

O'REILLY®



Cassandra

Полное руководство

Джефф Карпентер, Эбен Хьюитт



ОМК
издательство

Джефф Карпентер, Эбен Хьюитт

Cassandra.

Полное руководство

Jeff Carpenter and Eben Hewitt

Cassandra: The Definitive Guide

Distributed data at web scale

Second edition

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Джефф Карпентер, Эбен Хьюитт

Cassandra.

Полное руководство

**Распределенные данные
в масштабе веба**

Второе издание



Москва, 2017

УДК 004.73:004.65Apache Cassandra

ББК 32.972.134

К26

Карпентер Д., Хьюитт Э.

К26 Cassandra. Полное руководство. 2-е изд. / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2017. – 400 с.: ил.

ISBN 978-5-97060-453-3

Из этой книги вы узнаете, как система управления базами данных Cassandra справляется с обработкой сотен терабайтов данных, работая в нескольких ЦОДах и сохраняя высокую доступность. Во втором издании, дополненном и охватывающем версию Cassandra 3.0, вы найдете технические детали и практические примеры, которые помогут запустить эту систему в боевых условиях. Авторы демонстрируют достоинства нереляционного дизайна Cassandra, уделяя особое внимание моделированию данных.

Издание предназначено для разработчиков, администраторов баз данных и архитекторов, работающих с «большими данными» и стремящихся решить проблему масштабирования.

**УДК 004.73:004.65Apache Cassandra
ББК 32.972.134**

Authorized Russian translation of the English edition of Cassandra: The Definitive Guide, 2nd Edition.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.'

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-491-93366-4 (анг.)

ISBN 978-5-97060-453-3 (рус.)

© 2016 Jeff Carpenter, Eben Hewitt

© Оформление, издание, перевод,
ДМК Пресс, 2017

*Посвящаю своей возлюбленной, Элисон Браун.
Я слышу звуки скрипок задолго до того, как
они начинают играть¹.*

– Эбен Хьюитт

Посвящаю Стефани.

*Ты для меня источник вдохновения, неизмен-
ная опора, любовь всей моей жизни.*

– Джейфф Карпентер

¹ Цитата из песни Sway (M. D. House). – Прим. перев.

Содержание

Предисловие	14
Предисловие	16
Вступление	18
Глава 1. За пределами реляционных баз данных.....	25
Что не так с реляционными базами данных?.....	25
Краткий обзор реляционных баз данных	30
РСУБД: великие и не очень.....	31
Масштаб веба.....	39
Восхождение NoSQL	40
Резюме	42
Глава 2. Введение в Cassandra	44
Краткая презентация Cassandra.....	44
Cassandra в 40 словах.....	44
Распределенная и децентрализованная.....	45
Эластичная масштабируемость	46
Высокая доступность и отказоустойчивость	47
Настраиваемая согласованность	48
Теорема CAP Брюера.....	51
Строковая база.....	56
Высокая производительность	58
Как появилась Cassandra?.....	58
История версий	60
Подходит ли Cassandra для моего проекта?	67
Крупное развертывание.....	67
Много операций записи, статистика, анализ	67
Территориальная разнесенность	68
Быстро эволюционирующие приложения.....	68
Резюме	70
Глава 3. Установка Cassandra.....	71
Установка из дистрибутива Apache	71

Распаковка дистрибутива	71
Что внутри?	72
Сборка из исходного кода	73
Дополнительные цели сборки	75
ОС Windows	76
ОС Linux	77
Запуск сервера	77
Остановка Cassandra	79
Другие дистрибутивы Cassandra	80
Запуск оболочки CQL	81
Простые команды cqlsh	82
cqlsh Help	82
Описание окружения в cqlsh	84
Создание пространства ключей и таблицы в cqlsh	84
Запись и чтение данных в cqlsh	88
Резюме	89
 Глава 4. Язык Cassandra Query Language	 90
Реляционная модель данных	90
Модель данных Cassandra	91
Кластер	95
Пространства ключей	95
Таблицы	95
Столбцы	97
Типы данных в CQL	99
Числовые типы данных	99
Текстовые типы данных	100
Типы времени и идентификации	101
Прочие простые типы данных	103
Коллекции	104
Пользовательские типы	107
Вторичные индексы	110
Резюме	112
 Глава 5. Моделирование данных	 113
Построение концептуальной модели данных	113
Проектирование реляционной базы данных	115
Различия в проектировании для РСУБД и Cassandra	115
Определение запросов в приложении	119

Построение логической модели данных	120
Логическая модель данных отеля	122
Логическая модель данных о бронировании	124
Построение физической модели данных	126
Физическая модель данных отеля	127
Физическая модель данных о бронировании	128
Материализованные представления	129
Оценка и уточнение	131
Вычисление размера раздела	132
Оценка места, занятого на диске	133
Разбиение больших разделов	134
Определение схемы базы данных	135
DataStax DevCenter	138
Резюме	139
 Глава 6. Архитектура Cassandra	 140
Центры обработки данных и стойки	140
Сплетни и обнаружение отказов	142
Осведомители	144
Кольца и маркеры	145
Виртуальные узлы	147
Разделители	147
Стратегии репликации	148
Уровни согласованности	149
Запросы и узлы-координаторы	150
Таблицы в памяти, файлы SSTable и журналы фиксаций	151
Кэширование	154
Вручение напоминаний	154
Облегченные транзакции и Paxos	156
Надгробья	157
Фильтры Блума	158
Уплотнение	159
Антиэнтропия, исправление и деревья Меркля	160
Многоступенчатая событийно-ориентированная архитектура (SEDA) 162	162
Диспетчеры и службы	164
Демон Cassandra	164
Движок хранения	164
Служба хранения	165
Прокси хранения	165

Служба обмена сообщениями	166
Диспетчер потоков данных	166
Сервер транспортного протокола CQL.....	166
Системные пространства ключей.....	167
Резюме	169
Глава 7. Настройка Cassandra	170
Диспетчер кластера Cassandra.....	170
Создание кластера	171
Узлы-распространители	175
Разделители	176
Разделитель Murmur3Partitioner	176
Разделитель RandomPartitioner.....	176
Разделитель OrderPreservingPartitioner	176
Разделитель ByteOrderedPartitioner.....	177
Осведомители.....	178
Простой осведомитель	178
Осведомитель на основе файла свойств	178
Сплетничающий осведомитель с файлом свойств.....	179
Осведомитель, догадывающийся о стойках	179
Облачные осведомители	180
Динамический осведомитель	180
Конфигурация узлов.....	181
Маркеры и виртуальные узлы.....	181
Сетевые интерфейсы	182
Хранение данных	183
Параметры JVM и протоколирования	185
Добавление узлов в кластер	185
Динамическое присоединение ккольцу	187
Стратегии репликации	188
Стратегия SimpleStrategy.....	189
Стратегия NetworkTopologyStrategy.....	190
Изменение коэффициента репликации	191
Резюме	192
Глава 8. Клиенты	193
Hector, Astyanax и другие устаревшие клиенты.....	193
Драйвер DataStax для Java	194
Настройка среды разработки.....	195

Кластеры и точки контакта	195
Сеансы и пулы соединений.....	197
Объекты Statement.....	199
Политики.....	207
Метаданные	211
Отладка и мониторинг	215
Драйвер DataStax для Python.....	217
Драйвер DataStax для Node.js.....	218
Драйвер DataStax для Ruby	219
Драйвер DataStax для C#.....	219
Драйвер DataStax для C/C++	220
Драйвер DataStax для PHP	222
Резюме	222
 Глава 9. Чтение и запись данных	 223
Запись	223
Уровни согласованности при записи	224
Путь записи в Cassandra.....	226
Запись файлов на диск.....	228
Облегченные транзакции.....	230
Пакеты	233
Чтение.....	235
Уровни согласованности при чтении.....	236
Путь чтения в Cassandra.....	238
Исправление на этапе чтения	241
Запросы по диапазону, упорядочение и фильтрация.....	241
Функции и агрегаты	244
Разбиение на страницы.....	249
Упреждающее выполнение.....	252
Удаление	252
Резюме	254
 Глава 10. Мониторинг	 255
Протоколирование	255
Динамическое наблюдение за журналом	257
Изучение журналов.....	258
Мониторинг Cassandra средствами JMX.....	259
Подключение к Cassandra через JConsole	261
Краткий обзор MBean-объектов.....	264

МBean-объекты Cassandra	267
МBean-объекты, относящиеся к базе данных	270
МBean-объекты, относящиеся к сети	275
МBean-объекты, относящиеся к метрикам	276
МBean-объекты, относящиеся к потокам	277
МBean-объекты, относящиеся к службам	278
МBean-объекты, относящиеся к безопасности	278
Мониторинг с помощью nodetool	278
Получение информации о кластере	279
Получение статистики	282
Резюме	284
Глава 11. Обслуживание	285
Проверка исправности	285
Базовое обслуживание	286
Сброс на диск	286
Очистка	287
Исправление	288
Переиндексирование	293
Перемещение маркеров	294
Добавление узлов	294
Добавление узлов в существующий центр обработки данных	294
Добавление центра обработки данных в кластер	296
Обработка отказа узла	297
Ремонт узлов	298
Замена узлов	299
Исключение узлов	300
Переход на новую версию Cassandra	303
Резервное копирование и восстановление	305
Создание снимка	306
Удаление снимка	307
Включение инкрементного резервного копирования	307
Восстановление из снимка	308
Утилиты для работы с файлами SSTable	309
Средства обслуживания	310
DataStax OpsCenter	310
Netflix Priam	313
Резюме	313

Глава 12. Настройка производительности.....	314
Управление производительностью	314
Постановка целей	314
Мониторинг производительности	316
Анализ проблем с производительностью	317
Трассировка	318
Методика настройки.....	322
Кэширование	322
Кэш ключей.....	323
Кэш строк	323
Кэш счетчиков	324
Параметры, управляющие сохранением кэшей	324
Таблицы в памяти	325
Журналы фиксаций.....	326
Файлы SSTable.....	328
Вручение напоминаний	329
Уплотнение.....	330
Параллелизм и многопоточность	333
Сеть и тайм-ауты.....	335
Параметры JVM.....	337
Память	337
Сборка мусора.....	338
Утилита cassandra-stress.....	340
Резюме	343
 Глава 13. Безопасность	344
Аутентификация и авторизация	345
Аутентификация по паролю.....	345
Использование класса CassandraAuthorizer.....	350
Ролевое управление доступом	351
Шифрование.....	352
SSL, TLS и сертификаты	353
Шифрование трафика между узлами	354
Шифрование трафика между клиентами и узлами	357
Безопасность на уровне JMX	358
Обеспечение безопасности доступа через JMX	358
MBean-объекты, относящиеся к безопасности	360
Резюме	360

Глава 14. Развёртывание и интеграция.....	361
Планирование развертывания кластера	361
Оценка размера кластера.....	361
Выбор экземпляров.....	363
Хранилище.....	364
Сеть.....	365
Развертывание в облаке.....	366
Amazon Web Services.....	367
Microsoft Azure.....	369
Google Cloud Platform	370
Интеграция.....	370
Apache Lucene, SOLR и Elasticsearch.....	371
Apache Hadoop	371
Apache Spark	372
Резюме	380
Предметный указатель	381

Предисловие

Компания Facebook раскрыла исходный код Cassandra в июле 2008 года. Оригинальная версия была написана преимущественно двумя людьми: выходцами из Amazon и Microsoft. Большое влияние на нее оказала программа Dynamo – распределенное хранилище ключей и значений, впервые разработанное в Amazon. В Cassandra реализована модель репликации в духе Dynamo, не имеющая точек общего отказа, но при этом добавлена более эффективная модель данных на основе «семейства столбцов».

Я подключился к проекту в декабре того же года, когда компания Rackspace предложила мне разработать распределенную базу данных для своих нужд. Момент был выбран очень удачно, так как к моим услугам были все наиболее важные на тот момент масштабируемые базы данных с открытым исходным кодом – только выбирай. Несмотря на то что в активе Cassandra была только одна крупная система, ее архитектура показалась мне самой удачной, и я направил свои усилия на улучшение кода и создание сообщества.

Проект Cassandra был включен в инкубатор Apache и к моменту выхода из него в марте 2010 стал примером настоящей истории успеха. Среди его разработчиков числились компании Rackspace, Digg, Twitter и другие, которые не смогли бы написать свою базу данных с нуля, но вместе создали нечто значительное.

Сегодня Cassandra – совсем не та ранняя система, которая лежала (и до сих пор лежит) в основе поиска по папкам «Входящие» в Facebook; она превратилась в «безусловного лидера в области эффективной обработки транзакций» (по выражению Тони Бэйна) и пользуется заслуженной репутацией по части надежности и производительности в сочетании с высокой масштабируемостью.

По мере своего становления Cassandra привлекала все больше крупных пользователей, и, наконец, стало ясно, что без коммерческой поддержки не обойтись. Поэтому в апреле 2010 года мы вместе с Мэттом Пфейлем (Matt Pfeil) основали компанию Riptano. Способствовать внедрению Cassandra оказалось очень интересно и поучительно, в частности потому, что мы могли знакомиться с такими приложениями, которые не обсуждаются публично.

Также выявились нужда в такой книге, как эта. Как и многие другие проекты с открытым исходным кодом, Cassandra исторически отличалась не самой лучшей документацией. Но даже после того как дела с документацией наладились, представление материала в виде книги все равно полезно.

Благодарю Эбена за то, что он взял на себя труд раскрыть науку и искусство разработки в среде Cassandra и развертывания системы. А читателю предоставляется возможность познакомиться с методичным изложением новых концепций.

— *Джонатан Эллис*
руководитель проекта Apache Cassandra,
сооснователь и технический директор компании DataStax

Предисловие

Я очень волнуюсь, сочиняя предисловие для нового издания книги «Cassandra. Полное руководство». Вы спросите, почему? Да потому, что это новое издание! Когда появилось первое издание этой книги, проект Apache Cassandra только-только появился на свет. С годами изменилось так много, что тогдашние пользователи с трудом узнали бы в сегодняшней базе данных ту, прежнюю. Всем известно, как трудно угадаться за такими быстро развивающимися проектами, как Apache Cassandra, и я безумно благодарен Джейфу за то, что он решил поведать о текущем состоянии дел миру.

Одно из самых важных новшеств этого издания – раздел о моделировании данных. Я не раз публично заявлял: модель данных – это то, что отличает успешный проект на основе Apache Cassandra от провального. Значительная часть книги посвящена тому, как правильно построить модель. Но про эксплуатационников тоже не забыли. В современной базе Apache Cassandra есть такие вещи, как виртуальные узлы, а также многочисленные средства для обеспечения согласованности данных – и все это объясняется в новом издании. В общем, рассказать есть о чем, так что полное руководство окажется весьма кстати!

Вне зависимости от поставленной цели очень хорошо, что вы решили побольше узнать об Apache Cassandra. Сейчас самое время включить этот инструментарий в свой арсенал. А опытным пользователям стоит освежить знания, чтобы не отстать от жизни. Как показывают недавние опросы, специалисты, знакомые с Apache Cassandra, – одни из самых востребованных и высокооплачиваемых на рынке разработки приложений и построения инфраструктуры. И это отчетливая тенденция в нашей индустрии. Если организации нужна база данных с высоким уровнем масштабирования, размещенная в нескольких центрах обработки данных и постоянно готовая к работе, то лучше Apache Cassandra не найти. Первая же попытка поиска вернет сотни компаний, связавших свою судьбу с нашей любимой базой данных. И для такого доверия есть основания – вы убедитесь в этом, читая книгу. Приложения естественно мигрируют в облако, а Cassandra продолжает поддерживать работу с динамично изменяющимися глобальными данными. Эта книга научит вас применять Cassandra в соб-

ственных приложениях. Сделайте что-нибудь удивительное и предъявите свою историю успеха.

И наконец, приглашаю вас присоединиться к преуспевающему сообществу Apache Cassandra. Его членов можно найти в любом уголке мира, поэтому оно является одним из самых важных нетехнических ресурсов для новых пользователей. Нам повезло иметь такое процветающее сообщество, благодаря его совместной работе база данных Apache Cassandra стала еще лучше. Можете начать с простого – ходите на встречи и конференции, где сможете завязать знакомство со своими коллегами. Потом у вас, возможно, появится желание расширить свое участие, например писать статьи в блоге или проводить презентации, обогащая тем самым коллективный опыт и помогая новичкам, которые идут по вашим стопам. А там, глядишь, дойдет дело и до самого важного в любом проекте с открытым исходным кодом – деятельности технического характера. Напишите код, который исправит ошибку или добавит новую возможность. Отправьте в JIRA сообщение об ошибке или запрос на новую функциональность. Такие действия – показатель активности и доброго здоровья проекта. Вам не понадобится никакой особый статус, просто создайте учетную запись – и вперед! А если столкнетесь с затруднениями, загляните снова в эту книгу или обратитесь к сообществу. Мы всегда готовы прийти на помощь.

Заинтересовались? Это хорошо!

Но хватит слов, пора перелистнуть страницу и приступить к учебе.

— *Патрик Макфейдин,*
главный пропагандист
Apache Cassandra, компания DataStax

Вступление

Почему именно Apache Cassandra?

Apache Cassandra – бесплатная распределенная система хранения данных с открытым исходным кодом, которая принципиально отличается от реляционных систем управления базами данных (РСУБД).

Проект Cassandra получил статус инкубаторного проекта Apache в январе 2009 года. Вскоре после этого команда разработчиков, возглавляемая Джонатаном Эллисом, выпустила версию Cassandra 0.3, а затем версии стали выходить регулярно. Систему Cassandra применяют некоторые крупнейшие интернет-компании, в т. ч. Facebook, Twitter и Netflix.

Во многом ее популярность объясняется выдающимися техническими характеристиками. Она надежна, легко масштабируется и допускает настройку уровня согласованности данных. Операции записи выполняет с фантастической скоростью, система может хранить сотни терабайтов данных, децентрализована и симметрична, так что точки общего отказа отсутствуют. База данных характеризуется высокой доступностью, предлагаются также средства моделирования данных на основе языка Cassandra Query Language (CQL).

Интересна ли вам эта книга?

Эта книга ориентирована на различные аудитории. Она будет полезна:

- разработчикам высокомасштабируемых приложений для работы с очень большими объемами данных, в частности социальных приложений для Web 2.0 и интернет-магазинов;
- архитекторам приложений или данных, которым нужно понимать, какие есть варианты создания высокопроизводительных, децентрализованных, эластичных хранилищ данных;
- администратору или разработчику баз данных, который знаком со стандартными реляционными СУБД и хочет понять, как можно реализовать отказоустойчивое, согласованное в конечном счете хранилище данных;

- руководителю, желающему разобраться в достоинствах (и недостатках) Cassandra и других столбцовых баз данных, чтобы принять решение о технической стратегии;
- студентам, аналитикам и исследователям, занимающимся проектированием систем, в которых участвует Cassandra или иное нереляционное хранилище.

Эта книга представляет собой техническое руководство. В ряде отношений Cassandra предлагает новый взгляд на данные. Многие разработчики, получившие профессиональные навыки в последние 15–20 лет, хорошо освоили рассуждения о данных в реляционных или объектно-ориентированных терминах. В Cassandra модель данных совершенно иная, и поначалу осмыслить ее нелегко, особенно тем из нас, кто имеет предвзятые идеи о том, что такая база данных и какой она должна быть.

Работать с Cassandra могут не только Java-разработчики. Однако система написана на Java, поэтому если вы хотите изучить исходный код, то без ясного понимания Java никак не обойтись. И хотя знать Java, строго говоря, необязательно, знакомство с этим языком поможет лучше разобраться в исключениях, понять, как строить исходный код и как пользоваться некоторыми популярными клиентами. Многие примеры в книге написаны на Java. Но существуют также интерфейсы к Cassandra из C#, Python, node.js, PHP, Ruby и других языков.

Наконец, предполагается, что читатель ясно понимает, как работает веб, умеет пользоваться интегрированной средой разработки (IDE) и хотя бы немного знаком с типичными проблемами, которые возникают в приложениях, управляемых данными. Даже профессиональный разработчик или администратор, начав изучать Cassandra, может столкнуться с незнакомыми ранее инструментами. Так, для сборки Cassandra используется Apache Ant, а для доступа к исходному коду – Git. В тех случаях, когда для работы с примерами требуется что-то установить или настроить самостоятельно, мы стараемся помочь советом.

О содержании книги

Книга построена так, что каждая глава является независимым руководством – в той мере, в какой это возможно. Это важно для книги о проекте Cassandra, который ориентирован на разных пользователей и быстро изменяется. По аналогии с программным обеспечением книга организована по «модульному» принципу. Начинающим

изучать Cassandra лучше читать ее по порядку. Но даже если для вас начальные стадии – пройденный этап, все равно некоторые главы могут оказаться полезными автономными руководствами.

Ниже кратко описано содержание глав.

Глава 1 «За пределами реляционных баз» описывает историю невероятного успеха реляционных баз данных и недавнего восхождения нереляционных технологий, к каковым относится и Cassandra.

Глава 2 «Введение в Cassandra» содержит введение в Cassandra, в ней обсуждаются наиболее интересные особенности, отличающие ее от других продуктов, истоки и преимущества системы.

Глава 3 «Установка Cassandra» описывает порядок установки и запуска Cassandra, а также выполнение некоторых простейших операций.

Глава 4 «Cassandra Query Language» рассматривает модель данных в Cassandra с упором на отличия от традиционной реляционной модели. Здесь мы также изучим, как выразить модель на языке Cassandra Query Language (CQL).

Глава 5 «Моделирование данных» содержит введение в принципы и процессы моделирования данных в Cassandra. Для создания работоспособной схемы мы проанализируем хорошо известную предметную область.

Глава 6 «Архитектура Cassandra» поможет понять, что происходит при операциях чтения и записи и как базе данных удается достичь выдающихся характеристик в части надежности и высокой доступности. Мы рассмотрим некоторые внутренние механизмы, в т. ч. протокол распространения сплетен, вручение напоминаний (hinted handoff), исправление на этапе чтения, деревья Меркля и др.

В главе 7 «Настройка Cassandra» показано, как описывать распределители, стратегии размещения реплик и осведомители (snitch). Мы настроим кластер и понаблюдаем за последствиями выбора различных конфигурационных параметров.

Глава 8 «Клиенты» описывается клиентов для различных языков, в т. ч. Java, Python, node.js, Ruby, C# и PHP, позволяющих абстрагировать низкоуровневый API Cassandra. Мы поможем разобраться в общих функциях драйверов.

В главе 9 «Чтение и запись данных» на базе всего ранее изложенного мы узнаем, что Cassandra делает «под капотом» для чтения и записи данных. Мы также обсудим пакеты, облегченные транзакции и разбиение на страницы.

Глава 10 «Мониторинг» – после того как кластер запущен, встает задача о мониторинге его работы, характере использования памяти

и потоков и оценке общего состояния. В Cassandra реализован развитый интерфейс Java Management Extensions (JMX), которым мы воспользуемся для решения этих и других задач.

Глава 11 «Обслуживание» – текущее обслуживание кластера Cassandra упрощается благодаря инструментам, входящим в комплект поставки сервера. Мы покажем, как вывести узел из эксплуатации, балансировать нагрузку, получать статистику и решать другие стандартные задачи.

Глава 12 «Настройка производительности» – одна из наиболее выдающихся характеристик Cassandra – чрезвычайно высокое быстродействие. Но можно добиться еще большего за счет правильной настройки параметров памяти и хранилища данных, выбора оборудования, кэширования, задания размеров буферов и т. п.

Глава 13 «Безопасность» – часто можно встретить пренебрежительное отношение к технологиям NoSQL как недостаточно безопасным. Но Cassandra предоставляет средства аутентификации, авторизации и шифрования, настройке которых посвящена эта глава.

Глава 14 «Развертывание и интеграция» – и в конце мы поговорим о планировании развертывания кластера, в том числе на облачных платформах Amazon, Microsoft и Google. Мы также вкратце расскажем о нескольких технологиях, которые нередко используются совместно с Cassandra с целью расширения ее возможностей.



Версии Cassandra, используемые в этой книге

При написании книги мы использовали версии семейства Cassandra 3.X и драйвер DataStax для Java версии 3.0.

При упоминании возможностей, добавленных, начиная с версии 2.0, мы указываем, в какой именно версии они появились. Это может быть полезно читателям, работающим с ранними версиями и планирующим переход на новую.

ЧТО НОВОГО ВО ВТОРОМ ИЗДАНИИ

Первое издание книги «Cassandra. Полное руководство» было вообще первой книгой на тему Cassandra и на протяжении нескольких лет оставалось авторитетным. Но с 2010 года многое в Cassandra изменилось – как в части самой технологии, так и в отношении сообщества, которое эту технологию развивает и поддерживает. Ниже перечислены основные изменения, которые мы внесли, чтобы восстановить актуальность материала.

В ногу со временем

Первое издание относилось к версии 0.7, выпущенной в 2010 году. В 2016 году мы уже работаем с версиями семейства 3.X. Самое важное новшество – появление языка CQL и отказ от старого Thrift API. Из других архитектурных изменений отметим вторичные индексы, материализованные представления и облегченные транзакции. В главе 2 приведена история выпуска версий, она поможет проследить за изменениями. Упоминая новые возможности в тексте, мы часто указываем, в какой версии они появились.

Поддержка разработчиков

С годами техника разработки и тестирования систем на основе Cassandra претерпела серьезные изменения, чему способствовали внедрение оболочки CQL (`cqlsh`) и постепенная замена клиентов, разработанных сообществом, драйверами, предлагаемыми компанией DataStax. Подробному описанию `cqlsh` посвящены главы 3 и 4, а драйверам – главы 8 и 9. В главе 9 мы также приводим детальные сведения о выполнении операций чтения и записи в Cassandra, это поможет читателю разобраться во внутренних механизмах работы и понять, к каким последствиям приводят те или иные решения.

Новый уровень эксплуатации Cassandra

По мере роста количества внедрений Cassandra в производственных системах индивидуальными пользователями и организациями расширялась и база знаний о возникающих проблемах и способах их разрешения. Чтобы накопленные знания не остались втуне, мы добавили новые главы о безопасности (глава 13) и развертывании и интеграции (глава 14), а также значительно расширили главы о мониторинге, обслуживании и настройке производительности (главы 10–12).

Графические выделения

В книге применяются следующие графические выделения:

Курсив

Новые термины, имена и расширения имен файлов.

Моноширинный

Листинги программ, а также элементы кода в основном тексте: имена переменных и функций, базы данных, типы данных, переменные окружения, предложения и ключевые слова языка.

Моноширинный полужирный

Команды и иные строки, которые следует вводить буквально.

Моноширинный курсив

Текст, вместо которого следует подставить значения, заданные пользователем или определяемые контекстом.



Так обозначается замечание общего характера



Так обозначается предупреждение или предостережение.

О примерах кода

Примеры кода, встречающиеся в этой книге, можно скачать со страницы <https://github.com/jeffreyscarpenter/cassandra-guide>.

Эта книга призвана помочь вам в работе. Поэтому вы можете использовать приведенный в ней код в собственных программах и в документации. Спрашивать у нас разрешение необязательно, если только вы не собираетесь воспроизвести значительную часть кода. Например, никто не возбраняет включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров из книг издательства O'Reilly на компакт-диске разрешение требуется. Цитировать книгу и примеры в ответах на вопросы можно без ограничений. Но для включения значительных объемов кода в документацию по собственному продукту нужно получить разрешение.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «Cassandra: The Definitive Guide, Second Edition, by Jeff Carpenter. Copyright 2016 Jeff Carpenter, 978-1-491-93366-4».

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу permissions@oreilly.com.

Как с нами связаться

Вопросы и замечания по поводу этой книги отправляйте в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

800-998-9938 (в США и Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги имеется веб-страница, на которой публикуются списки замеченных ошибок, примеры и прочая дополнительная информация. Адрес страницы: <http://bit.ly/cassandra2e>.

Замечания и вопросы технического характера следует отправлять по адресу bookquestions@oreilly.com.

Дополнительную информацию о наших книгах, конференциях и новостях вы можете найти на нашем сайте по адресу <http://www.oreilly.com>.

Читайте нас на Facebook: <http://facebook.com/oreilly>.

Следите за нашей лентой в Twitter: <http://twitter.com/oreillymedia>.

Смотрите нас на YouTube: <http://www.youtube.com/oreillymedia>.

Благодарности

Мы благодарны многим замечательным людям, которые способствовали выходу книги в свет.

Спасибо нашим рецензентам: Стю Худу (Stu Hood), Роберту Шнейдеру (Robert Schneider) и Гэри Дасбейбеку (Gary Dusbabek), которые высказали немало ценных замечаний к первому изданию, а также Эндрю Бейкеру (Andrew Baker), Эвану Эллиоту (Ewan Elliot), Кирку Дэмрону (Kirk Damron), Кори Коулу (Corey Cole), Джейфу Джирса (Jeff Jirsa) и Патрику Макфейдину (Patrick McFadin), которые рецензировали второе издание. Замечания Криса Джадсона (Chris Judson) оказали решающее влияние на окончательный вид главы 14.

