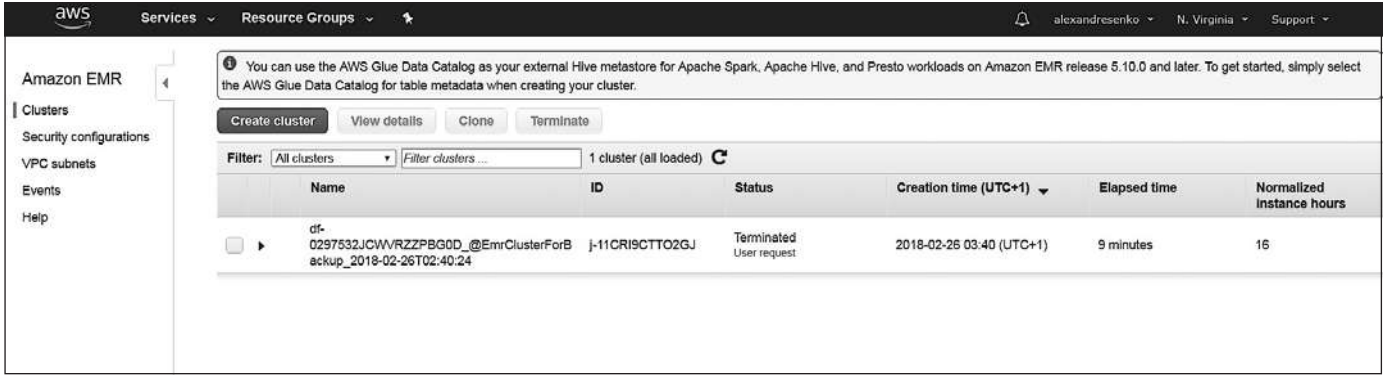


Рис. 12.47. Главная страница сервиса EMR со списком запущенных ранее кластеров

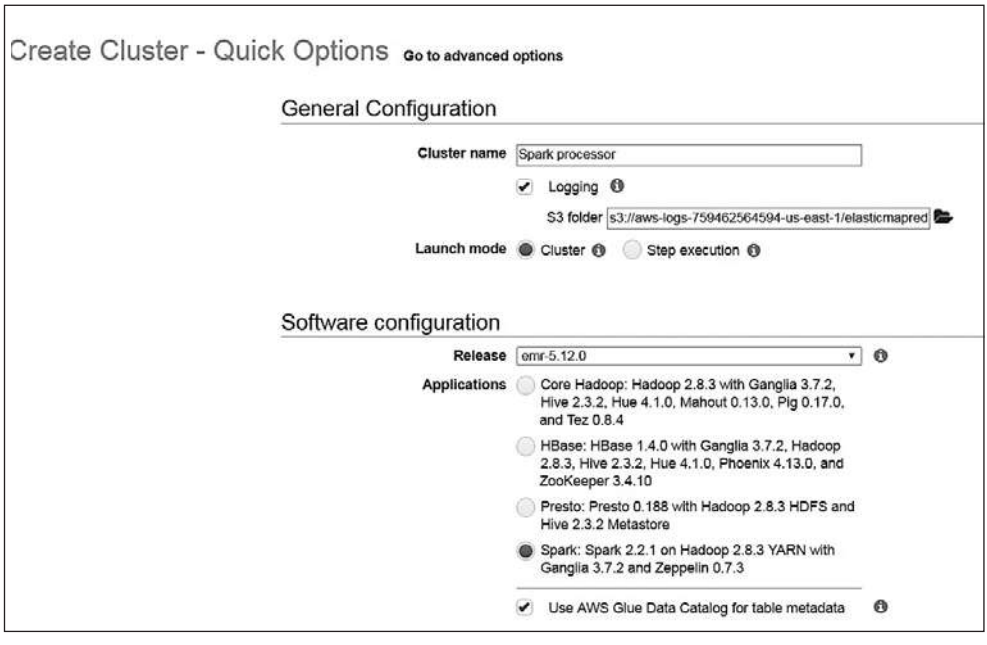


Рис. 12.48. Верхняя часть страницы конфигурирования кластера

Интерактивное взаимодействие пользователя со Spark может происходить двумя основными путями: с помощью онлайн-редакторов (представляют собой веб-страницы, размещаемые в головном узле кластера и позволяющие пользователю подключаться к ним извне через браузер (эти онлайн-редакторы еще называются блокнотами — notebook)), и по протоколу SSH, подключившись к головному узлу. Чтобы воспользоваться вторым путем, необходимо создать пару ключей (public и private). Для этого следует перейти на странице сервиса EC2 на вкладку Key Pairs (рис. 12.49) и нажать кнопку Create Key Pair (Создать пару ключей). В появившемся окне нужно ввести имя создаваемой пары ключей.

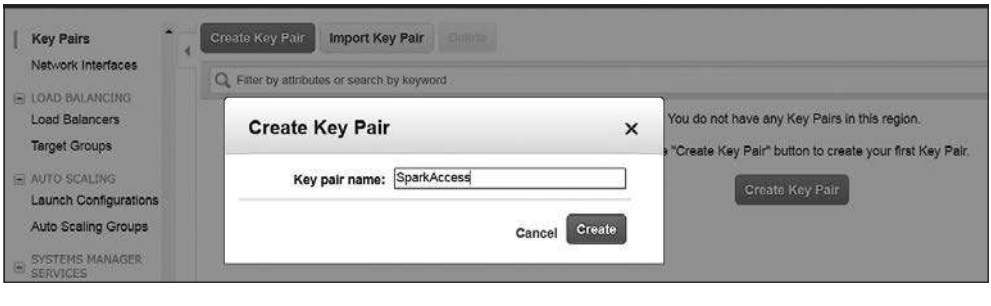


Рис. 12.49. Вкладка создания пары ключей сервиса EC2

После создания пары будет автоматически загружен приватный ключ, а публичный останется в списке ключей, доступных для EC2 (рис. 12.50).

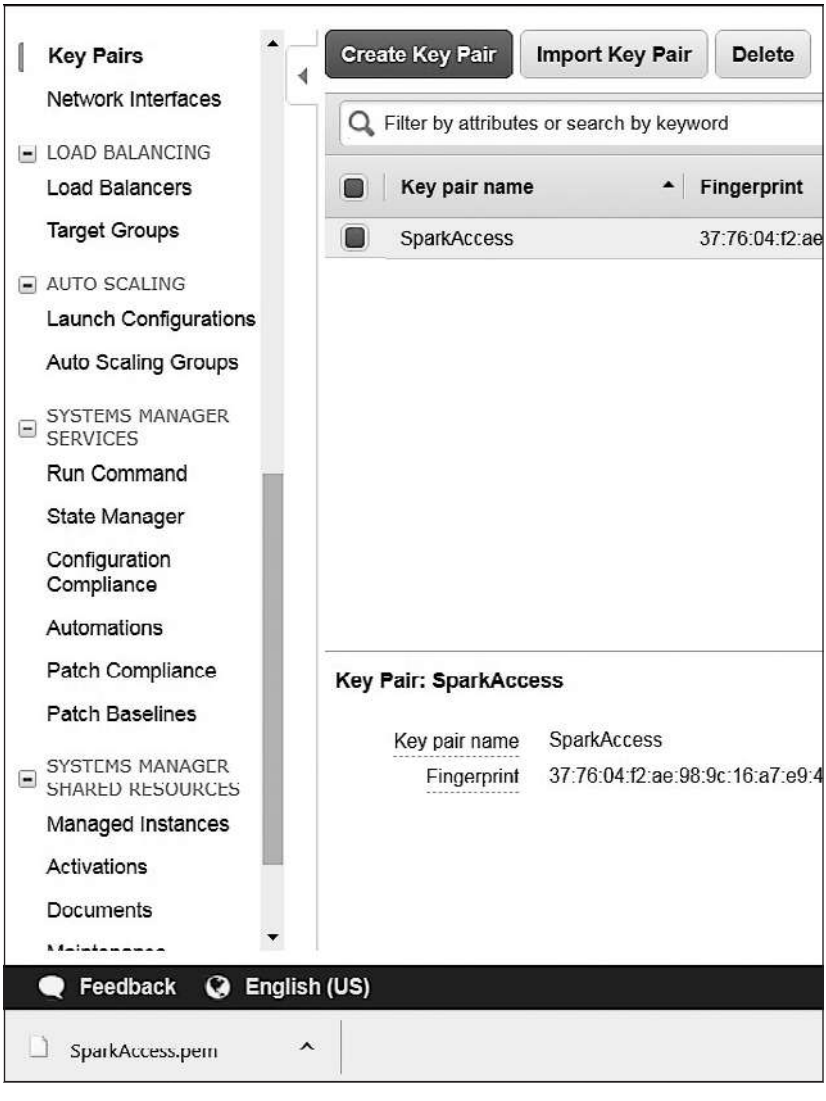


Рис. 12.50. Созданная пара ключей Spark Access

Можно сгенерировать ключи локально и импортировать их (этот функционал доступен при нажатии кнопки Import Key Pair (Импортировать пару ключей)). Далее вернемся к странице конфигурирования кластера. На рис. 12.51 показана нижняя ее часть.

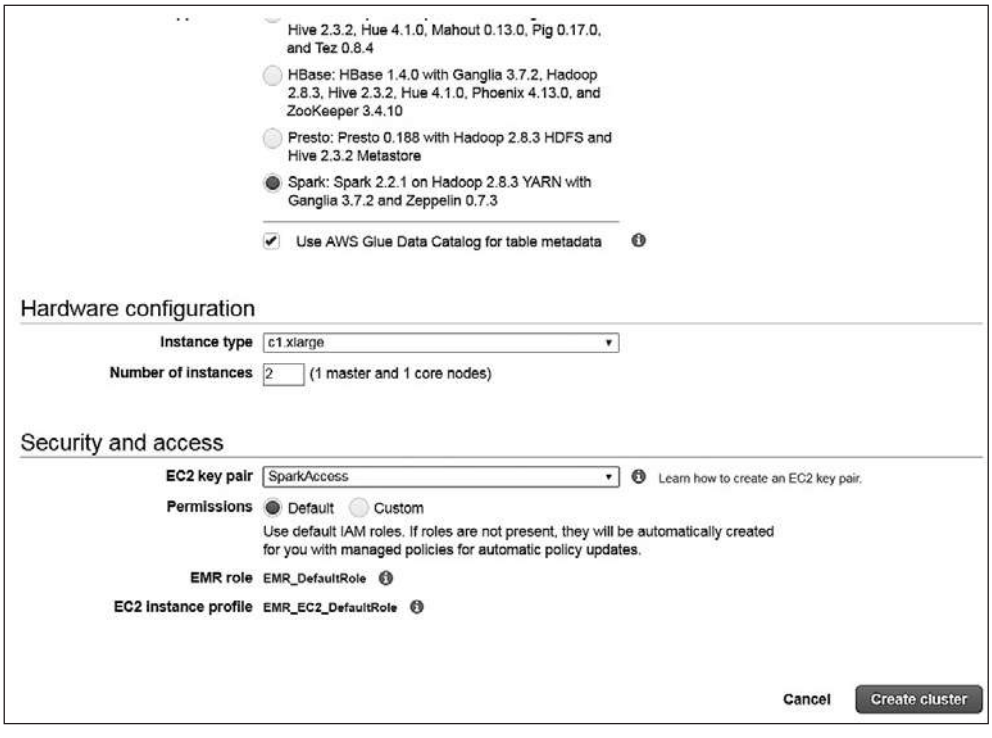


Рис. 12.51. Нижняя часть страницы конфигурирования кластера

Здесь нужно выбрать тип экземпляра (в данном случае это c1.xlarge), количество экземпляров (2 — один головной узел и один обычный) и созданную нами пару ключей. После чего следует нажать кнопку Create cluster (Создать кластер). Будьте внимательны при выборе типа экземпляров — различные типы поддерживаются в разных регионах, кроме того, наименьшие экземпляры могут иметь недостаточные ресурсы.

Далее произойдет автоматический переход на страницу создаваемого кластера. Дождитесь завершения процесса. На рис. 12.52 представлена страница обзора создаваемого кластера.

На странице сервиса EC2 можно наблюдать процесс создания экземпляров кластера (рис. 12.53).

Прежде чем работать с кластером через SSH или онлайн-редактор, опишу основные компоненты программной модели Spark.

Прежде всего, для работы со Spark по SSH служит интерактивная оболочка Spark Shell, которая основана на концепции Scala REPL (Read — Eval — Print — Loop, «чтение — исполнение — печать — циклическое повторение»). Наиболее естественным языком взаимодействия с Spark Shell, доступной по SSH, является Scala. Широко применяется также язык Python. Каждое приложение Spark

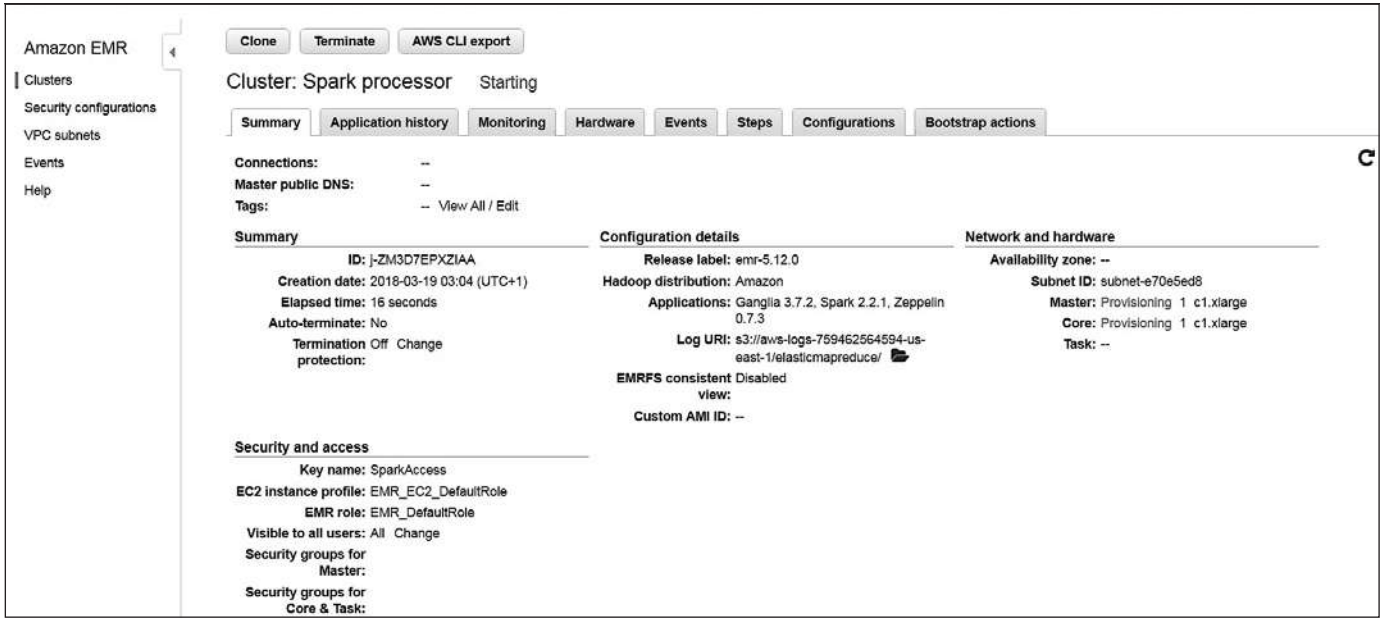


Рис. 12.52. Страница обзора создаваемого кластера

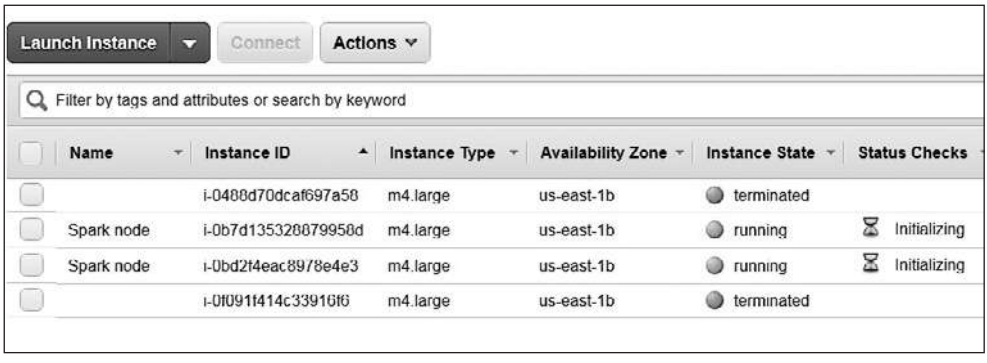


Рис. 12.53. Создание экземпляров узлов кластеров

включает программу-драйвер, запускающую главную функцию (main), которая содержит пользовательскую программу, исполняемую параллельно. Основные сведения о Spark Shell я буду давать на примере использования Scala.

Самый основной программный элемент Spark — *гибкий распределенный набор данных* (Resilient Distributed Dataset, RDD), представляющий собой коллекцию элементов, распределенную среди узлов кластера, благодаря чему можно выполнять параллельные запросы к данным. RDD создается из файла в HDFS или из коллекций Scala. RDD можно сконфигурировать как постоянно пребываемый в памяти объект (persist RDD). Кроме того, RDD отвечает за отказоустойчивость и автоматическое восстановление данных, потерянных при отказе узла кластера.

Следующий базовый программный элемент — *разделяемая переменная* (shared variable). Смысл использования переменных такого типа заключается в том, что Spark запускает исполняемые задания в различных узлах со своими копиями переменных. Разделяемые переменные применяются, когда необходимо иметь переменную, доступную во всех исполняемых заданиях. Существует два вида разделяемых переменных: *широковещательные переменные* (broadcast variables), расположенные в оперативной памяти, доступные для чтения во всех узлах, и *аккумуляторы* (accumulators), используемые для сценария типа переменной-счетчика/сумматора.

Посмотрим на практике, как работать с этими компонентами. Прежде всего необходимо получить доступ по SSH, для чего нужно открыть доступ к головному узлу с вашего IP-адреса. Для этого следует перейти к странице EC2, выбрать экземпляр с головным узлом и добавить новую запись в Inbound Rules.

Для получения инструкций по подключению пользовательского терминала следует перейти на страницу обзора кластера и нажать ссылку SSH (рис. 12.54).

Можно следовать этим инструкциям, а можно, задействуя пользовательский терминал, перейти к локальной папке, где хранится private-ключ, сгенерированный ранее (рис. 12.55).

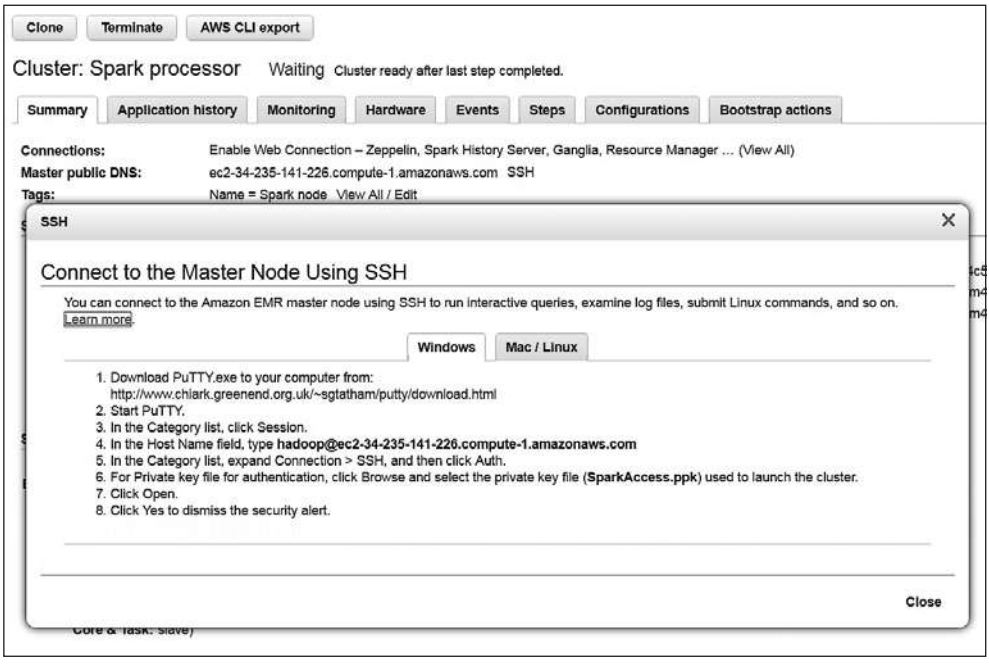


Рис. 12.54. Страница инструкций подключения к кластеру

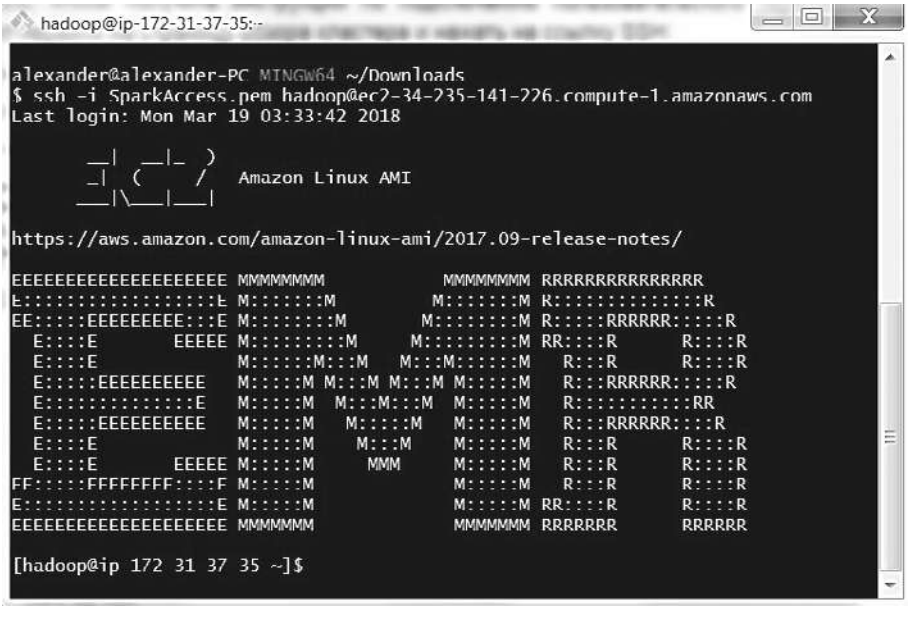


Рис. 12.55. Доступ к головному узлу Spark из пользовательского терминала

И вот вы зашли в кластер под именем пользователя hadoop. Возможность интерактивной работы обеспечит вход в оболочку Spark через команду `spark-shell` (рис. 12.56).