Мы признательны Джонатану Эллису и Патрику Макфейдину, написавшим предисловия к первому и второму изданиям соответственно. Патрик, спасибо тебе за вклад в раздел об интеграции со Spark в главе 14.

Спасибо нашим редакторам, Майку Лоукидесу (Mike Loukides) и Мари Божуро (Marie Beaugureau), за постоянную поддержку и стремление сделать книгу лучше.

Джефф благодарит Эбена за предоставленную возможность принять участие в обновлении пользующейся уважением книги и за поддержку на протяжении всей работы.

Наконец, нас вдохновляли многочисленные великолепные разработчики, внесшие свой вклад в проект Cassandra. Снимаем шляпу перед всеми, кто создал такую элегантную и эффективную базу данных.

Глава 1

За пределами реляционных баз данных

Если в первый момент идея не кажется абсурдной, она безнадежна.

— Альберт Эйнштейн

Цель этой книги — помочь разработчикам и администраторам баз данных в освоении важной технологии. По ходу дела мы будем сравнивать Cassandra с традиционными реляционными СУБД и покажем, как включить ее в свою среду.

Что не так с реляционными базами данных?

Если бы я спросил людей, чего они хотят, они бы попросили более быструю лошадь.

- Генри Форд

Представьте себе некую модель данных, придуманную небольшой командой разработчиков в компании, насчитывающей тысячу работников. Она была доступна по протоколу TCP/IP, а обращаться к ней можно было из программ на разных языках, включая Java и веб-службы. Поначалу в модели могли разобраться разве что самые квалифицированные специалисты по информатике, но с течением времени она распространялась все шире, и заложенные в ней идеи ста-

новились понятнее. Для использования базы данных, построенной на основе этой модели, пришлось выучить новые термины и по-новому взглянуть на хранение данных. Но по мере того как идея обрастала готовыми продуктами, ее принимали на вооружение все новые компании и правительственные учреждения, в немалой степени потому, что база данных работала быстро – могла выполнять тысячи операций в секунду. И приносила колоссальные доходы.

А затем появилась новая модель.

Новая модель несла в себе угрозу, в основном по двум причинам. Во-первых, она сильно отличалась от старой модели, демонстративно споря с ней. И это пугает, потому что понять и принять нечто новое и радикально отличающееся всегда трудно. Последующие споры могут лишь укрепить упорствующих в отстаивании своих взглядов – взглядов, которые, возможно, сложились на основе всего предшествующего опыта и условий работы. Во-вторых, – и это, пожалуй, более важное препятствие – новая модель несет угрозу, потому что компании вложили огромные деньги в старую модель и получают от ее использования немалую прибыль. Смена курса кажется нелепой, даже невозможной.

Разумеется, мы говорим об иерархической базе данных *Information Management System (IMS)*, придуманной в компании IBM в 1966 году.

IMS создавалась с целью использования в ракете для полета на Луну Сатурн-5. Ее архитектор Верн Уоттс (Vern Watts) посвятил ей всю свою профессиональную карьеру. Многие из нас знакомы с базой данных DB2, разработанной в IBM. Эта весьма популярная СУБД получила имя от своей предшественницы – программы DB1, построенной на основе иерархической модели данных IMS. IMS была выпущена в 1968 году и затем получила широкое распространение как составная часть системы *Customer Information Control System (CICS)* и других приложений. Она используется и по сей день.

Но спустя несколько лет после изобретения IMS появилась новая, угрожающая ей модель – реляционная база данных.

В статье «*A Relational Model of Data for Large Shared Data Banks*», вышедшей в 1970 году, д-р Эдгар Ф. Кодд пропагандировал теорию реляционной модели данных, созданную им во время работы в исследовательской лаборатории IBM в Сан-Хосе. Эта статья, и сейчас доступная по адресу <http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf>, легла в основу всех работ по реляционным системам управления базами данных.

Работа Кодда была полярно противоположна иерархической структуре IMS. Для понимания реляционной базы данных и работы с ней

понадобились новые термины: «отношение», «кортеж», «нормальная форма» и другие, которые были совершенно непривычны пользователям IMS. Новая система предлагала принципиальные улучшения по сравнению с предшествующей, в частности возможность выражать сложные связи между сущностями – намного более сложные, чем допускали иерархические базы данных.

За сорок лет эти идеи и их применения подверглись существенному развитию, но и до сих пор реляционные базы данных, без сомнения, являются одними из самых успешных программ в истории. Мы встречаем их как в форме персональных баз данных типа Microsoft Access, так и в гигантских транснациональных корпорациях, эксплуатирующих кластеры из сотен тщательно настроенных узлов, в совокупности составляющих многотерабайтное хранилище данных. В реляционных базах хранятся счета-фактуры, сведения о заказчиках, каталоги товаров, бухгалтерские книги, схемы аутентификации пользователей – чуть ли не все существующие в мире данные. Несомненно, реляционные базы – важнейшая грань современного технологического и делового ландшафта, и они пребудут с нами в различных формах еще много лет, как, впрочем, и IMS в ее разных ипостасях. Реляционная модель – это альтернатива IMS, и у каждой имеются свои применения.

Поэтому кратким ответом на вопрос «Что не так с реляционными базами данных?» будет «Все так».

Однако существует и более развернутый ответ: время от времени рождаются идеи, которые, на первый взгляд, меняют все и порождают всяческие революции. Но если взглянуть на это конструктивно, то такие революции – не более чем обычный ход истории. IMS, РСУБД, NoSQL. Лошадь, автомобиль, самолет. Новое всегда основывается на предшествующем ему, каждая новая технология стремится решить определенные проблемы; она хороша для одних целей и не столь хороша для других. Все они мирно сосуществуют, даже сейчас.

Давайте поговорим о том, почему на настоящем этапе развития стоит поискать альтернативу реляционной базе данных. Точно так же сам Кодд сорок лет назад анализировал систему Information Management System и думал, что это, возможно, не единственно допустимый способ организации информации и решения задач хранения данных. И что, возможно, для некоторых задач было бы более плодотворно подумать об альтернативах.

Когда приложение, в котором применяется реляционная база, становится популярным и активно используется, мы сталкиваемся

с проблемой масштабируемости. Даже в реляционных базах скромного размера никуда не деться от операции соединения, а она может оказаться медленной. Согласованность в базе данных обычно обеспечивается за счет транзакций, а это подразумевает блокировку некоторой части базы, из-за чего она становится недоступной другим клиентам. При очень высокой нагрузке это может стать нетерпимым, потому что из-за блокировок запросы пользователей выстраиваются в очередь в ожидании возможности выполнить операцию чтения или записи.

Обычно эти проблемы решаются одним или несколькими из перечисленных ниже способов, часто именно в указанном порядке.

- Бросить на решение проблемы аппаратные ресурсы: добавить память, поставить более быстрые процессоры, модернизировать диски. Такой подход называется *вертикальным масштабированием*. Он может спасти на некоторое время.
- Когда проблема возникает снова, применяется похожее решение: раз эта машина достигла своего потолка, добавим оборудование в виде дополнительных машин, образующих кластер базы данных. Теперь встает проблема репликации и обеспечения согласованности в обычном режиме работы и в случае отказа. Такой проблемы раньше не было.
- Нам придется изменить конфигурацию системы управления базами данных. Возможно, нужно будет оптимизировать каналы, по которым база данных производит запись в файловую систему. Мы отключаем протоколирование или журналирование, что обычно нежелательно (а иногда просто невозможно).
- Сделав все возможное на уровне СУБД, мы обращаем внимание на приложение. Пытаемся улучшить структуру индексов. Оптимизируем запросы. Но надо полагать, что при таком-то масштабе мы уже позаботились об индексах и запросах, так что они и так настроены оптимально. Наступает черед мучительной ревизии кода доступа к данным в попытках найти хоть какие-нибудь возможности для улучшения. Это может быть отказ от некоторых соединений или их реорганизация, выбрасывание таких накладных средств, как обработка XML в хранимой процедуре, и т. д. Разумеется, изначально у нас были причины обрабатывать XML, поэтому если где-то делать это необходимо, то придется перенести обработку на уровень приложения и молиться, чтобы нигде ничего не сломалось.
- Мы включаем в систему уровень кэширования. В больших системах это может быть какой-нибудь распределенный кэш типа

memcached, Redis, Riak, EHCache. Теперь возникает проблема согласованного обновления кэша и базы данных, которая к тому же осложняется наличием кластера.

- Мы снова обращаем внимание на базу данных и приходим к выводу, что теперь, когда приложение создано и мы понимаем, как выполняются основные запросы, можно пойти на дублирование некоторых данных, адаптировав их к структуре запросов. Этот процесс денормализации идет вразрез с пятью нормальными формами, характеризующими реляционную модель, и прямо нарушает 12 правил Кодда. Мы напоминаем себе, что живем в реальном, а не в идеальном мире и делаем все, чтобы приложение откликалось за приемлемое время, пусть даже в результате оно перестает быть «чистым».



Двенадцать правил Кодда

Кодд составил список из 12 правил (на самом деле их 13, занумерованных от 0 до 12), формализующих его определение реляционной модели. Это было сделано в ответ на отклонение коммерческих СУБД от оригинальных идей. Эти правила Кодд изложил в двух статьях в журнале CompuWorld за октябрь 1985 года и formalizoval во втором издании своей книги «The Relational Model for Database Management», которая больше не переиздается.

Наверное, все это вам знакомо. Инженеры, имеющие дело с системами масштаба веба, вправе задаться вопросом, а не напоминает ли ситуация известное высказывание Генри Форда о том, что более быстрая лошадь – не то, что нужно людям. И в поисках ответа на этот вопрос они проделали интересную и впечатляющую работу.

В этот момент мы должны осознать, что реляционная модель – это всего лишь модель. То есть полезный взгляд на мир, применимый к некоторому классу задач. Она никогда не задумывалась как исчерпывающий способ представления данных, который не подлежит пересмотру и не оставляет места альтернативам. Обратившись к истории, мы убедимся, что в свое время модель Кодда подрывала основы. В ней использовалась совершенно новая терминология, например знакомое слово «кортеж» в новом, дотоле не употреблявшемся смысле. К реляционной модели относились с подозрением, она подвергалась ожесточенным нападкам. В оппозицию к ней встал даже работодатель Кодда – компания IBM, которая построила приносящий немалые прибыли набор продуктов на основе IMS и вовсе не желала делиться своим куском пирога со всякими выскочками.

Но теперь реляционная модель занимает, наверное, лучшее место в доме под названием «мир данных». У языка SQL широкая поддержка, его хорошо понимают разработчики. SQL преподают на вводных курсах в университетах. Существуют базы данных с открытым исходным кодом, уже установленные на сервере интернет-провайдера, которыми может воспользоваться всякий, кто платит 4,95 доллара в месяц за хостинг. Поставщики облачных решений PaaS (Platform-as-a-Service – платформа как услуга) – Amazon Web Services, Google Cloud Platform, Rackspace и Microsoft Azure предлагают доступ к реляционной базе данных как услугу, включающую автоматизированный мониторинг и обслуживание. Зачастую выбор базы данных продиктован архитектурными стандартами, принятыми в организации. Но даже если таких стандартов нет, имеет смысл поинтересоваться, какая платформа уже используется в организации. Коллеги из отделов разработки и инфраструктуры тяжелым трудом накопили много полезных знаний.

За многие годы мы привыкли считать (быть может, по инерции), что реляционная база данных – решение на все случаи жизни.

Поэтому, наверное, лучше спрашивать не «Что не так с реляционными базами данных?», а «Какая у вас проблема?».

То есть мы хотим убедиться, что решение соответствует имеющейся задаче. Для некоторых задач реляционные базы подходят идеально. Но лавинообразный рост веба и в особенности социальных сетей ведет к такому же росту объема данных, которые необходимо обрабатывать. Когда в начале 1990-х годов Тим Бернерс-Ли обдумывал веб, предполагалось, что эта сеть будет использоваться для обмена научными документами между сотрудниками физических лабораторий. Но теперь веб проник повсюду, им пользуются как ученые, так и легионы пятилетних ребятишек, которые шлют друг другу смайлики с котятами. Среди прочего это означает необходимость поддерживать гигантские объемы данных, и тот факт, что веб с этим справляется, – памятник прекрасно продуманной архитектуре.

Однако часть этой инфраструктуры начинает прогибаться под непомерным весом.

Краткий обзор реляционных баз данных

Скорее всего, все это вам известно, но давайте все же освежим в памяти некоторые фундаментальные концепции реляционных баз данных. Это позволит заложить основу для рассмотрения недавних идей

касательно компромиссов, характерных для распределенных систем, в особенности очень больших – таких, которые необходимы в масштабе веба.

РСУБД: великие и не очень

Причин, по которым реляционные базы данных добились такого ошеломительного успеха за последние сорок лет, немало. Одной из самых важных является язык Structured Query Language (SQL), обладающий весьма развитой функциональностью и вместе с тем простым декларативным синтаксисом. Впервые SQL был принят в качестве стандарта ANSI в 1986 году; с тех пор вышло несколько версий, а производители включили нестандартные расширения синтаксиса, например Microsoft T-SQL и Oracle PL/SQL, для поддержки зависящих от реализации возможностей.

Своей эффективностью SQL обязан нескольким факторам. Он позволяет пользователю представлять сложные связи между данными с помощью команд языка манипулирования данными (Data Manipulation Language – DML) для вставки, выборки, обновления, удаления, усечения и объединения данных. Применяя функции, основанные на реляционной алгебре, мы можем выполнять различные операции, например находить минимальное и максимальное значения во множестве или фильтровать и сортировать результаты. Команды SQL поддерживают группировку и агрегирование данных. Благодаря языку определения данных (Data Definition Language – DDL) SQL предоставляет средства для непосредственного создания, изменения и удаления структурных элементов схемы без перезагрузки. SQL также позволяет предоставлять и отзывать права для отдельных пользователей и групп пользователей.

SQL прост в использовании. Для изучения базового синтаксиса нужно совсем немного времени, да и вообще на концептуальном уровне SQL и РСУБД не представляют особых сложностей. Начинающие разработчики быстро приобретают необходимые навыки, а, как часто бывает в отрасли с высоким темпом изменений, жесткими сроками и растущими затратами, простота использования может оказаться решающим фактором. А в данном случае простота свойственна не только синтаксису; существует много надежных инструментов, включающих интуитивно понятный графический интерфейс для просмотра и манипулирования данными.

Отчасти благодаря стандартизации SQL позволяет без труда интегрировать РСУБД с разнообразными системами. Нужно лишь

установить драйвер для языка, на котором написано приложение, – и можно начинать, причем уровень переносимости очень высок. Если вы решите переписать приложение на другом языке (или сменить базу данных), то часто для этого не потребуется много усилий, если, конечно, вы не загнали себя в угол чрезмерно усердным применением нестандартных расширений.

Транзакции, свойства ACID и двухфазная фиксация

Помимо всего вышеупомянутого, РСУБД и SQL поддерживают *транзакции*. Важнейшая особенность транзакций заключается в том, что сначала они выполняются виртуально, т. е. программист может отменить (откатить) изменения, если во время выполнения что-то пошло не так; если же все хорошо, то транзакцию можно зафиксировать. По выражению Джима Грея, транзакция – это «преобразование состояния», обладающее свойствами ACID (см. статью «The Transaction Concept: Virtues and Limitations» по адресу <http://research.microsoft.com/en-us/um/people/gray/papers/theTransactionConcept.pdf>).

Акроним ACID расшифровывается как «Atomic, Consistent, Isolated, Durable» – атомарность, согласованность, изолированность, долговечность. Это те свойства, которые позволяют оценить, насколько правильно и успешно выполнена транзакция.

Атомарность

Атомарность означает «все или ничего». Иными словами, чтобы выполнение транзакции было признано успешным, все команды обновления внутри нее должны завершиться успешно. Не может быть частичного успеха, когда одно обновление выполнилось, а другое – связанное с ним – нет. Стандартный пример – перевод денежных средств с одного счета на другой, т. е. уменьшение суммы средств на одном счете и увеличение на другом. Эта операция неделима – обе ее части должны быть успешно выполнены.

Согласованность

Согласованность означает, что данные переходят из одного корректного состояния в другое, тоже корректное, не оставляя стороннему наблюдателю возможности увидеть недопустимую совокупность значений. Например, транзакция, которая пытается удалить клиента и всю историю его заказов, не может оставить строки заказов, ссылающиеся на первичный ключ удаленного клиента; такое состояние является несогласованным, и попытка прочитать такие записи о заказах привела бы к ошибке.

Изолированность

Изолированность означает, что транзакции, выполняемые одновременно, не смешиваются – каждая выполняется в своем собственном пространстве. Иначе говоря, если две разные транзакции попытаются модифицировать одни и те же данные, то одна должна будет дождаться завершения другой.

Долговечность

После успешного завершения транзакции произведенные в ней изменения не будут потеряны. Это не означает, что другая транзакция не может впоследствии изменить те же самые данные, просто процессы-писатели могут быть уверены, что данные будут доступны последующим транзакциям в измененном виде.

Споры о поддержке транзакций – большое место в любом разговоре о нереляционных хранилищах данных, поэтому разберемся, в чем тут дело. На первый взгляд, свойства ACID кажутся настолько желательными, что и говорить не о чем. Трудно представить себе человека, работающего с базой данных, который не предполагал бы, что обновления сохраняются в течение некоторого времени, ведь именно в доступности другим пользователям для чтения и состоит смысл обновлений. Однако более пристальное изучение приводит к выводу, что было бы неплохо в какой-то степени управлять этими свойствами. Часто говорят, что в Интернете не бывает бесплатных завтраков, и, поняв, чем приходится платить за транзакции, мы, возможно, захотим поискать альтернативу.

При высокой нагрузке транзакции становятся источником проблем. Впервые попытавшись горизонтально масштабировать реляционную базу данных, вы столкнетесь с необходимостью учитывать *распределенные транзакции*, которые распространяются не на одну таблицу и даже не на одну базу данных, а на несколько систем. Чтобы гарантировать выполнение свойств ACID в такой ситуации, понадобится диспетчер транзакций, координирующий работу нескольких узлов.

Для выполнения транзакций на нескольких хостах придумана двухфазная фиксация (иногда можно встретить сокращение «2PC»). Но поскольку в этом случае блокируются все ассоциированные с транзакцией ресурсы, двухфазная фиксация полезна только для коротких операций. Возможно, что в вашем случае все распределенные операции завершаются меньше чем за секунду, но, очевидно, так бывает не всегда. Есть ситуации, когда требуется координация работы

нескольких хостов, которые вы не контролируете. Операции обновления, включающие несколько различных, но взаимосвязанных действий, могут выполняться часами.

Двухфазная фиксация – *блокирующая* операция, т. е. клиенты («конкурирующие потребители») должны ожидать завершения предыдущей транзакции, прежде чем смогут получить доступ к блокированному ресурсу. Протокол предполагает ожидание ответа от узла, даже если он перестал работать. В этом случае есть возможность избежать бесконечного ожидания, установив тайм-аут; по его истечении координатор транзакций решит, что узел уже никогда не ответит, и отменит транзакцию. Однако бесконечный цикл в протоколе двухфазной фиксации все же возможен, и связано это с тем, что узел может отправить координатору транзакций сообщение о том, что тот может зафиксировать транзакцию. Затем узел будет ждать от координатора ответного сообщения о состоявшейся фиксации (или откате, если, к примеру, какой-то другой узел не согласен на фиксацию). Если в такой ситуации координатор «упадет», то узел будет ждать ответа вечно.

Ввиду этих недостатков протокола двухфазной фиксации распределенных транзакций была введена идея *компенсации*. Компенсация часто используется в веб-службах и, попросту говоря, означает, что операция фиксируется немедленно, а в случае ошибки выполняется другая операция, призванная восстановить корректное состояние.

Существует ряд хорошо известных паттернов компенсационных действий, которые архитекторам часто приходится рассматривать в качестве альтернативы двухфазной фиксации. К ним относится, в частности, « списание » ошибочной транзакции, т. е. игнорирование ошибки в момент ее возникновения и устранение последствий позже. Другой вариант – повтор невыполненных операций через некоторое время после получения уведомления. В системах бронирования и биржевой торговли такие решения, скорее всего, не подойдут. Но в биллинговых системах и в приложениях для уведомления о проблемах они вполне допустимы.



Проблема двухфазной фиксации

Грегор Хоп, архитектор из Google, опубликовал в своем блоге замечательную, часто цитируемую статью под названием «Starbucks Does Not Use Two-Phase Commit»¹ (http://www.enterpriseintegration-patterns.com/ramblings/18_starbucks.html). В ней показано, как трудно масштабировать двухфазную фиксацию в реальных при-

¹ В Старбаксе двухфазная фиксация не используется. – *Прим. перев.*

ложении, и предложены некоторые альтернативы. Это легкое, забавное и поучительное чтение.

С точки зрения разработчика приложения, двухфазная фиксация может означать потерю доступности и большое время задержки в случае частичного отказа. То и другое крайне нежелательно. Поэтому если вам повезло – ваше приложение оказалось настолько успешным, что база данных не умещается на одной машине, – то придется решать, как обрабатывать транзакции, охватывающие несколько машин, чтобы не нарушились свойства ACID. Не важно, на скольких машинах размещается база данных – на 10, 100 или 1000, – атомарность необходима так же, как если бы узел был всего один. Только теперь эта пилюля намного горше.

Схема

К числу часто превозносимых особенностей реляционных баз данных нередко относят гибкость схем. Реляционная модель позволяет представить объекты предметной области. Даже развилась целая индустрия весьма дорогостоящих инструментальных средств (например, CA ERWin Data Modeler) для упрощения этой деятельности. Но чтобы схема была правильно нормализована, приходится создавать таблицы, которым не соответствуют никакие объекты предметной области. Например, в схеме университетской базы данных могут быть таблицы «student» и «course». Однако, поскольку между ними имеется связь типа «многие-ко-многим» (один студент может посещать много курсов, а один курс может посещать много студентов), необходима связующая таблица. Это «загрязняет» чистую модель данных, в которой есть только студенты и курсы, и, ко всему прочему, вынуждает нас писать более сложные SQL-команды для соединения таблиц. А операция соединения может работать долго.

Опять-таки, в небольшой системе это не так уж важно. Но когда таблиц много и в каждой много строк, сложные запросы с несколькими соединениями могут выполняться недопустимо долго.

Наконец, не каждая схема хорошо ложится на реляционную модель. В последние десять лет появились и стали популярными сложные системы обработки событий, в которых нужно без потерь представлять очень интенсивный поток изменений состояния. Это полезно, когда нужно согласовать одни события с другими и сделать выводы, позволяющие принять важное для бизнеса решение. Потоки событий можно представить в реляционной базе данных, но с большой натяжкой.

Любой разработчик приложений, без сомнения, знаком с различными системами объектно-реляционного отображения (ORM), которые в последние годы расплодились в неимоверном количестве и призваны сгладить трудности, присущие представлению объектов в реляционной модели. В небольшой системе ORM действительно может помочь. Но часто привносит свои собственные проблемы, например дополнительные требования к памяти, и «загрязняет» код приложения громоздким и малопонятным кодом отображения. Вот пример Java-метода, в котором система Hibernate используется, чтобы «устранить трудность» написания кода на SQL:

```
@CollectionOfElements  
 @JoinTable(name="store_description",  
     joinColumns = @JoinColumn(name="store_code"))  
 @MapKey(columns=@Column(name="for_store", length=3))  
 @Column(name="description")  
 private Map<String, String> getMap() {  
     return this.map;  
 }  
//... и т. д.
```

Не правда ли, это что угодно, только не решение проблемы? И конечно, встречаются системы, например связанные с обменом документами, в том числе в формате XML, в которых не существует очевидного отображения на реляционную модель. Это только усугубляет проблему.

Сегментирование и архитектура без разделения ресурсов

Если не можете разбить,
то не сможете и масштабировать.

— Рэнди Шоуп,
почетный архитектор, eBay

Другой подход к масштабированию реляционной базы данных – включить в архитектуру *сегментирование* (sharding). Он с успехом применяется на таких больших сайтах, как eBay, который обрабатывает миллиарды SQL-запросов в день, и в других современных веб-приложениях. Идея в том, чтобы не хранить все данные на одном сервере и не реплицировать все данные на каждый сервер в кластере, а разделить их на части и поместить каждую часть на отдельный сервер.

Рассмотрим, например, большую таблицу клиентов в реляционной базе данных. Меньше всего изменений (по крайней мере, с точки зрения программистов) система потребует в случае вертикального масштабирования путем добавления процессоров, памяти и более быстрых дисков. Но если систему преследует успех и число клиентов только растет, то в какой-то момент (скажем, когда число строк достигнет десятка миллионов) неизбежно придется подумать о дополнительных машинах. И что тогда? Просто скопировать все данные на каждую машину? Или разбить одну таблицу клиентов, так чтобы в каждой базе данных хранилась только часть записей, с сохранением порядка? Тогда при выполнении запросов, касающихся клиентов, нагрузка ляжет только на ту машину, где находятся искомые записи, а другие будут свободны.

Кажется очевидным, что для такого сегментирования необходимо найти подходящий ключ, по которому упорядочиваются записи. Например, можно было бы распределить данные между 26 машинами (по числу букв латинского алфавита), так что на каждой машине будут храниться клиенты, фамилии которых начинаются с одной и той же буквы. Но эту стратегию не назовешь удачной – фамилий, начинающихся с «Q» или «Z», скорее всего, окажется немного, так что соответствующие им машины будет простаивать, тогда как машины, где хранятся фамилии на «J», «M» или «S», будут перегружены. Можно было бы сегментировать на основе какого-то числового поля, например номера телефона, даты регистрации или названия штата, в котором находится клиент. Все зависит от того, как распределены конкретные данные.

Существуют три основные стратегии определения структуры сегмента.

Сегментирование по признакам, или функциональное сегментирование

Этот подход принял Рэнди Шоуп, почетный архитектор из компании eBay, который в 2006 году помог привести архитектуру сайта к современному состоянию, позволяющему обрабатывать миллиарды запросов ежедневно. При такой стратегии на части разбивается не одна таблица (как в рассмотренном выше примере таблицы клиентов), а выделяются отдельные базы данных, содержащие слабо перекрывающиеся признаки. Так, в eBay пользователи находятся в одном сегменте, а данные о продажах – в другом. На сайте Flixster в одном сегменте хранятся рейтинги фильмов, а в другом – комментарии. Этот подход требует ясного понимания предметной области.

Сегментирование по ключам

При таком подходе мы пытаемся найти в данных ключ, позволяющий равномерно распределить их по сегментам. Так, вместо того чтобы хранить на каждом сервере клиентов, фамилии которых начинаются с одной и той же буквы (наивное и неправильное решение), мы применим к данным ключевого поля одностороннюю функцию хэширования и будем распределять записи по машинам в соответствии с ее значением. Часто в этом случае для хэширования выбирается какое-нибудь числовое поле или поле, содержащее временную метку.

Справочная таблица

В этом случае один из узлов кластера выступает в роли справочника «желтые страницы», позволяя узнать, на каком узле хранятся интересующие нас данные. У такого подхода два очевидных недостатка. Во-первых, снижение производительности при каждом дополнительном обращении к справочной таблице. А во-вторых, справочная таблица становится не только узким местом, но и точкой общего отказа.

В зависимости от удачности выбранной стратегии сегментирование может уменьшить конкуренцию и позволяет не просто масштабировать базу данных по горизонтали, а сделать это более точно, чтобы можно было нарастить вычислительные ресурсы именно для тех сегментов, которые больше всего нуждаются в этом.

Сегментирование можно было бы назвать «архитектурой без разделения ресурсов» в применении к базам данных. Архитектурой *без разделения ресурсов* называется такая архитектура, в которой нет централизованного (разделяемого) состояния, а каждый узел распределенной системы независим, т. е. клиенты не конкурируют за разделяемые ресурсы. Термин пустил в обращение Майкл Стоунбрейкер из Калифорнийского университета в Беркли в статье 1986 года «The Case for Shared Nothing».

В последнее время архитектура без разделения ресурсов популяризуется компанией Google, разработавшей базу данных Bigtable и собственную реализацию технологии MapReduce, в которых нет никакого разделяемого состояния, т. е. допускается почти безграничное масштабирование. База данных Cassandra также построена на основе архитектуры без разделения ресурсов, поскольку не имеет центрального контроллера; ей чуждо понятие главного и подчиненного узла – все узлы равноправны.



Еще об архитектуре без разделения ресурсов

Статью 1986 года «The Case for Shared Nothing» можно прочитать по адресу <http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf>. Она занимает всего несколько страниц. Ознакомившись с ней, вы поймете, что многие характерные черты распределенной архитектуры данных без разделения ресурсов, в частности высокая доступность и способность к масштабированию на очень большое число компьютеров, – как раз то, чем славится Cassandra.

База данных MongoDB также предлагает средства автоматического сегментирования для управления отработкой отказов и балансирования нагрузки между узлами. Тот факт, что многие нереляционные базы данных предоставляют подобный механизм автоматически и в готовом виде, очень удобен, поскольку создание и обслуживание сегментов вручную – занятие неблагодарное. Важно понимать общее место сегментирования в архитектуре данных, и особенно в случае Cassandra, поскольку эта база допускает подход, аналогичный сегментированию по ключам, но в автоматическом режиме.

Масштаб веба

Короче говоря, реляционные базы данных очень хороши для решения некоторых задач хранения данных, но могут порождать проблемы при попытке масштабирования. Зачастую приходится искать способы избавления от операции соединения, что означает денормализацию, а это влечет за собой проблемы управления несколькими экземплярами данных и серьезно подрывает принципы проектирования как самой базы, так и приложения. Кроме того, почти наверняка придется как-то обходить распределенные транзакции, которые быстро становятся узким местом. Компенсационные действия если и поддерживаются напрямую, то разве что в самых дорогих РСУБД. И даже если вы в состоянии справиться с такой необъятной задачей, все равно нужно будет тщательно выбирать ключи разбиения, так что в результате полностью игнорировать ограничение все равно не получится.

Но важнее то, что в ходе рассмотрения ограничений РСУБД и стратегий, применяемых для преодоления трудностей масштабирования, начинает вырисовываться некая общая картина. И на ее фоне решения, принятые в технологиях NoSQL, кажутся уже не такими радикальными и пугающими, как поначалу, а выглядят как естественное выражение и инкапсуляция работы, которая уже и так делается для управления очень большими базами данных.

Из-за некоторых проектных решений, внутренне присущих РСУБД, масштабировать их не так просто, как другие системы, появившиеся недавно и принимающие в расчет структуру веба. Однако учитывать нужно не только структуру веба, но и феноменальные темпы его роста, потому что архитектура должна позволять компании справляться с постоянно растущим объемом данных почти в режиме реального времени, чтобы поддерживать принятие решений и предлагать все новые и новые возможности клиентам.



Масштаб данных раньше и теперь

Говорят, хотя проверить это очень трудно, что английский поэт XVII века Джон Мильтон прочитал все опубликованные к тому времени книги. Мильтон знал много языков (незадолго о смерти он даже начал изучать язык индейцев Навахо), и, принимая во внимание, что в то время насчитывалось несколько тысяч опубликованных книг, это вполне возможно. С тех пор объем хранимых во всем мире данных несколько подрос.

Вместе с быстрым ростом веба растет и многообразие данных, нуждающихся в хранении, обработке и поиске, а также разнообразие приложений, в которых эти данные используются. Учитывать нужно не только знакомые данные о клиентах, необходимые компаниям оптовой и розничной торговли, и не только цифровое видео, но и быстрый переход к цифровому телевидению, а также лавинообразный рост технологий электронной почты, мессенджеров, мобильной связи, радиометок (RFID), интернет-телефонии (VoIP) и Интернета вещей. Компаниям, поставляющим контент, а также формирующимися вокруг них компаниям, предоставляющим дополнительные услуги, необходимы высокомасштабируемые решения для хранения данных. Вспомните также, что типичный разработчик коммерческих приложений или администратор базы данных привык считать реляционные базы центром вселенной. И будет весьма удивлен, узнав, что 80% корпоративных данных не структурированы.

Восхождение NoSQL

Недавно возникший интерес к нереляционным базам данных отражает растущее в недрах сообщества разработчиков ПО осознание потребности в решениях масштаба веба. Сам термин «NoSQL» стал популярен примерно в 2009 году как краткая характеристика такого рода баз данных. О нем много спорили, но в результате сложилось

общее мнение: термин относится к нереляционным базам данных, которые поддерживают «не только семантику SQL».

Были предложены различные классификации таких баз данных. Мы рассмотрим лишь несколько наиболее часто встречающихся категорий.

Хранилища ключей и значений

В этом случае элемент данных имеет ключ, с которым ассоциировано множество атрибутов. Все данные, ассоциированные с ключом, хранятся вместе с ним. Данные часто дублируются. К популярным хранилищам ключей и значений можно отнести Dynamo DB компании Amazon, Riak и Voldemort. Кроме того, в роли хранилищ ключей и значений могут выступать многие популярные средства кэширования, в том числе Oracle Coherence, Redis и MemcacheD.

Столбцовые хранилища

Прототипом столбцовых хранилищ (или «хранилищ с широкими столбцами») является база данных Bigtable, созданная Google. К этой категории относятся Cassandra, Hypertable и HBase из проекта and Apache Hadoop.

Документные хранилища

Основной единицей хранения в документной базе данных является целый документ, часто он представляется в формате JSON, XML или YAML. Из популярных документных хранилищ отметим MongoDB и CouchDB.

Графовые базы данных

В графовой базе данные представлены в виде графа множества вершин и соединяющих их ребер. С вершинами и ребрами могут быть ассоциированы свойства. Поскольку в графовых базах, в частности FlockDB, Neo4J и Polyglot, основной упор делается на связях, они нашли применение при построении социальных сетей и семантического веба.

Объектные базы данных

В объектных базах данные хранятся не в виде отношений, состоящих из строк и столбцов, а в виде объектов, что упрощает работу с базой из объектно-ориентированного приложения. Такие объектные базы, как db4o и InterSystems Cache, позволяют избежать использования хранимых процедур и средств объектно-реляционного отображения (ORM).

Базы данных XML

База данных XML – это частный случай документной базы данных, оптимизированный для работы с XML. К числу «XML-ориентированных» баз данных относятся Tamino компании Software AG и eXist.

Полный перечень баз данных NoSQL см. на сайте <http://nosql-database.org>.

У таких баз разные цели и возможности, но можно выделить ряд общих характеристик. Самая очевидная следует из самого названия NoSQL – эти базы поддерживают модели данных, языки определения данных (DDL) и интерфейсы, не ограничивающиеся стандартными средствами SQL, которые присутствуют в популярных реляционных базах данных. Кроме того, обычно они представляют собой распределенные системы, не имеющие централизованного управления. На первое место в них выходят горизонтальная масштабируемость и высокая доступность, для чего иногда приходится жертвовать строгой согласованностью и семантикой ACID. Часто поддерживается быстрая разработка и развертывание. Подход к определению схемы гибкий, иногда вообще не требуется заранее определять схему. Поддерживаются большие данные и аналитические средства.

За последние несколько лет появилось немало баз данных NoSQL – как коммерческих, так и с открытым исходным кодом. Их распространенность и качество сильно варьируются, но в каждой из вышеупомянутых категорий обозначились лидеры с достаточно зрелыми технологиями, большим количеством внедрений и наличием коммерческой поддержки. Мы рады сообщить, что к таким технологиям принадлежит и Cassandra, которую мы начнем изучать в следующей главе.

Резюме

Реляционная модель верой и правдой служила индустрии программного обеспечения свыше сорока лет, но уровень доступности и масштабируемости, необходимый современным приложениям, превысил ее возможности.

Цель этой книги состоит не в том, чтобы хитроумными доводами склонить вас к принятию нереляционной базы данных, каковой является Apache Cassandra. Мы лишь хотим показать, что и как может делать Cassandra, чтобы вы могли принять обоснованное решение и приступить к практической работе с этой системой, если сочтете, что овчинка стоит выделки.

Таким образом, главный вопрос – не «Что не так с реляционными базами данных?», а «Что бы я стал делать с данными, если бы не эта проблема?». В мире, который уже функционирует в масштабе веба и нацелен в будущее, Apache Cassandra может стать частью ответа на этот вопрос.

Глава 2

• • • • • • • • • • • • • • • • • • • • • • • • •

Введение в Cassandra

Изобретение должно иметь смысл в том мире,
где оно завершено, а не в том, где начато.

— Рэй Курцвейл

В предыдущей главе мы говорили о возникновении технологий нереляционных баз данных в ответ на растущие требования со стороны современных приложений масштаба веба. Эту главу мы посвятим рассмотрению основных принципов и возможностей системы Cassandra, благодаря которым она способна принять этот вызов. Вы также узнаете об истории Cassandra и о том, как стать членом сообщества, развивающего эту базу данных.

Краткая презентация Cassandra

Голливудским сценаристам и программным стартапам часто рекомендуют иметь наготове краткую презентацию, которую можно было изложить за время поездки в лифте. Это краткое описание продукта — лаконичное, четкое и рассчитанное на одну-две минуты — на тот счастливый случай, если автор окажется в одном лифте с исполнительным продюсером, агентом или инвестором, который мог бы вложить деньги в проект. У проекта Cassandra захватывающая история, и мы сведем ее к краткой презентации, которую можно будет при удобном случае пересказать начальнику или коллегам.

Cassandra в 40 словах

«Apache Cassandra — открытая, распределенная, децентрализованная, эластично масштабируемая, высокодоступная, отказоустойчивая,

строковая, допускающая настройку согласованности база данных, дизайн распределенности которой основан на Amazon Dynamo, а модель данных – на Google Bigtable. Разработана в Facebook, ныне используется на некоторых из наиболее популярных веб-сайтах». Ровно 40 слов.

Разумеется, если вы выпалите это в лицо шефу в лифте, то, скорее всего, получите в ответ недоуменный взгляд. Поэтому в следующих разделах развернем каждое положение.

Распределенная и децентрализованная

Cassandra – *распределенная* база данных, т. е. может работать на нескольких машинах, которые пользователям представляются как единое целое. На самом деле запускать Cassandra на одном узле вообще не имеет особого смысла. Это, конечно, возможно и даже полезно, чтобы составить представление о ее быстродействии, но очень скоро вы поймете, что по-настоящему воспользоваться преимуществами Cassandra можно только при наличии нескольких машин. Мало того что система специально проектировалась и писалась для работы на многих компьютерах, она еще и оптимизирована с учетом наличия нескольких стоек в центре обработки данных и даже для случая одного кластера Cassandra, размещенного в территориально разнесенных ЦОДах. Мы можем записывать данные на любом узле кластера и быть уверены, что Cassandra их получит.

При масштабировании многих других хранилищ данных (MySQL, Bigtable) некоторые узлы должны быть назначены главными, а все остальные – подчиненными. Но Cassandra является *децентрализованной* базой данных в том смысле, что все узлы равноправны – никакой узел не выполняет организационных операций, которых не выполняли бы все остальные узлы. В Cassandra используется одноранговый протокол, а для поддержания актуального списка «живых» и «мертвых» узлов применяется механизм распространения сплетен.

Децентрализация Cassandra означает отсутствие точки общего отказа. Все узлы кластера Cassandra функционируют одинаково. Иногда это свойство называют «серверной симметрией». Поскольку все узлы делают одно и то же, по определению не может быть никакого специального узла, который координировал бы их работу, как в конфигурации главный–подчиненный, характерной для MySQL, Bigtable и многих других баз данных.

Во многих распределенных решениях (например, в кластерах РСУБД) мы настраиваем несколько копий данных на разных серверах

рах. Процесс копирования данных на несколько машин называется репликацией; его смысл в том, что обрабатывать запросы может любой сервер, и тем самым повышается производительность. Обычно этот процесс не является децентрализованным, как в Cassandra, для него нужно определить *связь главный–подчиненный* между серверами. Это означает, что не все серверы кластера работают одинаково. Мы настраиваем кластер, назначая один сервер главным, а остальные – подчиненными. Главный сервер служит авторитетным источником данных, между ним и подчиненными серверами существует односторонняя связь – подчиненные серверы должны синхронизировать свои копии данных. Если главный сервер выходит из строя, вся база данных оказывается под угрозой. Таким образом, децентрализация – одна из важнейших причин высокой доступности Cassandra. Отметим, что хотя мы привыкли к репликации главный–подчиненный в мире РСУБД, она применяется и в базах данных NoSQL, например в MongoDB.

Итак, у децентрализации два основных достоинства: она проще репликации главный–подчиненный и помогает избежать простоев. Эксплуатировать и обслуживать децентрализованное хранилище проще, потому что все узлы одинаковы. Поэтому для организации масштабирования не нужно никаких специальных знаний; настроить 50 узлов ничуть не сложнее, чем один. Конфигурировать почти ничего не придется. Наконец, в конфигурации главный–подчиненный главный сервер может оказаться точкой общего отказа. Чтобы предотвратить это, часто приходится усложнять систему, включая несколько главных серверов. В Cassandra же все реплики идентичны, поэтому выход из строя любого узла не приводит к прерыванию работы.

Короче говоря, благодаря распределенности и децентрализации в Cassandra нет точек общего отказа, что обеспечивает высокую доступность.

Эластичная масштабируемость

Масштабируемостью называется архитектурное свойство системы, позволяющее ей обрабатывать большее число запросов с малой потерей производительности. Проще всего добиться этой цели с помощью вертикального масштабирования – механического увеличения пропускной способности оборудования и добавления памяти в уже имеющуюся машину. Под горизонтальным масштабированием понимается увеличение числа машин, на каждой из которых размещаются все данные или их часть, так что ни одна отдельная машина не долж-

на обрабатывать все поступающие запросы. Но тогда в программное обеспечение должен быть встроен механизм синхронизации данных между узлами кластера.

Эластичной масштабируемостью называют специальное свойство горизонтальной масштабируемости. Это означает, что размер кластера может легко увеличиваться и уменьшаться. Для этого кластер должен уметь включать в себя новые узлы, которые получают данные полностью или частично и начинают обслуживать запросы без прерывания работы и кардинальной перенастройки всего кластера. Не требуется перезапускать никакие процессы. Не нужно изменять запросы в приложении. Нет нужды самостоятельно заниматься перемещением данных между узлами. Достаточно просто добавить компьютер – Cassandra найдет его и станет направлять ему запросы.

Понижающее вертикальное масштабирование означает изъятие из кластера части вычислительных мощностей. Иногда это вызвано коммерческими соображениями, например в связи с сезонным изменением нагрузки в приложениях розничной торговли или бронирования туров. Но могут быть и технические причины, например перенос частей приложения на другую платформу. Несмотря на все наши усилия, подобные вещи все-таки случаются. Но и в таком случае мы не хотим, чтобы все развалилось.

Высокая доступность и отказоустойчивость

Вообще говоря, архитектурно доступность определяется способностью системы обслуживать запросы. Но компьютеры подвержены самым разным сбоям, от отказа аппаратных компонентов до недоступности сети и повреждения данных. Выйти из строя может любой компьютер. Конечно, существуют «навороченные» (и баснословно дорогие) компьютеры, способные самостоятельно справляться со многими отказами, поскольку в них предусмотрены резервирование оборудования, средства отправки уведомлений о сбоях и горячая замена компонентов. Но кто угодно может случайно разрезать кабель Ethernet, а центр обработки данных не застрахован от стихийных бедствий. Поэтому для обеспечения высокой доступности система должна включать несколько объединенных в сеть компьютеров, а исполняемые программы должны уметь работать в кластере, распознавать отказы узлов и перенаправлять запросы другим узлам.

Cassandra является высокодоступной системой. Отказавшие узлы кластера можно заменять без простоя, а данные реплицировать на несколько ЦОДов с целью повышения локальной производительности

и предотвращения простоя в случае стихийного бедствия – например, пожара или наводнения – в одном ЦОДе.

Настраиваемая согласованность

По существу, *согласованность* означает, что операция чтения всегда возвращает последнее записанное значение. Рассмотрим случай, когда два покупателя интернет-магазина пытаются поместить в корзину один и тот же товар. Если осталась только одна единица товара и я пытаюсь поместить ее в корзину после вас, то в вашей корзине товар должен оказаться, а я должен получить сообщение о том, что товар закончился. Именно так и произойдет, если состояние после операции записи согласовано на всех узлах, где хранятся данные.

Но, как мы вскоре увидим, масштабирование хранилища данных предполагает определенные компромиссы между согласованностью, доступностью узлов и устойчивостью к разбиению. Cassandra часто называют «согласованной в конечном счете», хотя этот термин не вполне точен. Без дополнительной настройки Cassandra жертвует полной согласованностью ради достижения высокой доступности. Однако правильнее говорить, что Cassandra допускает «настраиваемую согласованность», т. е. позволяет выбрать необходимый уровень согласованности и соответствующий уровень доступности.

Задержимся здесь ненадолго, потому что термин «согласованность в конечном счете» стал причиной ожесточенных споров в индустрии. Некоторые специалисты-практики с опаской относятся к «согласованной в конечном счете» системе.

Оппоненты согласованности в конечном счете рассуждают следующим образом: для социальных сетей, где точные данные не так уж и важны, согласованность в конечном счете, может, и хороша. Ну действительно, что страшного случится, если отправленное маме сообщение о том, что малыш Билли съел на завтрак, потеряется? Но *мои-то* данные действительно важны, и я даже на секунду не могу допустить согласованность в конечном счете в своей модели.

Но примем во внимание тот факт, что все самые популярные веб-приложения (Amazon, Facebook, Google, Twitter) пользуются этой моделью, так что в ней, наверное, что-то все-таки есть. Надо полагать, данные очень важны для компаний, эксплуатирующих эти приложения, поскольку данные – их основной продукт, а сами компании стоят миллиарды долларов и обслуживают миллиарды пользователей в мире, где конкуренция очень высока. Может быть, и удалось бы обеспечить гарантированную идеальную согласованность в высоко-

нагруженной системе, которая параллельно выполняет запросы, поступающие по самым разным сетям, но если требуется, чтобы клиент получил результат сейчас, а не в будущем году, то сделать это будет ой как нелегко.

Оппоненты говорят, что некоторые базы для работы с большими данными, например Cassandra, реализуют всего лишь согласованность в конечном счете, тогда как все остальные распределенные системы обеспечивают *строгую* согласованность. Но реальный мир редко бывает черно-белым, и противопоставление согласованного и несогласованного на практике не совсем верно. Существует *степень* согласованности, которая в реальности сильно зависит от внешних обстоятельств.

Согласованность в конечном счете – это одна из нескольких моделей согласованности, имеющихся в распоряжении архитектора. Рассмотрим эти модели и оценим присущие им компромиссы.

Строгая согласованность

Иногда встречается также название «последовательная согласованность», и это самый обязывающий уровень. Требуется, чтобы любая операция чтения всегда возвращала самое последнее записанное значение. Звучит замечательно, как раз то, что мне нужно. Беру! Но давайте приглядимся внимательнее. Что точно понимается под словами «самое последнее записанное значение»? Самое последнее для кого? В одном компьютере с одним процессором такую операцию нетрудно наблюдать, потому что вся последовательность операций известна с точностью до такта. Но в системе, которая работает в нескольких территориально удаленных ЦОДах, все далеко не так определенно. Чтобы определить последнюю операцию, нужны какие-то глобальные часы, способные проставлять для всех операций временные метки вне зависимости от местоположения данных, отправившего запрос пользователя и количества (возможно, разнородных) сервисов, необходимых для получения ответа.

Причинно-следственная согласованность

Это несколько ослабленная форма строгой согласованности. Мы отказываемся от фантастических глобальных часов, которые могут волшебным образом синхронизировать все операции, не создавая нетерпимых узких мест. Вместо временных меток используется семантический подход – предпринимается попытка определить причины событий и тем самым в какой-то мере согласовать их по

порядку. Это означает, что результаты потенциально взаимосвязанных операций записи должны читаться в том же порядке. Если две не связанные между собой операции одновременно пишут значение в одно и то же поле, то считается, что между ними нет причинно-следственной связи. Но если одна операция следует за другой, то мы должны предположить наличие причинно-следственной связи между ними. Причинно-следственная согласованность означает, что результаты таких операций записи должны читаться в том же порядке.

Слабая (в конечном счете) согласованность

Согласованность в конечном счете означает, что все обновления рано или поздно распространяются во все реплики, имеющиеся в распределенной системе, но для этого потребуется некоторое время. В конечном счете все реплики будут согласованы.

И вот результат – согласованность в конечном счете не кажется такой уж непривлекательной на фоне условий, необходимых для достижения более строгих форм согласованности.

В любой распределенной системе можно достичь лишь двух из трех целей: согласованности, доступности и устойчивости к раздлению. Это утверждение называется теоремой CAP (ее смысл мы рассмотрим более подробно в следующем разделе). Суть проблемы состоит в репликации обновлений данных. Для достижения строгой согласованности все операции обновления должны выполняться синхронно, т. е. необходимо заблокировать все реплики до завершения операции и заставить ждать одновременно работающих клиентов. Побочным эффектом такого решения является тот факт, что в случае отказа часть данных будет полностью недоступна в течение всего времени до его устранения. Технический директор Amazon Верне Фогельс говорит так: «вместо того чтобы иметь дело с неопределенностью относительно правильности ответа, данные делаются недоступными до того момента, когда можно будет уверенно сказать, что они корректны»¹.

Можно было бы вместо этого принять оптимистический подход к репликации и в фоновом режиме распространять обновления всем репликам, чтобы избежать зависания клиента. Но с этим подходом связана другая трудность: теперь мы должны обнаруживать и раз-

¹ «Dynamo: Amazon’s Highly Distributed Key-Value Store» (http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html), 2007.

решать конфликты. И при проектировании нужно решить, когда это делать: во время чтения или во время записи. То есть проектировщик распределенной базы данных должен выбрать одно из двух: система всегда допускает чтение или всегда допускает запись.

В случае Dymamo и Cassandra выбрана допустимость записи, а сложности согласования перенесены на этап чтения. Это дает огромный выигрыш в производительности. Альтернатива – отвергать обновления, произошедшие во время отказа сети или сервера.

В Cassandra мы не обязаны принимать согласованность в полном объеме или вообще отказываться от нее. Точнее, это свойство называется «настраиваемой согласованностью», потому что клиент может контролировать, сколько реплик блокируется на время операций обновления. Для этого задаются уровень согласованности и коэффициент репликации.

Коэффициент репликации определяет, какой частью производительности мы готовы пожертвовать ради большей согласованности. Он задается в терминах числа узлов кластера, на которые должны быть распространены обновления (напомним, что под обновлениями понимаются операции добавления, обновления и удаления).

Уровень согласованности – это параметр, который клиент должен задать для каждой операции; он определяет, сколько реплик в кластере должны подтвердить операцию записи или ответить на операцию чтения, чтобы она была признана успешной. Это та часть решения о согласованности, которую Cassandra отдает на откуп клиенту.

Итак, мы могли бы задать уровень согласованности равным коэффициенту репликации и тем самым получить более строгую согласованность ценой синхронного выполнения операций, когда мы не возвращаем управление, пока все узлы не обновятся и не сообщат об успехе. На практике так поступают редко по причинам, которые теперь должны быть понятны (мешает достижению высокой доступности, негативно отражается на производительности и вообще идет вразрез с целями, ради которых было принято решение использовать Cassandra). Поэтому клиент устанавливает уровень согласованности меньшим, чем коэффициент репликации, и тогда обновление считается успешным, даже если некоторые узлы не работают.

Теорема CAP Брюера

Чтобы понять дизайн Cassandra и почему ее называют «согласованной в конечном счете», нужно усвоить теорему CAP, которую иногда называют теоремой Брюера по имени автора, Эрика Брюера.

Сотрудник Калифорнийского университета в Беркли Эрик Брюер сформулировал теорему CAP в 2000 году на симпозиуме ACM по принципам распределенных вычислений. Теорема касается трех взаимосвязанных требований, существующих в большой распределенной системе данных.

Согласованность

Все клиенты базы данных должны прочитать одно и то же значение в ответ на один и тот же запрос, даже если одновременно производятся обновления.

Доступность

Все клиенты базы данных всегда имеют возможность читать и записывать данные.

Устойчивость к разделению

Базу данных можно разделить между несколькими машинами; она может продолжать работу даже в случае отказа сегментов сети.

Теорема Брюера утверждает, что в любой системе можно гарантированно обеспечить выполнение только двух из этих трех требований. Это аналог поговорки, распространенной среди разработчиков программного обеспечения: «программа может быть хорошей, может быть быстрой и может быть дешевой: выбирай любые два свойства». Выбирать приходится потому, что между свойствами существует зависимость. Например, чем большая согласованность требуется от системы, тем меньше будет ее устойчивость к разделению, если только мы не готовы пойти на уступки в части доступности.

Формальное доказательство теоремы CAP дали Сет Гильберт (Seth Gilbert) и Нэнси Линч (Nancy Lynch) из МТИ в 2002 году. Но в распределенной системе весьма вероятны сетевые сбои, в результате которых часть машин оказывается недоступной. Сетевые проблемы, в частности потеря пакетов или большая задержка, почти неизбежны и могут вызывать временное разделение сети на части. Это приводит нас к выводу, что распределенная система должна сделать все возможное для продолжения работы в условиях разделения сети (чтобы быть устойчивой к разделению), а значит, для компромисса остаются только два свойства: доступность и согласованность.

На рис. 2.1 видно, что нет ни одной области, в которой выполнялись бы все три свойства.



Рис. 2.1 ♦ Теорема CAP говорит, что можно добиться одновременного выполнения лишь двух из трех свойств

Сейчас будет полезно посмотреть, в какой части спектра CAP оказываются нереляционные хранилища данных, которые мы будем обсуждать. Рисунок 2.2 подсказан слайдом, который Дуайт Мерри-мэн, генеральный директор и основатель MongoDB, продемонстрировал группе пользователей MySQL в Нью-Йорке (<http://leadit.databasemonth.com/hands-on-tech/MongoDB-High-Performance-SQL-Free-Database>). Правда, мы изменили место некоторых систем в соответствии с результатами собственных исследований.

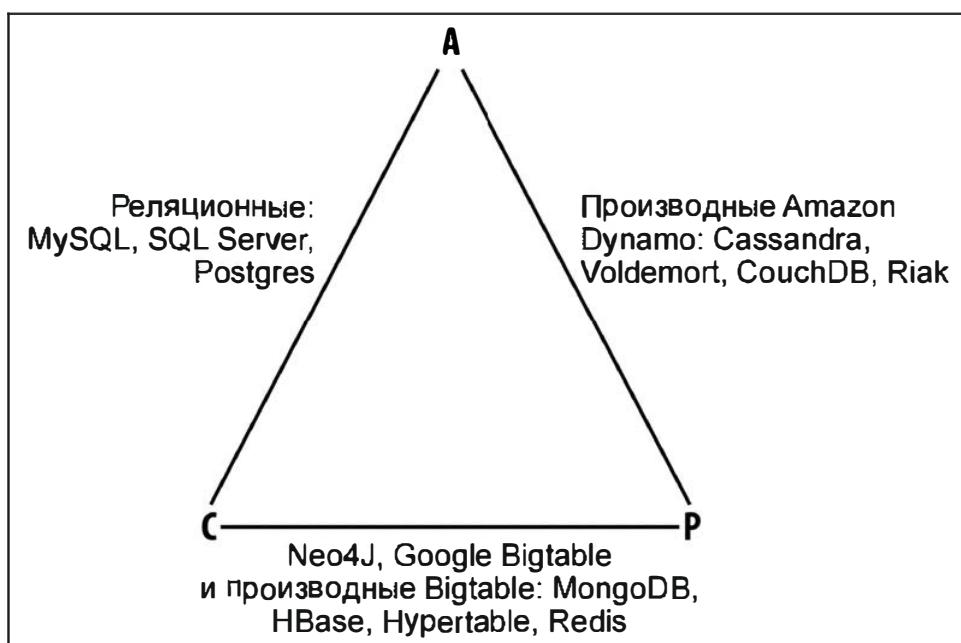


Рис. 2.2 ♦ Местоположение различных баз данных в спектре CAP

На рис. 2.2 изображено положение некоторых баз данных, обсуждаемых в этой главе. Отметим, что в зависимости от конфигурации база данных может перемещаться из одного места в другое. Как отметил Стюарт Худ, распределенную базу данных MySQL можно считать согласованной системой, только если применены исправления Google, относящиеся к синхронной репликации; в противном случае она является всего лишь доступной и устойчивой к разделению (AP).

Интересно отметить, что место системы в треугольнике CAP не зависит от характера механизма хранения данных; например, вдоль ребра CP расположены как графовые, так и документоориентированные базы данных.

На этом рисунке реляционные базы данных занимают место между согласованностью и доступностью, а это означает, что они могут выходить из строя в случае отказа сети (в том числе из-за разрыва кабеля). Обычно это вызвано либо наличием единственного главного сервера, который сам может «упасть», либо массива серверов, в который не встроены механизмы, позволяющие продолжать функционирование после разделения сети.

Графовые базы данных типа Neo4J и семейство баз данных, по крайней мере частично наследующих дизайн базы Google Bigtable (MongoDB, HBase, Hypertable и Redis), в меньшей степени ориентированы на доступность и в большей – на обеспечение согласованности и устойчивости к разделению.