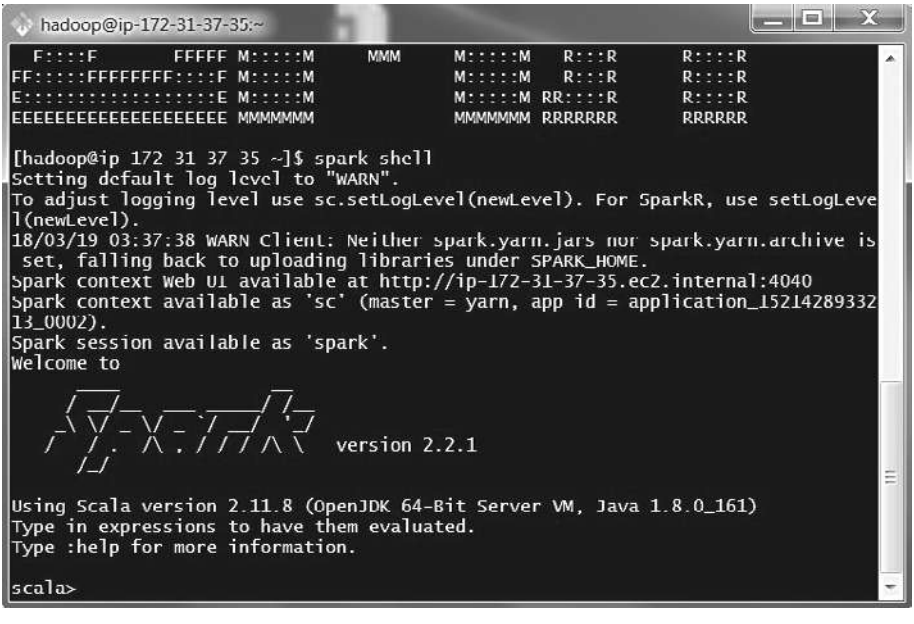


Рис. 12.56. Вход в оболочку Spark

Посмотрим, как создавать RDD в оболочке. В качестве источника данных можно использовать как коллекции Scala, так и файлы, расположенные локально в узле или в S3. Чтобы создать RDD, необходимо иметь доступ к глобальной переменной `sc` — объекту SparkContext — контексту Spark и в этой коллекции выполнить метод `sc.parallelize` данного контекста. В результате выполнения метода создастся RDD, который будет доступен по имени переменной `distData` (рис. 12.57).

Чтобы добавить файл в RDD, нужно использовать метод `sc.textFile` контекста Spark (рис. 12.58).

После того как мы создали RDD, к получившимся переменным можно применять различного рода операции преобразования данных. Все операции над RDD разделяются на две группы: *преобразования*, создающие новый RDD из существующего и возвращающие его пользовательскому драйверу, а также *действия*, возвращающие драйверу некоторое значение после своего выполнения.

Все *преобразования* в Spark Shell являются *ленивыми* (lazy), то есть выполняются только в тот момент, когда требуется получить результат выполнения и вернуть его драйверу пользовательской программы. Каждый раз при выполнении действия или трансформации происходит пересчет RDD; при больших размерах набора на это может уйти довольно много времени. Чтобы обойти данный нюанс, можно

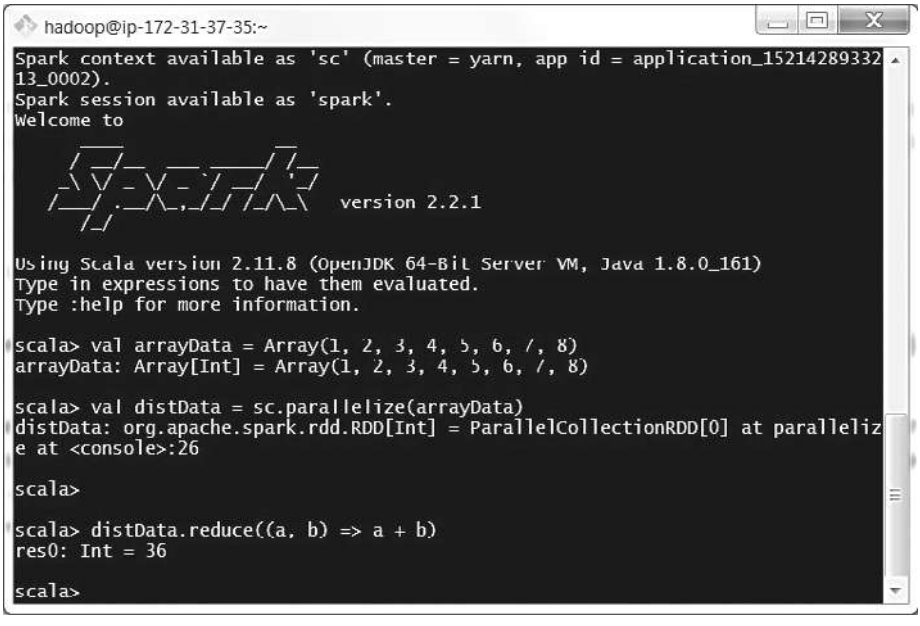


Рис. 12.57. Создание RDD из коллекции Scala и выполнение операции в ней

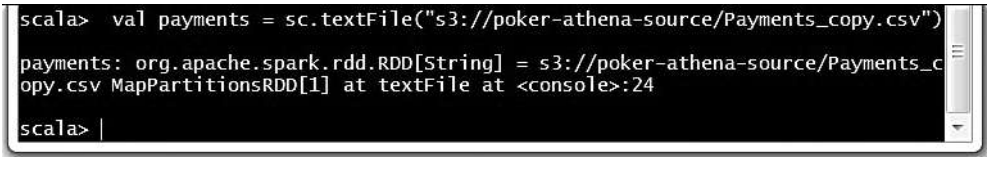


Рис. 12.58. Добавление файла в RDD

использовать метод `persist`, который сохраняет (кэширует) в памяти постоянное значение RDD.

Существуют различные варианты кэширования RDD. Основные из них представлены ниже.

- `MEMORY_ONLY` — это вариант по умолчанию, при котором RDD сохраняется как объект Java в JVM. Если памяти виртуальной машины не хватает, чтобы вместить весь RDD, то незакэшированные элементы будут пересчитаны при последующих действиях и трансформациях, примененных к RDD.
- `MEMORY_AND_DISK` — отличается от `MEMORY_ONLY` тем, что при нехватке памяти JVM данные сохраняются на жестком диске.
- `MEMORY_ONLY_SER` — этот вариант соответствует `MEMORY_ONLY`, за исключением того, что объекты RDD хранятся в памяти JVM в сериализованном виде, то есть как массив байтов (по одному на раздел). Данный тип задействует память

эффективнее MEMORY_ONLY, однако требует более интенсивного использования процессора для операций сериализации-десериализации. Этот вариант доступен только для программ на языке Java и Scala.

- MEMORY_AND_DISK_SER — отличается от MEMORY_AND_DISK тем, что данные в памяти JVM хранятся в сериализованном виде (как и для MEMORY_ONLY_SER).
- DISK_ONLY — обеспечивает хранение данных только на диске.
- MEMORY_ONLY_2, MEMORY_AND_DISK_2 и пр. — варианты с реплицированием информации между двумя узлами кластера.

Вариант кэширования указывается следующим образом (использован RDD payments, полученный нами из CSV-файла): payments.persist(StorageLevel.MEMORY_AND_DISK_SER). Объект StorageLevel содержит поля, представляющие собой перечисление вариантов кэширования. Для отмены кэширования нужно применить операцию payments.unpersist().

Теперь обратим внимание на основные операции трансформации, выполняемые с RDD:

- payments.map(func) — применяет к каждому элементу RDD пользовательскую функцию func и возвращает новый RDD;
- payments.filter(func) — выбирает из итогового RDD payments те элементы, для которых предикат func является истинным;
- payments.union(another_payments_RDD) — возвращает результат объединения (логическая операция UNION) двух RDD (в данном случае payments и another_payments_RDD);
- payments.intersection(another_payments_RDD) — возвращает результат пересечения (логическая операция INTERSECT) двух RDD;
- payments.distinct() — возвращает только уникальные элементы RDD;
- payments_KVP.groupByKey() — для RDD, представляемого в виде набора «ключ — значение», выполняет группировку по ключу;
- payments_KVP.reduceByKey(func) — для RDD, представляемого в виде набора «ключ — значение», применяется func, редуцирующая входной набор. Например, из двух смежных пар набора создается одна, являющаяся комбинацией двух значений этих пар (сумма, разность и пр.);
- payments_KVP.sortByKey() — сортирует по ключу RDD, представленному в виде набора «ключ — значение»;
- payments_KVP.join(another_payments_RDD_KVP) — выполняет операцию соединения двух RDD, представленных в форматах коллекции «ключ — значение». Имеется вариант leftOuterJoin, rightOuterJoin, fullOuterJoin.

В реальности существует гораздо больше доступных трансформаций, я привел наиболее простые и употребительные. Обратите внимание: многие операторы над RDD, с одной стороны, концептуально похожи на SQL (объединение, пересечение,

группировка, сортировка), а с другой — требуют специального формата входных данных (например, для группировки — это представление в виде коллекции из элементов «ключ — значение»). Чтобы получить входные данные требуемого формата, необходимо построить данный набор «ключ — значение» из исходного с помощью функции `map`.

Следующая особенность этих операторов по сравнению с операторами SQL (я сравниваю концептуально, а не интерпретируемый DSL язык запросов к реляционной СУБД и интерпретируемый язык общего назначения) — применение пользовательских функций. Их синтаксис зависит от используемого языка (Scala, Java, Python). Однако некоторые из перечисленных языков (например, Scala) могут применять специальный синтаксис — лямбда-выражения. Этот синтаксис представляет собой анонимную функцию, объявляемую прямо в месте ее использования, например: `(x,y) => x + y`, где `x` и `y` — входные аргументы, а функция возвращает значение их суммы. Лямбда-функции очень удобны, когда необходимо описать однократно использованную функцию с малым количеством операторов. Я не буду описывать пользовательские функции, классы и др., поскольку они специфичны для языка, а не для технологии Spark.

Теперь познакомимся со второй группой операций над RDD — с действиями. Ниже приведены только самые важные из них:

- `input_RDD.reduce(func)` — агрегирует значение объектов RDD функцией с сигнатурой `(in1,in2) => out`. Для того чтобы имелась возможность использовать операцию в параллельном режиме в кластере, передаваемая функция `func` должна быть ассоциативной и коммутативной;
- `input_RDD.collect()` — возвращает все объекты входного RDD в виде массива. Эта операция имеет смысл для отфильтрованных RDD малых объемов;
- `input_RDD.count()` — подсчитывает количество элементов RDD;
- `input_RDD.first()` — возвращает первый элемент из RDD;
- `input_RDD.take(n)` — возвращает `n` первых элементов RDD;
- `input_RDD_KVP.countByKey()` — для RDD, представленного в виде коллекции «ключ — значение», подсчитывает количество элементов для каждого ключа.

Таким образом, и здесь мы видим концептуальные аналоги SQL-операторов `TAKE`, `COUNT`.

Некоторые из приведенных выше операций запускают выполнение *тасования* (shuffle) — перераспределения данных среди разделов и узлов кластера, что в случае больших объемов данных является ресурсозатратной операцией.

Помимо интерактивной работы в оболочке Spark, имеется возможность развертывания программ, созданных вне его командной оболочки, в кластер. В этом случае они могут работать в фоновом режиме.

Итак, я привел краткий обзор возможности Spark Shell и описание некоторых его программных интерфейсов. Сразу бросается в глаза, что для написания программ (а это именно программы, а не просто запросы к данным) необходимо знать

один из языков программирования, который поддерживается оболочкой Shell. Но все поддерживаемые языки, включая Scala, работают с распределенными наборами данных в той манере, в какой позволяет синтаксис работы с коллекциями или экземплярами классов, что в ряде случаев не так удобно, как это сделано в SQL. Например, для операций группировки требуется предварительное преобразование RDD к набору «ключ —значение». В то же время для SQL не требуется никаких дополнительных преобразований. Кроме того, специалистам по анализу и обработке данных (data analyst и data scientist) для работы в оболочке нужно знать ее поддерживаемые языки, что требует времени на обучение.

Указанные выше соображения привели к тому, что в рамках Spark был разработан компонент Spark SQL, который обеспечивает интерактивную обработку данных в Spark с помощью языка SQL. Рассмотрим этот компонент. Он позволяет выполнять запросы к наборам данных, используя синтаксис и SparkSQL, и HiveQL, и поддерживает такие его компоненты, как SerDes (компоненты сериализации и десериализации источников данных), UDF (User defined functions — «определяемые пользователем функции»), а также Hive Meta Store (каталог метаданных Hive) (рис. 12.59).

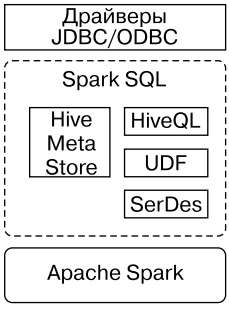


Рис. 12.59. Архитектура компонента SparkSQL

Кроме того, SparkSQL представляет интерфейс JDBC/ODBC, позволяющий получить доступ из сторонних приложений (в том числе средств бизнес-аналитики) для выполнения запросов к данным.

Существующие возможности взаимодействия с SparkSQL представлены ниже.

- *Доступ по Spark Shell*, используя программные конструкции Spark API. Технологически это представляет собой ту же программу оболочки Spark, где операции трансформации и действий заменены на выполнение метода, аргументом которого будут операторы SQL. Подготовка данных в этом случае отличается от подготовки данных для RDD, что мы подробнее рассмотрим далее.
- *Доступ из программы, задеплоенной в кластер,* — в сущности, те же программные конструкции, которые используются в Spark Shell, только собранные в исполняемый программный пакет и задеплоенный в кластере Spark.
- *Доступ извне по протоколам JDBC/ODBC* — возможность работать с SQL как внешним редакторам SQL, так и приложениям бизнес-аналитики, в том числе системам построения диаграмм (например, PowerBI).
- *Доступ из браузера через специализированные онлайн-редакторы* — кластеры Spark предоставляют доступ как к оболочке, так и к компоненту SQL из веб-страниц, которые отдаются по HTTP-запросу веб-сервером одного из узлов кластера. Эти онлайн-редакторы называются еще блокнотами (notebooks). Чаще всего в качестве таковых выступают Jupyter и Zeppelin. Первый более широко используется для языков Python, R, Bash и ряда других. Доступ к Spark SQL

из этого редактора возможен таким же образом, как из Spark Shell. Редактор Zeppelin поддерживает Spark, PySpark (библиотеки Python для Spark), R и Spark SQL. Поэтому он является наиболее удобными и чаще всего используется в качестве онлайн-редактора для Spark.

Базовый элемент организации данных — *кадры данных* (Data Frames). Они представляют собой распределенные источники данных, информация в которых организована в виде проименованных колонок, что существенно отличается от RDD. В качестве первоначальных источников для RDD могут выступать структурированные файлы, таблицы, описанные в источнике метаданных Hive, и таблицы во внешних реляционных базах данных. Возможно чтение из файлов форматов JSON, CSV, Parquet, ORC. Сама операция создания объекта `DataFrames` из файла включает в себя чтение файла, определение его схемы, выполнение запроса к данным и отображение результатов.

Создание `DataFrames` из файла выглядит следующим образом: `val paymentsDataFrame = spark.read,json("path_to_file")`, где `spark` — это экземпляр `SparkSession`. Чтобы посмотреть контент файла, необходимо выполнить метод `paymentsDataFrame.show()`. Распечатать схему можно с помощью операции `paymentsDataFrame.printSchema()`. Чтобы программно запустить SQL на объекте `DataFrame`, следует сначала зарегистрировать именованное временное представление (temporary view), используя оператор `paymentsDataFrame.createOrReplaceTempView("payments")`. Далее выполняются непосредственно запрос и оператор вывода результата на экран (рис. 12.60).

```
val commonResponse = spark.sql("SELECT * FROM payments")
commonResponse.show()
```

Рис. 12.60. Применение операторов SQL к именованному временному представлению

Это временное представление имеет ограниченное время жизни — в пределах одной сессии Spark. Для создания представления, доступного из разных сессий, используется *глобальное временное именованное представление* (global temporary view). Необходимо создать его с помощью метода `paymentsDataFrame.createGlobalTempView("payments")`. На этом я заканчиваю описание Spark SQL. Дальнейшую информацию можно получить в тематической документации.

12.6. Встроенные редакторы запросов сервиса CosmosDB

В приведенных выше разделах мы рассмотрели сервисы анализа данных, которые, прежде чем стать доступными для анализа, требуют переноса и/или трансформации, а именно применения процедур ETL/ELT. При этом в системе будет две копии данных: непосредственно в основном хранилище и в хранилище сервиса

аналитики. Основное хранилище — это, как правило, нереляционное хранилище данных, отличающееся низкой стоимостью и высокой масштабируемостью, например, типа «ключ — значение» (HBase, Azure Table Storage или AWS DynamoDB и др.) или документоориентированное (MongoDB). Подобные СУБД обеспечивают высокую производительность запросов типа выборки отфильтрованных значений, сортировки и поиска по ключу. Очевидно, что этого может быть достаточно для построения программных продуктов, которые будут извлекать данные без сложной трансформации или агрегации.

В то же время для целей аналитики требуется хранение данных в системах, дающих гораздо более широкие возможности выполнения запросов, в том числе и агрегацию, комбинирование различных источников данных. Если данные хранятся в РБД, то там можно использовать встроенный движок SQL, но в случае нереляционных БД, как правило, требуется загрузить данные в хранилище, допускающее аналитические запросы: в реляционное хранилище типа Azure DWH или AWS Redshift либо нереляционное типа Azure Data Lake Store. Однако сервис CosmosDB содержит встроенный редактор и движок, позволяющий выполнять запросы SQL, в том числе с помощью определяемых пользователем функций и хранимых процедур для запросов к данным. Чрезвычайная масштабируемость CosmosDB и приемлемая стоимость хранения позволяет использовать этот сервис для целей как OLTP, так и OLAP. Мы уже рассматривали основы работы с этими возможностями в главе 6, а здесь мы изучим более подробно особенности синтаксиса SQL для DocumentDB. Этот компонент CosmosDB так и называется — SQL API. Рассмотрим SQL API CosmosDB, поскольку встроенный в него язык запросов наиболее развит по сравнению с другими компонентами.

Итак, для CosmosDB форматом представления как входной, так и выходной информации является JSON. И его не нужно преобразовывать явно в табличный формат, как происходит, например, в случае AWS Athena или Spark. Это естественный формат DocumentDB. Он накладывает определенный отпечаток на синтаксис SQL, имеющий расширения, которые позволят форматировать выход как документ JSON. Это проявляется, в частности, в том, что документы SQL могут содержать в своем составе коллекции и объекты, работать с которыми можно так же, как и с коллекциями и объектами JSON. Предположим, у нас есть документ со следующей структурой (рис. 12.61).

Простой запрос, показывающий особенности таковых для документов JSON, показан на рис. 12.62. В данном случае я выполнил запрос непосредственно из коллекции, доступ к которой возможен по псевдониму (alias) `c`. Из этого примера видно, что для выборки данных могут применяться элементы синтаксиса JavaScript. Так, например, для выбора первого значения из массива `Players` задействован синтаксис `c.Players[0]`, а для доступа к полю `Bank` объекта `TableState` используется синтаксис `c.TableState.Bank`. При этом возможно применение синтаксиса типа `c["id"]`, что позволяет работать с полями документа, имена которых содержат пробелы.

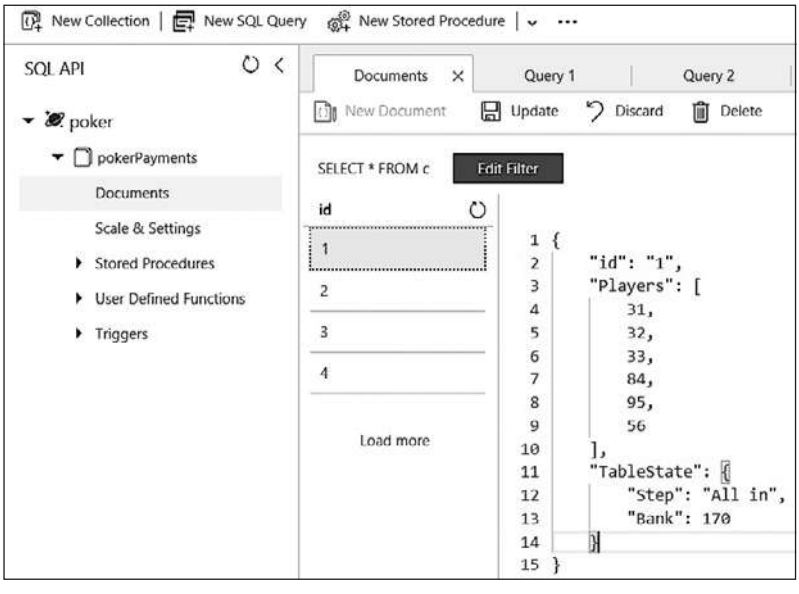


Рис. 12.61. Пример JSON-документа, доступного для построения запросов SQL

Рис. 12.62. Запрос к сложным JSON документам

Кроме того, можно явно форматировать вывод, указав формат выходного документа (рис. 12.63).

Рис. 12.63. Явное форматирование выходной информации в SQL API CosmosDB

Помимо форматирования вывода, можно применять более сложные операции, доступные из синтаксиса JavaScript (арифметические операции: сложение, вычитание, умножение и деление), а также операций из синтаксиса SQL (соединение (join)). Существенная особенность SQL DocumentDB — применение функций, определяемых пользователем, которые задействуют JavaScript. Возможности этого языка позволяют создавать достаточно сложную логику обработки типов данных, поддерживаемых DocumentDB. Кроме того, существует достаточно много встроенных функций, которые могут быть разделены на следующие типы:

- *математические функции* (`ABS`, `ASIN`, `TAN` и др.) — функции, результатом выполнения которых будет применение широко известных математических операторов ко входным параметрам;

- *функции проверки типов* (`IS_ARRAY`, `IS_BOOL` и пр.) — функции, возвращающие булево значение — истинно или ложно, в зависимости от того, принадлежит ли объект тому или иному типу (например, `IS_ARRAY` проверяет, является ли переменная массивом);
- *функции работы со строками* (`CONCAT`, `LENGTH`) — функции, преобразующие входные строки в выходные (например, `CONCAT` объединяет, конкатенирует строки в одну общую) или вычисляющие метрику строки (`LENGTH` — вычисляет длину строки);
- *функции работы с массивом* (`ARRAY_LENGTH`, `ARRAY_CONCAT` и др.) — аргументами этой функции являются массивы, а выходным значением — или новый массив, или численная величина, характеризующая его, например длина;
- *пространственные функции* — работающие с геопространственными типами данных;
- *функции агрегирования* (`COUNT`, `SUM`, `MAX`, `MIN`, `AVG`) — функции, работающие с коллекциями объектов.

Особенность базы данных CosmosDB — автоматическое индексирование документов. При этом возможно указание первичного ключа и построение вторичных индексов (ключей сортировки). В качестве недостатков языка запросов CosmosDB можно указать неполную поддержку синтаксиса SQL, например отсутствие ряда операторов группировки, что требует применения пользовательских функций, написанных на языке JavaScript.

12.7. Резюме

В этой главе мы рассмотрели наиболее распространенные системы интерактивного анализа больших данных. Все они могут быть подразделены на две большие группы — системы анализа данных, размещенных в реляционных и нереляционных хранилищах. Наличие современных систем, предоставляющих интерфейс SQL, позволяет нивелировать различия в синтаксисах запросов и алгоритмах работы специалиста по обработке данных. Обе эти группы подразумевают копирование данных из внешних источников в хранилище сервиса аналитики. Чаще всего для реляционного хранилища используется подход ETL — извлечение, преобразование, загрузка. Облачные провайдеры Azure и AWS имеют сервисы, автоматизирующие этот процесс: AWS Glue, AWS Data Pipeline и Azure Data Factory. Все эти системы могут на ходу трансформировать данные, запуская соответствующие сценарии в управляемых кластерах, в которых может быть запущен фреймворк из экосистемы Hadoop. Такой традиционный подход удобен тем, что непосредственно анализ данных производится на стандартном языке SQL (T-SQL или PostgreSQL), который применяется к данным, представленным в виде таблиц. Однако этап трансформации требует создания сценария для работы в управляемом кластере на языке,

отличном от стандартного SQL, и, таким образом, нарушается однородность всей системы. Особенно много сложностей в этой модели доставляют копирование и трансформация данных, хранящихся в файлах.

Использование нереляционного хранилища позволяет сначала копировать данные без применения трансформации, которую можно выполнить уже после копирования. Но в таком случае и трансформация, и анализ данных выполняется в рамках одной и той же программной модели. При этом все данные из всех источников преобразуются в файлы, которые в конечном итоге и обрабатываются.

Какую же систему следует выбирать? Это очень сложный вопрос, решаемый комплексно и индивидуально для каждой системы отдельно. Здесь нужно учитывать много нюансов работы системы. Если источниками данных являются в основном реляционные базы, то наиболее естественным хранилищем будет реляционное. Более того, если требуется агрегация данных, то можно выполнить процедуру ELT, при которой данные сначала загружаются в хранилище в промежуточную таблицу, затем для нее назначаются соответствующие индексы и выполняется запрос (сценарий) группировки, помещающий информацию в «рабочую» таблицу. Такой поток очень прост и позволяет применять на каждом шаге широко известные технологии в виде стандартных серверов баз данных и хранилищ. Кроме того, средства BI в этом случае очень хорошо интегрируются с любым компонентом такого хранилища.

С другой стороны, для нереляционных данных, хранящихся в соответствующих базах или в файлах, более естественным хранилищем будет нереляционное. Файлы просто копируются в него без каких бы то ни было преобразований. Такое копирование может быть выполнено как стандартными программами копирования файлов (например, AdCopy для случая Azure Data Lake Store) (в подобном случае данные просто свалены в это хранилище, потому и называются «озеро данных»), так и специализированными сервисами, развертываемыми в кластере (например, Apache Sqoop и Apache DistCp). Задача построения таблиц со схемой ложится на сервисы аналитики нереляционных данных, которые извлекают метаданные из таблицы и сохраняют их в хранилище метаданных. Сервисы аналитики нереляционных данных сейчас также предоставляют интерфейсы SQL (например, Apache Spark), доступные в том числе по протоколам ODBC/JDBC. Основная трудность такого хранилища — создание таблицы метаданных из файла в хранилище, для чего следует знать формат файла, символы-разделители и др. Таким образом, если среди источников данных преобладают файлы, то естественно выбрать нереляционное хранилище. Однако подобные системы достаточно сложны, настройка кластера с системой аналитики и первоначальное подключение к ней сервиса BI может занимать достаточно длительное время.

Оба этих подхода имеют общий недостаток, связанный с тем, что перед выполнением анализа данные должны быть скопированы в хранилище. То есть перед стадией анализа будут присутствовать стадии трансформации и копирования, что в ряде случаев значительно увеличит общее время анализа. Существует альтернатива

такому подходу: использовать распределенные системы анализа, которые «подключаются» к источникам данных, расположенным в их первоначальных местах, в том числе в файловых хранилищах. К таковым системам можно отнести Apache Presto, Microsoft PolyBase, AWS Redshift Spectrum и AWS Athena. Если источниками являются таблицы реляционных баз данных, то такие системы, безусловно, нагружают базу, хранящую эти таблицы.

Ну и наконец, ряд систем управления базами поддерживает выполнение сложных запросов к данным, непосредственно хранящимся в этих базах, а именно реляционные СУБД (Azure SQL и AWS RDS) и документоориентированная СУБД SQL API из семейства CosmosDB. Однако такое решение оправданно только для небольших баз, не содержащих миллионы строк, одновременно участвующих в OLTP-операциях.

13 Потоковый анализ данных

— Я сяду, — ответил кот, садясь, — но возражу относительно последнего. Речи мои представляют отнюдь не пачкотню, как вы изволите выражаться в присутствии дамы, а вереницу прочно увязанных силлогизмов, которые оценили бы по достоинству такие знатоки, как Секст Эмпирик, Марциан Капелла, а то, чего доброго, и сам Аристотель.

Михаил Булгаков. Мастер и Маргарита

13.1. Общие сведения

Потоковый анализ состоит в анализе потока в виде последовательности сообщений, то есть в применении алгоритма анализа к каждому сообщению потока. Этот анализ может состоять в выделении из всего потока сообщений, соответствующих определенным признакам, например сообщения об ошибках, которые должны быть направлены в отдельное хранилище или сервис, занимающийся обработкой ошибок. Следующий случай — задача маршрутизации сообщений потока в различные направления в зависимости от признаков, ассоциированных с сообщением (например, в зависимости от значения в определенном поле сообщения). Очень интересный пример потокового анализа сообщений, одновременно относящийся к интеллектуальному, — анализ банковских транзакций и определение в этом потоке потенциально подозрительных (скажем, выдача большой суммы в нетипичной для плательщика стране или выдача мелких сумм в разных банкоматах в течение малого интервала времени и др.).

По сути, потоковый анализ вполне представи́м как своеобразная фильтрация входящего потока. Вы совершенно резонно можете спросить: а чем, собственно,

такой анализ отличается от потокового анализа оцифрованного сигнала в классических системах цифровой обработки и фильтрации сигналов? Разберемся.

Единица информации, поступающая в систему потокового анализа BigData, — *сообщение*. Это порция информации, состоящая из ряда полей, а именно как минимум из идентификатора, временной метки и собственно поля, несущего полезную информацию (она еще называется *полезной нагрузкой* — payload). Кроме того, возможно наличие других полей: номера сообщения в последовательности, полей типа, метаданных и пр. В то же время классические системы анализа сигналов имеют дело с периодическими отсчетами какой-либо величины, характеризующимися временны́м положением и собственно величиной отсчетов. Это может быть, например, последовательность импульсов с тем или иным видом модуляции (амплитудная, временна́я, широтная, дельта-модуляция и др.) или цифровых отсчетов, представляющих собой последовательность закодированных двоичным кодом импульсов, несущих ту же информацию, но в двоичном виде. И каждый элементарный импульс или группа импульсов двоичного кода несут только скалярную информацию в виде числа, которому соответствует код, или числа, соответствующего величине модулирующего напряжения (например, амплитуды, временно́го сдвига). И классическая задача анализа импульсных сигналов состоит в анализе потоков скалярных величин.

В отличие от этого потоковый анализ *данных* состоит в анализе потока векторных величин, поскольку каждый элемент данных можно представить векторной величиной, состоящей из набора полей. Следующее отличие — случайный момент поступления сообщения, что отличает его от строго периодического потока величин отсчетов у сигнала (въедливый читатель может привести контрпример с время-импульсной модуляцией, где момент поступления каждого импульса случаен, но именно в этой случайности, а точнее, в величине сдвига момента поступления импульса от детерминированной несущей последовательности и закодирована полезная информация). В этом плане ближе аналогия с IP-пакетами — они тоже представляют собой пакеты, состоящие из набора полей и поступающие в устройства обработки (телекоммуникационное оборудование) в произвольные моменты времени и перенаправлены по разным каналом в зависимости от величин, хранящихся в их полях (например, адреса, контрольной суммы, битов QoS и пр.).

Форматы потокового анализа данных могут быть как текстовыми (обычно JSON, CSV), так и бинарными (Apache Avro или ProtocolBuffer). На мой взгляд, самый удобный формат в плане отладки, безусловно, JSON, поскольку он позволяет непосредственно отслеживать сообщения на любом этапе (хранение, отправка в концентратор, прием сервисом потокового анализа, и др.), не прибегая к необходимости программной десериализации. Кроме того, в отличие от CSV, JSON позволяет оперировать более структурированными данными (вложенными объектами, массивами и пр.). Как отмечалось в части I, его основной недостаток — повышенный объем сообщения, что становится критическим фактором при боль-

ших сообщениях и огромном потоке сообщений, поскольку требует применения более компактного бинарного формата. Но принципы работы сервиса потокового анализа со всеми этими форматами одинаковы, а потому далее рассмотрим только JSON.

Сервисы потокового анализа должны обладать высоким быстродействием, то есть минимальным временем отклика при всем диапазоне нагрузок потоков сообщений, быть надежными, масштабируемыми, легко сопрягаться с внешними сервисами (источники, приемники, захват). Вот тут-то и проявляются все преимущества облачных сервисов PaaS — они обеспечивают автоматическое масштабирование и требует минимального администрирования.

Как правило, сервисы потокового анализа состоят из таких компонентов, как:

- *источники* — входные каналы поступления сообщений. Обычно это концентраторы сообщений или IoT-концентраторы. Возможны и другие сервисы анализа;
- *приемники* — выходные каналы, по которым сообщения будут выходить из сервиса потокового анализа;
- *интеграция* — со сторонними сервисами (например, машинного обучения);
- *выполняемые задания* — алгоритм, применяемый к потоку данных и записанный с помощью специализированного языка, поддерживаемого сервисом потокового анализа.

Широко известные масштабируемые сервисы анализа потоковых данных — Apache Storm, Apache Spark Streaming и Streaming API, входящие в состав Apache Kafka. Apache Spark содержит компонент Spark Streaming, предоставляющий программный интерфейс для работы с потоковыми данными. Apache Kafka Streaming API предоставляет интерфейс для внешних программ, позволяющий потреблять поток сообщения из них. Например, связка Apache Kafka + Apache Spark является типовой для систем анализа и обработки потоковых данных. Apache Storm целиком предназначен для анализа потоковых данных с соответствующей архитектурой.

Указанные сервисы в облаках могут быть развернуты в управляемых кластерах (AWS EMR и Azure HDInsight), что существенно ускоряет процедуру первоначального конфигурирования. Подобный вариант развертывания позволяет, с одной стороны, автоматизировать процесс развертывания и управления кластера, а с другой — обеспечивать гибкость конфигурирования самой системы. Однако настройки всей системы «концентратор сообщений — сервис потокового анализа» могут быть достаточно сложными, что повлияло на появление облачных сервисов PaaS. К таковым относятся сервисы Azure Stream Analytics и AWS Kinesis Analytics. Эти сервисы легко интегрируются с концентраторами сообщений Azure Event Hub и AWS Kinesis Data Stream соответственно.

13.2. Azure Stream Analytics

Итак, рассмотрим облачный сервис потоковой обработки Microsoft-сообщений Stream Analytics. Он позволяет проводить анализ потока сообщений, поступающих на его входы, с помощью специального запроса, написанного с привлечением SQL-подобного синтаксиса. Каждый такой запрос запускается в виде специального задания (job). Как уже указывалось выше, концептуально входной поток сообщений представляется как таблица, состоящая из столбцов — полей сообщения и строк собственно конкретных сообщений. При этом неважна частота повторяемости сообщений: одно в час или десять в секунду — задание будет отрабатывать в любом случае.

Сервис поддерживает следующие форматы сообщений: JSON, CSV и Avro.

Источниками сообщений для сервиса потокового анализа могут быть: концентратор сообщений (Event Hub), шлюз IoT (IoT Hub) или напрямую Azure BLOB Storage. Типовой случай использования источника последнего типа — потоковый анализ лог-файлов, загружаемых в Azure BLOB Storage. В этом случае формат сообщений будет CSV. Применительно к Event Hub или IoT Hub сообщения имеют формат или JSON, или Avro.

Итак, экземпляр сервиса Azure Stream Analytics, называемый *потоковым заданием* (Streaming Job) включает в себя один или несколько входов, запросов и выходов, куда будут направлены результаты применения запроса к одному или нескольким выходным потокам сообщений (рис. 13.1).

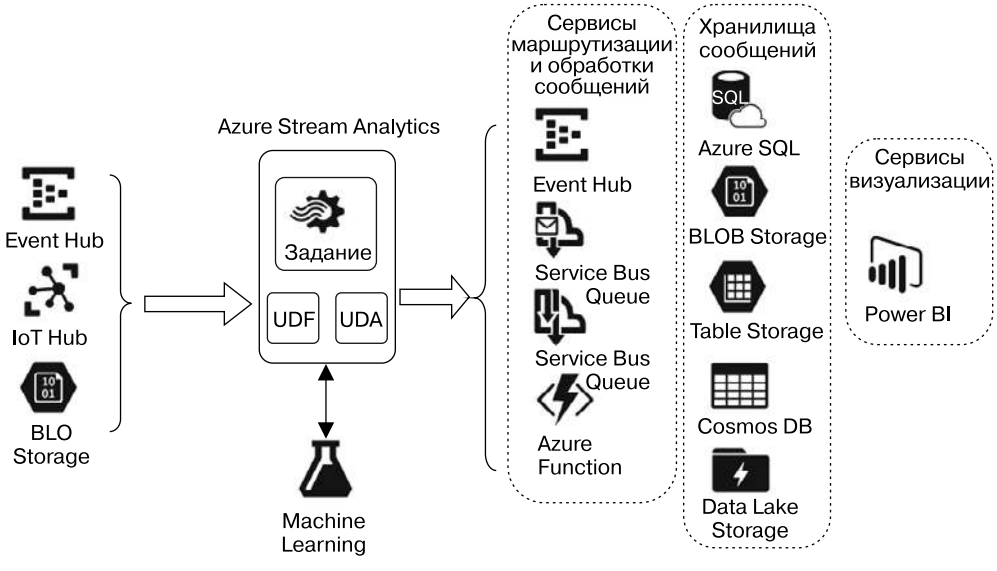


Рис. 13.1. Общая структура сервиса Azure Stream Analytics

Все разнообразие выходных потоков может быть разделено на три группы.

- *Сервисы маршрутизации и обработки сообщений* — к ним относятся облачные сервисы, реализующие архитектуру Event Driven. Это концентраторы сообщений (Event Hub) и сервисы шины интеграции (Service Bus Queue и Topic). Предусматривается также прямой вызов бессерверного сервиса Azure Function. Сценарий использования этой группы выходных потоков может быть разным. Например, для построения цепочки потоковых фильтров необходимо соединить их через промежуточные Event Hub. Service Bus Queue или Topic можно применить для интеграции с сервисами обработки и реакции на сообщения, созданные на базе Service Bus SDK и размещенные на виртуальных машинах в облаке или в стороннем окружении. Azure Function может реализовать функцию реакции на сообщения в бессерверном окружении. В настоящий момент не реализована возможность отправки сообщений в сервисы Azure Queue Storage, напрямую в другой экземпляр исполняемого задания и в Azure IoT Hub.
- *Хранилища сообщений* — обеспечивают непосредственное сохранение результатов потоковой обработки в том или ином хранилище. Это может быть Azure SQL, Azure BLOB и Table Storage, Azure CosmosDB и Azure Data Lake Storage. Сохранение сообщений в Azure File Storage не предусмотрено. Подобный вид выхода используется в сценариях, относящихся к BigData: есть входной поток данных, требуется его предварительная фильтрация и сохранение результатов, которые в последующем нужно будет, например, агрегировать за длительный период, прибегнув к потоковому анализу (об этом — в следующей главе).
- *Сервисы визуализации* — при потоковом анализе сообщений зачастую возникают задачи, требующие визуального мониторинга и наблюдения за текущим потоком. Для этого могут использоваться BI-сервисы визуализации, которые и будут графически отображать результаты фильтрации. Наглядный пример подобного применения Azure Stream Analytics — отображение количества ошибок в единицу времени, получаемого путем непрерывной фильтрации лог-файлов, поступающих в BLOB Storage. Само потоковое задание на фильтрацию включает в себя SQL-запрос, выбирающий только те записи логов, которые соответствуют ошибкам, и суммирующий их количество в заданном временно́м окне (о синтаксисе запросов — чуть ниже). В настоящее время прямая интеграция реализована только с сервисом Power BI.