Наконец, к числу баз данных, производных от Amazon Dynamo, относятся Cassandra, Project Voldemort, CouchDB и Riak. Они больше ориентированы на обеспечение доступности и устойчивости к разделению. Но это не значит, что согласованность считается чем-то несущественным. Точно так же и Bigtable не игнорирует согласованность. Согласно данным, приведенным в статье о Bigtable, средняя доля времени работы сервера, в течение которого «часть данных» была недоступна, составляет 0,0047 % (раздел 4), это не так уж много, учитывая, что речь идет об очень надежных системах. Если считать каждую из букв C, A, P кнопкой, на которую можно нажать и получить систему с желаемыми характеристиками, то производные Dynamo предназначены для эксплуатации в условиях, когда «согласованность в конечном счете» допустима, при этом время до достижения согласованности измеряется миллисекундами, исправление на этапе чтения означает, что операция чтения возвращает согласованные значения, и при желании можно обеспечить строгую согласованность.

Рассмотрим, что на практике означает поддержка только двух свойств CAP.

CA

Приоритет поддержке согласованности и доступности означает, что для распределенных транзакций, вероятно, используется двухфазная фиксация. Следовательно, система блокируется в случае разделения сети, а стремление сгладить эту проблему приводит к тому, что система может работать только в кластере, размещенном в одном ЦОДе. Если вашему приложению достаточно такого уровня масштабирования, то администрирование упрощается, и вы сможете полагаться на простые, хорошо знакомые структуры.

CP

Отдавая приоритет поддержке согласованности и устойчивости к разделению, вы можете попытаться усложнить архитектуру, введя в нее сегментирование. Данные будут согласованы, но остается риск недоступности некоторых данных в случае отказа узлов.

AP

Система, ориентированная на поддержку прежде всего доступности и устойчивости к разделению, может возвращать неточные данные, зато всегда будет доступна, даже в случае разделения сети. Пожалуй, самым известным примером системы, которая масштабируется в очень широких пределах, высокодоступна и устойчива к разделению, является система доменных имен DNS.

Отметим, что рисунок выше призван дать лишь общую картину, описывающую приблизительные границы различных классов систем, не стоит ждать от него точности. Например, вовсе не очевидно, в какое место следует поместить базу данных Google Bigtable. В статье Google эта база описывается как «высокодоступная», но позже говорится, что если Chubby (служба постоянных блокировок, являющаяся частью Bigtable) «оказывается недоступна в течение длительного периода времени [в результате ошибки в Chubby или сетевых проблем], то Bigtable станет недоступна» (раздел 4). По вопросу операций чтения данных в статье сказано, что «мы не рассматриваем возможность нескольких копий одних и тех же данных, в том числе в различных формах, вследствие наличия представлений или индексов». Наконец, в статье отмечается, что «перед Bigtable не ставилась цель реализовать централизованное управление или византийскую отказоустойчивость» (раздел 10). Понятно, что при наличии такой

противоречивой информации определение места базы данных в треугольнике CAP – не точная наука.

Новый взгляд на CAP

В феврале 2012 года Эрик Брюер предложил по-новому взглянуть на теорему CAP, опубликовав в журнале «Computer», издаваемом IEEE, статью «CAP Twelve Years Later: How the ‘Rules’ Have Changed»¹. Теперь Брюер считает аксиому «два из трех» вводящей в заблуждение. Он отмечает, что при наличии разделения проектировщик должен пожертвовать только согласованностью или доступностью и что прогресс технологий восстановления после разделения дал возможность одновременно достичь высокого уровня согласованности и доступности.

Говоря о прогрессе, следует, безусловно, включить использование в Cassandra таких механизмов, как вручение напоминаний и исправление на этапе чтения. Мы рассмотрим их в главе 6. Но важно понимать, что механизмы восстановления после разделения не являются непогрешимыми. Настраиваемая согласованность Cassandra по-прежнему играет важную роль, потому что позволяет системе эффективно работать в условиях, когда полностью предотвратить разделение сети невозможно.

Строковая база

Модель данных в Cassandra можно описать как разделенное строковое хранилище, в котором данные хранятся в разреженных многомерных хэш-таблицах. «Разреженность» означает, что в любой строке может быть один или несколько столбцов, но схожие строки не обязаны состоять из одинакового количества столбцов (как в реляционной модели). «Разделенность» означает, что у каждой строки имеется уникальный ключ, по которому можно обратиться к ее данным, и что эти ключи используются для распределения строк между несколькими хранилищами.



Строковые и столбцовые базы данных

Cassandra часто называют «столбцовой» базой данных, и это служит источником недоразумений. В столбцовых базах данные хранятся по столбцам, в отличие от реляционных баз, где данные хранятся по строкам. Недоразумение отчасти вызвано различием между API

¹ <https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed>.

самой базы данных и лежащего в ее основе хранилища на диске. Таким образом, Cassandra на самом деле не является столбцовой базой данных, поскольку данные не организованы в виде столбцов.

В реляционной модели хранения все столбцы таблицы определяются заранее, и для каждого столбца резервируется место вне зависимости от того, есть в нем данные или нет. Напротив, в Cassandra данные хранятся в многомерной отсортированной хэш-таблице. Данные, записываемые в столбец, сохраняются в отдельном элементе хэш-таблицы. Значения столбцов хранятся в согласованном порядке, а незаполненные столбцы пропускаются, что позволяет более эффективно использовать место на диске и обрабатывать запросы. Более подробно модель данных Cassandra рассматривается в главе 4.

Правда ли, что в Cassandra нет схемы?

Первые версии Cassandra были верны оригинальному техническому описанию Bigtable и поддерживали «бессхемную» модель данных, в которой новые столбцы можно было определять динамически. Бессхемные базы данных, например Bigtable и MongoDB, обладают тем преимуществом, что очень просто расширяются и позволяют эффективно обращаться к данным большого объема. Основной же их недостаток – трудность определения семантики и формата данных, что ограничивает возможности выполнения сложных запросов. Эти недостатки стали преградой на пути внедрения таких баз во многих проектах, особенно стартапах, которым повезло вырасти из первоначально гибких фирмочек в более сложно организованные предприятия с большим штатом разработчиков и администраторов.

Решением для таких пользователей стало появление языка Cassandra Query Language (CQL), который дает возможность определить схему с помощью синтаксиса, похожего на Structured Query Language (SQL) и знакомого специалистам, работавшим с реляционными базами. Поначалу CQL представлял собой просто еще один интерфейс к Cassandra наряду с бессхемным интерфейсом, основанным на проекте Apache Thrift. В течение переходного периода употреблялся термин «с необязательной схемой» для описания моделей данных, которые можно определить с помощью схемы на CQL, но допускали динамическое добавление новых столбцов средствами Thrift API. В это время базовое хранилище данных по-прежнему основывалось на модели Bigtable.

Начиная с версии 3.0 API для поддержки динамического создания столбцов, основанный на Thrift, объявлен нерекомендуемым, а базовое хранилище Cassandra переделано для лучшей совместимости

с CQL. Нельзя сказать, что Cassandra вообще не позволяет динамически расширять схему, но принцип работы значительно изменился. Коллекции CQL – списки, множества и особенно словари – представляют возможность добавлять данные в слабо структурированной форме и тем самым расширять существующую схему. CQL также позволяет в некоторых случаях изменять тип столбца и поддерживает хранение текста в формате JSON.

Поэтому сейчас правильнее всего звать Cassandra базой данных, поддерживающей «гибкие схемы».

Высокая производительность

Cassandra специально проектировалась для работы на десятках машин в нескольких ЦОДах и так, чтобы в полной мере использовались преимущества многопроцессорных и многоядерных компьютеров. Она легко масштабируется для обработки данных объемом сотни терабайтов. Продемонстрирована исключительно высокая производительность Cassandra при высокой нагрузке. Она стабильно показывает очень большое количество операций записи в секунду на простых стандартных компьютерах – все равно, на физических или виртуальных машинах. При добавлении новых серверов все желательные характеристики Cassandra сохраняются без потери производительности.

Как появилась Cassandra?

Хранилище данных Cassandra сейчас является проектом Apache с открытым исходным кодом (<http://cassandra.apache.org/>). Проект был начат в компании Facebook в 2007 году для решения задачи поиска по папкам «Входящие», так как компании приходилось иметь дело с огромными объемами данных, обработка которых с трудом поддавалась масштабированию традиционными методами. Точнее, требовалось обрабатывать огромное количество сообщений и инвертированные индексы сообщений, а также выполнять много одновременных операций чтения и записи с произвольным доступом.

Составилась группа под руководством Джеффа Хаммербахера (Jeff Hammerbacher), в которую на правах ведущих специалистов входили Авинаш Лакшман (Avinash Lakshman), Картик Ранганатан (Karthik Ranganathan) и инженер из группы поиска Прашант Малик (Prashant Malik). В июле 2008 года код был раскрыт в виде проекта на Google Code. В 2008 году, пока проект пребывал в этом статусе,

изменения в код могли вносить только сотрудники Facebook, поэтому вокруг проекта сформировалось очень узкое сообщество. В марте 2009 года произошла смена статуса – проект стал инкубаторным проектом Apache, а 17 февраля 2010 по результатам голосования он был повышен до проекта верхнего уровня. На странице Apache Cassandra Wiki опубликован список соавторов (<http://wiki.apache.org/cassandra/Committers>), многие из которых участвуют в проекте еще с 2010–2011 годов. Некоторые соавторы работают в таких компаниях, как Twitter, LinkedIn, Apple, другие являются независимыми разработчиками.



Статья, в которой Cassandra была впервые представлена миру

Основной работой о Cassandra стала статья Лашмана и Малика из компании Facebook «A Decentralized Structured Storage System» (<http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf>). Джонатан Эллис опубликовал комментарий к этой статье (<http://docs.datastax.com/en/articles/cassandra/cassandra-thenandnow.html>), соответствующий версии 2.0, отметив, какие изменения технология претерпела с момента перехода под эгиду Apache. Некоторые изменения мы подробно рассмотрим ниже.

Откуда взялось название Cassandra?

В греческой мифологии Кассандра – дочь троянского царя Приама и его жены Гекубы. Кассандра была такой красавицей, что бог Аполлон наделил ее даром предвидения будущего. Но она отвергла его ухаживания, и бог наложил на нее проклятье – она сможет предсказывать все, что случится, но ей никто не будет верить. Кассандра предвидела разрушение своей родной Трои, но оказалась не в силах предотвратить его. Распределенная база Cassandra названа в ее честь. Мы полагаем, что тут кроется шутливый намек на Дельфийского оракула, еще одного прорицателя, в честь которого названа база данных Oracle.

По мере роста интереса к Cassandra со стороны коммерческих организаций стала очевидна необходимость полноценной технической поддержки. В апреле 2010 года Джонатан Эллис, руководитель проекта Apache Cassandra, и его коллега Мэтт Пфейль создали сервисную компанию DataStax (первоначальное название Riptano). DataStax взяла на себе руководство развитием проекта Cassandra и его поддержку, приняв в штат нескольких соавторов.

DataStax предлагает бесплатные продукты, в т. ч. драйверы Cassandra для различных языков и инструментальные средства для разработки и администрирования. За отдельную плату предоставляются корпоративные версии сервера Cassandra и инструментов, интеграция с другими технологиями обработки данных и техническая поддержка. В отличие от других проектов с открытым исходным кодом, имеющих коммерческую составляющую, все изменения сначала вносятся в открытый код проекта Apache, а вскоре после выхода очередной версии включаются в коммерческую версию.

DataStax также поддерживает сайт Planet Cassandra (<http://www.planetcassandra.org/>), являющийся ресурсом для сообщества. Здесь можно ознакомиться с постоянно растущим перечнем коммерческих и академических организаций, использующих Cassandra (<http://www.planetcassandra.org/companies/>). Представлены компании из самых разных секторов экономики: финансовые услуги, телекоммуникации, образование, социальные сети, развлечения, маркетинг, розничная торговля, гостиничный бизнес, транспорт, здравоохранение, энергетика, филантропия, авиакосмическая промышленность, оборона, технологический сектор. Очень может быть, что вы найдете целый ряд примеров, близких к вашим потребностям.

История версий

Познакомившись с людьми и организациями, стоявшими у истоков Cassandra, посмотрим, как происходило развитие после получения статуса официального проекта Apache. Если вы раньше не сталкивались с Cassandra, не переживайте по поводу незнакомых концепций и терминов – в свое время мы обо всем расскажем. Можете вернуться к этому разделу позже, чтобы получить представление о путях развития Cassandra в прошлом и будущем. Если же вам уже доводилось работать с Cassandra, то эта краткая сводка поможет вспомнить, что и когда происходило.



Усовершенствования в части производительности и надежности

Ниже перечислены главным образом новые функциональные возможности, добавленные за время существования Cassandra. Но не следует забывать о постоянных и значительных улучшениях в части надежности и производительности чтения/записи.

Версия 0.6

Первая версия, выпущенная после того, как Cassandra из инкубаторного проекта Apache превратилась в проект верхнего уровня.

Первой версией в этой серии была 0.6.0 (апрель 2010), последней – 0.6.13 (апрель 2011). Основные функциональные возможности:

- интеграция с Apache Hadoop, позволяющая легко получать данные из Cassandra по технологии MapReduce;
- интегрированное кэширование строк, устранившее необходимость развертывать наряду с Cassandra другие технологии кэширования.

Версия 0.7

Первой версией в этой серии была 0.7.0 (январь 2012), последней – 0.7.10 (октябрь 2011). Основные функциональные возможности и улучшения:

- вторичные индексы, т. е. индексы по столбцам, не совпадающим с первичным ключом;
- поддержка больших строк, содержащих до двух миллиардов столбцов;
- оперативное изменение схемы, в т. ч. добавление, переименование и удаление пространств ключей и семейств столбцов в работающем кластере без перезапуска с помощью Thrift API;
- задания срока хранения (TTL) данных на уровне столбцов;
- для поддержки развертывания в нескольких ЦОДах реализована стратегия репликации NetworkTopologyStrategy, позволяющая задавать коэффициенты репликации для каждого ЦОДа и каждого пространства ключей;
- формат конфигурационных файлов изменен с XML на более удобочитаемый YAML.

Версия 0.8

С этой версии началось изменение Cassandra API в сторону внедрения CQL. Первой версией в этой серии была 0.8.0 (июнь 2011), последней – 0.8.10 (февраль 2012). Основные функциональные возможности и улучшения:

- добавлен новый тип данных, распределенный счетчик, допускающий увеличение или уменьшение;
- добавлена программа sstableloader для поддержки массовой загрузки данных в кластеры Cassandra;
- реализован кэш строк вне кучи, что позволило использовать память операционной системы вместо кучи виртуальной машины Java;
- добавлено параллельное уплотнение, позволившее реализовать многопоточное выполнение и контроль над темпом уплотнения файлов SSTable;

- улучшенные параметры настройки памяти позволили реализовать более гибкое управление размером таблиц в памяти.

Версия 1.0

В соответствии с общепринятой практикой нумерации версий это первая официальная версия, хотя многие компании и раньше использовали Cassandra в производственном режиме. Первой версией в этой серии была 1.0.0 (октябрь 2011), последней – 1.0.12 (октябрь 2012). Стремясь обеспечить промышленное качество, разработчики уделили внимание прежде всего повышению производительности и улучшению уже имеющихся функций:

- в язык CQL 2 добавлен ряд усовершенствований, в т. ч. возможность изменять таблицы и столбцы, поддержка счетчиков и срока хранения (TTL), а также возможность получить число элементов, удовлетворяющих условиям запроса;
- введена многоуровневая стратегия уплотнения как альтернатива первоначальной ступенчатой стратегии, что позволило ускорить чтение за счет увеличения числа операций ввода-вывода при записи;
- сжатие файлов SSTable настраивается на уровне таблицы.

Версия 1.1

Первой версией в этой серии была 1.1.0 (апрель 2012), последней – 1.1.13 (май 2013). Основные функциональные возможности и улучшения:

- в язык CQL 3 добавлены тип `timeuuid` и возможность создавать таблицы с составным первичным ключом, в т. ч. включающим кластерные столбцы. Кластерные ключи поддерживают семантику «*order by*», чтобы обеспечить возможность сортировки. Эту функциональность ждали давно и с нетерпением, теперь стало возможно создавать «широкие строки» средствами CQL;
- поддержка импорта и экспорта CSV-файлов в оболочке `cqlsh`;
- гибкие параметры, задаваемые на уровне таблицы, допускают хранение данных на SSD или магнитных дисках;
- заново реализован механизм изменения схемы, появилась возможность производить параллельные изменения, и повысилась надежность. Теперь схемы хранятся в таблицах в пространстве ключей `system`;
- механизм кэширования изменен и теперь допускает более простую настройку размеров кэшей;

- реализована утилита массовой загрузки из Hadoop, обеспечивающая эффективный экспорт данных из Hadoop в Cassandra;
- добавлена изоляция на уровне строк, гарантирующая, что при обновлении сразу нескольких столбцов никакая операция чтения не увидит комбинацию старых и новых значений.

Версия 1.2

Первой версией в этой серии была 1.2.0 (январь 2013), последней – 1.2.19 (сентябрь 2014). Основные функциональные возможности и улучшения:

- в язык CQL 3 добавлены типы коллекций (множества, списки и словари), подготовленные команды и двоичный протокол вместо Thrift;
- виртуальные узлы более равномерно распределяют данные между узлами кластера, что повышает производительность, особенно при добавлении и замене узлов;
- атомарные пакеты гарантируют, что все операции записи в пакете завершаются либо успешно, либо неудачно;
- пространство ключей system содержит таблицу local, в которой хранится информация о локальном узле, и таблицу peers, в которой описываются остальные узлы кластера;
- можно включить режим трассировки запросов, чтобы клиенты могли видеть взаимодействия между узлами во время операций чтения и записи. Трассировка дает ценную информацию о происходящем за кулисами и помогает разработчику оценить последствия выбора тех или иных проектных решений;
- большинство структур данных перенесено из кучи JVM в память операционной системы;
- с помощью политики обработки дисковых отказов можно гибко настраивать поведение, например удаление узла из кластера в случае отказа диска или попытку любой ценой получить данные из памяти, пусть даже устаревшие.

Версия 2.0

Версия 2.0 стала важной вехой в истории Cassandra, поскольку знаменовала кульминацию развития CQL, а также новый уровень зрелости продукта. В нее были включены средства, существенно повысившие производительность, а также произведена вычистка кода, чтобы оплатить накопившийся за 5 лет технический долг. Первой версией в этой серии была 2.0.0 (сентябрь 2014), последней – 2.0.16 (июнь 2015). Основные нововведения:

- добавлены облегченные транзакции по протоколу консенсуса Paxos;
- реализован ряд улучшений CQL 3; добавлены семантика DROP в команде ALTER, условная модификация схемы (IF EXISTS, IF NOT EXISTS) и возможность создавать вторичные индексы по столбцам, входящим в состав первичного ключа;
- платформенные зависимости в реализации протокола CQL позволили продемонстрировать более высокую производительность по сравнению с Thrift;
- добавлена реализация прототипа триггеров, дающая расширяемый способ реагировать на операции записи. Триггеры можно писать на любом языке, работающем на платформе JVM;
- впервые стала обязательной версия Java 7;
- в версии 2.0.6 добавлены статические столбцы.

Версия 2.1

Первой версией в этой серии была 2.1.0 (сентябрь 2014), последней – 2.1.8 (июнь 2015). Основные функциональные возможности и улучшения:

- в CQL 3 добавлены пользовательские типы (UDT) и возможность создавать вторичные индексы над коллекциями;
- добавлены конфигурационные параметры, позволяющие перемещать таблицы в память из кучи в память операционной системы;
- расширены средства настройки кэширования строк, так что стало возможно задавать количество кэшируемых строк на уровне раздела;
- счетчики полностью переработаны для повышения производительности и надежности.

Версия 2.2

В оригинальном плане выпуска версий, составленном разработчиками Cassandra, версия 2.2 не значилась. После серии 2.1 предполагались крупная переработка и последующий выпуск версии 3.0. Но из-за объема и сложности изменений было решено выпустить некоторые завершенные части отдельно, а более сложным изменениям дать «созреть». Версия 2.2.0 вышла в июле 2015 года, а дополнительные к ней запланированы вплоть до осени 2016-го. Основные функциональные возможности и улучшения:

- усовершенствования в CQL 3, включая поддержку ввода-вывода в формате JSON и пользовательских функций;

- начиная с этой версии, официально поддерживается операционная система Windows. Хотя Cassandra по-прежнему лучше всего работает в системах на базе Linux, улучшения файлового ввода-вывода и поддержки скриптов существенно упростили запуск Cassandra в Windows;
- реализована технология Date Tiered Compaction Strategy (DTCS), что позволило повысить производительность работы с временными рядами;
- реализовано ролевое управление доступом (role-based access control – RBAC), что повысило гибкость управления авторизацией.

Модель «тик-так» выпуска версий

В июне 2015 года разработчики Cassandra объявили о планах перехода на модель «тик-так» выпуска версий с целью повысить гибкость разработки и качество версий.

Эта модель, популяризируемая компанией Intel, первоначально предназначалась для проектирования микропроцессоров. Идея заключается в том, что попаременно изменяются архитектура процессора и технологический процесс. Подробнее об этом подходе можно прочитать на странице <http://www.intel.com/content/www/us/en/silicon-innovations/intel-tick-tock-model-general.html>.

Модель «тик-так» оказалась полезной и при разработке программного обеспечения. Начиная с версии Cassandra 3.0 версии с четными номерами содержат новые функциональные возможности и исправления некоторых ошибок, а в версиях с нечетными номерами только исправляются ошибки. При этом версии предполагается выпускать ежемесячно.

Версия 3.0 (новая функциональность – ноябрь 2015-го):

- базовый движок хранения переписан и теперь более точно соответствует конструкциям CQL;
- добавлена поддержка материализованных представлений (иногда их еще называют глобальными индексами);
- поддерживается версия Java 8;
- исключен командный интерфейс, основанный на Thrift.

Версия 3.1 (исправление ошибок – декабрь 2015-го)

Версия 3.2 (новая функциональность – январь 2016-го):

- переработан способ выделения места для файлов SSTable на нескольких дисках в конфигурации «просто пачка дисков»

(JBOD) с целью повысить надежность и производительность, а также поддержать резервное копирование и восстановление отдельных дисков;

- добавлена возможность сжатия и шифрования напоминаний.

Версия 3.3 (исправление ошибок – февраль 2016-го)

Версия 3.4 (новая функциональность – март 2016-го)

- SSTableAttachedSecondaryIndex, или SASI – реализация интерфейса Cassandra SecondaryIndex, которую можно использовать вместо существующих реализаций.

Версия 3.5 (исправление ошибок – апрель 2016-го)

На осень 2016 года запланирован выпуск первой версии из серии 4.0.

Вы, наверное, обратили внимание на следующие тенденции:

- постоянное совершенствование возможностей CQL;
- растущий список клиентов для популярных языков, построенный на базе общего набора метафор;
- расширение набора конфигурационных параметров для настройки производительности и оптимизации использования ресурсов;
- улучшение в части производительности и надежности, погашение технического долга.

Поддерживаемые версии

В любой момент времени официально поддерживаются две версии Cassandra: последняя стабильная версия, которая считается пригодной для работы в производственном режиме, и последняя разрабатываемая версия. Официально поддерживаемые версии указаны на странице загрузки проекта (<http://cassandra.apache.org/download/>).

Пользователям Cassandra настоятельно рекомендуется работать с последней стабильной версией. Как это ни смешно, подавляющее большинство ошибок и вопросов в списке рассылки Cassandra-users относится к версиям, которые уже не поддерживаются. Специалисты вежливо отвечают на такие вопросы и помогают разобраться с ошибками, но чаще рекомендуют как можно скорее перейти на новую версию, в которой проблема решена.

Подходит ли Cassandra для моего проекта?

Итак, мы раскрыли тезисы, содержащиеся в краткой презентации, и понимаем основные достоинства Cassandra. Несмотря на тщательно продуманный дизайн и набор хитроумных функций, этот инструмент годится не для всякой работы. И в этом разделе мы поговорим о том, в каких проектах имеет смысл использовать Cassandra.

Крупное развертывание

Вряд ли вы поедете на грузовой фуре забирать вещи из химчистки, фуры для такой работы не приспособлены. В реализацию высокой доступности, настраиваемой согласованности, однорангового протокола и органичного масштабирования в Cassandra вложена уйма труда и инженерного гения, и именно этим она интересна. Но все эти качества обесцениваются при развертывании на одном узле, в такой конфигурации невозможно в полной мере раскрыть потенциал системы.

Однако существует множество разнообразных ситуаций, когда реляционной базы, работающей на одном узле, вполне достаточно. Поэтому нужны измерения. Примите во внимание ожидаемый трафик, необходимую пропускную способность и соглашения об уровнях обслуживания. Никаких твердых правил тут нет, но если вы ожидаете, что сможете надежно обслужить поток запросов с приемлемым уровнем производительности, ограничившись всего лишь несколькими реляционными базами, то имеет смысл на этом и остановиться, просто потому, что эксплуатировать РСУБД на одной машине легче, да и дело это знакомое.

Если же вы полагаете, что понадобится хотя бы несколько узлов, то Cassandra, возможно, подойдет. Ну а если для работы приложения необходимы десятки узлов, то Cassandra – как раз то, что надо.

Много операций записи, статистика, анализ

Взгляните на свое приложение с точки зрения соотношения числа операций чтения и записи. Cassandra оптимизирована для записи.

Во многих ранних внедрениях Cassandra использовалась для хранения данных об активности пользователей, данных об использовании социальной сети, рекомендаций и отзывов, статистики приложения. Все это разумные сценарии использования Cassandra, потому что число операций записи заведомо велико, а число операций чтения

не столь предсказуемо, и потому что запись может производиться неравномерно, с внезапными пиками нагрузки. Вообще, способность справляться с задачами, требующими высокой производительности при большом числе операций записи и наличии многочисленных параллельных клиентов, – одна из основных характеристик Cassandra.

Согласно вики-странице проекта, Cassandra использовалась для создания разнообразных приложений, включая хранилище оконных временных рядов, поиска в инвертированном индексе документов и организации распределенной очереди работ с приоритетами.

ТERRITORIALNAYA RAZNESENNOST'

Cassandra изначально поддерживает территориально разнесенные данные. Систему можно настроить так, что она будет осуществлять репликацию из одного ЦОДа в другие. Если ваше приложение глобально и позволяет получить выигрыш в производительности, когда данные находятся рядом с пользователями, то Cassandra прекрасно подойдет.

БЫСТРО ЭВОЛЮЦИОНИРУЮЩИЕ ПРИЛОЖЕНИЯ

Если ваше приложение быстро развивается и вы находитесь в «режиме стартапа», то Cassandra может оказаться подходящим кандидатом, поскольку поддерживает гибкие схемы. Это упростит синхронизацию базы данных с темпом изменения приложения.

Как принять участие

Любая технология тем сильнее и привлекательнее, чем больше вклад отдельных лиц в деятельность динамично развивающегося сообщества. По счастью, сообщество Cassandra активно, полно сил и предлагает самые разные способы участия, например:

Чат

Многие разработчики и члены сообщества Cassandra «зависают» в канале `#cassandra` на сайте webchat.freenode.net. Это прекрасное место для неформального общения, где можно получить ответ на свой вопрос и ответить кому-то.

Списки рассылки

В проекте Apache поддерживаются несколько списков рассылки, на которые вы можете подписаться и получать информацию по интересующим темам:

- user@cassandra.apache.org – общие вопросы, часто используется начинающими, а также нуждающимися в помощи;
- dev@cassandra.apache.org – используется разработчиками для обсуждения изменений, расстановки приоритетов и одобрения версий;
- client-dev@cassandra.apache.org – здесь обсуждается разработка клиентов Cassandra для различных языков программирования;
- commits@cassandra.apache.org – отслеживание записи в репозиторий кода Cassandra. Этот список с очень активным трафиком представляет интерес главным образом для работающих с репозиторием.

Объявления о выходе новых версий обычно публикуются в списках для пользователей и разработчиков.

Проблемы

Если вы столкнулись с проблемами при работе с Cassandra и подозреваете, что обнаружили дефект, то не стесняйтесь и отправляйте отчет в систему Cassandra JIRA (<https://issues.apache.org/jira/browse/cassandra>). Более того, пользователям, которые сообщают о дефектах в списке user@cassandra.apache.org, часто рекомендуют написать о проблеме в JIRA.

Блоги

В блоге разработчиков, который администрирует DataStax (<http://www.datastax.com/dev/blog>), публикуются материалы об использовании Cassandra, анонсы выхода продуктов Apache Cassandra и DataStax, а также технические статьи о деталях реализации Cassandra и разрабатываемой функциональности. В блоге Planet Cassandra (<http://www.planetcassandra.org/blog/>) предлагаются похожие материалы, но с упором на сведения о компаниях, применяющих Cassandra.

На сайте Apache Cassandra Wiki (<http://wiki.apache.org/cassandra>) размещены полезные статьи о том, как приступить к работе и как настроить систему, но следует иметь в виду, что некоторые материалы могут относиться к прошлым версиям.

Неформальные встречи

На неформальных встречах собираются члены местного сообщества и обсуждают проблемы, представляющие взаимный интерес. Такие встречи – отличная возможность пообщаться, чему-то на-

учиться или поделиться своими знаниями, проведя презентацию. Подобные группы существуют на всех континентах (<http://live-pc-development.pantheonsite.io/join-your-local-meetup/>), поэтому вам не составит труда найти что-нибудь поблизости.