Теперь подробнее ознакомимся с тем, как создавать экземпляр потокового задания с помощью портала Azure. Для этого в веб-портале следует нажать ссылку + New и в появившемся окне поиска набрать Stream Analytics Job (рис. 13.2).

Затем должна появиться страничка с описанием Stream Analytics Job. Нажмите кнопку Create (Создать) (рис. 13.3).

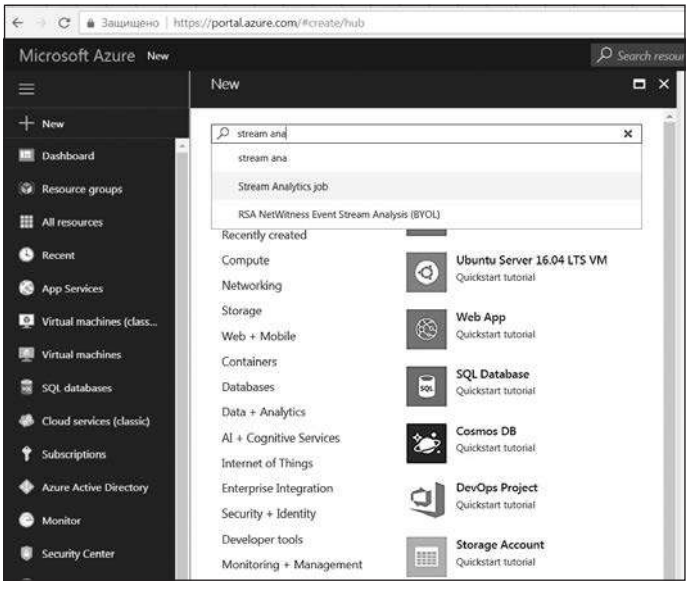


Рис. 13.2. Поиск сервиса Azure Stream Analytics Job среди доступных для создания

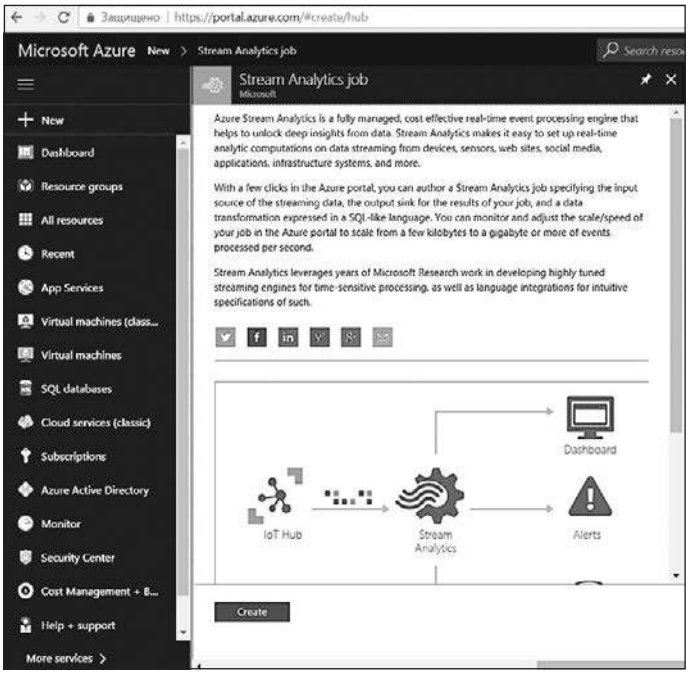


Рис. 13.3. Страница портала с кратким описанием сервиса

В появившейся форме (рис. 13.4) следует указать имя вашего сервиса (в нашем примере это `mainGameFilter`), выбрать существующую или создать новую группу ресурсов (сейчас это общая группа PockerRumExample), задать местоположение датацентра (в примере это Central US). Далее нужно выбрать облако как хостинг (Hosting Environment — Cloud) и установить флажок Pin to Dashboard (Поместить виджет на диаграмму).

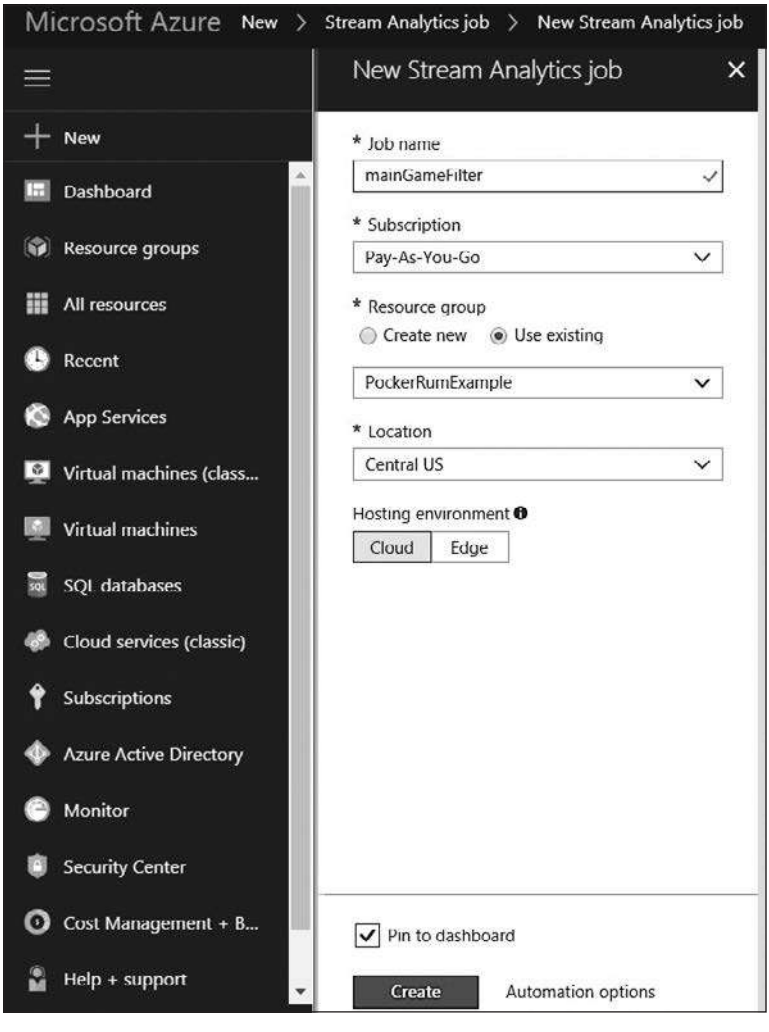


Рис. 13.4. Начальная форма сервиса Stream Analytics Job

Пока что сервис не настроен и содержит пустую стартовую страницу с примером запроса (рис. 13.5).

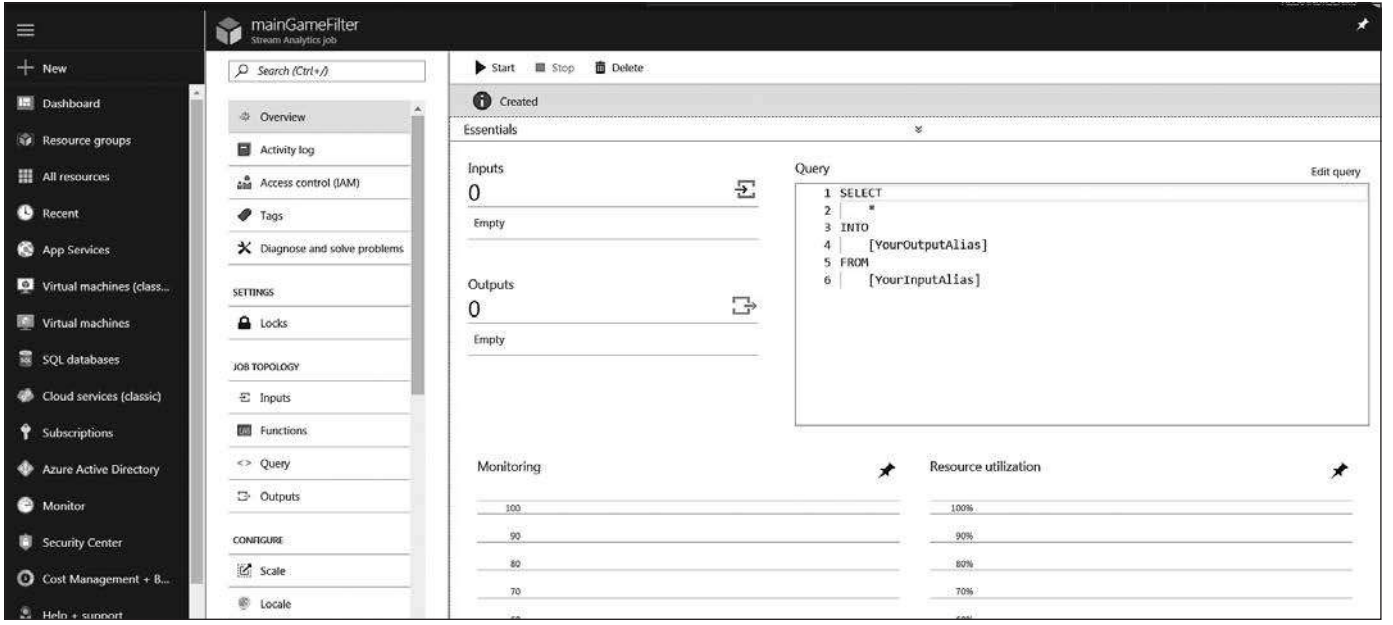


Рис. 13.5. Начальная страница сервиса Stream Analytics Job

Займемся настройкой сервиса на нашем примере покер-рума. Как уже отмечалось, сервис потокового анализа может выступать в качестве фильтра сообщений, перенаправляющего сообщения заданного вида из источника в приемник, а также для аналитических преобразований, необходимых для онлайн-мониторинга или построения системы реагирования на определенные сообщения или на определенную агрегируемую величину.

Итак, прежде всего следует настроить входы, выходы и собственно выполняемый запрос. Остальные возможности Stream Analytics Job рассмотрим далее. Как уже отмечалось, входами для Stream Analytics могут быть Event Hub, IoT Hub и BLOB Storage. Архитектура нашего покер-рума построена на основе приема сообщений через Event Hub, и этот источник лучше всего подходит для демонстрации работы Stream Analytics Job. Для настройки входов следует нажать на ссылку Inputs (рис. 13.5, см. выше), а затем на ссылку + Add stream input и выбрать Event Hub (рис. 13.6).

Input alias — имя, которое будет использовано для доступа к входному потоку в SQL запросе. Это очень важный параметр.

Основные параметры входного источника Event Hub выберите из доступных для вашей подписки (Select Event Hub from your subscription). Кроме того, выберите пространство имен (Event Hub namespace), которое соответствует главному концентратору, и имя политики ключей (Event Hub policy name). В данном примере параметр Event Hub consumer group оставлен пустым, и потому используется группа по умолчанию $Default. Формат сообщения — JSON. Успешно сконфигурированный и добавленный вход показан на рис. 13.7.

Теперь настроим выход. Как уже указывалось в общем описании примера, главное хранилище игровых сообщений и инициатор связанных с этим событий (я напоминаю: в основе примера лежит архитектура Event Driven) — сервис CosmosDB. Для перехода к странице настройки необходимо выбрать вкладку Outputs (рис. 13.8).

В этой форме настраиваются следующие параметры: Output Alias (`MainEventStorage`) — имя выхода, которое будет использоваться в коде запроса. Мы выбираем опцию Select Cosmos DB from your subscription, а затем нужный нам аккаунт (maingamestorage), базу данных в пределах этого аккаунта (Database — Use existing — GameStorage) и имя коллекции в этой базе данных (`GameEvents`). Добавленный и настроенный выход показан на рис. 13.9.

Теперь посмотрим, как настроить собственно фильтрующий запрос. Для этого нажмите ссылку Query и подкорректируйте запрос следующим образом (рис. 13.10).

Подобный запрос означает, что все поля (символ *) всех сообщений будут перенаправлены из входного потока `MainGameProcessor` в выходной `MainEventStorage`. То есть все сообщения со входа передаются на выход.

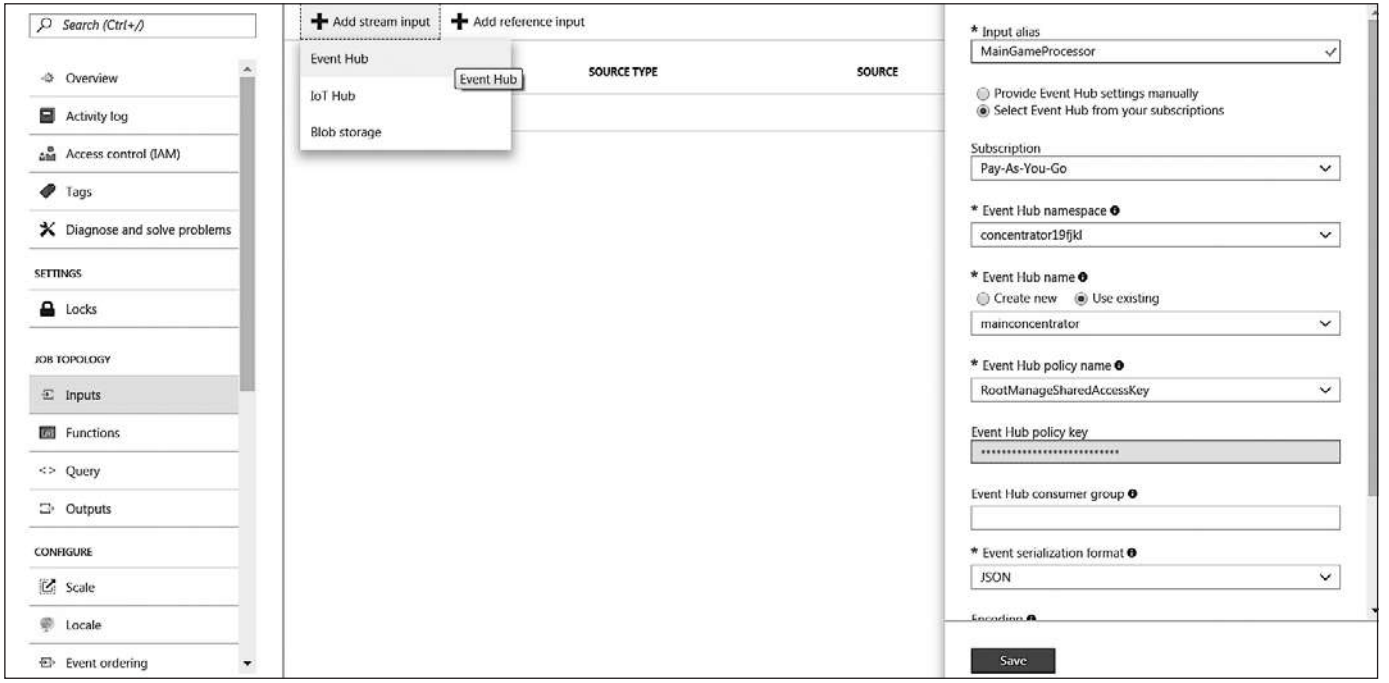


Рис. 13.6. Выбор Event Hub как источника для Stream Analytics Jobs

Рис. 13.7. Успешно сконфигурированный и добавленный вход

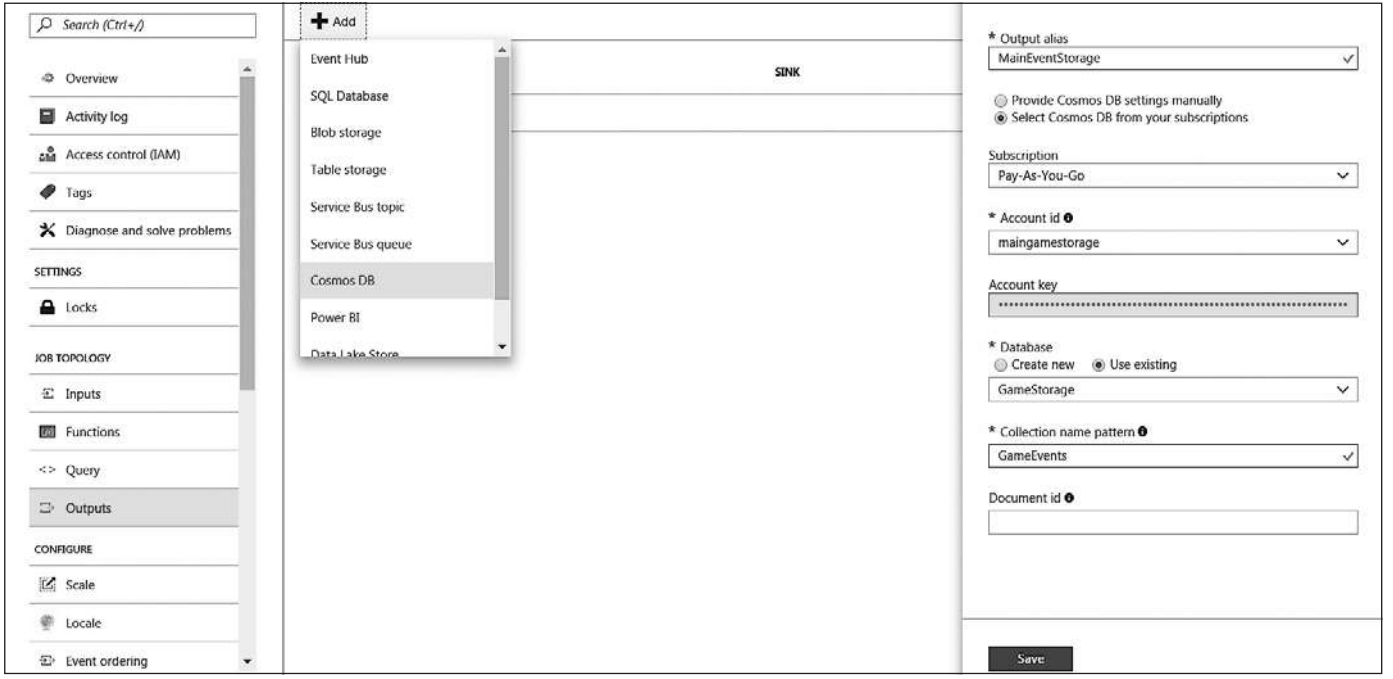


Рис. 13.8. Настройка выхода

Рис. 13.9. Добавленный и настроенный выход

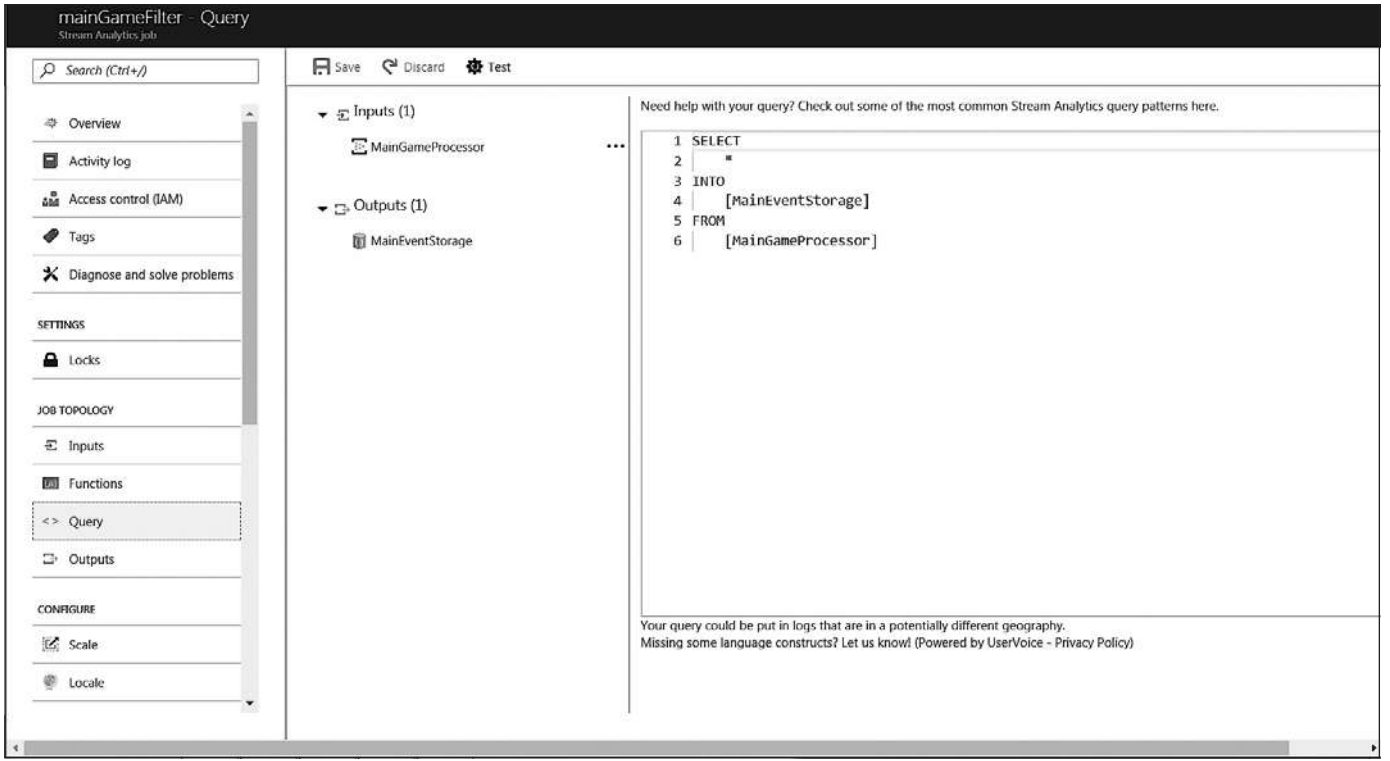


Рис. 13.10. Настройка Stream Analytics Job для непосредственного направления входного потока на выход

Рассмотрим подробнее возможности языка запросов Azure Stream Analytics.

Прежде всего, можно подключиться и выполнять запросы ко многим источникам данных. При этом источники могут быть применены в запросах в виде соединений (JOIN) и объединены (UNION). Вообще язык запросов Azure Stream Analytics поддерживает большинство стандартных операторов языка SQL, что, безусловно, является большим преимуществом такого сервиса и делает его использование удобным. Более того, поддерживается оператор `CREATE TABLE`, который применяется для указания схемы источника или приемника данных.

Помимо стандартных аналитических операторов SQL, язык Azure Stream Analytics поддерживает широкий набор встроенных функций.

- *Агрегирующие функции* (aggregate functions) выполняют агрегацию набора данных — вычисление среднего значения (`AVG`), подсчет количества (`COUNT`), вычисление максимального (`MAX`) и минимального (`MIN`) значений.
- *Аналитические функции* (analytic function) являются специфическими для Azure Stream Analytics и обрабатывают именно сообщения, в частности вычисляют аномалии в потоках данных.
- *Функции работы с массивами* (array functions) обрабатывают входные данные типа массива.
- *Функции преобразования типов* (cast function) преобразуют типы данных (`CAST`, `TRY_CAST`) или возвращают их тип (`GetType`).
- *Функции, работающие с временем и датой* (date and time functions) оперируют данными временного формата, в частности вычисляют разность между временны́ми моментами (`DATEDIFF`), получающие значение дня (`DAY`), месяца (`MONTH`), года (`YEAR`) и пр.
- *Функции, работающие с геопространственными данными* (geospatial functions), облегчают анализ потоковых геопространственных данных.
- *Математические функции* (mathematical functions) выполняют стандартные математические операции (`SIN`, `ABS` и др.) над входными элементами данных.
- *Строковые функции* (string functions) работают со строковыми данными.
- *Специальные функции* — группы функций, работающих с различными специальными аспектами сервиса Azure Stream Analytics, например с метаданными источника.

Очень мощный механизм расширения языка Azure Stream Analytics — *определяемые пользователем функции* (user defined functions, UDF). Они создаются на языке JavaScript, и из них напрямую доступны как стандартные функции языка JavaScript, так и встроенные функции Azure Stream Analytics. Чтобы создать функцию, определяемую пользователем, нужно перейти на вкладку `Functions` сервиса Azure Stream Analytics, в результате будет доступна следующая панель (рис. 13.11).

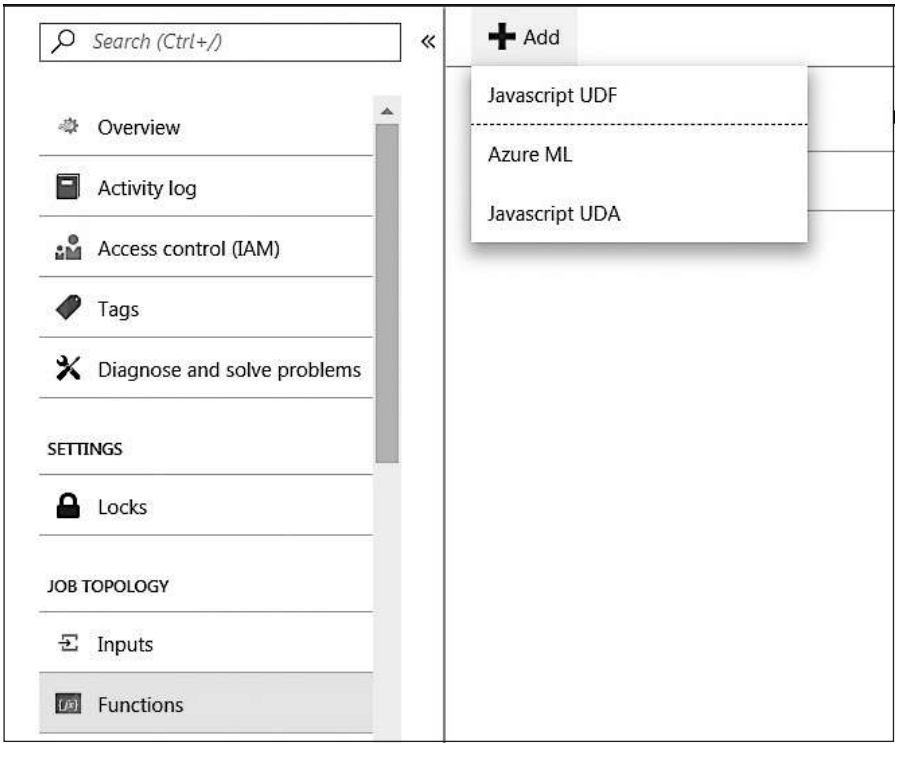


Рис. 13.11. Панель выбора добавляемой функции, определяемой пользователем

Рассмотрим возможные типы функций.

Javascript UDF — определяемая пользователем функция, доступная из тела основного запроса по имени `UDF.functionFame(arguments)` и служащая для «сквозного» выполнения, то есть она не содержит в своем составе переменных-аккумуляторов (счетчиков). Создание этой функции предствлено на рис. 13.12.

Особенностями UDF по сравнению с традиционной функцией на языке JavaScript является то, что она имеет название `main`, которое указывается в коде. Следует также указать тип выходного значения, выбрав из доступных в списке Output Type. Имя функции указывается в поле Function alias. Существенно упрощает процесс написания JavaScript-кода функция подсказки (автозаполнения), доступная в онлайн-редакторе.

Javascript UDA — определяемая пользователем функция (агрегатор), которая содержит переменные-аккумуляторы и позволяет агрегировать данные. Создание UDA представлено на рис. 13.13.

В отличие от UDF, структура определяемого пользователем агрегатора более сложна и включает встроенные методы, запускаемые при инициализации (`init`), накоплении (`accumulate`) и вычислении результатов (`computeResult`).

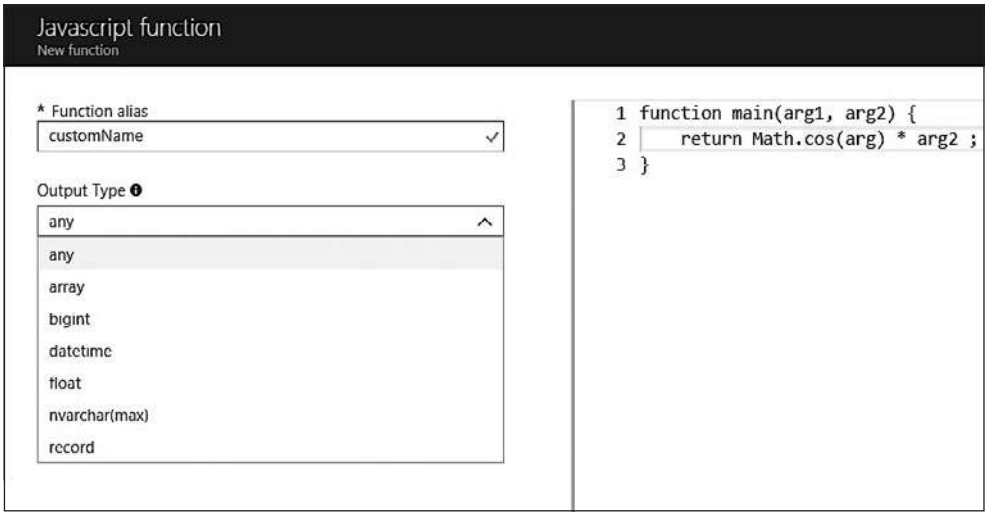


Рис. 13.12. Создание UDF

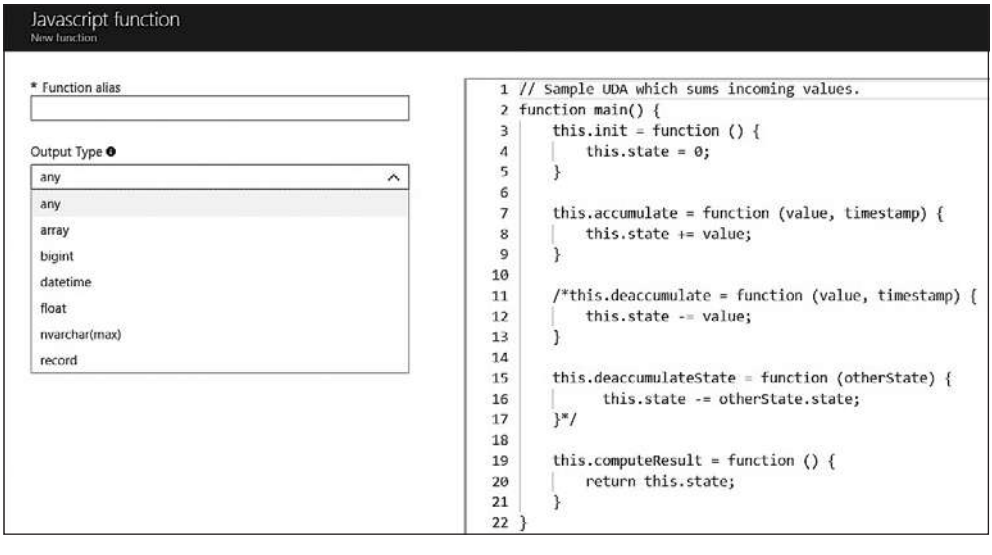


Рис. 13.13. Создание UDA

И наконец, третий доступный тип — интеграция с сервисом Azure ML. Дело в том, что этот сервис представляет собой облачный сервис создания моделей, полученных в результате применения алгоритмов машинного обучения и развертывания этих моделей в качестве REST API, доступных извне по адресу конечной точки HTTPS. Таким образом, сначала необходимо получить («натренировать»)

модель с помощью сервиса Azure ML, затем, после ее размещения в качестве REST-сервиса, получить ее адрес и использовать его для интеграции с Azure Stream Analytics. Описанные действия позволяют применять сложную модель обработки данных, в том числе на основе нейронных сетей, для потокового анализа данных. Это очень перспективное направление, позволяющее создавать, например, системы онлайн-анализа банковских транзакций, которые выделяют паттерны, являющиеся мошенническими и подозрительными.

Помимо интеграции с сервисом Azure ML, возможна интеграция со сторонним сервисом, предоставляемым по REST API.

Таким образом, Azure Stream Analytics представляет собой облачный сервис, который позволяет анализировать потоковые данные с помощью синтаксиса SQL и обладает возможностью интегрирования с сервисом машинного обучения Azure ML.

13.3. Amazon Kinesis Analytics

Один из компонентов группы сервисов AWS Kinesis — сервис потокового анализа данных AWS Kinesis Analytics. Он легко интегрируется с сервисами AWS Kinesis Data Streams и AWS Kinesis Firehouse в качестве источников сообщений, а также AWS S3, Redshift, Elasticsearch, AWS Lambda или AWS Data Stream в качестве приемников сообщений (рис. 13.14).

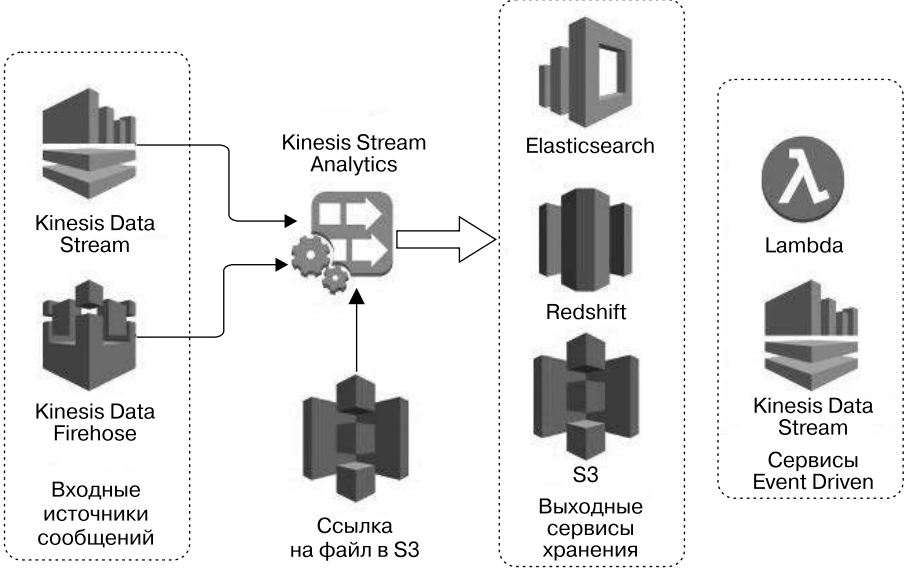


Рис. 13.14. Архитектура связей сервиса AWS Kinesis Stream Analytics

Рассмотрим, как работает этот сервис. Главный ресурс сервиса Amazon Kinesis Stream Analytics — *приложение* (application), которое создается в рамках группы сервисов Kinesis в аккаунте Amazon. Приложения можно создать и управлять ими с помощью веб-консоли Amazon или через SDK/AWS CLI. Приложения непрерывно читают сообщения из входных источников, обрабатывают их, применяя запросы на языке SQL, и помещают данные в выходной поток. Каждое приложение включает в себя следующие атрибуты: имя, описание, версию и статус. Версия назначается автоматически при создании приложения и обновляется при обновлении кода приложения.

Каждое приложение содержит обязательные компоненты (описаны ниже).

- *Вход* (input) — источник сообщений для приложения. В качестве такового выступает Kinesis Firehouse или Kinesis Data Stream. В конфигурации входа необходимо связать (map) входной источник с входом, который будет доступен по имени в коде приложения в качестве источника данных для выполняемого запроса SQL. Иначе говоря, вход в данном случае можно представить абстрактно как непрерывно обновляемую таблицу, к которой применяется, также непрерывно, запрос SQL. При этом возможно разделение (partitioning) потока от входного источника с большой интенсивности на несколько входов. Для каждого входа Amazon Kinesis Data Analytics добавляет дополнительные колонки временных меток — `Timestamp` и `ROWTIME`, которые отражают различные временные моменты попадания сообщения в Kinesis Data Analytics, на связанный именованный вход и др. Эти параметры важны для построения запросов с временно́й агрегацией. Дополнительно можно создать ссылку на источник в виде файла, расположенного в S3, который читается и обрабатывается при запуске приложения.
- *Код приложения* (application code) — набор операторов на языке SQL, преобразующих поток входных сообщений в выходной. Запрос может работать с входными потоками или связанными с файлами таблицами в качестве источников данных. Все эти источники логически представлены таблицами, которые могут участвовать в том числе в операциях соединения (JOIN) и комбинироваться в любых сочетаниях. В простейшем случае запрос читает сообщения из входного потока и помещает их на выход. В более сложных случаях запросы могут разделять сообщения в разные выходные потоки, в зависимости от значения того или иного признака, или объединять потоки нескольких входных источников в один выходной.
- *Выходы* (outputs) — SQL-запрос, представляющий код приложения, помещает результат выполнения в один или несколько выходных потоков, каждый из которых должен быть связан с конкретным сервисом AWS, который будет приемником сообщений. Это могут быть как сервисы хранения (AWS Redshift, S3-бакет или кластер Elasticsearch), так и сервисы дальнейшей обработки

сообщений (Amazon Kinesis Data Stream и Amazon Lambda). Интеграция с другими источниками возможна с помощью кода, размещаемого в Lambda. В то же время наличие в качестве выходного и входного источника Kinesis Data Stream позволяет строить иерархические, многостадийные системы обработки потоковых данных. Возможно также создание выхода, к которому будет послано сообщение при возникновении ошибок обработок сообщений приложением. При этом Amazon Kinesis Analytics обеспечивает гарантированную доставку и запись сообщения в источник, модель доставки будет «как минимум одно».

Рассмотрим подробнее концепции и особенности синтаксиса SQL-кода приложения. В основу синтаксиса положен стандарт SQL 2008. Для создания входа приложения требуется связать входной источник с входным потоком в приложении. Кроме того, можно создать специальные потоки в самом приложении для сохранения промежуточных результатов. Создание таких промежуточных потоков состоит из двух этапов: сначала создается пустой поток, потом в него «закачиваются» (pump) данные с помощью оператора `PUMP`.

Ключевой особенностью языка Amazon Kinesis Analytics SQL является то, что операторы группировки присутствуют в виде *оконных операторов* (windowed queries). Они бывают двух типов: *время-ориентированные* (time based) и *строчно-ориентированные* (row based). Суть оконных операторов заключается в том, что весь поток разбивается на отдельные множества, отделяемые от других по временно́му признаку или признаку количества сообщений. Например, для время-ориентированных оконных функций задается интервал и в его рамках обрабатывается подмножество сообщений. Строчно-ориентированные оконные функции группируют строки по заданному количеству строк. Можно применять операторы временного окна типа *«скользящее окно»*, непрерывно перемещающееся по потоку сообщений окно фиксированного размера, а также *«барабанное окно»* (thumbling window), когда происходит группировка нефиксированным окном, при котором сообщения разделяются в непересекающиеся группы с помощью `GROUP BY`.

Посмотрим, как работать с сервисами Kinesis Stream Analytics. Стартовая страница создания сервиса Amazon Kinesis Analytics выглядит так (рис. 13.15).

Чтобы создать приложение, следует нажать кнопку Create application (Создать приложение), после чего откроется страница настройки приложения (рис. 13.16).

Далее откроется страница с шагами конфигурирования входов, кода приложения и выходов (рис. 13.17).

Сначала следует подключить входной источник, для чего нужно нажать на ссылку Connect to a source. В результате откроется страница конфигурирования входных источников сервиса, верхняя и нижняя части которой показаны на рис. 13.18 и 13.19 соответственно. В качестве входного источника следует выбрать экземпляр Kinesis Data Stream. Для наполнения концентратора тестовыми данными можно использовать bash-скрипт, описанный в главе 10, посвященной доставке в облако потоковых данных.