Учебные курсы и конференции

DataStax предлагает онлайновое обучение (<https://academy.datastax.com/>), а в июне 2015 года объявлено о заключении партнерского соглашения с O'Reilly Media о выдаче свидетельств о прохождении курса по Cassandra. DataStax также проводит ежегодные конференции Cassandra Summit (<http://cassandra-summit-datastax.com/>) в различных уголках мира.



Спрос на рынке труда

Устойчиво растет спрос на разработчиков и администраторов, знакомых с Cassandra. В исследовании Dice.com за 2015 год Cassandra занимает второе место по размеру зарплаты.

Резюме

В этой главе мы познакомились с определяющими характеристиками Cassandra, историей проекта и основными возможностями. Мы узнали о сообществе пользователей Cassandra и о том, как ее используют различные компании. Теперь можно приступать к практическим занятиям.

Глава 3

Установка Cassandra

Чтобы порадовать тех, кто предпочитает немедленное вознаграждение, приступим к установке Cassandra. Возможно, по ходу дела вам встретятся незнакомые термины. Ничего страшного, сейчас мы просто хотим быстро настроить простую конфигурацию и убедиться, что все работает как надо. Это послужит нам ориентиром. Затем мы вернемся назад и осмыслим Cassandra в более широком контексте.

Установка из дистрибутива Apache

Cassandra можно скачать с сайта <http://cassandra.apache.org>. После щелчка по ссылке на домашней странице загрузится tar.gz-файл. Как правило, предлагаются две версии Cassandra. *Последняя версия* рекомендуется для тех, кто только начинает новый проект, не планируя сразу запускать его в производственном режиме. Для «боевых» систем рекомендуется *стабильная версия*. Для любой версии файл с уже собранным двоичным дистрибутивом называется *apache-cassandra-x.x.x-bin.tar.gz*, где *x.x* – номер версии. Размер файла составляет примерно 23 МБ.

Распаковка дистрибутива

Чтобы побыстрее начать работу, проще всего скачать готовый двоичный дистрибутив. Для распаковки сжатого файла годится любая стандартная утилита для работы с ZIP-файлами. В системах на базе Unix, например Linux или MacOS, должна уже присутствовать программа распаковки GZip; в Windows придется откуда-то скачать под-

ходящую программу, например WinZip (комерческая) или 7-Zip (<http://www.7-zip.org/> – бесплатная).

Запустите программу распаковки. Возможно, придется сначала распаковать ZIP-файл, а затем – TAR-файл. После того как в файловой системе появится папка *apache-cassandra-x.x.x*, можно будет запускать Cassandra.

Что внутри?

Распаковав архив, вы увидите несколько файлов и каталогов.

Файл *NEWS.txt* содержит замечания к версии с описанием функциональности, включенной в текущую и предыдущую версии, а файл *CHANGES.txt* – сведения об исправлениях ошибок. Не забудьте ознакомиться с этими файлами при переходе на новую версию, чтобы знать, чего ожидать.

Заглянем в отдельные каталоги и познакомимся с их содержимым.

bin

Здесь находятся исполняемые файлы сервера Cassandra, а также клиенты, в т. ч. оболочка языка запросов (*cqlsh*) и командный интерфейс (CLI). Здесь же вы найдете скрипты для запуска *nodetool* – утилиты, которая проверяет правильность конфигурации кластера и выполняет ряд операций обслуживания. Подробнее мы изучим эту утилиту позже. В этом же каталоге находится несколько утилит для работы с файлами SSTable, в т. ч. для вывода списка ключей (*sstablekeys*), массового извлечения и восстановления содержимого SSTable (*sstableloader*) и модификации SSTable при переходе на новую версию Cassandra (*sstableupgrade*).

conf

Здесь находятся файлы, необходимые для настройки экземпляра Cassandra. Основной конфигурационный файл называется *cassandra.yaml*, а в файле *logback.xml* задаются параметры протоколирования. Для настройки топологии сети, команд архивации и восстановления и триггеров служат дополнительные файлы. Мы рассмотрим их, когда будем обсуждать настройку в главе 7.

interface

В этом каталоге есть всего один файл *cassandra.thrift*. В нем определен унаследованный API удаленного вызова процедур (Remote Procedure Call – RPC), основанный на синтаксисе Thrift. Интерфейс Thrift использовался при создании клиентов для языков Java, C++, PHP, Ruby, Python, Perl и C# до появления CQL. В версии 3.2

Thrift API официально объявлен нерекомендуемым, и в версии 4.0 он будет исключен.

javadoc

Здесь находится код сайта документации, сгенерированный с помощью утилиты JavaDoc. Отметим, что JavaDoc знает только о комментариях, которые хранятся непосредственно в коде Java, поэтому полной эту документацию не назовешь. Но она полезна для тех, кто хочет знать, как структурирован код. Правда, несмотря на все плюсы проекта Cassandra, его код прокомментирован весьма скучно, поэтому полезность документации в формате JavaDoc ограничена. Читатели, знакомые с языком Java, пожалуй, извлекут больше информации, обратившись напрямую к файлам классов. Так или иначе, чтобы почитать документацию, откройте в браузере файл *javadoc/index.html*.

lib

В этом каталоге находятся все внешние библиотеки, необходимые Cassandra, например: две библиотеки сериализации в формате JSON, проект коллекций Google и несколько библиотек из проекта Apache Commons.

pylib

Здесь находятся библиотеки на Python, используемые в программе *cqlsh*.

tools

Здесь находятся инструменты, применяемые для обслуживания узлов кластера Cassandra. Мы будем рассматривать их в главе 11.



Дополнительные каталоги

Если вы уже запускали Cassandra в конфигурации по умолчанию, то будет создано еще два каталога: *data* и *log*. Их содержимое мы обсудим чуть ниже.

Сборка из исходного кода

Для сборки Cassandra используется скриптовый язык Apache Ant и система управления зависимостями Maven.



Получение Ant

Скачать Ant можно с сайта <http://ant.apache.org>. Отдельно скачивать Maven для сборки Cassandra необязательно.

Для сборки из исходного кода необходим полный комплект средств разработчика Java 7 или 8, а не только среда выполнения JRE. Если появляется сообщение о том, что Ant не хватает файла *tools.jar*, значит, либо у вас не установлен полный JDK, либо вы указали неправильный путь в переменной среды. Maven загружает файлы из Интернета, поэтому если нет подключения или Maven не может найти прокси-сервер, то сборка завершится с ошибкой.



Получение сборок для разработчиков

Если вы хотите скачать самую последнюю сборку, то можете забрать исходный код из системы Jenkins, которая в проекте Cassandra используется в качестве средства непрерывной интеграции. Последние сборки и сведения о покрытии тестами можно найти на сайте <http://cassci.datastax.com>.

Любители Git могут получить доступную только для чтения столовую версию исходного кода Cassandra такой командой:

```
$ git clone git://git.apache.org/cassandra.git
```



Что такое Git?

Git – это система управления исходным кодом, которую Линус Торвальдс создал для управления разработкой ядра Linux. Она набирает популярность и используется в таких проектах, как Android, Fedora, Ruby on Rails, Perl и многие клиенты Cassandra (как мы убедимся в главе 8). Если вы работаете с каким-нибудь дистрибутивом Linux, например Ubuntu, то получить Git элементарно. Просто введите на консоли команду `>apt-get install git` – и программа будет установлена и готова к работе. Дополнительные сведения см. на сайте <http://git-scm.com>.

Поскольку обо всех зависимостях заботится Maven, собрать Cassandra из исходного кода очень просто. Перейдите в корневой каталог развернутого дистрибутива и запустите программу ant, которая найдет файл *build.xml* в текущем каталоге и выполнит указанную в нем цель сборки по умолчанию.

```
$ ant
```

Вот и все. Maven сам загрузит все необходимые зависимости, а Ant откомпилирует сотни исходных файлов и прогонит тесты. Если не случится ошибок, то будет выведено сообщение BUILD SUCCESSFUL. В противном случае проверьте, что пути заданы правильно, что установлены последние версии всех необходимых программ и что скачана

стабильная версия Cassandra. Можете посмотреть сформированный Jenkins отчет и убедиться, что скачанный исходный код компилируется.



Дополнительная информация о процессе сборки

Если вы хотите видеть подробную информацию обо всем происходящем в процессе сборки, запустите Ant с флагом `-v` – тогда будут выводиться сведения о каждой выполняемой операции.

Дополнительные цели сборки

Чтобы откомпилировать сервер, нужно просто набрать команду `ant`, как описано выше. Она выполняет цель по умолчанию `-jar`. При этом производится полная сборка, включая автономные тесты, а созданный файл `apache-cassandra.x.x.jar` записывается в каталог `build`.

Чтобы ознакомиться с перечнем всех поддерживаемых целей, запустите Ant с флагом `-р`. Ниже перечислено несколько наиболее интересных целей.

test

Эта цель, пожалуй, наиболее полезна пользователям, поскольку запускает выполнение всего комплекта автономных тестов. Можете заглянуть в исходный код тестов – там имеются полезные примеры взаимодействия с Cassandra.

stress-build

Эта цель строит программу нагружочного тестирования Cassandra, мы познакомимся с ней в главе 12.

clean

Эта цель удаляет локальные временные файлы, в частности сгенерированные файлы с исходным кодом и результаты автономных тестов. Цель `realclean` сначала выполняет `clean`, а затем удаляет дистрибутивные jar-файлы Cassandra и jar-файлы, скачанные Maven.

Запуск Cassandra

В первых версиях Cassandra перед запуском сервера нужно было отредактировать конфигурационные файлы и задать переменные среды. Но разработчики приложили немало усилий, чтобы Cassandra можно было запустить немедленно, безо всяких церемоний. Некоторые конфигурационные параметры мы рассмотрим по ходу изложения.



Требуемая версия Java

Для работы Cassandra необходима версия Java 7 или 8, лучше последняя стабильная версия. Система тестировалась для Open JDK и Oracle JDK. Узнать, какая версия установлена на вашем компьютере, можно, выполнив команду `java -version`. Скачать нужный JDK можно, зайдя на страницу <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.

ОС Windows

После того как скачан двоичный дистрибутив или скачан и откомпилирован исходный дистрибутив, можно запустить сервер базы данных.

Рекомендуется установить переменную среды `JAVA_HOME`. Если вы работаете в Windows 7, то нажмите кнопку Пуск, затем щелкните правой кнопкой мыши по пункту меню «Компьютер» и выберите «Свойства». Выберите команду «Дополнительные параметры системы» и нажмите кнопку «Переменные среды». Нажмите «Создать...» под списком системных переменных. В поле «Имя переменной» введите `JAVA_HOME`, а в поле «Значение переменной» – путь к установочной папке Java (скорее всего, она называется `C:\Program Files\Java\jre7` в случае Java 7 или `C:\Program Files\Java\jre1.8.0_25` в случае Java 8).

Не забывайте, что после создания новой переменной среды нужно заново открыть окно терминала, чтобы система увидела новую переменную. Для проверки того, что переменная среды задана правильно и что Cassandra сможет найти Java, выполните в новом окне терминала команду `echo %JAVA_HOME%`. Она напечатает значение переменной среды.

Можно также определить переменную среды `CASSANDRA_HOME`, которая должна указывать на каталог верхнего уровня, куда был загружен или где был собран дистрибутив Cassandra, тогда не придется думать о том, откуда запущена Cassandra. Это полезно для инструментов, отличных от самого сервера, например `nodetool` и `cqlsh`.

После первого запуска сервера Cassandra создаст в системе каталоги для размещения файлов данных. По умолчанию они создаются внутри каталога, на который указывает `CASSANDRA_HOME`.

data

В этом каталоге Cassandra хранит данные. По умолчанию в нем есть три подкаталога, отведенных для различных файлов данных: `commitlog`, `data` и `saved_caches`. Их назначение мы рассмотрим в главе 6. Если вы экспериментируете с разными версиями базы и не

боитесь потерять данные, то можете просто удалить эти каталоги и перезапустить сервер.

logs

Здесь Cassandra хранит свои журналы в файле *system.log*. В случае каких-либо затруднений откройте журнал и посмотрите, что случилось.

Расположение файлов данных

Места расположения файлов данных можно задать в файле *cassandra.yaml*, который находится в каталоге *conf*. Соответствующие свойства называются *data_file_directories*, *commit_log_directory* и *saved_caches_directory*. Рекомендуемую конфигурацию этих каталогов мы обсудим в главе 7.

ОС Linux

В Linux и других операционных системах, производных от Unix (включая Mac OS), процедура примерно такая же, как в Windows. Проверьте, что переменная среды *JAVA_HOME* установлена в соответствии с приведенными выше инструкциями. Затем с помощью программы *gunzip* распакуйте архив Cassandra. Многие пользователи предпочитают хранить данные в каталоге */var/lib*. Если вы собираетесь изменить место хранения, то нужно будет отредактировать файл *conf/cassandra.yaml* и создать соответствующие каталоги для данных и журналов Cassandra, не забыв предоставить право записи в них пользователю, от имени которого будет запущен сервер.

```
$ sudo mkdir -p /var/lib/cassandra  
$ sudo chown -R username /var/lib/cassandra
```

Вместо *username* подставьте нужное имя пользователя.

Запуск сервера

Чтобы запустить сервер Cassandra в любой ОС, откройте окно терминала, перейдите в каталог *<cassandra-directory>/bin*, куда был распакован дистрибутив, и выполните команду *cassandra -f*.



Запуск Cassandra в приоритетном режиме

Флаг *-f* означает, что Cassandra должна работать в приоритетном, а не в фоновом процессе. При этом все сообщения сервера выводятся в стандартный вывод, т. е. отображаются в окне терминала, что полезно для тестирования. В любом случае, сообщения записываются также в вышеупомянутый файл *system.log*.

Если запустить сервер сразу после установки, то в журнал будут выведены дополнительные сообщения. Их вид зависит от используемой версии, но есть и общие черты. Поискав по фразе «*cassandra.yaml*», вы обнаружите такие строки:

```
DEBUG [main] 2015-12-08 06:02:38,677 YamlConfigurationLoader.java:104 -  
    Loading settings from file:/.../conf/cassandra.yaml  
INFO [main] 2015-12-08 06:02:38,781 YamlConfigurationLoader.java:179 -  
    Node configuration: [authenticator=AllowAllAuthenticator;  
    authorizer=AllowAllAuthorizer; auto_bootstrap=false; auto_snapshot=true;  
    batch_size_fail_threshold_in_kb=50; ...
```

В этих сообщениях указано местоположение файла *cassandra.yaml*, содержащего конфигурационные параметры. В сообщении Node configuration перечислены прочитанные из этого файла параметры.

Теперь поищите строку «JVM».

```
INFO [main] 2015-12-08 06:02:39,239 CassandraDaemon.java:436 -  
JVM vendor/version: Java HotSpot(TM) 64-Bit Server VM/1.8.0_60  
INFO [main] 2015-12-08 06:02:39,239 CassandraDaemon.java:437 -  
Heap size: 519045120/519045120
```

Здесь приведена информация об используемой JVM, в т. ч. о настройках памяти.

Затем поищите номера версий: строки «Cassandra version», «Thrift API Version», «CQL supported versions»:

```
INFO [main] 2015-12-08 06:02:43,931 StorageService.java:586 -  
Cassandra version: 3.0.0  
INFO [main] 2015-12-08 06:02:43,932 StorageService.java:587 -  
Thrift API version: 20.1.0  
INFO [main] 2015-12-08 06:02:43,932 StorageService.java:588 -  
CQL supported versions: 3.3.1 (default: 3.3.1)
```

Можно также поискать сообщения об инициализации внутренних структур данных Cassandra, например кэшей:

```
INFO [main] 2015-12-08 06:02:43,633 CacheService.java:115 -  
Initializing key cache with capacity of 24 MBs.  
INFO [main] 2015-12-08 06:02:43,679 CacheService.java:137 -  
Initializing row cache with capacity of 0 MBs  
INFO [main] 2015-12-08 06:02:43,686 CacheService.java:166 -  
Initializing counter cache with capacity of 12 MBs
```

Поиск строк «JMX», «gossip» и «clients» покажет такие сообщения:

```

WARN [main] 2015-12-08 06:08:06,078 StartupChecks.java:147 -
JMX is not enabled to receive remote connections.
Please see cassandra-env.sh for more info.
INFO [main] 2015-12-08 06:08:18,463 StorageService.java:790 -
Starting up server gossip
INFO [main] 2015-12-08 06:02:48,171 Server.java:162 -
Starting listening for CQL clients on /127.0.0.1:9042 (unencrypted)

```

Эти сообщения говорят, что сервер начал инициализацию обмена данными с другими серверами в кластере и уведомляет о публично доступных интерфейсах. По умолчанию интерфейс управления на основе Java Management Extensions (JMX) для удаленного доступа закрыт. Этот интерфейс мы обсудим в главе 10.

Наконец, поищем строку «state jump»:

```

INFO [main] 2015-12-08 06:02:47,351 StorageService.java:1936 -
Node /127.0.0.1 state jump to normal

```

Поздравляем! Сервер Cassandra запустился и работает в новом кластере Test Cluster с одним узлом, прослушивая порт 9160. Продолжая следить за сообщениями сервера, вы увидите, что периодически выводится сообщение о сбросе и уплотнении таблек; скоро мы вернемся к этому вопросу.



Перезапуск

Разработчики прилагают массу усилий, чтобы данные можно было читать после перехода с текущей версии (основной или дополнительной) на следующую. Но журнал фиксаций (commit log) нужно полностью очищать при смене версий (даже дополнительных).

Если на вашей машине установлена предыдущая версия Cassandra, то для запуска новой, возможно, придется очистить каталоги данных. Если вы напортачили с установкой и хотите быстро запуститься с чистого листа, то можете вообще удалить эти каталоги.

Остановка Cassandra

Итак, сервер Cassandra успешно запущен. Но как его остановить? Вы, наверное, обратили внимание на команду `stop-server` в каталоге `bin`. Попробуйте выполнить ее. В системах Unix будет напечатано следующее сообщение:

```

$ ./stop-server
please read the stop-server script before use

```

То есть сервер не остановился, а нам предлагают почитать скрипт. Открыв его в редакторе, мы увидим, что для остановки Cassandra нужно снять процесс JVM, в котором работает сервер. В файле предложено несколько способов идентификации и завершения этого процесса.

Первый вариант – запускать Cassandra с флагом `-p`, указывая имя файла, в который Cassandra должна записать идентификатор процесса (PID) в момент запуска. Это, пожалуй, самый простой способ точно узнать, какой процесс снимать.

Но мы-то уже запустили Cassandra без флага `-p`, поэтому должны определить нужный процесс самостоятельно. Скрипт предлагает воспользоваться командой `pgrep` для поиска процессов, запущенных текущим пользователем, в именах которых есть слово «`cassandra`»:

```
user=`whoami`  
pgrep -u $user -f cassandra | xargs kill -9
```



Остановка Cassandra в Windows

В Windows для поиска и завершения процесса JVM следует использовать диспетчер задач.

Другие дистрибутивы Cassandra

Выше мы видели, как установить Cassandra из дистрибутива Apache. Но, помимо этого дистрибутива, есть еще несколько способов получить Cassandra.

DataStax Community Edition

Этот бесплатный дистрибутив предлагается компанией DataStax на сайте Planet Cassandra. Имеются пакеты для различных платформ: RPM и Debian для Linux, MSI для Windows и библиотека для Mac OS. В издание для сообщества включены дополнительные инструменты, в т. ч. интегрированная среда разработки DevCenter и средство мониторинга OpsCenter. Еще одна полезная возможность – запуск Cassandra в виде управляемой операционной системой службы в Windows. Версии издания для сообщества обычно выходят спустя короткое время после выхода версии Apache.

DataStax Enterprise Edition

Компания DataStax также предлагает полностью поддерживаемую версию, сертифицированную для работы в производственном режиме. В линейку продуктов входит интегрированная платформа

базы данных с поддержкой дополнительных технологий, в частности Hadoop и Apache Spark. Некоторые виды интеграции мы рассмотрим в главе 14.

Образы виртуальных машин

Часто Cassandra распространяется в виде одного из ранее описанных дистрибутивов, оформленного как образ виртуальной машины. Много таких образов есть в магазине Amazon Web Services (AWS) Marketplace.

В главе 14 мы внимательно рассмотрим некоторые варианты развертывания Cassandra в производственных средах, в том числе в облаке.

Выбор дистрибутива зависит от среды, в которой производится развертывание, потребностей в масштабировании, стабильности и наличии технической поддержки, а также от средств, выделяемых на разработку и обслуживание. Поскольку можно выбирать между бесплатными и коммерческими версиями, то у любой организации имеются достаточно гибкие возможности сделать правильный выбор.

Запуск оболочки CQL

Итак, Cassandra установлена и запущена, теперь проверим, что все настроено правильно. Воспользуемся оболочкой CQL (`cqlsh`), чтобы подключиться к нашему серверу и немного оглядеться.



Использовать CLI не рекомендуется

Если раньше вы работали с версией Cassandra ниже 3.0, то, наверное, знакомы с интерфейсом командной строки (CLI) `cassandra-cli`. Из версии 3.0 он исключен, потому что основан на устаревшем Thrift API.

Для запуска оболочки откройте окно терминала, перейдите в домашний каталог и введите следующую команду:

```
$ bin/cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.0.0 | CQL spec 3.3.1 | Native protocol v4]
Use HELP for help.
cqlsh>
```

Поскольку мы не указали, к какому узлу хотим подключиться, оболочка проверила, работает ли узел на локальном хосте, нашла запущенный нами сервер и сообщила, что подключилась к кластеру

с именем «Test Cluster». Так произошло, потому что по умолчанию мы настроили кластер из одного узла на компьютере localhost.



Переименование кластера по умолчанию

В производственной среде не забудьте поменять имя кластера на более подходящее для приложения.

Чтобы подключиться к конкретному узлу, укажите при запуске команды имя хоста и номер порта. Например, для подключения к локальному узлу нужно написать:

```
$ bin/cqlsh localhost 9042
```

Другой способ указать, к какому узлу должна подключаться cqlsh, – задать переменные среды \$CQLSH_HOST и \$CQLSH_PORT. Это особенно полезно, если вы часто подключаетесь к одному и тому же узлу, расположенному на другом хосте. Хост и порт, заданные в командной строке, имеют приоритет перед переменными среды.



Ошибки при подключении

При попытке подключиться к серверу появилось такое сообщение:

```
Exception connecting to localhost/9160. Reason: Connection refused.
```

В таком случае проверьте, что на указанном хосте и порте запущен экземпляр Cassandra и что вы можете прозвонить этот хост командой ping. Возможно, на брандмауэре заданы правила, блокирующие подключение.

Для получения полного списка параметров команды cqlsh введите команду cqlsh -help.

Простые команды cqlsh

Рассмотрим простейшие команды, которые cqlsh позволяет отправлять серверу. Мы покажем команды инспекции окружения и команды вставки и выборки данных.



Регистр в cqlsh

Все команды cqlsh нечувствительны к регистру. В примерах мы будем использовать верхний регистр, потому что именно так команды выглядят в оперативной справке и в списке.

cqlsh Help

Для получения справки по cqlsh введите HELP или ?; будут выведены все существующие команды.

```
cqlsh> HELP
Documented shell commands:
=====
CAPTURE      COPY      DESCRIBE  EXPAND   PAGING   SOURCE
CONSISTENCY  DESC      EXIT      HELP     SHOW     TRACING

CQL help topics:
=====
ALTER CREATE_TABLE_TYPES PERMISSIONS
ALTER_ADD CREATE_USER REVOKE
ALTER.Alter DATE_INPUT REVOKE_ROLE
ALTER_DROP DELETE SELECT
ALTER_RENAME DELETE_COLUMNS SELECT_COLUMNFAMILY
ALTER_USER DELETE_USING SELECT_EXPR
ALTER_WITH DELETE_WHERE SELECT_LIMIT
APPLY DROP SELECT_TABLE
ASCII_OUTPUT DROP_AGGREGATE SELECT_WHERE
BEGIN DROP_COLUMNFAMILY TEXT_OUTPUT
BLOB_INPUT DROP_FUNCTION TIMESTAMP_INPUT
BOOLEAN_INPUT DROP_INDEX TIMESTAMP_OUTPUT
COMPOUND_PRIMARY_KEYS DROP_KEYSPACE TIME_INPUT
CREATE DROP_ROLE TRUNCATE
CREATE_AGGREGATE DROP_TABLE TYPES
CREATE_COLUMNFAMILY DROP_USER UPDATE
CREATE_COLUMNFAMILY_OPTIONS GRANT UPDATE_COUNTERS
CREATE_COLUMNFAMILY_TYPES GRANT_ROLE UPDATE_SET
CREATE_FUNCTION INSERT UPDATE_USING
CREATE_INDEX INT_INPUT UPDATE_WHERE
CREATE_KEYSPACE LIST USE
CREATE_ROLE LIST_PERMISSIONS UUID_INPUT
CREATE_TABLE LIST_ROLES
CREATE_TABLE_OPTIONS LIST_USERS
```



Разделы справки по cqlsh

Обратите внимание, что названия разделов справки несколько отличаются от истинного синтаксиса команд. Например, в разделе CREATE_TABLE описывается команда > CREATE TABLE

Для получения дополнительных сведений о команде наберите `HELP <command>`. Многие команды cqlsh можно использовать без параметров, и в таком случае печатается текущее значение параметра, например: `CONSISTENCY, EXPAND, PAGING`.

Описание окружения в cqlsh

Если вы пользуетесь двоичным дистрибутивом, то после подключения к экземпляру Test Cluster создается пустое *пространство ключей*, т. е. база данных Cassandra. С этой базой можно экспериментировать.

Чтобы узнать о текущем кластере, в котором вы работаете, введите:

```
cqlsh> DESCRIBE CLUSTER;
Cluster: Test Cluster
Partitioner: Murmur3Partitioner
...
```

Начиная с версии 3.0 эта команда также печатает список диапазонов маркеров (*token range*), принадлежащих каждому узлу кластера. Для краткости мы его опустили.

Чтобы узнать, какие пространства ключей имеются в кластере, наберите:

```
cqlsh> DESCRIBE KEYSPACES;
system_auth system_distributed system_schema
system system_traces
```

Первоначально список состоит из нескольких системных пространств ключей. После того как вы создадите свои собственные, они тоже будут показываться. Пространства ключей с префиксом *system* предназначены для внутренних нужд Cassandra, пользователь не может записывать в них данные. В этом смысле они похожи на базы данных *master* и *temp* в Microsoft SQL Server. Cassandra использует их для хранения информации о схемах, трассировке и безопасности. Подробнее мы будем говорить о них в главе 6.

Чтобы узнать о версиях клиента, сервера и протокола, наберите команду:

```
cqlsh> SHOW VERSION;
[cqlsh 5.0.1 | Cassandra 3.0.0 | CQL spec 3.3.1 | Native protocol v4]
```

Обратите внимание, что та же самая информация о версиях печатается при запуске *cqlsh*. Есть еще много команд, с которыми вы можете поэкспериментировать. А мы сейчас займемся добавлением данных в базу и извлечением их.

Создание пространства ключей и таблицы в cqlsh

Пространство ключей в Cassandra напоминает реляционную базу данных. В нем определяется одна или несколько таблиц, или «семейств

столбцов». Если запустить cqlsh, не указав пространство ключей, то будет напечатано приглашение `cqlsh>`, в котором имя пространства ключей отсутствует.

Создадим свое пространство ключей, чтобы было куда записывать данные. При создании пространства ключей необходимо задать некоторые параметры. Чтобы узнать о них, можно было бы выполнить команду `HELP CREATE_KEYSPACE`, но мы вместо этого воспользуемся удобным механизмом завершения команды, встроенным в cqlsh. Наберите показанную ниже строку и нажмите клавишу **Tab**:

```
cqlsh> CREATE KEYSPACE my_keyspace WITH
```

После нажатия **Tab** cqlsh приступит к завершению команды:

```
cqlsh> CREATE KEYSPACE my_keyspace WITH replication = {'class': '
```

Нам подсказывают, что для задания пространства ключей необходимо указать стратегию репликации. Еще раз нажмите **Tab**, чтобы узнать, какие есть варианты:

```
cqlsh> CREATE KEYSPACE my_keyspace WITH replication = {'class': '  
NetworkTopologyStrategy SimpleStrategy  
OldNetworkTopologyStrategy
```

Теперь cqlsh предлагает на выбор три стратегии. Что они означают, мы узнаем в главе 6. А пока выберем `SimpleStrategy`, введя эту строку. Сообщим, что закончили, введя закрывающую кавычку, и еще раз нажмем **Tab**:

```
cqlsh> CREATE KEYSPACE my_keyspace WITH replication = {'class':  
'SimpleStrategy', 'replication_factor':
```

Далее нам предлагают ввести коэффициент репликации. Для простой стратегии нужно задать, на сколько узлов в этом пространстве ключей производится запись. В производственной среде копии данных должны храниться на нескольких узлах, но поскольку сейчас у нас всего один узел, то и запросим одну копию. Введите значение «1» и снова нажмите **Tab**:

```
cqlsh> CREATE KEYSPACE my_keyspace WITH replication = {'class':  
'SimpleStrategy', 'replication_factor': 1};
```

Теперь cqlsh добавила закрывающую скобку, показывая, что все обязательные параметры заданы. Завершим команду, поставив точку с запятой и нажав **Enter**, – пространство ключей создано.