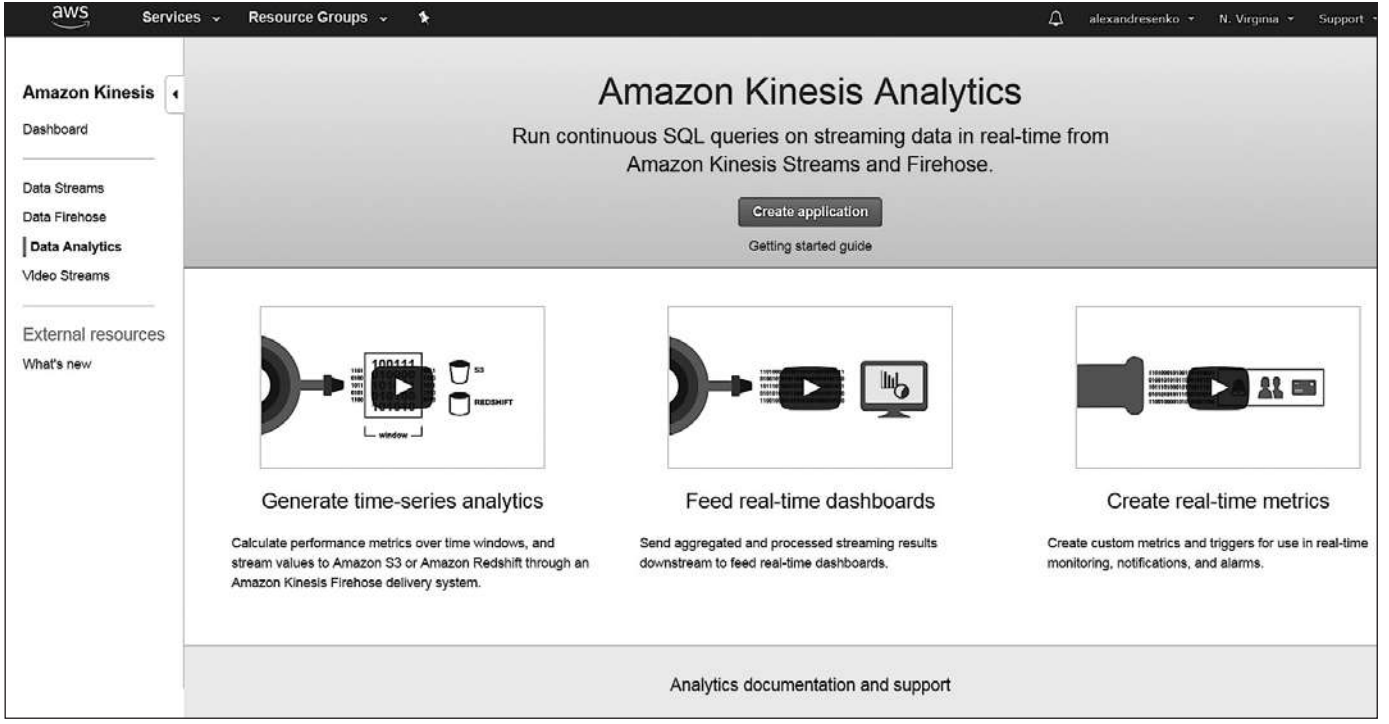


Рис. 13.15. Стартовая страница создания сервиса Amazon Kinesis Analytics

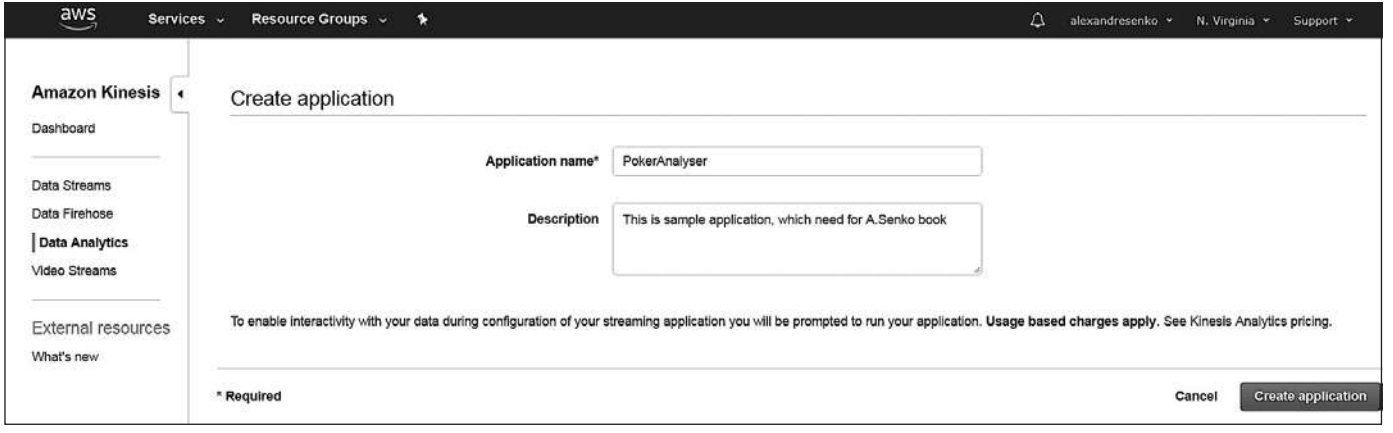


Рис. 13.16. Страница конфигурирования приложения

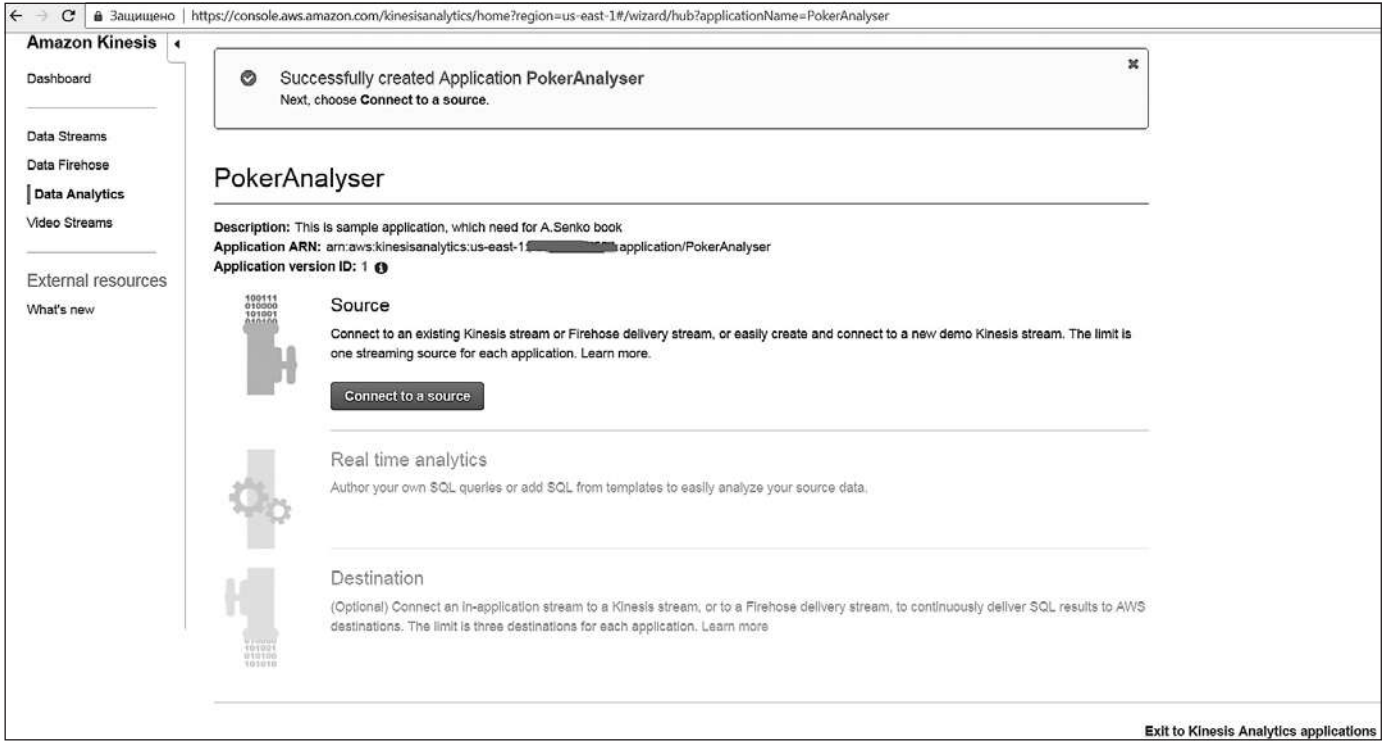


Рис. 13.17. Страница конфигурирования приложения

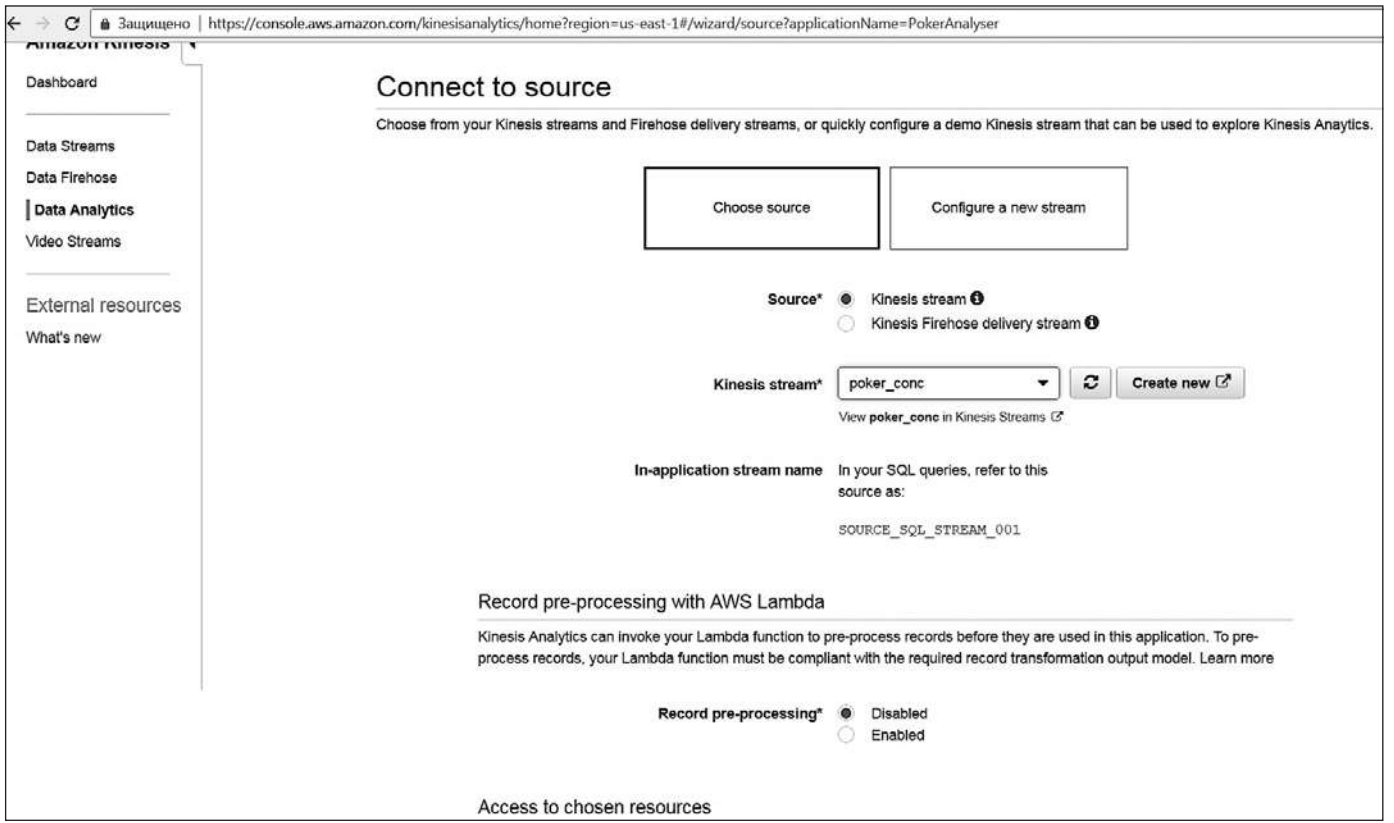


Рис. 13.18. Страница конфигурирования источника данных (верхняя половина)

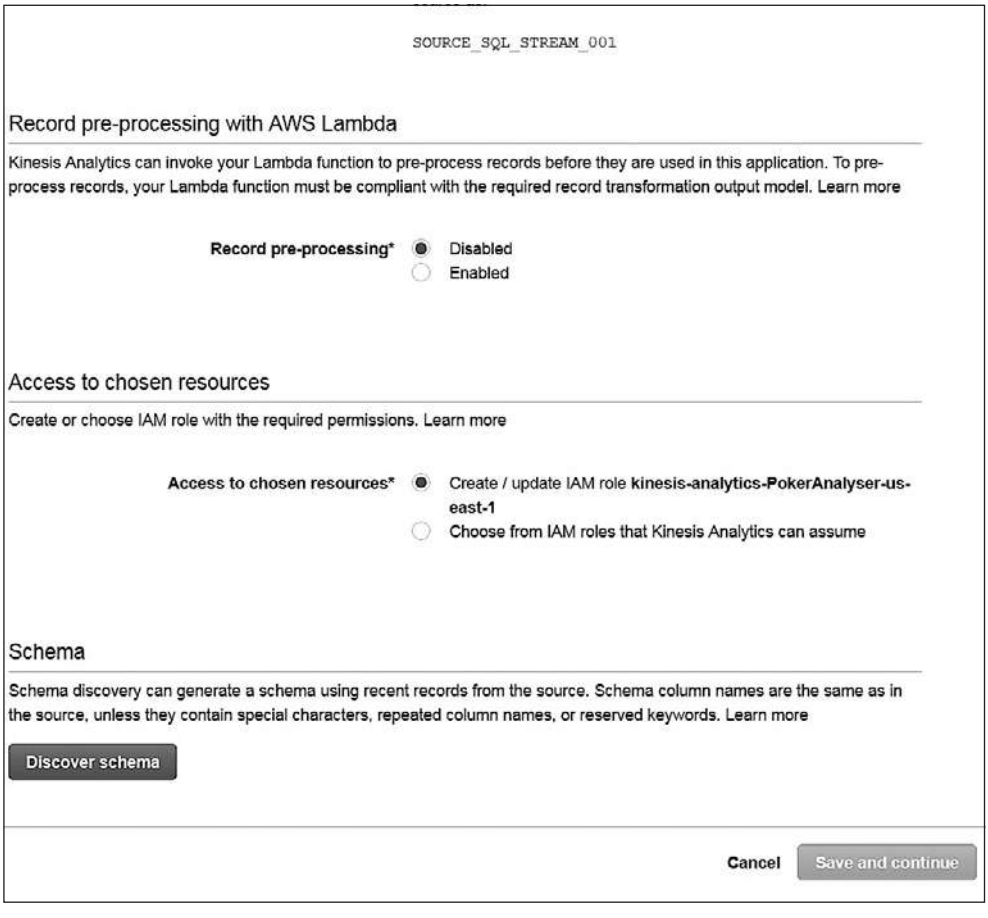


Рис. 13.19. Страница конфигурирования источника данных (нижняя половина)

В коде приложения входной источник будет иметь имя `SOURCE_SQL_STREAM_001`. Поскольку нам не надо предварительно обрабатывать сообщения с помощью AWS Lambda, то выберем опцию Disabled свойства Record pre-processing.

Далее следует нажать кнопку Discover schema, чтобы определить схему сообщений, поступающих на вход сервиса Kinesis Data Stream, подключенного к Kinesis Stream Analytics. Для этого сценарий, посылающий тестовые данные в Kinesis Data Stream, должен работать.

В случае успешного распознавания схемы можно наблюдать следующую вкладку (рис. 13.20).

Нажав на ссылку Save and continue, вы автоматически перейдете на страницу конфигурирования приложения.

Затем следует перейти на страницу редактора кода, нажав на ссылку Go to SQL editor, показанную на рис. 13.21. На рисунке показан сценарий простого сквозного фильтра, выбирающего поля PlayerId, TableId и Data из входного потока. Создается промежуточный «насос» STREAM_PUMP, который «накачивает» выходной поток под именем DESTINATION_SQL_STREAM.

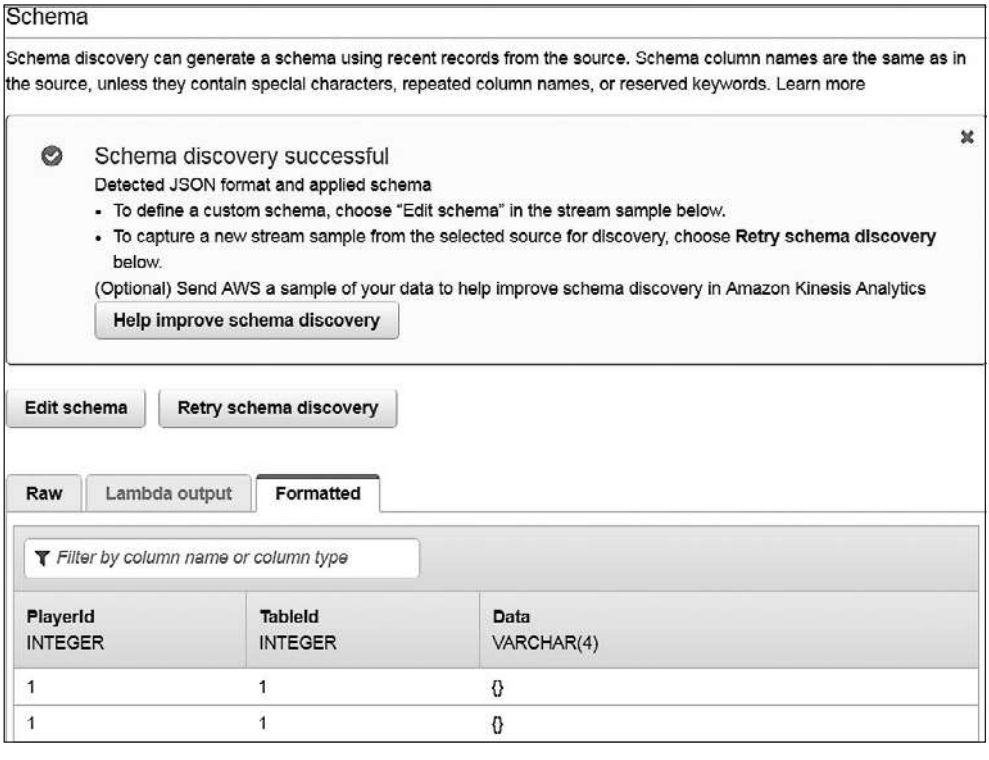


Рис. 13.20. Распознанная схема сообщения

Данные из источника отображаются на вкладке Source data (см. рис. 13.21), а отфильтрованные данные будут помещены в выходной поток DESTINATION_SQL_STREAM (рис. 13.23).

Теперь подключим к выходному источнику другой концентратор сообщений, для чего необходимо перейти на вкладку Destination и нажать на ссылку Connect to destination. В результате откроется вкладка конфигурирования выходного источника, показанная на рис. 13.24. Здесь следует выбрать сервис, в который будем посылать данные. В нашем случае это второй экземпляр Amazon Kinesis Data Stream, в который данные передаются в формате JSON (возможен еще CSV) из потока DESTINATION_SQL_STREAM.

В результате приложение будет полностью сконфигурировано (рис. 13.25).

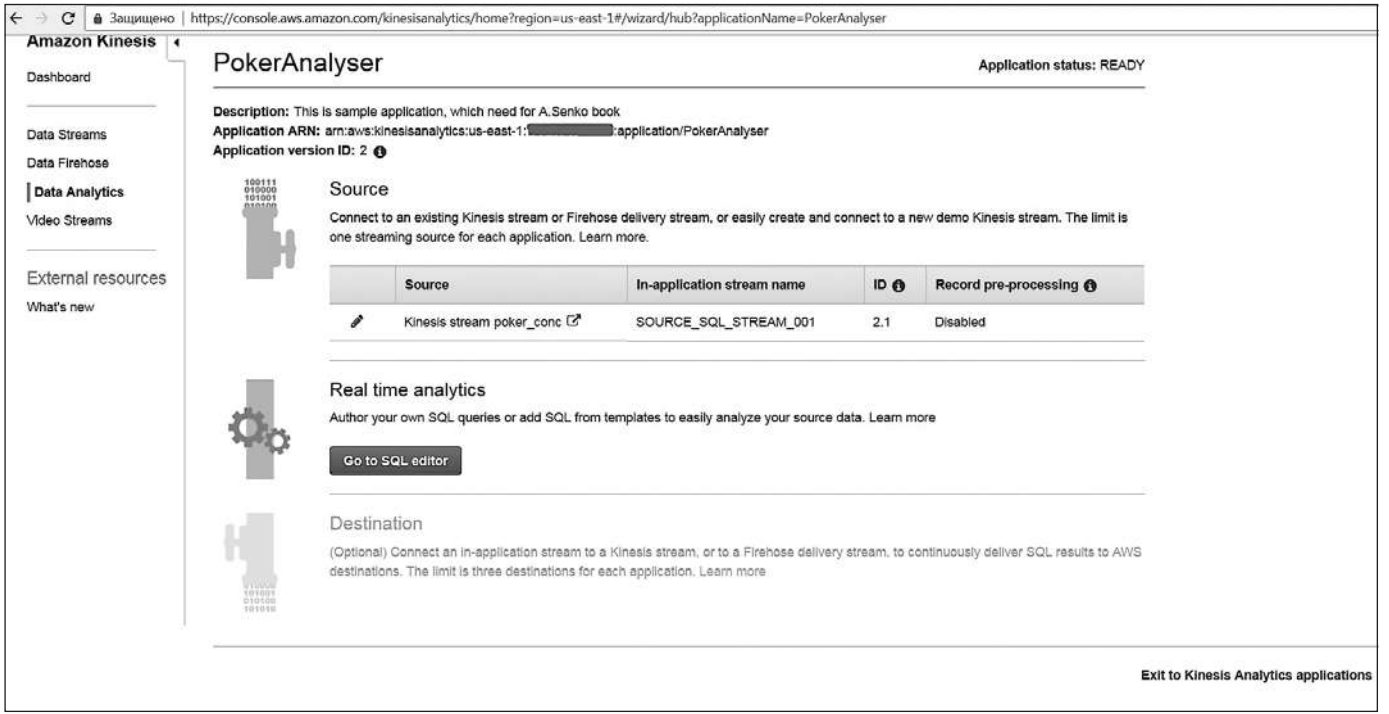


Рис. 13.21. Страница конфигурирования приложения с добавленным входным источником

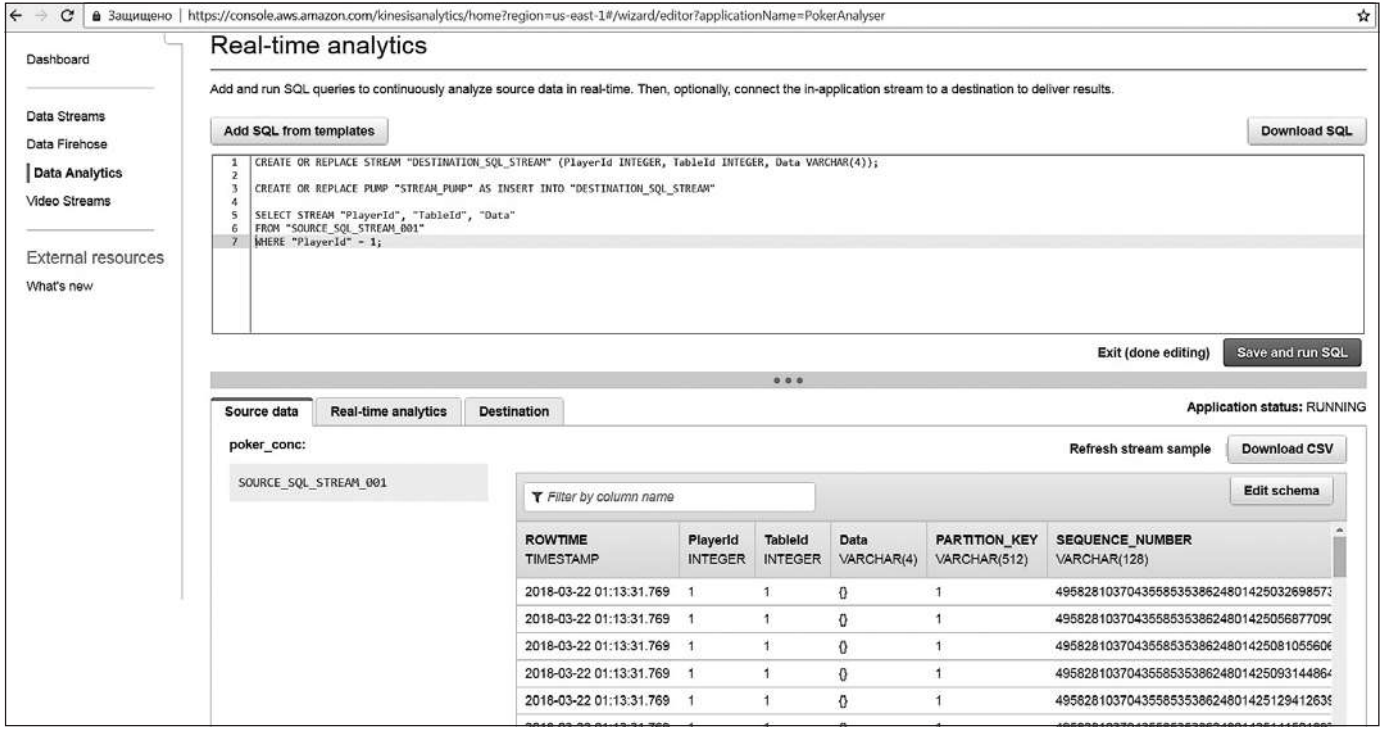


Рис. 13.22. Страница редактора кода приложения с примером простого фильтра

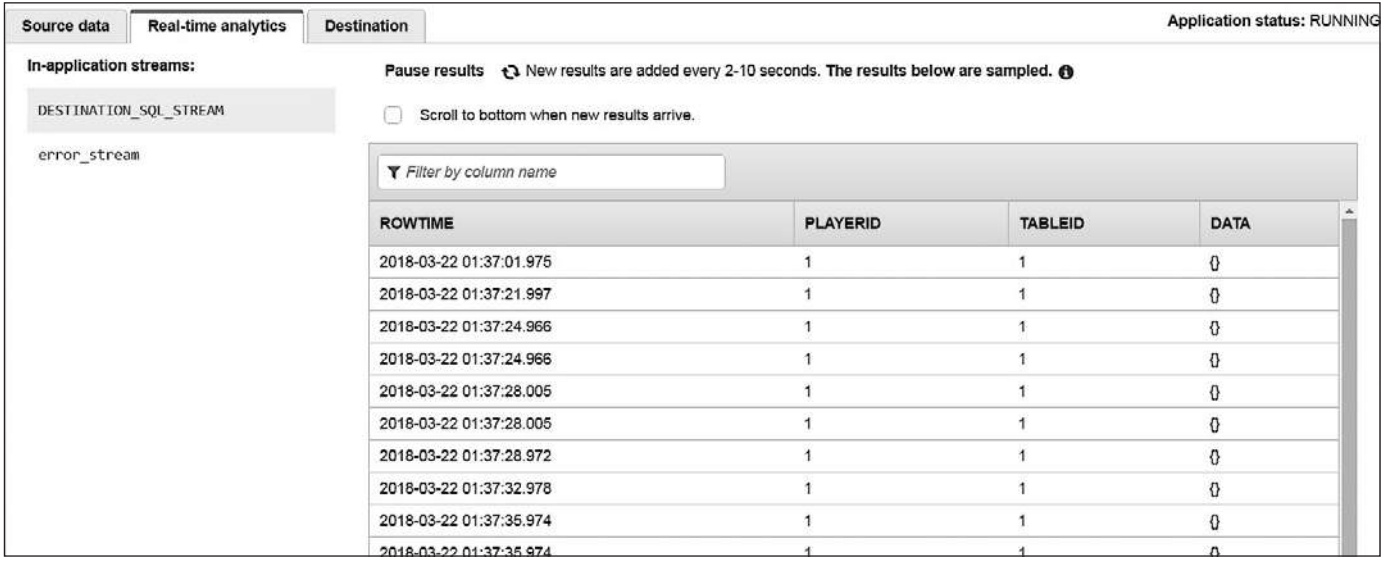


Рис. 13.23. Отфильтрованные данные, направленные в выходной поток DESTINATION_SQL_STREAM

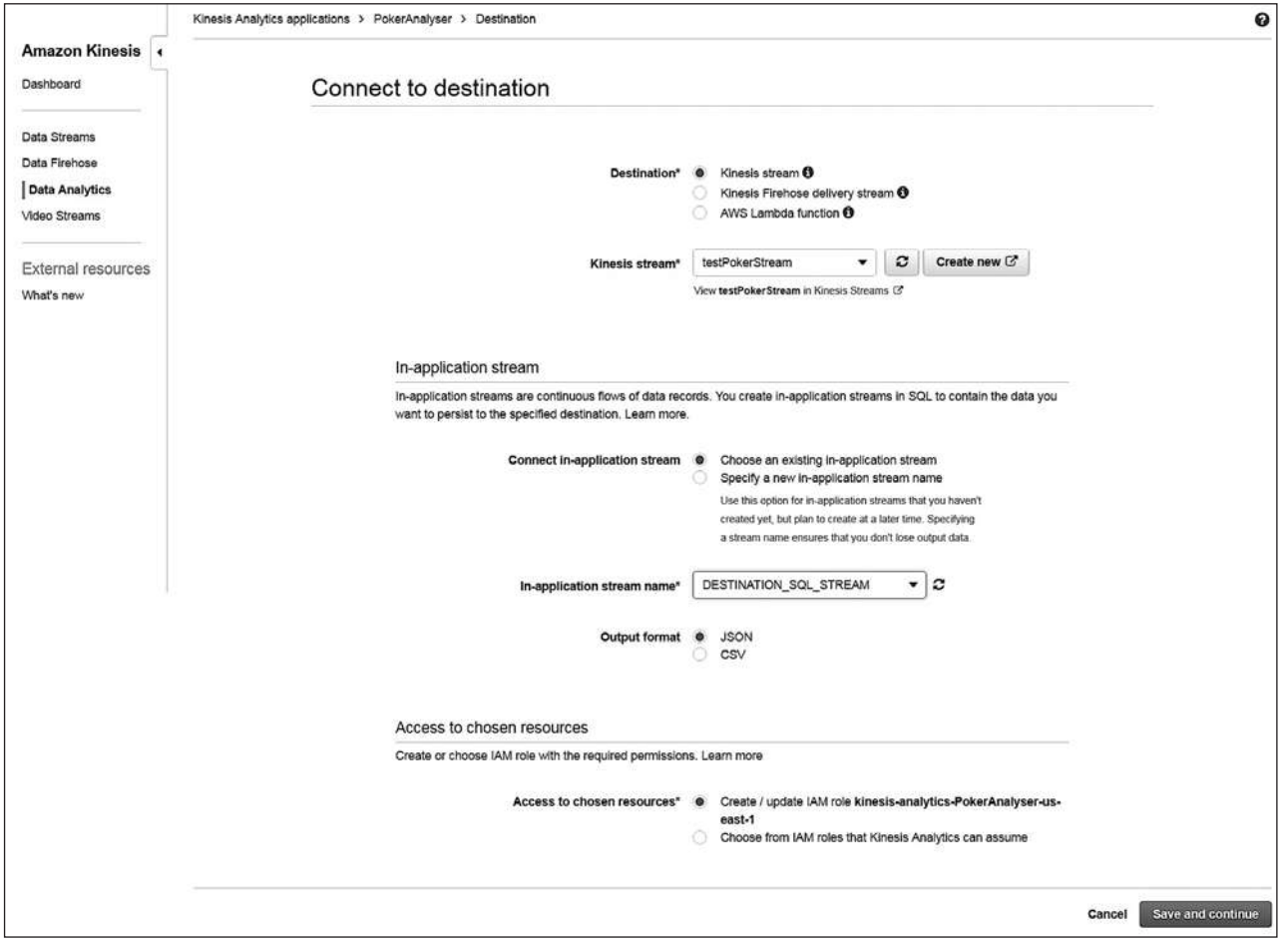


Рис. 13.24. Страница подключения выходного источника

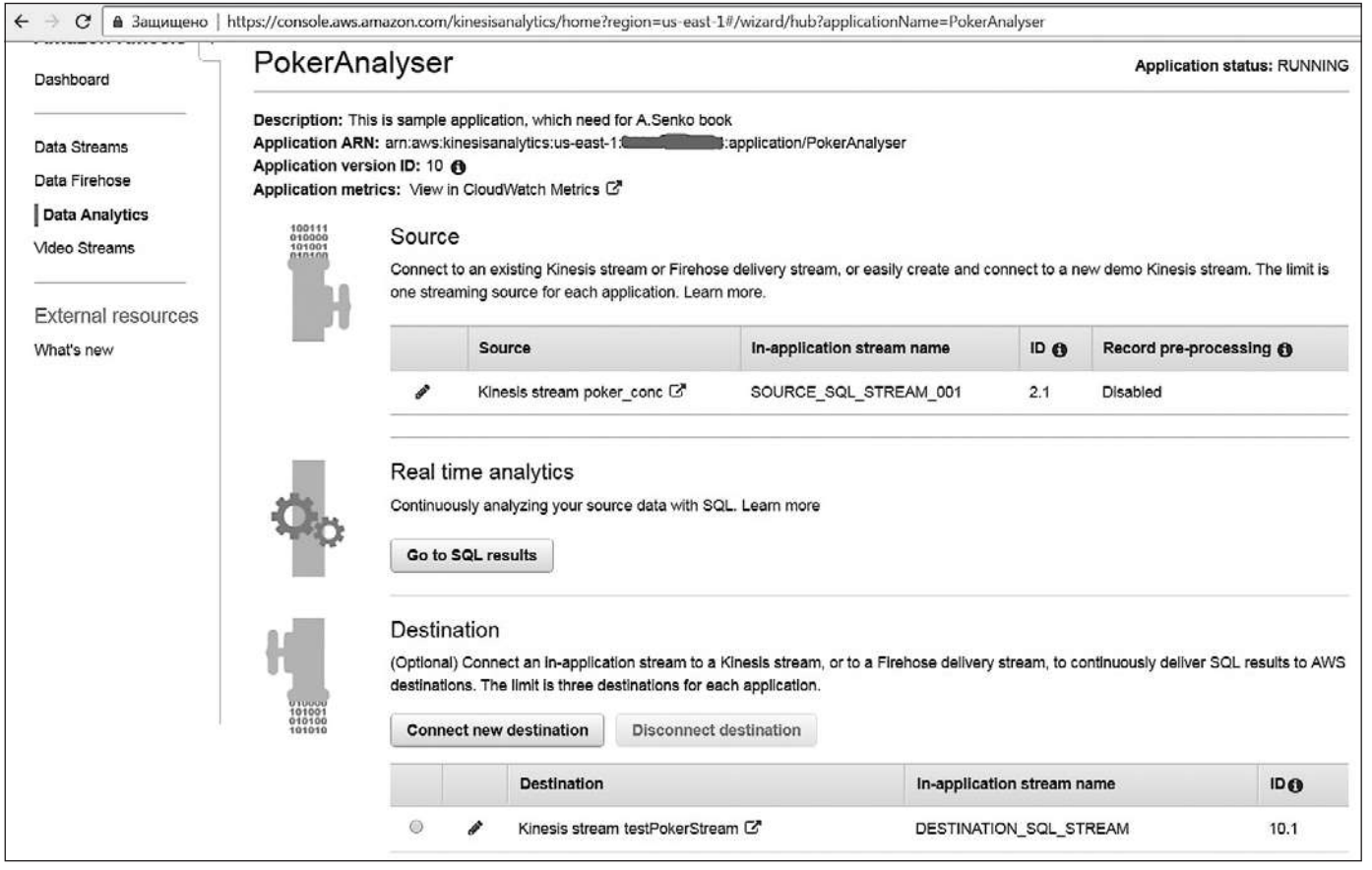


Рис. 13.25. Страница полностью сконфигурированного приложения Kinesis Stream Analytics

13.4. Apache Storm

Apache Storm представляет собой программный продукт для анализа потоковых данных. Storm в облачных средах размещается в управляемых кластерах (на момент написания книги — только в HDInsight). Этот фреймворк написан на Closure и изначально разработан для построения отказоустойчивых, масштабируемых систем анализа потоковых данных, обладающих высоким быстродействием.

Концептуально обработка сообщений происходит с помощью так называемых *топологий* (topologies) — непрерывно выполняемых заданий обработки, логически представляемых в виде графов. Топологии состоят из соединенных *струй* (spouts) и *засовов* (bolts), которые соединены с *потоками* (streams). Потоки — это ключевая абстракция Storm, являющаяся последовательностью *кортежей* (tuples) — объектов, представляющих сообщения. Все кортежи в потоке обрабатываются параллельно распределенным образом. Для каждого потока определена схема, так называемые имена и типы полей кортежей.

Струи — источники для потоков в топологиях. В общем случае задача струй — получить сообщения из внешних источников (например, из Kafka), представить их в виде кортежей и передать вдоль топологии. Струи бывают *надежными* (reliable) и *ненадежными* (unreliable). Надежные позволяют повторять отправку сообщений в топологии в случае возникновения ошибок обработки. Ненадежные не обладают такой возможностью. Каждая струя может наполнять один или несколько потоков.

Засов выполняет основную обработку потоковых данных. Может фильтровать, агрегировать, объединять и пр., а также выполнять простые трансформации потоков. Для сложных операций документация Storm рекомендует разбивать обработку на несколько стадий и комбинировать ее в несколько последовательно применяемых засовов. Каждый засов может наполнять один или несколько потоков.

Каждая струя и засов выполняются в виде заданий, реплицированных среди узлов кластера (рис. 13.26).

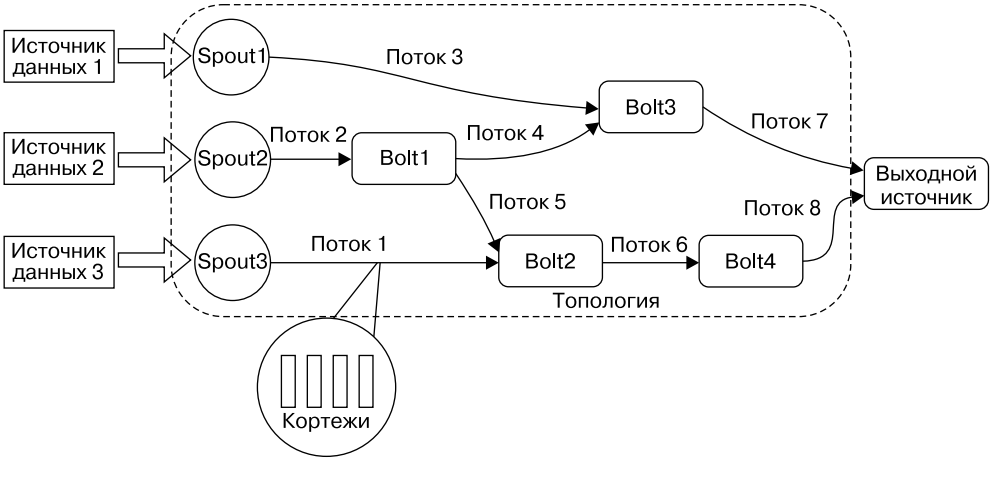


Рис. 13.26. Внутренняя архитектура Storm

Кластер Storm состоит из *рабочих узлов* (worker nodes), в которых расположен Supervisor (управляемый из головного узла компонент), и головного узла, в котором расположены сервисы Zookeeper и Nimbus. Последний отвечает за распределенное развертывание топологий, представимых в виде исполняемых пакетов в рабочие узлы кластера, контролируя их выполнение через супервизор.

Все топологии определяются в языках C#, Java или Python с помощью программной библиотеки Apache Storm. Затем они компилируются в программные продукты и развертываются в кластер с применением Nimbus.

13.5. Резюме

Рассмотренные облачные сервисы потокового анализа концептуально схожи между собой. Входные источники представляются в виде таблицы, непрерывно обновляемой входными данными, к которой применяется, также непрерывно, запрос на языке SQL, преобразующий входные потоки в выходные. Оба сервиса анализа потоковых данных от Azure и AWS позволяют применять SQL-подобный синтаксис для построения алгоритма анализа. Кроме того, оба сервиса доставляют сообщения, обработанные непрерывно выполняемым запросом, в различные назначения. В качестве выхода выступают как хранилища (в том числе файловые, реляционные и др.), сервисы обработки сообщений (типа бессерверных лямбда-функций) и другие концентраторы сообщений.

Сервис Spark Streaming требует использования программ на Java, Scala или Python, которые задействуют программные интерфейсы Spark, но концептуальная модель та же, что и в случае сервисов PaaS. А вот Storm предлагает принципиально другую модель онлайн-анализа данных, в котором ключевой является топология в виде графа.

Вы уже, вероятно, заметили, что с точки зрения удобства и простоты использования PaaS-сервисы вне конкуренции. Более того, в ряде случаев допускается их расширение сторонними, весьма сложными функциями (как в случае Azure Stream Analytics) благодаря интеграции с сервисами машинного обучения Azure ML.

14 Пакетный анализ данных

— А-а! Вы историк? — с большим облегчением и уважением спросил Берлиоз.

— Я — историк, — подтвердил ученый и добавил ни к селу ни к городу:

— Сегодня вечером на Патриарших прудах будет интересная история!

Михаил Булгаков. Мастер и Маргарита

14.1. Общие сведения о пакетном анализе данных

Итак, мы подобрались к последнему виду анализа больших данных. Он еще называется историческим, поскольку подразумевает анализ исторических данных, то есть данных, накопленных в течение некоторого времени работы информационной системы. Анализ исторических данных традиционно считается исторически (я умышленно применил эту тавтологию) первой областью обработки больших данных. Суть его состоит в том, что алгоритмы обработки применяются к данным, уже находящимся в хранилищах. Сами алгоритмы запускаются автоматически и не требуют интерактивного участия человека. Сервисы AWS Data Pipeline и Azure Data Factory позволяют запускать задания анализа данных в хранилищах, и поэтому, строго говоря, мы их уже рассматривали. В этой главе я приведу сведения, обойденные нашим вниманием в предыдущих главах.