Параметры создания пространства ключей

В производственной среде не следует задавать коэффициент репликации 1. В зависимости от выбранной стратегии репликации могут потребоваться дополнительные параметры пространства ключей. Механизм завершения команды в любом случае подскажет, что вводить.

Теперь посмотрим, что мы создали, выполнив команду DESCRIBE KEYSPACE:

```
cqlsh> DESCRIBE KEYSPACE my_keyspace
CREATE KEYSPACE my_keyspace WITH replication = {'class':
  'SimpleStrategy', 'replication_factor': '1'} AND
  durable_writes = true;
```

Как видим, создано пространство ключей со стратегией SimpleStrategy, коэффициент репликации replication_factor равен единице, а операции записи долговечны (параметр durable_writes). Обратите внимание, что для описания пространства ключей применяется практически такой же синтаксис, как при создании, с одним дополнительным параметром: durable_writes = true. Смысл параметров сейчас не так важен, мы обсудим его позднее.

Создав пространство ключей, мы можем переключиться на него:

```
cqlsh> USE my_keyspace;
cqlsh:my_keyspace>
```

Заметьте, что приглашение изменилось – теперь в нем входит имя пространства ключей.

Змеиная нотация

Наверное, вам интересно, почему для именования пространства ключей мы выбрали «змеиную нотацию» (`my_keyspace`), а не «верблюжью» (`MyKeyspace`), как принято в Java и других языках.

Дело в том, что внутри себя Cassandra хранит имена пространств ключей, таблиц и столбцов в нижнем регистре, т. е. автоматически переводит введенные имена в нижний регистр. Это поведение можно изменить, заключив имя в двойные кавычки (например, `CREATE KEYSPACE "MyKeyspace"`...). Но проще не плевать против ветра, а воспользоваться змеиной нотацией.

Имея пространство ключей, мы можем создать в нем таблицу. В `cqlsh` для этого служит такая команда:

```
cqlsh:my_keyspace> CREATE TABLE user ( first_name text ,
last_name text, PRIMARY KEY (first_name) ) ;
```

В результате в текущем пространстве ключей создается таблица «user» с двумя столбцами для хранения имени и фамилии, оба типа text. Типы text и varchar – синонимы и применяются для хранения строк. Мы сказали, что столбец first_name будет первичным ключом, а для всех остальных параметров таблицы оставили значения по умолчанию.



Об именах пространств ключей в cqlsh

Можно было бы создать эту таблицу, не переключаясь в нужное пространство ключей, а воспользовавшись синтаксисом CREATE TABLE my_keyspace.user (....

С помощью команды DESCRIBE TABLE можно получить описание только что созданной таблицы:

```
cqlsh:my_keyspace> DESCRIBE TABLE user;
CREATE TABLE my_keyspace.user (
    first_name text PRIMARY KEY,
    last_name text
) WITH bloom_filter_fp_chance = 0.01
    AND caching = { 'keys': 'ALL', 'rows_per_partition': 'NONE' }
    AND comment = ''
    AND compaction = { 'class': 'org.apache.cassandra.db.compaction.
        SizeTieredCompactionStrategy', 'max_threshold': '32',
        'min_threshold': '4' }
    AND compression = { 'chunk_length_in_kb': '64', 'class':
        'org.apache.cassandra.io.compress.LZ4Compressor' }
    AND crc_check_chance = 1.0
    AND dclocal_read_repair_chance = 0.1
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair_chance = 0.0
    AND speculative_retry = '99PERCENTILE';
```

Отметим, что cqlsh выводит отформатированный вариант команды CREATE TABLE, включив также значения всех параметров, которые мы явно не задавали. Пока не думайте о них, нам и так есть чем заняться.

Запись и чтение данных в cqlsh

Теперь у нас есть пространство ключей и таблица, так давайте запишем в базу какие-нибудь данные, а затем прочитаем их. На данном этапе детали процесса нам не важны, в модели данных Cassandra мы разберемся позже. А пока достаточно знать, что есть пространство ключей (база данных), а в нем таблица, содержащая столбцы – автоматные единицы хранения.

Для записи значения служит команда INSERT:

```
cqlsh:my_keyspace> INSERT INTO user (first_name, last_name )
    VALUES ('Bill', 'Nguyen');
```

Здесь мы создали новую строку с двумя столбцами и ключом Bill для хранения набора связанных с этим ключом данных. Столбцы называются first_name и last_name. Воспользовавшись командой SELECT COUNT, убедимся, что строка действительно записана:

```
cqlsh:my_keyspace> SELECT COUNT (*) FROM user;
    count
-----
      1
(1 rows)
```

Зная, что данные сохранены, прочитаем их командой SELECT:

```
cqlsh:my_keyspace> SELECT * FROM user WHERE first_name='Bill';
  first_name | last_name
-----+-----
    Bill    |    Nguyen
(1 rows)
```

Здесь мы попросили вернуть строки с первичным ключом Bill, выбрав все столбцы. Команда DELETE позволяет удалить столбец. Ниже из строки с ключом Bill удаляется столбец last_name:

```
cqlsh:my_keyspace> DELETE last_name FROM USER WHERE
    first_name='Bill';
```

Убедимся, что столбец действительно удален:

```
cqlsh:my_keyspace> SELECT * FROM user WHERE first_name='Bill';
  first_name | last_name
-----+-----
    Bill    |      null
(1 rows)
```

Теперь почистим за собой, удалив всю строку. Команда та же самая, только имя столбца не указывается:

```
cqlsh:my_keyspace> DELETE FROM USER WHERE first_name='Bill';
```

Убедимся, что строка удалена:

```
cqlsh:my_keyspace> SELECT * FROM user WHERE first_name='Bill';  
first_name | last_name  
-----+-----
```

(0 rows)

Можно было удалить из таблицы все данные командой TRUNCATE или вообще удалить схему таблицы командой DROP TABLE.

```
cqlsh:my_keyspace> TRUNCATE user;  
cqlsh:my_keyspace> DROP TABLE user;
```



История команд cqlsh

Поработав некоторое время с cqlsh, вы, наверное, заметили, что по списку команд можно перемещаться клавишами со стрелками вверх и вниз. История команд хранится в файле *cqlsh_history*, который находится в скрытом подкаталоге *.cassandra* домашнего каталога. Механизм такой же, как для истории команд оболочки bash: команды записаны в обычном текстовом файле в порядке выполнения. Удобно!

Резюме

Мы установили и запустили Cassandra. Мы воспользовались клиентом cqlsh для вставки и выборки данных. Теперь можно отступить назад, чтобы составить общее представление о Cassandra, а затем уже переходить к деталям.

Глава 4

• • • • • • • • • • • • • • • • • • • • • • • • •

Язык Cassandra Query Language

В этой главе мы обсудим модель данных в Cassandra и ее реализацию в виде языка Cassandra Query Language (CQL). Мы покажем, как CQL поддерживает проектные цели Cassandra, и остановимся на некоторых общих характеристиках поведения.

Разработчикам и администраторам, привыкшим к реляционным базам данных, модель данных Cassandra поначалу может показаться трудной и непонятной. Некоторые термины, например «пространство ключей», вообще новые, а другие, к примеру «столбец», употребляются тут и там, но их семантика несколько отличается. Синтаксис CQL во многих отношениях напоминает SQL, но имеет ряд важных отличий. Читателей, знакомых с такими технологиями NoSQL, как Dynamo или Bigtable, модель данных тоже может повергнуть в смятение, потому что хотя Cassandra и основана на этих технологиях, ее модель данных устроена совершенно по-другому.

Поэтому мы начнем эту главу с описания реляционной терминологии, а затем познакомимся со взглядом на мир, принятым в Cassandra. Попутно мы сведем знакомство с CQL и узнаем, как в нем реализована модель данных.

Реляционная модель данных

В реляционном мире база данных является самым внешним контейнером всего, что относится к одному приложению. База данных содержит таблицы. У таблицы есть имя, и она состоит из одного или нескольких столбцов, также имеющих имена. Добавляя данные в таблицу, мы задаем значения для каждого столбца; если в некотором

столбце не должно быть никакого значения, указывается `null`. Новая запись становится строкой таблицы, которую мы впоследствии можем прочитать, если знаем ее уникальный идентификатор (первичный ключ), или с помощью команды SQL, выражающей критерий, которому должна удовлетворять строка. Обновление таблицы может затрагивать все или только некоторые строки – в зависимости от фильтра, заданного во фразе «`where`» SQL-команды.

После этого краткого обзора мы можем перейти к описанию модели данных Cassandra, акцентируя внимание на сходстве и различии.

Модель данных Cassandra

В этом разделе мы примем восходящий подход к описанию модели данных Cassandra. Простейшим из возможных хранилищ данных, наверное, является массив или список (рис. 4.1).

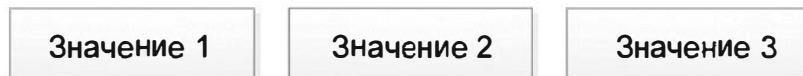


Рис. 4.1 ♦ Список значений

Сохранив этот список, мы впоследствии могли бы предъявлять к нему запросы, но для этого пришлось бы либо просмотреть каждое значение, либо хранить значения в неизменных местах и включить во внешнюю документацию сведения о том, в каком элементе массива какое значение хранится. Это означало бы, что необходимо определить специальное значение (`null`), чтобы сохранить предопределенный размер массива в случае, когда значения некоторых необязательных атрибутов, скажем номер факса или квартиры, неизвестны или отсутствуют. Конечно, массив – полезная структура данных, но ее семантика ограничена. Поэтому хотелось бы добавить к списку второе измерение: имена, сопоставляемые со значениями. Присвоив каждому элементу имя, мы получим структуру словаря, показанную на рис. 4.2.

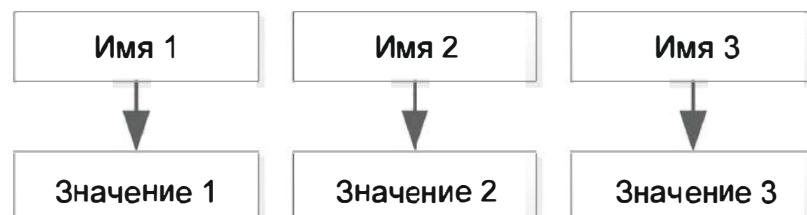


Рис. 4.2 ♦ Словарь, отображающий ключи на значения

Это уже лучше, поскольку в нашем распоряжении имеются имена значений. Если, к примеру, словарь содержит сведения о пользователе, столбцы можно было бы назвать `first_name`, `last_name`, `phone`, `email` и т. д. С такой структурой работать удобнее.

Но такая структура годится, только если имеется всего один экземпляр каждой сущности: один человек, один пользователь, одна гостиница, один твит. Она мало что дает, если требуется хранить несколько сущностей с одинаковой структурой, а нам-то нужно именно это. Нет никакого механизма для представления коллекции пар имя-значение как единого целого и никакого способа повторно использовать одни и те же имена столбцов. Поэтому необходимо придумать, как сгруппировать значения нескольких столбцов в нечто, допускающее адресацию. Нам нужен ключ для ссылки на группу столбцов, которые должны рассматриваться совместно. Нужны строки. А имея одну строку, мы сможем получить сразу все пары имя-значение, относящиеся к одной сущности, или только значения с интересующими нас именами. Такие пары имя-значение можно было бы назвать *столбцами*, а отдельные сущности, состоящие из некоторого множества столбцов, – *строками*. Уникальный идентификатор каждой строки можно было бы назвать *ключом строки* или *первичным ключом*. На рис. 4.3 показано содержимое простой строки: первичный ключ, состоящий из одного или нескольких столбцов, и дополнительные столбцы.



Рис. 4.3 ♦ Страна в Cassandra

В Cassandra *таблицей* называется логическая структура, объединяющая похожие данные. Например, могут быть таблицы `user`, `hotel`, `address book` и т. д. В этом смысле таблица в Cassandra аналогична реляционной таблице.

Но мы не обязаны хранить значения каждого столбца при сохранении новой сущности. Возможно, значения некоторых столбцов неизвестны. Например, у одних людей есть второй номер телефона,

а у других – нет. Или в форме на странице сайта, реализованного с использованием Cassandra, одни поля обязательны, а другие факультативны. Это нормально. Вместо того чтобы хранить null в качестве признака отсутствия значения и расходовать на это место, мы вообще не храним соответствующий столбец в данной строке. Получается разреженный многомерный массив, показанный на рис. 4.4.

При проектировании таблицы в традиционной реляционной базе данных мы обычно имеем дело с «сущностями», т. е. наборами атрибутов, описывающих некоторое существительное (hotel, user, product и т. д.). Мы не задумываемся о размере строк, потому что после того как решение о том, какое существительное представлено таблицей, принято, размер строки уже не подлежит обсуждению. Но при работе с Cassandra размер строки фиксирован не так жестко: строка может быть широкой или узкой в зависимости от количества присутствующих в ней столбцов.

Широкая строка может содержать очень много (десятки тысяч или даже миллионы) столбцов. Но обычно строк, содержащих столько столбцов, относительно немного. Наоборот, мы можем приблизиться к реляционной модели, когда число столбцов невелико, а строк много – это узкая (skinny) модель. На рис. 4.4 показана именно узкая модель.

Таблица

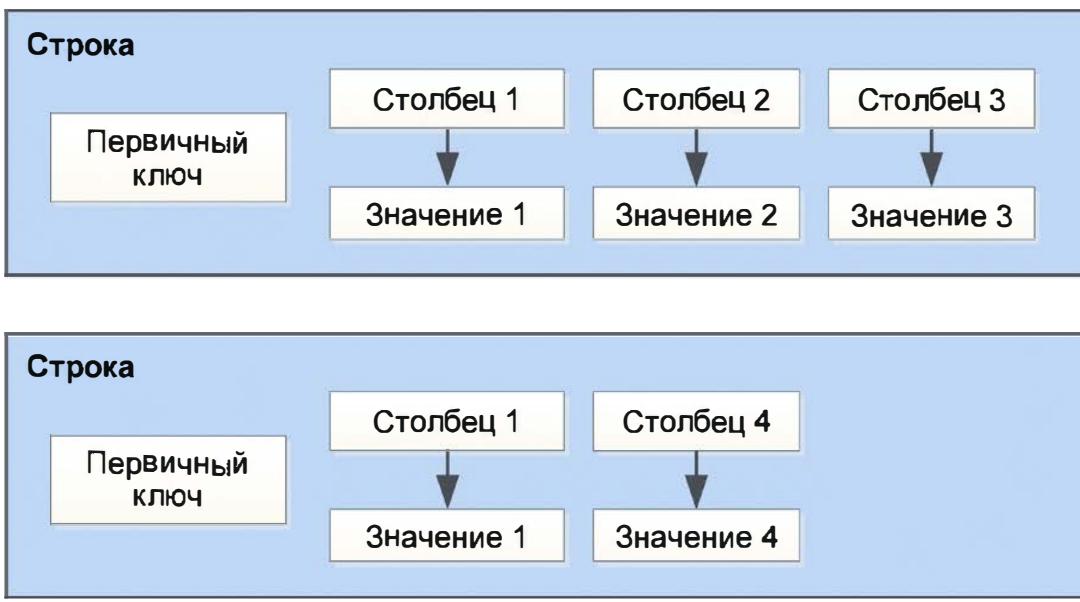


Рис. 4.4 ♦ Таблица в Cassandra

В Cassandra для представления широких строк, называемых также *разделами* (partition), используется специальный первичный ключ, называемый *составным ключом* (composite, или compound key). Составной ключ состоит из *ключа раздела* (partition key) и необязательного набора *кластерных столбцов* (clustering column). Ключ раздела служит для определения узлов, на которых хранятся строки, и сам может состоять из нескольких столбцов. Кластерные столбцы используются для управления сортировкой данных при хранении внутри раздела. Cassandra поддерживает также дополнительную конструкцию – *статический столбец* (static column), такие столбцы служат для хранения данных, которые не являются частью первичного ключа, но одинаковы во всех строках одного раздела.

На рис. 4.5 показано, как раздел уникально идентифицируется ключом раздела и как кластерные ключи используются для уникальной идентификации строк внутри раздела.

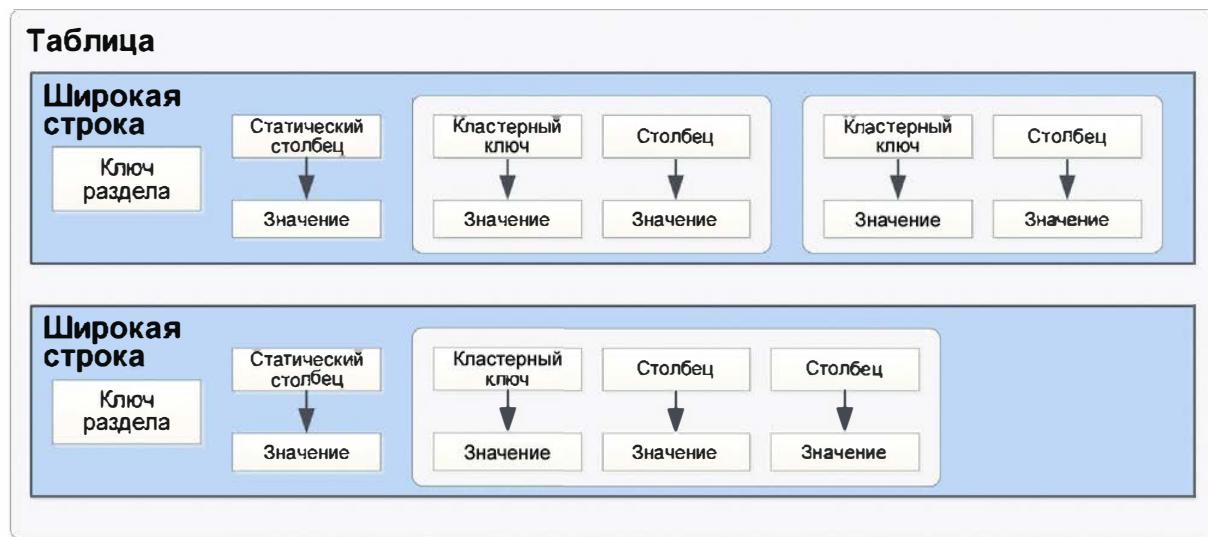


Рис. 4.5 ❖ Широкая строка в Cassandra

В этой главе нас будут интересовать простые первичные ключи, состоящие из одного столбца. В таком случае первичный ключ и ключ раздела – одно и то же, поскольку кластерных столбцов нет. Более сложные первичные ключи будут рассмотрены в главе 5.

Соберем все вместе. Ниже перечислены основные структуры данных в Cassandra:

- *столбец* – пара имя-значение;
- *строка* – контейнер столбцов, на который можно сослаться по первичному ключу;
- *таблица* – контейнер строк;

- *пространство ключей* – контейнер таблиц;
- *кластер* – контейнер пространств ключей, расположенных в одном или нескольких узлах.

Это взгляд снизу вверх на модель данных Cassandra. Познакомившись с основными терминами, перейдем к детальному изучению каждой структуры.

Кластер

Выше уже отмечалось, что база данных Cassandra изначально спроектирована для распределения между несколько машинами, которые работают совместно и представляются пользователю единым целым. Самая внешняя структура в Cassandra называется *клестером*, или *кольцом*, потому что Cassandra распределяет данные между узлами кластера, считая их закольцованными.

Пространства ключей

Кластер – это контейнер пространств ключей. *Пространство ключей* (*keyspace*) – самый внешний контейнер данных в Cassandra, во многом аналогичный реляционной базе данных. Как база данных является контейнером таблиц в реляционной модели, так пространство ключей – контейнер таблиц в модели данных Cassandra. Как и у реляционной базы данных, у пространства ключей есть имя и набор атрибутов, определяющих его поведение в целом.

Поскольку сейчас мы говорим о модели данных, отложим на будущее (до главы 7) вопросы конфигурирования кластеров и пространств ключей.

Таблицы

Таблица – это контейнер для упорядоченной коллекции строк, каждая из которых представляет собой упорядоченную коллекцию столбцов. Порядок определяется столбцами, назначенными в качестве ключей. Ниже мы увидим, что в Cassandra могут существовать и дополнительные ключи, помимо первичного.

В Cassandra при записи данных в таблицу задаются значения одного или нескольких столбцов. Эта коллекция значений называется *строкой*. Хотя бы одно из заданных значений должно быть *первичным ключом*, который играет роль уникального идентификатора строки.

Вернемся к таблице *user*, созданной в предыдущей главе. Напомним, что мы записали в нее строку данных, а затем прочитали ее командой `SELECT` в `cqlsh`:

```
cqlsh:my_keyspace> SELECT * FROM user WHERE first_name='Bill';
first_name | last_name
-----+-----
Bill      |    Nguyen
(1 rows)
```

Из последней строки видно, что возвращена одна строка. В этой строке поле `first_name` равно «`Bill`», это первичный ключ, идентифицирующий строку.



Для доступа к данным необходим первичный ключ

Это важная деталь – команды `SELECT`, `INSERT`, `UPDATE` и `DELETE` в CQL оперируют в терминах строк.

Выше мы сказали, что при добавлении строки в таблицу необязательно задавать значения всех столбцов. Проверим это на таблице `user`, для чего воспользуемся командой `ALTER TABLE`, а затем проконтролируем результат командой `DESCRIBE TABLE`:

```
cqlsh:my_keyspace> ALTER TABLE user ADD title text;
cqlsh:my_keyspace> DESCRIBE TABLE user;

CREATE TABLE my_keyspace.user (
    first_name text PRIMARY KEY,
    last_name text,
    title text
) ...
```

Как видим, добавлен столбец `title`. Мы опустили часть выходной информации с различными параметрами таблицы. Об этих параметрах и их настройке речь пойдет в главе 7.

Теперь запишем две строки, задав в них разные столбцы, и посмотрим, что получилось.

```
cqlsh:my_keyspace> INSERT INTO user (first_name, last_name, title)
VALUES ('Bill', 'Nguyen', 'Mr.');
cqlsh:my_keyspace> INSERT INTO user (first_name, last_name) VALUES
('Mary', 'Rodriguez');
cqlsh:my_keyspace> SELECT * FROM user;

first_name | last_name | title
-----+-----+-----
Mary      | Rodriguez | null
Bill      | Nguyen    | Mr.
(2 rows)
```

Разобравшись со структурой таблицы и поэкспериментировав с моделированием данных, перейдем к рассмотрению столбцов.

Столбцы

Столбец – самая элементарная структурная единица в модели данных Cassandra. До сих пор мы видели, что у столбца есть имя и значение. Также при определении столбца задается тип, чтобы ограничить множество возможных значений. Мы изучим различные типы данных, но сначала познакомимся с другими атрибутами столбцов, которые еще не обсуждали: временными метками и временем жизни. Эти атрибуты – ключ к пониманию того, как в Cassandra используется время для поддержания актуальности данных.

Временные метки

Всякий раз, как мы записываем данные, Cassandra генерирует временную метку для каждого обновленного столбца. Эти метки используются для разрешения конфликтов при изменении одного и того же значения со стороны нескольких клиентов. В общем случае сохраняется изменение с самой поздней временной меткой.

Посмотрим на временные метки, сгенерированные в ходе предыдущих операций записи, добавив в команду SELECT функцию `writetime()` для столбца `lastname`:

```
cqlsh:my_keyspace> SELECT first_name, last_name,
    writetime(last_name) FROM user;

first_name | last_name | writetime(last_name)
-----+-----+-----
Mary | Rodriguez | 1434591198790252
Bill | Nguyen | 1434591198798235

(2 rows)
```

Естественно было бы ожидать, что, запросив временную метку для столбца `first_name`, мы получим аналогичный результат. Однако оказывается, что запрашивать временные метки для столбцов, составляющих первичный ключ, запрещено.

```
cqlsh:my_keyspace> SELECT WRITETIME(first_name) FROM user;
InvalidRequest: code=2200 [Invalid query] message="Cannot use
selection function writeTime on PRIMARY KEY part first_name"
```

Cassandra позволяет также задать желаемую временную метку при выполнении записи. Для этого воспользуемся командой CQL `UPDATE` с необязательной фразой `USING TIMESTAMP`:

```
cqlsh:my_keyspace> UPDATE user USING TIMESTAMP 1434373756626000
    SET last_name = 'Boateng' WHERE first_name = 'Mary' ;
cqlsh:my_keyspace> SELECT first_name, last_name,
    WRITETIME(last_name) FROM user WHERE first_name = 'Mary';
first_name | last_name | writetime(last_name)
-----+-----+-----
Mary | Boateng | 1434373756626000
(1 rows)
```

В результате в строку с первичным ключом «Mary» добавляется столбец `last_name`, и для него устанавливается заданная нами временная метка.



Работа с временными метками

Задавать временную метку при записи необязательно. Эта возможность используется главным образом, когда есть опасения, что какая-то операция записи может затереть актуальные данные устаревшими. Прибегать к ней следует с осторожностью.

В настоящее время в `cqlsh` нет способа преобразовать временные метки, генерируемые функцией `writetime()`, во что-то более понятное.

Время жизни (TTL)

Весьма полезное свойство Cassandra – возможность задать срок хранения данных, необходимость в которых отпала. Это очень гибкий механизм, работающий на уровне значений отдельных столбцов. Время жизни (time to live – TTL) хранится для каждого значения в столбце и показывает, как долго это значение следует сохранять.

По умолчанию TTL равно `null`, т. е. срок хранения записанных данных не ограничен. Убедимся в этом, добавив в команду `SELECT` функцию `TTL()` для столбца `last_name` в строке, относящейся к Mary:

```
cqlsh:my_keyspace> SELECT first_name, last_name, TTL(last_name)
    FROM user WHERE first_name = 'Mary';
first_name | last_name | ttl(last_name)
-----+-----+-----
Mary | Boateng | null
(1 rows)
```

Теперь установим TTL для столбца `last_name` равным одному часу (3600 секунд), добавив в команду `UPDATE` фразу `USING TTL`:

```
cqlsh:my_keyspace> UPDATE user USING TTL 3600 SET last_name =
    'McDonald' WHERE first_name = 'Mary' ;
cqlsh:my_keyspace> SELECT first_name, last_name, TTL(last_name)
    FROM user WHERE first_name = 'Mary';
first_name | last_name | ttl(last_name)
-----+-----+-----
Mary | McDonald | 3588
(1 rows)
```

Как видим, уже пошел обратный отсчет TTL – учтены те несколько секунд, которые были потрачены на ввод второй команды. Если выполнить эту же команду через час, то мы увидим, что столбец last_name в строке Mary будет содержать null. Время жизни можно задать и в команде INSERT с помощью той же фразы USING TTL.



Использование TTL

Напомним, что TTL хранится на уровне столбца. В настоящее время не существует механизма задания TTL на уровне всей строки. Как и в случае временной метки, невозможно ни установить, ни получить значение TTL для столбцов, входящих в состав первичного ключа. Кроме того, задавать TTL можно только одновременно с заданием значения столбца.

Чтобы задать TTL для всей строки, придется указать значения всех столбцов, не входящих в состав первичного ключа, в команде INSERT или UPDATE.

Типы данных в CQL

Мы стали лучше понимать, как в Cassandra представляются столбцы и в т. ч. метаданные, связанные со временем. Теперь посмотрим, какие типы данных имеются в нашем распоряжении.

Мы уже видели, что у любого столбца в таблице имеется тип. До сих пор мы использовали только тип varchar, но в CQL есть и много других: символьные и числовые, коллекции и пользовательские типы. Мы опишем все эти типы и приведем примеры их использования, чтобы вам было проще сделать правильный выбор.

Числовые типы данных

CQL поддерживает все обычные числовые типы, включая целочисленные и с плавающей точкой. Они аналогичны стандартным типам в Java и других языках.

int

32-разрядное целое со знаком (как в Java).

bigint

64-разрядное целое со знаком (эквивалентно типу long в Java).

smallint

16-разрядное целое со знаком (эквивалентно типу short в Java).

tinyint

8-разрядное целое со знаком (как в Java).

varint

Целое со знаком переменной точности (эквивалентно java.math.BigInteger).

float

32-разрядное с плавающей точкой, удовлетворяющее стандарту IEEE-754 (как в Java).

double

64-разрядное с плавающей точкой, удовлетворяющее стандарту IEEE-754 (как в Java).

decimal

Десятичное переменной точности (эквивалентно java.math.BigInteger).



Дополнительные целые типы

Типы smallint и tinyint добавлены начиная с версии Cassandra 2.2.

Во многих языках программирования имеются перечисляемые типы, но в CQL нет их прямого эквивалента. Обычно перечисляемые значения хранятся в виде строк. Так, метод Enum.name() можно использовать для преобразования перечисляемого значения в тип String и последующей записи в Cassandra в виде текста, а метод Enum.valueOf() – для преобразования текста обратно в перечисляемое значение.

Текстовые типы данных

CQL предлагает два типа данных для представления текста, и с одним из них (`text`) мы уже неоднократно встречались.

`text`, `varchar`

Синонимы, служат для представления строки символов в кодировке UTF-8.

ascii

Строка символов в кодировке ASCII.

UTF-8 – более поздний стандарт, он широко используется для представления текста и поддерживает интернационализацию, поэтому при создании новых таблиц рекомендуем использовать тип `text`, а не `ascii`. Тип `ascii` полезен, когда приходится иметь дело с унаследованными данными в кодировке ASCII.



Установка локали в cqlsh

По умолчанию `cqlsh` выводит управляющие и другие не имеющие графического представления символы в виде числового кода с обратной косой чертой. Но отображение символов, отсутствующих в ASCII, можно контролировать, установив локаль в переменной среды `$LANG` до запуска программы. Для получения дополнительных сведений выполните в `cqlsh` команду `HELP TEXT_OUTPUT`.

Типы времени и идентификации

Идентификация таких элементов данных, как строки и разделы, важна в любой модели данных, так как позволяет обращаться к данным. В Cassandra имеется несколько типов, полезных для определения уникальных ключей разделов.

timestamp

Мы уже отмечали, что у каждого столбца имеется временная метка, показывающая, когда он был в последний раз модифицирован. Но временная метка может быть и значением столбца. Время можно представить 64-разрядным целым со знаком, но обычно гораздо полезнее вводить временную метку в одном из нескольких форматов данных, описанных в стандарте ISO 8601, например:

```
2015-06-15 20:05-0700  
2015-06-15 20:05:07-0700  
2015-06-15 20:05:07.013-0700  
2015-06-15T20:05-0700  
2015-06-15T20:05:07-0700  
2015-06-15T20:05:07.013+-0700
```

Рекомендуется всегда явно указывать часовой пояс, а не полагаться на конфигурацию часовых поясов в операционной системе.

date, time

В версиях до Cassandra 2.1 включительно для представления времени существовал только тип `timestamp`, включающий как дату, так

и время. В версии 2.2 были добавлены типы `date` и `time`, позволяющие представлять эти компоненты независимо, т. е. дату без времени и время суток без привязки к дате. Как и тип `timestamp`, они поддерживают форматы, описанные в стандарте ISO 8601.

В Java 8 появились новые типы в пакете `java.time`, но тип `date` отображается на внутренний тип Cassandra ради сохранения совместимости с предыдущими версиями JDK. Тип `time` отображается на тип Java `long` и представляет число наносекунд после полуночи.

uuid

Универсальный уникальный идентификатор (UUID) – это 128-разрядное значение, биты которого организованы по-разному, но наиболее часто встречаются типы 1 и 4. В CQL тип `uuid` соответствует UUID типа 4, основанного исключительно на случайных числах. Обычно UUID записывают в виде разделенных дефисами групп шестнадцатеричных цифр, например:

```
1a6300ca-0572-4736-a393-c0b7229e193e
```

Часто тип `uuid` применяется в качестве суррогатного ключа самостоятельно или в сочетании с другими значениями.

Поскольку длина UUID конечна, нельзя дать гарантию абсолютной уникальности. Но в большинстве операционных систем и языков программирования, в частности в `cqlsh`, имеются утилиты для генерации достаточно уникальных идентификаторов. Можно получить UUID типа 4 от функции `uuid()` и использовать его в команде `INSERT` или `UPDATE`.

timeuuid

Это UUID типа 1, основанный на МАС-адресе компьютера, системном времени и порядковом номере во избежание дубликатов. Он часто используется в качестве бесконфликтной временной метки. В `cqlsh` есть несколько вспомогательных функций для работы с типом `timeuuid`: `now()`, `dateOf()` и `unixTimestampOf()`.

Наличие таких функций – одна из причин, по которым тип `timeuuid` используется чаще, чем `uuid`.

В нашем сквозном примере было бы разумно приписать каждому пользователю уникальный идентификатор, потому что встречаются люди с одинаковыми именами и, следовательно, столбец `first_name` не будет уникальным ключом. Если бы мы начинали с нуля, то сделали бы первичным ключом такой идентификатор, но сейчас просто добавим содержащий его столбец.



Первичный ключ – это навсегда

После создания таблицы модифицировать первичный ключ уже невозможно, потому что он определяет распределение данных в кластере и, что еще важнее, порядок хранения данных на диске.

Добавим столбец `id` типа `uuid`:

```
cqlsh:my_keyspace> ALTER TABLE user ADD id uuid;
```

Затем определим идентификатор для Mary с помощью функции `uuid()` и посмотрим, что получилось:

```
cqlsh:my_keyspace> UPDATE user SET id = uuid() WHERE first_name =
  'Mary';
cqlsh:my_keyspace> SELECT first_name, id FROM user WHERE
  first_name = 'Mary';
first_name | id
-----+-----
  Mary | e43abc5d-6650-4d13-867a-70cbad7feda9
(1 rows)
```

Обратите внимание, что `id` напечатан в формате UUID.

Теперь структура таблицы стала более надежной, но мы можем расширять ее и дальше.

Прочие простые типы данных

CQL предоставляет и другие простые типы данных, не попадающие ни в одну из рассмотренных выше категорий.

`boolean`

Может принимать значения `true` и `false`. `cqlsh` позволяет вводить их в любом регистре, но выводит `True` или `False`.

`blob`

Большой двоичный объект (блоб) – так в разговорной речи называют произвольный массив байтов. Тип `blob` полезен для хранения мультимедийных и других двоичных файлов. Cassandra не проверяет байты, хранящиеся в блоке. В CQL такие данные представляются в шестнадцатеричном виде, например `0x00000ab83cf0`. Чтобы преобразовать в тип `blob` произвольные текстовые данные, можно воспользоваться функцией `textAsBlob()`. Для получения дополнительных сведений выполните команду `HELP BLOB_INPUT`.

inet

Этот тип служит для представления интернет-адресов версий IPv4 и IPv6. cqlsh принимает IPv4-адреса в любом допустимом формате – с точками и без, записанные в десятичном, восьмеричном или шестнадцатеричном виде. Но при выводе адреса печатаются в точечно-десятичной нотации, например 192.0.2.235.

IPv6-адреса представляются восьмью группами по четыре шестнадцатеричные цифры, разделенными двоеточием, например 2001:0db8:85a3:0000:0000:8a2e:0370:7334. Спецификация IPv6 позволяет опускать нули в начале группы, так что при выводе результатов SELECT это значение печатается следующим образом: 2001:db8:85a3:a::8a2e:370:7334.

counter

Тип данных counter позволяет представить 64-разрядное целое со знаком, значение которого нельзя установить непосредственно, а можно только увеличивать или уменьшать. Cassandra – одна из немногих баз данных, в которых реализовано свободное от конкуренции увеличение счетчика, видимого в нескольких ЦОДах. Счетчики нередко применяются для ведения статистики, например количества просмотров страницы, твитов, сообщений в журнале и т. д. На тип counter наложено несколько специфических ограничений. Он не может быть частью первичного ключа. Если в строке имеется счетчик, то все остальные столбцы, кроме входящих в состав первичного ключа, тоже должны иметь тип counter.



Предупреждение по поводу счетчиков

Напоминаем: операторы инкремента и декремента не являются идемпотентными. Не существует операции прямого сброса счетчика, но ее можно аппроксимировать: прочитать значение счетчика, а затем уменьшить на это значение. К сожалению, правильная работа в этом случае не гарантируется, потому что значение счетчика может измениться между чтением и записью.

Коллекции

Допустим, мы хотим добавить в таблицу user поддержку нескольких адресов электронной почты. Можно было бы дополнительно создать столбцы email2, email3 и т. д. Это работает, но плохо масштабируется и грозит большим объемом переделок. Гораздо проще рассматривать почтовые адреса как группу, или «коллекцию». CQL

предлагает три типа коллекций: множества, списки и словари. Рассмотрим их поочередно.

`set`

Тип данных `set` предназначен для хранения коллекции элементов. Элементы множества не упорядочены, но `cqlsh` возвращает их в отсортированном виде. В частности, текстовые значения возвращаются в алфавитном порядке. Множества могут содержать как элементы рассмотренных ранее простых типов, так и пользовательских типов (о них чуть ниже) и даже другие коллекции. Одно из преимуществ типа `set` – возможность вставлять дополнительные элементы без предварительного чтения множества.

Модифицируем нашу таблицу `user`, добавив в нее множество адресов:

```
cqlsh:my_keyspace> ALTER TABLE user ADD emails set<text>;
```

Затем добавим почтовый адрес для Mary и проверим, что получилось:

```
cqlsh:my_keyspace> UPDATE user SET emails = {  
    'mary@example.com' } WHERE first_name = 'Mary';  
cqlsh:my_keyspace> SELECT emails FROM user WHERE first_name =  
    'Mary';  
  
emails  
-----  
{ 'mary@example.com' }  
(1 rows)
```

Отметим, что, добавив первый адрес, мы заменили предыдущее содержимое множества, в данном случае – `null`. Следующий адрес можно добавить, не заменяя множество целиком, применив оператор конкатенации:

```
cqlsh:my_keyspace> UPDATE user SET emails = emails + {  
    'mary.mcdonald.AZ@gmail.com' } WHERE first_name = 'Mary';  
cqlsh:my_keyspace> SELECT emails FROM user WHERE first_name =  
    'Mary';  
  
emails  
-----  
{ 'mary.mcdonald.AZ@gmail.com', 'mary@example.com' }  
(1 rows)
```



Другие операции над множествами

С помощью оператора вычитания можно удалить элементы из множества: `SET emails = emails - { 'mary@example.com' }`. Или очистить сразу все множество, воспользовавшись нотацией пустого множества: `SET emails = {}`.

`list`

Тип данных `list` служит для хранения упорядоченного списка элементов. По умолчанию значения хранятся в порядке вставки. Добавим в таблицу `user` список телефонов:

```
cqlsh:my_keyspace> ALTER TABLE user ADD
    phone_numbers list<text>;
```

Затем добавим телефон Mary и проверим, что все прошло удачно:

```
cqlsh:my_keyspace> UPDATE user SET phone_numbers = [
    '1-800-999-9999' ] WHERE first_name = 'Mary';
cqlsh:my_keyspace> SELECT phone_numbers FROM user WHERE
    first_name = 'Mary';

phone_numbers
-----
[ '1-800-999-9999' ]
(1 rows)
```

Добавим второй номер телефона:

```
cqlsh:my_keyspace> UPDATE user SET phone_numbers =
    phone_numbers + [ '480-111-1111' ] WHERE first_name = 'Mary';
cqlsh:my_keyspace> SELECT phone_numbers FROM user WHERE
    first_name = 'Mary';

phone_numbers
-----
[ '1-800-999-9999', '480-111-1111' ]
(1 rows)
```

Как видим, второй номер оказался в конце списка.



Можно было бы добавить номер в начало списка, поменяв порядок слагаемых: `SET phone_numbers = ['4801234567'] + phone_numbers`.

Можно заменить один конкретный элемент списка, указав его индекс:

```
cqlsh:my_keyspace> UPDATE user SET phone_numbers[1] =
    '480-111-1111' WHERE first_name = 'Mary';
```

Как и в случае множеств, оператор вычитания применяется для удаления элемента с указанным значением:

```
cqlsh:my_keyspace> UPDATE user SET phone_numbers =
    phone_numbers - [ '480-111-1111' ] WHERE first_name = 'Mary';
```

Наконец, можно удалить элемент по индексу:

```
cqlsh:my_keyspace> DELETE phone_numbers[0] from user WHERE
    first_name = 'Mary';
```

`map`

Тип данных `map` служит для хранения пар ключ-значение. Ключи и значения могут иметь любой тип, кроме `counter`. Воспользуемся типом `map` для хранения информации о входах пользователя в систему. Создадим столбец, в котором будем отслеживать продолжительность сеанса в секундах, причем ключом будет значение типа `timeuuid`:

```
cqlsh:my_keyspace> ALTER TABLE user ADD
    login_sessions map<timeuuid, int>;
```

Затем добавим информацию о двух сеансах для Mary и полюбуемся на результат:

```
cqlsh:my_keyspace> UPDATE user SET login_sessions =
    { now(): 13, now(): 18} WHERE first_name = 'Mary';
cqlsh:my_keyspace> SELECT login_sessions FROM user WHERE
    first_name = 'Mary';

login_sessions
-----
{6061b850-14f8-11e5-899a-a9fa1d00bce: 13,
 6061b851-14f8-11e5-899a-a9fa1d00bce: 18}

(1 rows)
```

Можно также сослаться на отдельный элемент словаря по ключу.

Типы коллекций весьма полезны в случаях, когда нужно сохранить переменное число элементов в одном столбце.

Пользовательские типы

А что, если мы захотим хранить физические адреса пользователей? Можно было бы завести для этого один текстовый столбец, но тогда задача разбора адреса ляжет на приложение. Лучше бы определить

структуру для хранения адреса, выделив в ней поля для отдельных его частей.

По счастью, Cassandra позволяет определять собственные типы, а затем создавать столбцы, имеющие пользовательские типы (UDT). Создадим тип address, разбив для наглядности команду на несколько строк:

```
cqlsh:my_keyspace> CREATE TYPE address (
    ... street text,
    ... city text,
    ... state text,
    ... zip_code int);
```

Видимость UDT ограничена пространством ключей, в котором он определен. Можно было бы написать CREATE TYPE my_keyspace.address. Выполнив затем команду DESCRIBE KEYSPACE my_keyspace, мы увидим, что тип address является частью определения пространства имен.

Теперь попробуем воспользоваться определенным типом адреса в таблице user. Если вы работаете с версией Cassandra 2.1 или более ранней, то возникнет ошибка:

```
cqlsh:my_keyspace> ALTER TABLE user ADD
    addresses map<text, address>;
InvalidRequest: code=2200 [Invalid query] message="Non-frozen
collections are not allowed inside collections: map<text,
address>"
```

В чем дело? Оказывается, что пользовательский тип данных рассматривается как коллекция, поскольку его реализация похожа на реализацию set, list или map.



Замораживание коллекций

Версии Cassandra младше 2.2 не в полной мере поддерживают вложенные коллекции. Точнее, не поддерживается возможность доступа к отдельным атрибутам вложенной коллекции, поскольку в этой реализации вложенная коллекция сериализуется как один объект.

В качестве механизма совместимости с будущими версиями сообщество Cassandra ввело концепцию замораживания. Пока что вложить одну коллекцию в другую можно, пометив ее как замороженную. В будущем, когда будет реализована полноценная поддержка вложенных коллекций, появится механизм «размораживания» вложенных коллекций, позволяющий обратиться к отдельным атрибутам.

Замороженная коллекция может быть первичным ключом.

После этого краткого отступления о замораживании и вложенных коллекциях вернемся к нашей таблице и на этот раз сделаем адрес замороженным:

```
cqlsh:my_keyspace> ALTER TABLE user ADD addresses map<text,
frozen<address>>;
```

Теперь добавим домашний адрес Mary:

```
cqlsh:my_keyspace> UPDATE user SET addresses = addresses +
{ 'home': { street: '7712 E. Broadway', city: 'Tucson',
state: 'AZ', zip_code: 85715} } WHERE first_name = 'Mary';
```

Итак, мы закончили изучение различных типов. Теперь отступим на шаг назад и посмотрим, что мы создали в пространстве ключей my_keyspace:

```
cqlsh:my_keyspace> DESCRIBE KEYSPACE my_keyspace ;

CREATE KEYSPACE my_keyspace WITH replication = {'class':
'SimpleStrategy', 'replication_factor': '1'} AND
durable_writes = true;

CREATE TYPE my_keyspace.address (
    street text,
    city text,
    state text,
    zip_code int
);

CREATE TABLE my_keyspace.user (
    first_name text PRIMARY KEY,
    addresses map<text, frozen<address>>,
    emails set<text>,
    id uuid,
    last_name text,
    login_sessions map<timeuuid, int>,
    phone_numbers list<text>,
    title text
) WITH bloom_filter_fp_chance = 0.01
AND caching = '{"keys":"ALL", "rows_per_partition":"NONE"}'
AND comment = ''
AND compaction = {'min_threshold': '4', 'class':
'org.apache.cassandra.db.compaction.
SizeTieredCompactionStrategy', 'max_threshold': '32'}
AND compression = {'sstable_compression':
'org.apache.cassandra.io.compress.LZ4Compressor'}
```

```
AND dclocal_read_repair_chance = 0.1
AND default_time_to_live = 0
AND gc_grace_seconds = 864000
AND max_index_interval = 2048
AND memtable_flush_period_in_ms = 0
AND min_index_interval = 128
AND read_repair_chance = 0.0
AND speculative_retry = '99.0PERCENTILE';
```

Вторичные индексы

Попытавшись опросить таблицу Cassandra по столбцу, не являющемуся первичным ключом, вы обнаружите, что это запрещено. Рассмотрим, к примеру, таблицу пользователей из предыдущей главы, в которой первичным ключом был столбец `first_name`. Попытка выполнить запрос по столбцу `last_name` дает такой результат:

```
cqlsh:my_keyspace> SELECT * FROM user WHERE last_name = 'Nguyen';
InvalidRequest: code=2200 [Invalid query] message="No supported
secondary index found for the non primary key columns restrictions"
```

Как следует из сообщения об ошибке, мы должны создать *вторичный индекс* по столбцу `last_name`. Вторичным называется индекс по столбцу, не являющемуся частью первичного ключа.

```
cqlsh:my_keyspace> CREATE INDEX ON user ( last_name );
```

Мы можем присвоить индексу необязательное имя: `CREATE INDEX <name> ON....` Если имя не задано, то `cqlsh` построит его автоматически по шаблону `<имя таблицы>_<имя столбца>_idx`. Имя только что созданного индекса можно узнать с помощью команды `DESCRIBE KEYSPACE`:

```
cqlsh:my_keyspace> DESCRIBE KEYSPACE;
...
CREATE INDEX user_last_name_idx ON my_keyspace.user (last_name);
```

После того как индекс создан, запрос работает, как и ожидалось:

```
cqlsh:my_keyspace> SELECT * FROM user WHERE last_name = 'Nguyen';
```

first_name	last_name
Bill	Nguyen

(1 rows)

Индексы можно строить не только по столбцам простых типов. Разрешаются индексы по значениям элементов коллекций. Например, нас может интересовать поиск по физическому адресу пользователя, по адресу его электронной почты, по номеру телефона, а эти атрибуты в нашей реализации имеют типы `map`, `set` и `list` соответственно.

```
cqlsh:my_keyspace> CREATE INDEX ON user ( addresses );
cqlsh:my_keyspace> CREATE INDEX ON user ( emails );
cqlsh:my_keyspace> CREATE INDEX ON user ( phone_numbers );
```

Что касается словарей, то можно индексировать как ключи – с помощью синтаксической конструкции `KEYS(addresses)`, так и значения – это режим по умолчанию. Создавать индексы одновременно по ключам и значениям словаря не разрешается.

Наконец, для удаления индекса служит команда `DROP INDEX`:

```
cqlsh:my_keyspace> DROP INDEX user_last_name_idx;
```



Подвохи вторичных индексов

Поскольку данные в Cassandra распределены между несколькими узлами, на каждом узле должна быть собственная копия вторичного индекса, построенная по тем же данным, хранящимся в размещенных на нем разделах. Поэтому запросы с участием вторичного индекса обычно затрагивают больше узлов, что делает их более накладными.

Использовать вторичные индексы не рекомендуется в следующих специальных случаях.

- Столбцы с большим числом различных элементов. Так, индексирование столбца `user.addresses` обойдется очень дорого, потому что подавляющее число адресов уникально.
- Столбцы с очень малым числом различных элементов. Так, не имеет смысла индексировать столбец `user.title` (обращение), чтобы поддерживать поиск всех обращений «Mrs.» (миссис) в таблице пользователей, поскольку это приведет к появлению очень длинной строки в индексе.
- Столбцы, для которых часто производятся операции обновления или удаления. При работе с такими индексами могут возникать ошибки, если объем удаленных данных (надгробий) растет так быстро, что процесс уплотнения не справляется.

Для достижения оптимальной производительности чтения обычно лучше использовать не вторичные индексы, а материализованные представления, о которых мы узнаем в главе 5. Однако вторичные индексы могут оказаться подспорьем для поддержки запросов, которые не были учтены на этапе начального проектирования модели данных.

SASI: новая реализация вторичных индексов

В версию Cassandra 3.4 включена альтернативная реализация вторичных индексов – SSTable Attached Secondary Index (SASI). Эта технология была разработана компанией Apple, и ее открытый код был выпущен в виде API вторичных индексов для Cassandra. Как следует из названия, SASI-индексы вычисляются и хранятся как часть каждого файла SSTable, в отличие от оригинальной реализации Cassandra, где индексы хранились в отдельных «скрытых» таблицах.

Реализация SASI существует с традиционными вторичными индексами, а для создания SASI-индекса служит команда CQL `CREATE CUSTOM INDEX`:

```
CREATE CUSTOM INDEX user_last_name_sasi_idx ON user (last_name)
USING 'org.apache.cassandra.index.sasi.SASIIndex';
```

Функциональность SASI-индексов шире, чем у традиционных вторичных индексов; в частности, они допускают поиск со сравнением (на больше или меньше) по индексированным столбцам. Для индексированных столбцов можно также пользоваться новым ключевым словом CQL `LIKE`. Так, чтобы найти пользователей, фамилии которых начинаются с буквы «N», нужно написать:

```
SELECT * FROM user WHERE last_name LIKE 'N%';
```

И хотя SASI-индексы работают быстрее традиционных, так как исключено чтение дополнительных таблиц, все равно при работе с ними нужно читать данные с большего числа узлов, чем при денормализованной структуре.

Резюме

В этой главе мы вкратце рассмотрели модель данных Cassandra, состоящую из кластеров, пространств ключей, таблиц, ключей, строк и столбцов. По ходу дела мы много узнали о синтаксисе CQL и приобрели опыт работы с таблицами и столбцами в `cqlsh`. Если вы хотите больше узнать о языке CQL, обратитесь к полной спецификации языка по адресу <https://cassandra.apache.org/doc/latest/cql/index.html>.

Глава 5

Моделирование данных

В этой главе мы расскажем о проектировании моделей данных для Cassandra, в т. ч. о процессе моделирования и о нотации. Для иллюстрации мы возьмем пример приложения, с которым будем работать на протяжении нескольких последующих глав. Это поможет понять, как все кусочки складываются в единое целое. Попутно мы познакомимся с инструментом для управления CQL-скриптами.

Построение концептуальной модели данных

Для начала создадим простую модель предметной области, вполне понятную в реляционном мире, а затем посмотрим, как перейти от реляционной к распределенной модели на основе хэш-таблиц, принятой в Cassandra.

Для этого примера нам понадобится нечто такое, что позволило бы продемонстрировать различные структуры данных и паттерны проектирования, но не настолько сложное, чтобы погрязнуть в деталях. К тому же предметная область должна быть всем знакома, чтобы сосредоточиться на Cassandra, а не на описании того, что мы пытаемся смоделировать.

Поэтому возьмем всем известное приложение для бронирования номеров в отеле.

В концептуальной модели будут участвовать отели, постояльцы, коллекции номеров в каждом отеле, цены и доступность номеров, а также сведения о бронировании. Отели обычно предлагают также

«список достопримечательностей»: парки, музеи, крупные магазины, памятники и другие места поблизости, которые могут заинтересовать гостей. Для отелей и достопримечательностей необходимо еще хранить географические координаты, чтобы их можно было найти на карте и вычислить расстояния между объектами.

Концептуальная модель предметной области изображена на рис. 5.1 в виде диаграммы сущность–связь, предложенной Питером Чэнем. Сущности предметной области представлены прямоугольниками, а их атрибуты – овалами. Атрибуты, соответствующие уникальным идентификаторам объектов, подчеркнуты. Связи между атрибутами представлены ромбами, а на линии, соединяющей связь с сущностью, показана кратность связи.

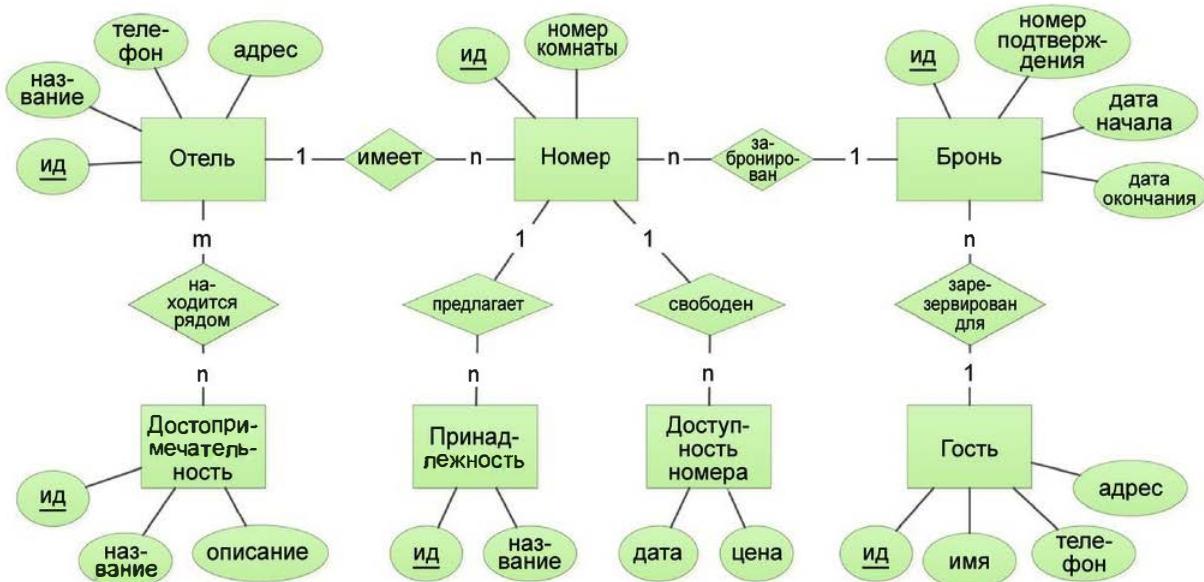


Рис. 5.1 ♦ Диаграмма сущность–связь для предметной области бронирования номеров в отеле

Понятно, что в реальности нужно учитывать еще много деталей, так что модель будет гораздо сложнее. Например, известно, что цены гостиничных номеров изменяются динамически, а при вычислении цены учитывается много факторов. Мы ограничимся картиной, которая достаточно сложна, чтобы не умереть со скуки и продемонстрировать важные идеи, но достаточно проста, чтобы не отвлекаться от изучения Cassandra.

Проектирование реляционной базы данных

Приступая к разработке нового приложения с использованием реляционной базы данных, мы начинаем с создания модели предметной области в виде набора правильно нормализованных таблиц и внешних ключей, связывающих данные в разных таблицах.

На рис. 5.2 показано, как могла бы быть устроена реляционная база данных нашего приложения. В реляционной модели имеются «связующие» таблицы, необходимые для реализации связей типа многое-ко-многим между отелями и достопримечательностями, номерами и принадлежностями (amenity), номерами и доступностью и гостями и номерами (в результате бронирования).

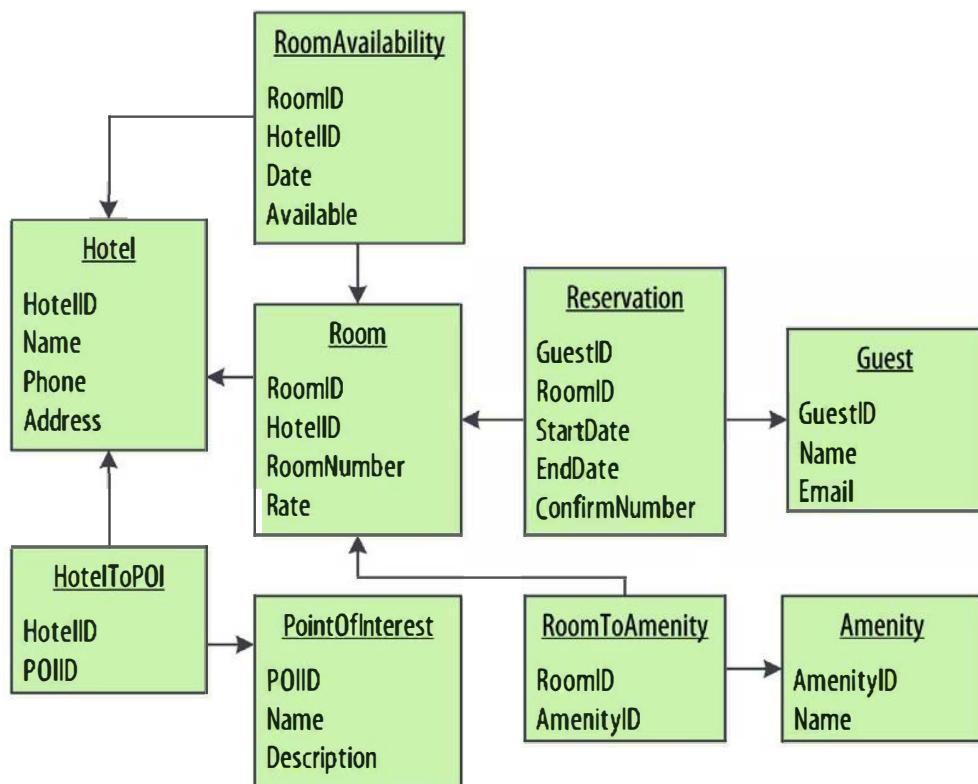


Рис. 5.2 ♦ Реляционная база данных для простой системы бронирования гостиничных номеров

Различия в проектировании для РСУБД и Cassandra

Поскольку эта книга о Cassandra, то мы, конечно, хотим построить модель данных так, чтобы их можно было хранить в Cassandra. Но прежде чем браться за дело, рассмотрим основные различия между моделированием данных для Cassandra и для реляционной базы.