Таким образом, типовая архитектура системы пакетного анализа данных включает в себя:

- хранилище данных, допускающее выполнение массивно-параллельных алгоритмов анализа (в качестве такового может выступать HDFS, DWH, реляци-

онная БД и пр.). Очевидно, что для больших массивов это хранилища на основе группы серверов — кластера;

- вычислительные средства, обеспечивающие распараллеливание аналитического запроса и сведение результатов его выполнения в единый вид опять-таки в виде кластера;
- алгоритм анализа данных, выполняемый в виде задания в кластере вычислительных средств и представленный как исполняемый модуль на компилируемом языке или сценарий (SQL, Python);
- сервис управления подсистемы хранения, вычисления и планирования выполнения задания.

Традиционное средство анализа исторических данных — Hadoop, являющийся родоначальником практически всех современных систем больших данных. Рассмотрим его более подробно.

14.2. Hadoop

Hadoop представляет собой фреймворк, который обеспечивает выполнение распределенных алгоритмов анализа данных, выполняемых на кластерах. Он включает в себя библиотеки, утилиты и другие программные средства, позволяющие создавать программы анализа данных с помощью языка Java, на котором, собственно, и написан этот фреймворк.

Hadoop — продукт OpenSource, то есть свободно и бесплатно распространяемое программное обеспечение, разработанное организацией Apache Software Foundation. Эта система лежит в основе поисковых движков многих крупных веб-сервисов, например Yahoo!, Facebook и пр. Она оказалась настолько удачной, что на базе ее самой или компонентов построена целая экосистема из программ, фреймворков и пр. Некоторые из программных продуктов этой экосистемы мы уже рассматривали: Apache Spark, Storm, Sqoop, Kafka и др. Рассмотрим теперь сам Hadoop. Он состоит из четырех главных компонентов:

- *Hadoop Common* (Hadoop Core) — набор связующих программ (утилит), обеспечивающих работу системы как единого целого;
- *HDFS* (Hadoop Distributed File System) — распределенная файловая система Hadoop;
- *YARN* (Yet Another Resource Negotiator) — система управления кластером и планирования выполнения заданий на нем;
- *Hadoop MapReduce* — паттерн параллельного распределенного анализа данных.

Итак, в Hadoop данные хранятся в HDFS — распределенной файловой системе, допускающей параллельное применение алгоритмов анализа. Архитектура этой

системы была рассмотрена в главе 8. Таким образом, данные нужно скопировать в Hadoop, прежде чем они будут проанализированы. Данные в HDFS хранятся в виде файлов, а сама процедура копирования может применяться без трансформации, то есть используется ELT.

YARN функционально состоит из двух компонентов: подсистем управления ресурсами и планирования выполнения вычислительных заданий. Архитектура YARN имеет следующий вид (рис. 14.1).

Менеджер ресурсов (resource manager) предназначен для централизованного выделения ресурсов выполняемым заданиям, то есть распределения их по кластерам. На рис. 14.1 показан случай выполнения нескольких экземпляров одного задания (обозначены как контейнеры MapReduce) в нескольких кластерах. Каждый кластер включает в себя *менеджер узла* (node manager), отвечающий за мониторинг ресурсов узла (использование CPU, памяти и пр.). Непосредственно за выполнением задания следит *монитор приложения* (application master), который получает сообщения о выполняемых заданиях и, принимая от менеджера узла метрики, отсылает в менеджер ресурсов статус выполняемых заданий.

Менеджер ресурсов состоит из двух компонентов: *планировщика заданий* (Scheduler) и *менеджера приложений* (application manager). Планировщик отвечает за выделение ресурсов узла для выполняемых экземпляров заданий при указанных ограничениях на ресурсы. При этом он не осуществляет мониторинг выполняемого задания. Кроме того, этот планировщик не отвечает за перезапуск задания при физическом отказе узла или программной ошибке. Задания планируются на основе требований к ресурсам со стороны выполняемых приложений, выполняющихся в виде абстрактных контейнеров приложений, которые потребляют строго обозначенное количество физических ресурсов узла кластера (CPU, памяти, полосы пропускания сети и пр.).

Менеджер приложений отвечает за прием заданий, переданных клиентом на кластер для выполнения, за «переговоры» (negotiations) о выделении ресурсов контейнерам приложений. Этот процесс представляет собой реализацию протокола выделения ресурсов и синхронизации множества выполняемых заданий и включает взаимодействие с планировщиком. Кроме того, менеджер приложений обеспечивает перезапуск контейнеров приложений в случае ошибки выполнения и отслеживает их статус и прогресс выполнения.

Кроме того, YARN включает множество других элементов, например компонент *федерации YARN* (YARN federation), который позволяет совместно использовать несколько узлов YARN в одном кластере, разбивая его на несколько независимых подкластеров или, наоборот, соединяя и синхронизируя несколько кластеров в один. Последнее позволяет выполнять очень большие (то есть ресурсоемкие) задания в кластере. Помимо этого, существует расширение, позволяющее применять Hadoop для анализа потоковых данных (hadoop streaming).

Непосредственно задания, которые выполняются в виде контейнеров приложений, представляют собой исполняемые программы на языке Java, созданные с помощью паттерна MapReduce. Он изначально был разработан компанией Google

Рис. 14.1. Архитектура подсистемы Hadoop — YARN

специально для выполнения вычислений над огромными объемами данных (петабайты) в кластерах из тысяч узлов. Каждое приложение, построенное с помощью этого паттерна, состоит из шагов двух типов.

- *Шаг Map* служит для разбиения набора данных на независимые подмножества, которые будут анализироваться в кластерах параллельно, и непосредственной обработки выбранного подмножества данных.
- *Шаг Reduce* обеспечивает «свертку», то есть комбинирование результатов выполнения параллельных шагов Map и выдачу результатов внешнему программному клиенту.

Паттерн MapReduce работает с данными в виде пар «ключ — значение», что требует или представления их в таком виде в файле, или преобразования из файла в этот тип. Как пример использования анализа исторических данных можно привести систему анализа активности конкретного пользователя покер-рума в течение нескольких лет. В качестве анализируемых величин будут выступать тип шага (колл, бет, олл-ин и пр.) и соответствующая ему величина ставки. Требуется подсчитать предпочтения пользователя, а конкретнее — указать, какую величину ставки он предпочитает (мы предполагаем, что в нашей системе возможны только дискретные величины ставок). На шаге Map весь объем данных разделяется на отдельные подмножества. В пределах каждого из них выполняется фильтрация по идентификатору пользователя и создается объект типа «ключ — значение», который в качестве ключа тип шага будет использовать связку `"тип_шага – величина_ставки"`, а значение будет равняться `"1"`. При этом если игрок делал подобную ставку несколько раз, то на шаге Map следует создать соответствующее количество пар `<"тип_шага – величина_ставки", "1">`. Далее эти пары нужно сгруппировать по ключу `"тип_шага – величина_ставки"` и направить их в алгоритм, соответствующий шагу Reduce, который подсчитывает количество пар, просто суммируя их и пересылая результат на выход.

Как уже отмечалось, выполняемые задания нужно писать на языке Java с использованием программных библиотек Hadoop. Это значительное ограничение системы, поскольку требует знание этого языка, кроме того, интерактивная отладка в данном случае существенно затруднена. Следующим ограничением системы является то, что задача должна быть представлена в формате MapReduce. В предыдущих главах достаточно подробно описывался SQL для анализа как реляционных, так и нереляционных данных и упоминались все сложности, связанные с распараллеливанием задачи. Предоставление удобного интерактивного интерфейса брал на себя соответствующий сервис. В случае Hadoop такое разбиение — это прямая обязанность пользователя. С одной стороны, это дает очень большую гибкость, а с другой — требует хорошего знания Java и высокой программистской квалификации.

Следующим недостатком «чистого» Hadoop является то, что при работе выполняемых заданий осуществляется много дисковых запросов, что происходит достаточно медленно и требует увеличения количества узлов кластера в целях

повышения производительности. Эту проблему решает Apache Spark, который оперирует данными, предварительно загруженными в оперативную память.

Несмотря на эти ограничения, Hadoop в настоящее время очень широко применяется в системах пакетной обработки больших данных. И поскольку этот фреймворк работает поверх кластера, то в облачных провайдерах AWS и Microsoft Azure он входит в состав сервисов управляемых кластеров — Azure HDInsight и AWS EMR.

14.3. Apache Pig

Платформа Apache Pig позволяет применять специальный интерпретируемый язык Pig Latin для обработки данных, хранящихся в Hadoop, а также Apache Tez и Spark. Синтаксически Pig близок к SQL, что делает его очень удобным для использования. Эта платформа изначально была разработана в Yahoo!. И в настоящее время распространяется как свободное ПО Apache Software Foundation и является частью экосистемы Hadoop.

Apache Pig преобразует операторы языка Pig Latin в задания MapReduce и выполняет их в кластере Hadoop. Это решает две проблемы «чистого» Hadoop: позволяет, во-первых, выполнять задания интерактивно (что значительно упрощает отладку), а во-вторых, автоматизировать процесс создания сценариев и использования кластера Hadoop для анализа данных с помощью алгоритмов, созданных другими сервисами. Мы уже рассматривали сервис AWS Data Pipeline, применяющий для копирования и трансформации данных кластер AWS EMR, в котором можно использовать Pig или Spark. В обоих случаях для упрощения трансформации данных используются сценарии, напрямую входящие в состав шага трансформации. Очевидно, что применять для этого компилируемые программы весьма непросто. Вместе с тем, в отличие от декларативного SQL, язык Pig Latin является процедурным. Кроме того, он не требует четкого задания схемы данных.

Итак, Pig работает поверх Hadoop с использованием всех его основных компонентов: YARN, HDFS и MapReduce. На рис. 14.2 показана архитектура Pig.

Прежде всего, сценарий Pig Latin можно выполнить в различных режимах: *интерактивном* в оболочке Grunt Shell (при этом возможен интерактивный режим взаимодействия пользователя с Pig через эту оболочку); в виде *определяемой пользователем функции* (UDF), написанной на языке Java во *«встроенном» режиме* (Embedded); и наконец, в *пакетном режиме* (Batch Mode), при котором запускается сценарий, представленный в виде файла `.pig`-формата. В любом случае конечным итогом преобразования сценария Pig является исполняемый файл MapReduce, который посылается в кластер Hadoop. Чтобы создать этот MapReduce, сценарий Pig Latin последовательно обрабатывается компонентами Apache Pig, описанными ниже.

- *Парсер* служит для проверки синтаксиса, соответствия типов данных и представляет выражения и логические операторы Pig Latin в виде направленного ациклического графа. В последнем эти логические операторы расположены

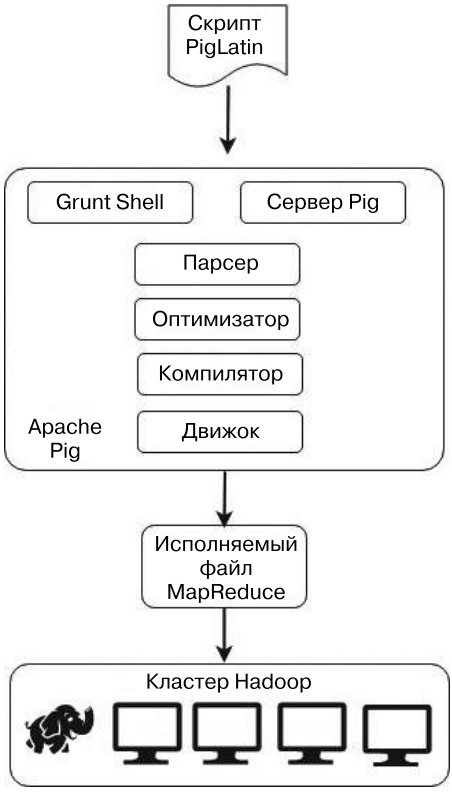


Рис. 14.2. Архитектура Apache Pig

в вершинах, а ребра представляют собой потоки операторов преобразования данных.

- *Оптимизатор* принимает на вход направленный ациклический граф потока обработки данных и преобразует его к графу с модифицированной топологией, который при исполнении потребляет меньше ресурсов, чем исходный.
- *Компилятор* создает файлы заданий MapReduce, компилируя граф потока исполнения данных в исполняемый файл на языке Java.
- *Движок* отвечает за взаимодействие с основным кластером Hadoop.

Для Apache Pig используются следующие модели данных.

- *Атомарный тип* (Atom) — одиночный тип данных, хранящий строковые, целочисленные или числовые величины с плавающей точкой, например `'32'`, `'Alexander'`.
- *Кортеж* (Tuple) — наборы данных в виде упорядоченного набора атомарных значений. Этот тип похож на строку в таблице реляционной базы данных, например `(Alexander, Bet, 30)`.

- *Вложенный тип, «сумка»* (Bag) — комплексный тип, состоящий из неупорядоченного набора кортежей, причем не обязательно одинаковой сигнатуры, например `{(Alexander, Bet, 30), (Dmity, All-in)}`. Существует также «внутренняя сумка» (inner Bag), содержащая некое подобие вложенной иерархической конструкции: `{Alexander, {Bet, 400}}`.
- *Карта* (Map) — набор «ключ — значение», имеющий вид `[Name#Alexander, Status#winner]`.

Вообще PigLatin — очень богатый язык, содержащий множество атомарных типов данных, а также большое количество операторов языка (арифметические, логические, операторы разделения, соединения, объединения, группировки и т. д.).

Подобно Hadoop, платформа Pig доступна в управляемых кластерах как Azure HDInsight, так и AWS EMR.

14.4. Apache Hive

Последний компонент экосистемы, который будет рассмотрен, — Apache Hive. Мы уже встречались с этой платформой. Ей следует уделить бóльшее внимание. Apache Hive представляет собой СУБД, построенную поверх Hadoop. Первоначальным разработчиком системы являлся Facebook, который затем передал его Apache Software Foundation. В официальной документации платформа описана как хранилище данных (DWH). Apache Hive позволяет работать с данными, хранящимися в HDFS или HBase, с помощью языка HiveQL, очень близкого к стандартному SQL. Более того, можно подключить сторонние средства BI, задействуя JDBC. При этом Apache Hive не является реляционной базой данных, не поддерживает онлайн-транзакций (OLTP), в том числе обновления отдельных строк таблиц.

Архитектура платформы представлена на рис. 14.3.

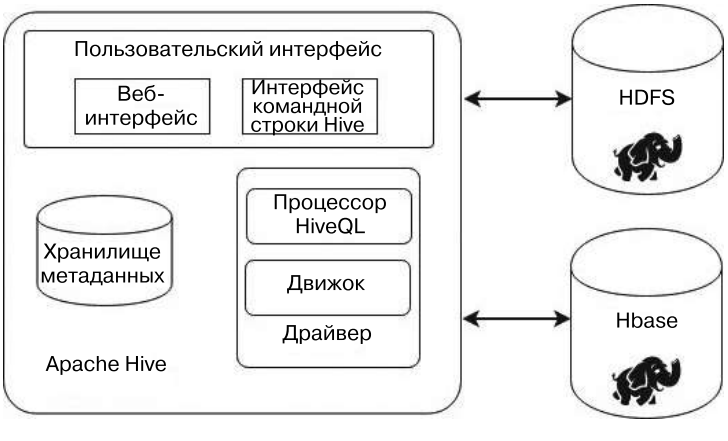


Рис. 14.3. Архитектура платформы Apache Hive

Помимо поддержки JDBC и REST API, у Apache Hive имеется интерфейс взаимодействия с пользователем, включающий веб-интерфейс и интерфейс командной строки. Поскольку использование стандартного языка SQL требует четкого указания схемы данных, представленной в виде таблицы, а данные в HDFS/HBase могут быть и неструктурированными, то необходимо хранить метаданные в специализированном *хранилище метаданных* (meta store). Еще раз подчеркнем, что оно не содержит данных, а лишь выполняет «маппинг» между таблицами, необходимыми для применения SQL и физическими файлами в HDFS. *Процессор HiveQL* (HiveQL Process Engine) и *движок* (Execution Engine) предназначены для создания исполняемых модулей MapReduce и их запуска.

Рассмотрим более подробно процесс взаимодействия Hadoop и Apache Hive.

1. На первом шаге сценарий на HiveQL с помощью веб-интерфейса, интерфейса командной строки или JDBC добавляется в Hive.
2. Драйвер, используя процессор HiveQL, выполняя анализ и парсинг сценария, строит план выполнения (execution plan). Для последнего необходимы также метаданные источников данных, которые берутся в хранилище.
3. Построенный план пересылается в движок, который преобразует его в исполняемый файл MapReduce.
4. Движок пересылает файл MapReduce в Hadoop. Он же следит за ходом выполнения его на кластере Hadoop.

Как уже подчеркивалось выше, HiveQL — язык, очень близкий к стандартному SQL. Он также содержит операторы DDL и DML. Apache Hive более удобна в использовании по сравнению с Hadoop и Pig, поскольку не требует изучения специального доменно-специфичного языка (Pig Latin) или языка программирования (Java). А это значит, что платформа Hive является универсальной, то есть может быть задействована как система пакетного анализа, нереляционное хранилище данных и часть процесса ETL. Apache Hive доступна в Azure HDInsight и AWS ERM. Кроме того, хранилище метаданных этой платформы лежит в основе хранилища метаданных AWS Athena и AWS Glue.

14.5. Резюме

В этой главе мы кратко рассмотрели архитектуры ряда платформ, которые используются для пакетного анализа. Все эти платформы представлены в облачных средах как части сервисов управляемых кластеров AWS EMR и Azure HDInsight. Кроме того, эти сервисы могут использоваться в самостоятельной установке на виртуальных машинах IaaS или в кластерах Docker.

Наиболее зрелая и традиционная платформа пакетного анализа данных — Hadoop, по сути являющаяся первой BigData-платформой. На ее основе путем устранения недостатков и введения улучшений было создано очень много сервисов и платформ, называемых экосистемой Apache Hadoop. Все они вместе и их отдель-

ные элементы составляют базис любой системы, работающей с BigData. Более того, в основе всех облачных сервисов анализа данных лежит та или иная платформа из этой экосистемы.

Однако, несмотря на зрелость, Apache Hadoop достаточно трудна в использовании, поскольку требует создания программ на языке программирования общего назначения Java. Для облегчения процедуры создания программ для Hadoop используются «надстройки» в виде Apache Hive, Apache Pig и др. Упрощение в обоих случаях достигается за счет того, что пользователю не нужно напрямую создавать исполняемые пакеты MapReduce, за него это будут делать внутренние компоненты этих платформ. Пользователю в данном случае следует лишь написать сценарий анализа данных на SQL-подобном языке (Pig Latin) или стандартном SQL и выполнить его на платформе.

Вы можете спросить: а как же отдельные облачные сервисы пакетного анализа? В настоящее время специализированных PaaS-сервисов пакетного анализа у AWS и Azure нет. Да и в таком сервисе нет надобности, поскольку имеются сервисы интерактивного анализа и сервисы периодического запуска заданий анализа. И это очень удобно, поскольку перед настройкой периодического выполнения нужно произвести отладку алгоритма в интерактивном режиме.

Заключение

Вот вы и дошли до конца книги, пройдя сквозь весь «зоопарк» сервисов, фреймворков и платформ обработки данных в облачных средах. Несмотря на все их разнообразие, качественно все указанные объекты делятся на следующие группы:

- хранение данных в реляционных и нереляционных базах данных и в виде файлов;
- загрузка данных в хранилища в виде сервисов доставки потоковых данных и сервисов трансформации и копирования данных;
- анализ данных: потоковый, интерактивный и пакетный.

В общем и целом систему анализа больших данных в облаках можно строить тремя способами:

- с помощью облачных PaaS-сервисов;
- используя сервисы из экосистемы Apache Hadoop;
- применяя гибридный подход.

Как показывает мой опыт, гибридный подход характерен тем, что привносит дополнительные сложности интеграции между «миром Hadoop» и «миром PaaS», а потому, вероятно, следует выбрать один из них.

Безусловно, к моменту выхода книги и позже облачные сервисы будут интенсивно развиваться, поменяется пользовательский интерфейс, появится новая функциональность и т. д. Чтобы не допустить тотального устаревания книги, я постарался уделить достаточно много внимания архитектуре систем, которые еще долго будут актуальными.

Ну и, безусловно, я описывал базовые концепции, подходы и методики, знание которых позволит вам самостоятельно освоить аналогичные сервисы, предоставляемые другими облачными средами.

Надеюсь, что вы испытали хотя бы часть того удовольствия, которое я испытывал при знакомстве и работе с описанными сервисами. Конечно, реальное их использование далеко не всегда удовольствие, иногда это и головная боль, нервы, часы отладок и сотни страниц прочитанных документов. И если после прочтения книги вы избавитесь хотя бы от части этой боли и построите свою систему анализа больших данных, то я буду считать поставленную мной задачу выполненной.

А. Сенько

Работа с BigData в облаках. Обработка и хранение данных с примерами из Microsoft Azure

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 09.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 25.09.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 36,120. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87