Отсутствие соединений

В Cassandra нет операций соединения. Соединение таблиц, если оно необходимо, придется производить на стороне клиента, либо можно создать дополнительную денормализованную таблицу для хранения результатов соединения. При создании моделей данных для Cassandra предпочтителен второй вариант. Соединение на стороне клиента применяется очень редко, лучше продублировать (денормализовать) данные.

Отсутствие ссылочной целостности

Несмотря на поддержку облегченных транзакций и пакетов команд, в Cassandra нет понятия ссылочной целостности таблиц. В реляционной базе данных мы можем определить в таблице внешний ключ для ссылки на первичный ключ записи в другой таблице. В Cassandra такого механизма нет. При проектировании схемы по-прежнему следует хранить идентификаторы связанных сущностей, но операции типа каскадного удаления отсутствуют.

Денормализация

При проектировании реляционной базы данных нас учат, как важна нормализация. Но при работе с Cassandra это вовсе не преимущество, потому что оптимальная производительность достигается, когда модель данных денормализована. Часто бывает, что и в реляционной базе приходится идти на денормализацию, и тому есть две причины. Первая – производительность. Компании попросту не удается добиться нужной производительности, если необходимо соединять накопившиеся за много лет данные, поэтому она производит денормализацию, подстраиваясь под нужды известных запросов. Этот подход работает, но идет вразрез с идеологией реляционной модели, так что в конце концов возникает вопрос, а является ли реляционная база оптимальным решением в данных условиях.

Вторая причина намеренной денормализации – структура документа, нуждающегося в длительном хранении. То есть имеется основная таблица, которая ссылается на много внешних таблиц, данные в которых со временем изменяются, однако нам необходимо сохранить для истории документ в том виде, в котором он существовал в момент создания. Типичный пример – счета-фактуры. У нас уже есть таблицы заказчиков и продуктов, и, казалось бы, счет-фактура может просто сослаться на эти таблицы. Но на практике так никогда

не делается. Информация о заказчике или ценах может измениться, и тогда будет нарушена целостность счета-фактуры в том виде, в котором она существовала на момент создания. А это приведет к ошибкам при аудите, в отчетах, к нарушению законов и прочим проблемам.

В реляционном мире денормализация нарушает нормальные формы Кодда, и мы стремимся этого избежать. Но в Cassandra денормализация – благо и считается совершенно нормальным делом. Если модель простая, то денормализовывать ее необязательно. Но и бояться этого не надо.



Материализованные представления как средство денормализации на сервере

Исторически для денормализации в Cassandra требовалось проектировать несколько таблиц и управлять ими, применяя методы, с которыми мы скоро познакомимся. Но, начиная с версии 3.0, появились так называемые *материализованные представления*, которые позволяют создавать несколько денормализованных представлений данных на основе структуры базовой таблицы. Cassandra управляет материализованными представлениями на сервере, заботясь, в частности, о синхронизации с базовой таблицей. В этой главе мы приведем примеры классической денормализации и материализованных представлений.

Методика проектирования от запроса

Говоря простыми словами, реляционное моделирование означает, что мы начинаем с концептуальной модели предметной области, а затем представляем существительные, оказавшиеся в этой модели, таблицами. После этого назначаются первичные и внешние ключи для моделирования связей. Связи многие-ко-многим представляются связующими таблицами. В реальном мире связующим таблицам ничего не соответствует, это просто необходимый побочный эффект строения реляционных моделей. Определившись с таблицами, мы начинаем писать запросы, в которых данные из разных таблиц связываются с помощью ключей. В реляционном мире запросы – вещь вторичная. Предполагается, что если таблицы спроектированы правильно, то уж данные мы всяко достанем. И обычно это правда, даже если для решения задачи придется прибегнуть к сложным подзапросам или соединениям.

Напротив, в Cassandra мы начинаем не с модели данных, а с модели запросов. Вместо того чтобы сначала построить модель данных, а затем писать запросы, при работе с Cassandra мы сначала продумываем

запросы, а уже вокруг них организуем данные. Решите, какие запросы чаще всего будут предъявляться в приложении, а затем создавайте таблицы, поддерживающие их эффективное выполнение.

Критики высказывали мнение, что проектирование от запросов чрезмерно ограничивает область применимости приложения, не говоря уже о моделировании базы данных. Но разве не разумно предположить, что мы обязаны тщательно продумать запросы приложения – уж не менее тщательно, чем обдумываем особенности предметной области. Допустив ошибку, мы получим проблемы и в том, и в другом случае. Со временем запросы могут измениться, и тогда придется изменить структуру данных. Но это ничем не отличается от неправильного определения таблиц или необходимости добавления новых таблиц в РСУБД.

Оптимизация хранения на этапе проектирования

В реляционных базах структура хранения таблиц на диске обычно прозрачна для пользователя, и в контексте моделирования данных редко можно встретить рекомендации, касающиеся физического хранения таблиц. Но в Cassandra это важный аспект. Поскольку каждая таблица хранится в отдельном файле, важно, чтобы взаимосвязанные столбцы были определены вместе в одной и той же таблице.

Как мы вскоре увидим, главная цель при проектировании модели данных в Cassandra – минимизировать количество разделов, которые нужно просматривать при обработке запроса. Поскольку каждый раздел находится только на одном узле, быстрее всего выполняются запросы, которым нужно просматривать только один раздел.

Решение о сортировке принимается при проектировании

В РСУБД порядок, в котором возвращаются записи, легко изменить, включив в запрос фразу ORDER BY. Порядок сортировки по умолчанию не задается, и в отсутствии других указаний записи возвращаются в том порядке, в котором добавлялись. Чтобы изменить порядок сортировки, нужно всего лишь модифицировать запрос, причем сортировать можно по любому набору столбцов.

Но в Cassandra к сортировке отношение другое: решение о ней принимается на этапе проектирования. Порядок сортировки в запросе фиксирован и определяется кластерными столбцами, заданными в команде CREATE TABLE. Команда CQL SELECT поддерживает семантику ORDER BY, но только в порядке, определяемом кластерными столбцами.

Определение запросов в приложении

Опробуем подход проектирования от запроса на примере модели данных для гостиничного приложения. Структура пользовательского интерфейса часто помогает, когда мы начинаем продумывать, какие необходимы запросы. Предположим, что мы переговорили со всеми заинтересованными сторонами, и дизайнеры предложили набросок интерфейса для основных выявленных сценариев. По-видимому, мы подготовим примерно такой список запросов.

- Q1. Найти отели поблизости от заданной достопримечательности.
- Q2. Получить информацию о заданном отеле, например его название и местоположение.
- Q3. Найти достопримечательности вблизи заданного отеля.
- Q4. Найти свободный номер в заданном диапазоне дат.
- Q5. Найти цены и принадлежности, зная номер.



Количество запросов

Часто бывает удобнее ссылаться на запросы по коротким идентификаторам, чем выписывать их целиком. Запросы в нашем списке обозначены Q1, Q2 и т. д., именно так мы будем обозначать их на последующих диаграммах.

Разумеется, мы хотим, чтобы пользователи нашего приложения имели возможность бронировать номера в отелях. Для этого понадобится выбирать свободный номер и вводить сведения о госте. Следовательно, потребуются какие-то запросы, относящиеся к сущностям концептуальной модели «Бронь» и «Гость». Но и здесь мы должны мыслить не только в терминах того, как данные записываются, но и в терминах того, как мы будем запрашивать данные в последующих сценариях.

Проектируя модель данных, мы естественно хотим сначала сосредоточиться на структуре таблиц для хранения записей о бронировании и гостях, а только потом заняться запросами к этим данным. Возможно, подобный дискомфорт уже возник у вас, когда вы прочитали список запросов и подумали «а откуда берутся данные об отелях и достопримечательностях?». Не переживайте, мы скоро к этому пойдем. Вот несколько запросов, касающихся доступа к информации о бронировании со стороны пользователей.

- Q6. Найти бронь по номеру подтверждения.
- Q7. Найти бронь по отелю, дате и имени гостя.

- Q8. Найти все брони по имени гостя.
- Q9. Просмотреть подробную информацию о госте.

На рис. 5.3 все эти запросы показаны в контексте порядка работы приложения. Каждый блок представляет один шаг работы приложения, а стрелками обозначены переходы между шагами и соответствующие им запросы. Если приложение спроектировано хорошо, то на каждом шаге выполняется действие, которое «разблокирует» последующие шаги. Например, шаг «Просмотреть отели рядом с достопримечательностью» дает приложению возможность получить информацию о нескольких отелях и в т. ч. их уникальные ключи. Ключ выбранного отеля можно использовать в запросе Q2, чтобы получить подробное описание отеля. Акт бронирования номера приводит к созданию записи о бронировании, которую впоследствии смогут найти гость и персонал отеля с помощью дополнительных запросов.

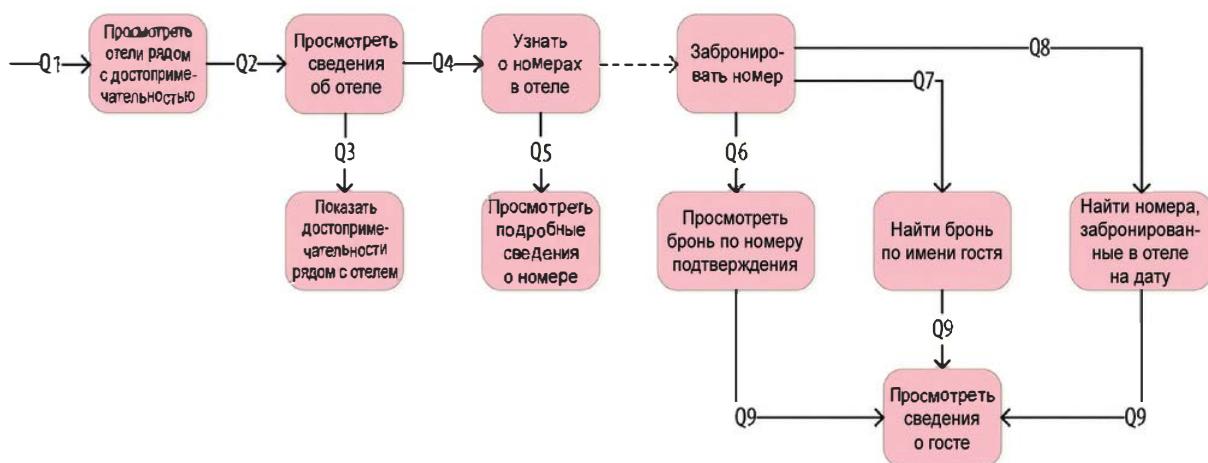


Рис. 5.3 ♦ Запросы в гостиничном приложении

Построение логической модели данных

После того как запросы определены, мы можем приступить к проектированию таблиц Cassandra. Сначала построим логическую модель, содержащую таблицу для каждого запроса с учетом сущностей и связей из концептуальной модели.

Имя таблицы выбирается, исходя из типа главной опрашиваемой сущности. Если производится запрос по атрибутам других связанных сущностей, то мы добавляем их к уже сформированному имени таблицы, отделяя строкой `_by_`. Например, `hotels_by.poi`.

Затем определяем, каким должен быть первичный ключ таблицы, составляя его из ключевых столбцов разделов в зависимости от тре-

буемых в запросе атрибутов и кластерных столбцов, гарантирующих уникальность и требуемый порядок сортировки.

И напоследок добавим в таблицы дополнительные атрибуты, состав которых определяется запросом. Если какой-нибудь дополнительный атрибут одинаков во всех экземплярах с одним и тем же ключом раздела, пометим такой столбец как статический.

Это было весьма краткое описание довольно сложного процесса, поэтому стоит потратить время на подробный разбор примера. Сначала введем нотацию, которая поможет представить логические модели.

Введение в диаграммы Чеботко

Несколько членов сообщества Cassandra предлагали нотацию для представления моделей данных в виде диаграмм. Мы остановились на предложении Артема Чеботко, поскольку оно дает простой и информативный способ наглядно представить связи между запросами и таблицами. На рис. 5.4 показана диаграмма Чеботко логической модели данных.

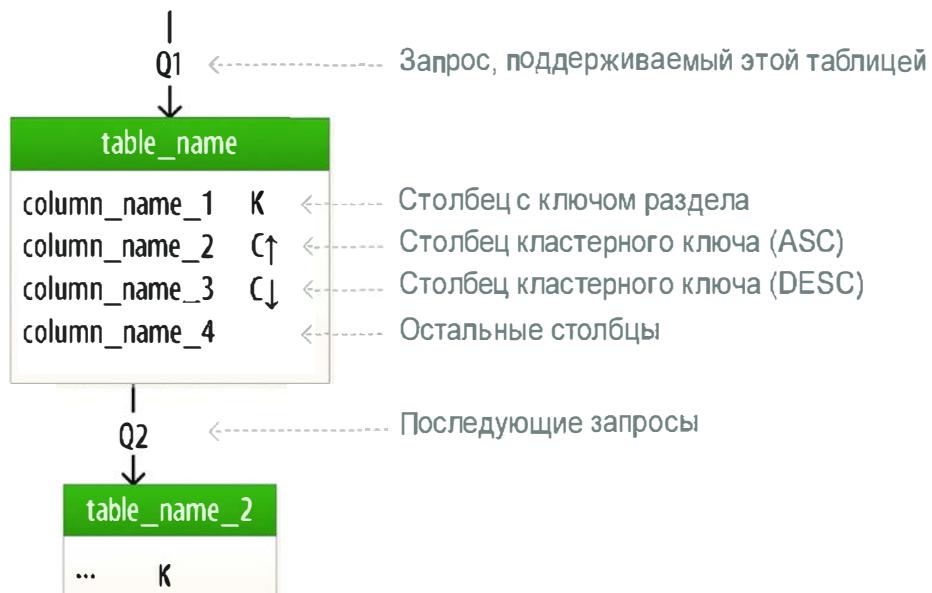


Рис. 5.4 ♦ Логическая диаграмма Чеботко

Для каждой таблицы показан заголовок и список столбцов. Столбцы, составляющие первичный ключ раздела, обозначаются символом **K**, а столбцы кластерных ключей – символами **C[↑]** (сортировка в порядке возрастания) или **C[↓]** (сортировка в порядке убывания). Линии при входе в таблицу или между таблицами обозначают запросы, для поддержки которых таблицы существуют.

Логическая модель данных отеля

На рис. 5.5 показана диаграмма Чеботко логической модели данных для запросов, касающихся отелей, достопримечательностей, номеров и принадлежностей. Сразу видно, что в нашем проекте не предусмотрено отдельных таблиц для номеров и принадлежностей, как было бы в реляционной базе. Объясняется это тем, что мы не выявили запросов, требующих прямого доступа к этим сущностям.

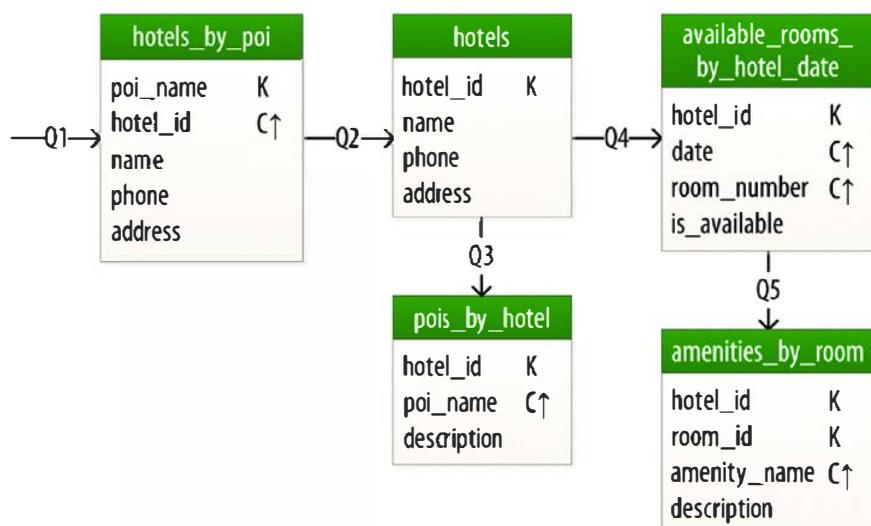


Рис. 5.5 ❖ Логическая модель отеля

Рассмотрим каждую таблицу подробнее.

В первом запросе Q1 требовалось найти отели рядом с достопримечательностью, поэтому таблицу мы назвали `hotels_by_poi`. Мы ищем достопримечательность по названию, следовательно, название должно быть частью первичного ключа. Именно с достопримечательности пользователи приложения будут начинать поиск.

Конечно, рядом с данной достопримечательностью может быть несколько отелей, поэтому в первичный ключ нужно включить еще одну компоненту, чтобы для каждого отеля был уникальный раздел. Сделаем идентификатор отеля кластерным столбцом.



Первичные ключи должны быть уникальными

При проектировании первичного ключа таблицы важно следить за его уникальностью. В противном случае есть риск случайно затереть данные.

Для второго запроса (Q2) нам нужна таблица, из которой можно получить информацию о конкретном отеле. Можно было бы поместить все атрибуты отеля в таблицу `hotels_by_poi`, но мы реши-

ли оставить в ней только те, которые диктуются порядком работы приложения.

Из диаграммы процесса работы мы знаем, что таблица `hotels_by_roi` используется для отображения списка отелей, содержащего основные сведения об отеле, и приложение знает уникальные идентификаторы возвращенных отелей. После того как пользователь выбрал отель, мы можем с помощью запроса Q2 получить подробные сведения о нем. Поскольку в результате выполнения Q1 мы уже знаем `hotel_id`, то можем использовать этот идентификатор для поиска отеля. Таким образом, наша вторая таблица будет называться просто `hotels`.

Альтернативно можно было бы хранить множество названий достопримечательностей `roi_names` в таблице отелей. Этот подход ничем не хуже. Как лучше поступить в конкретном приложении, познается на опыте.



Использование уникальных идентификаторов для ссылки

Часто бывает удобно использовать UUID в качестве уникальных идентификаторов элементов и хранить их в таблицах для ссылки на другие сущности. Это сводит к минимуму связи между сущностями разных типов и особенно полезно, если в основе приложения лежит архитектура микросервисов, в которой за сущности каждого типа отвечает свой сервис.

Но в этой книге мы в основном используем в качестве идентификаторов текстовые атрибуты, чтобы примеры были проще и понятнее. Так, в гостиничном бизнесе принято обозначать свойства короткими кодами, например «AZ123» или «NY229». Мы будем использовать такие строки в качестве значений `hotel_id`, понимая, что они необязательно глобально уникальны.

Запрос Q3 является обратным к Q1 – найти достопримечательности рядом с отелем, а не отели рядом с достопримечательностями. Но теперь нам нужны сведения о каждой достопримечательности, хранящиеся в таблице `pois_by_hotel`. Как и раньше, включим название достопримечательности в качестве кластерного ключа, чтобы гарантировать уникальность.

Теперь посмотрим, как поддержать запрос Q4, чтобы пользователь мог найти в выбранном отеле номера, свободные на интересующие его даты. Отметим, что в запросе нужно указать начальную и конечную даты. Из того, что в запросе участвует диапазон, а не одна дата, мы заключаем, что дату придется сделать кластерным ключом. Будем использовать `hotel_id` в качестве первичного ключа, чтобы все номера

из одного отеля оказались в одном разделе, это поможет ускорить поиск. Назовем таблицу `available_rooms_by_hotel_date`.



Поиск по диапазону

Используйте кластерные столбцы для хранения атрибутов, к которым нужно обращаться в запросе по диапазону. Напомним, что порядок кластерных столбцов важен. Подробнее о запросах по диапазону мы будем говорить в главе 9.

И, завершая проектирование этой части модели данных, добавим таблицу `amenities_by_room` для поддержки запроса Q5. Она позволит пользователю посмотреть, какие принадлежности (телефон, фен, шампунь и т. п.) предусмотрены в номерах, свободных в течение указанного периода.

Логическая модель данных о бронировании

Теперь перейдем к запросам, касающимся бронирования. На рис. 5.6 показана соответствующая логическая модель. Обратите внимание, что таблицы денормализованы – одни и те же данные встречаются в нескольких таблицах с разными ключами.

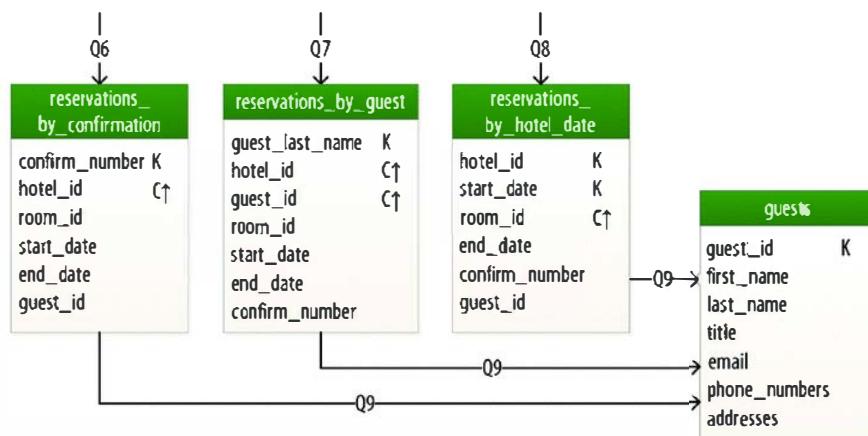


Рис. 5.6 ♦ Денормализованная логическая модель бронирования

Для выполнения запроса Q6 таблица `reservations_by_confirmation` поддерживает поиск брони по уникальному номеру подтверждения, который система сообщила клиенту в момент бронирования.

Если номер подтверждения неизвестен, для поиска брони по имени гостя можно воспользоваться таблицей `reservations_by_guest`. Можно представить себе, что запрос Q7 используется на сайте самообслуживания или оператором колл-центра, который пытается помочь гостю.

Поскольку имя гостя не обязательно уникально, мы добавили кластерный столбец, содержащий идентификатор гостя.

Персоналу отеля нужно знать, какие номера забронированы на указанную дату, чтобы понять, насколько эффективно работает отель: когда все номера заняты, а когда пустуют. Запрос Q8 поддерживает поиск бронирований в данном отеле на указанную дату.

Наконец, создадим таблицу *guests*. Обратите внимание, что по составу атрибутов она напоминает таблицу *user* из главы 4. Это то единственное место, в котором хранится информация о гостях. В данном случае мы ввели отдельный уникальный идентификатор гостя, потому что встречаются гости с одинаковыми именами. Во многих организациях база данных клиентов, похожая на нашу таблицу *guests*, являлась бы частью отдельного приложения управления работой с клиентами, поэтому мы не включили в пример запросы для доступа к информации о гостях.



Проектируйте запросы с учетом интересов всех участников

Запросы Q8 и Q9 призваны напомнить, что мы должны иметь в виду различных потенциальных пользователей приложения: не только гостей, но и персонал, а быть может, еще и аналитиков, поставщиков и т. д.

Паттерны и антипаттерны

Как и при проектировании других видов программ, в моделировании данных для Cassandra есть свои паттерны и антипаттерны. Один из наиболее распространенных паттернов – широкую строку – мы уже встретили в модели отеля.

Паттерн временных рядов является обобщением паттерна широкой строки. Речь идет о хранении в широкой строке последовательности измерений, сделанных в определенные моменты времени, причем время измерения является частью ключа раздела. Этот паттерн часто встречается в таких предметных областях, как бизнес-аналитика, управления показаниями датчиков и научные эксперименты.

Паттерн временных рядов полезен не только для хранения результатов измерений. Рассмотрим, к примеру, банковское приложение. Мы могли бы сохранить в строке баланс клиента, но тогда проверка баланса или выполнение транзакции любым пользователем может привести к высокой конкуренции при чтении и записи. Вероятно, мы предпочли бы обернуть все операции записи транзакциями, чтобы защитить баланс от неправильного обновления. С другой стороны, подход на основе временных рядов предполагает хранение каждой

транзакции в виде строки с временной меткой, а задача вычисления текущего баланса возлагается на приложение.

Многие начинающие пользователи совершают одну и ту же ошибку, пытаясь использовать Cassandra в качестве очереди. Каждый элемент очереди хранится вместе с временной меткой в широкой строке. Элементы добавляются в конец очереди и читаются из начала, прочитанные элементы удаляются. Идея кажется привлекательной, особенно ввиду ее кажущегося сходства с паттерном временных рядов. Беда, однако, в том, что удаленные элементы становятся «надгробиями», которые Cassandra должна пропустить при чтении, чтобы добраться до начала очереди. Со временем растущее число надгробий начинает отрицательно сказываться на производительности.

Антитиптерн очереди напоминает о том, что любое проектное решение, зависящее от удаления данных, потенциально способно привести к проблемам с производительностью.

Построение физической модели данных

После того как логическая модель данных определена, построить физическую уже сравнительно просто.

Мы просматриваем все таблицы, вошедшие в логическую модель, и назначаем тип каждому столбцу. Можно использовать любые типы, рассмотренные в главе 4: простые, коллекции и пользовательские. Для упрощения структуры можно определить новые пользовательские типы.

Назначив тип данных, мы переходим к анализу модели: оцениваем размер базы и тестируем работу модели. На основе полученных результатов можно внести корректизы. Мы продемонстрируем детали процесса моделирования данных в контексте нашего примера.

Но прежде рассмотрим несколько добавлений к нотации диаграмм Чеботко для физических моделей.

Физические диаграммы Чеботко

Для изображения физических моделей мы должны включить в каждый столбец информацию о типе. На рис. 5.7 показано, как это может выглядеть.

Дополнительно включено обозначение пространства ключей, содержащего таблицу, и специальные обозначения для столбцов, имеющих тип коллекции или пользовательский тип. Обратите также внимание на пометку статических столбцов и столбцов, по которым построены вторичные индексы. Никто не мешает включить все это и в логиче-

скую модель, но по существу такая информация относится скорее к физическим аспектам моделирования.

keyspace_name		
table_name		
column_name_1	CQL Type	K
column_name_2	CQL Type	C↑
column_name_3	CQL Type	C↓
column_name_4	CQL Type	S
column_name_5	CQL Type	IDX
column_name_6	CQL Type	++
[column_name_7]	CQL Type	
{column_name_8}	CQL Type	
<column_name_9>	CQL Type	
column_name_10	UDT Name	
(column_name_11)	CQL Type	
column_name_12	CQL Type	

Столбец с ключом раздела
Столбец кластерного ключа (ASC)
Столбец кластерного ключа (DESC)
Статический столбец
Столбец вторичного индекса
Столбец счетчика
Столбец списка
Столбец множества
Столбец словаря
Столбец UDT
Столбец кортежа
Обычный столбец

Рис. 5.7 ♦ Обобщение нотации Чеботко на физическую модель данных

Физическая модель данных отеля

Перейдем к физической модели для нашего приложения. Прежде всего нам нужны пространства ключей для таблиц. Чтобы не слишком усложнять проект, создадим пространство ключей `hotel`, в котором будут находиться таблицы с данными об отелях и доступности номеров, и пространство ключей `reservation` для таблиц с данными о бронировании и о гостях. В реальной системе мы могли бы завести дополнительные пространства ключей, чтобы лучше разделить обязанности.

В таблице `hotels` зададим тип `text` для идентификатора отеля `id`. Для адреса воспользуемся типом `address`, созданным в главе 4. Телефон представим типом `text`, потому что в разных странах приняты различные соглашения о форматировании номеров телефонов.

Для других таблиц, определенных в логической модели данных отеля, поступаем аналогично. В результате получается физическая модель, изображенная на рис. 5.8.

Обратите внимание, что в проект включен тип `address`. Он помечен звездочками, чтобы было понятно, что это пользовательский тип данных, столбцов первичного ключа в нем нет. Этот тип используется в таблицах `hotels` и `hotels_by.poi`.

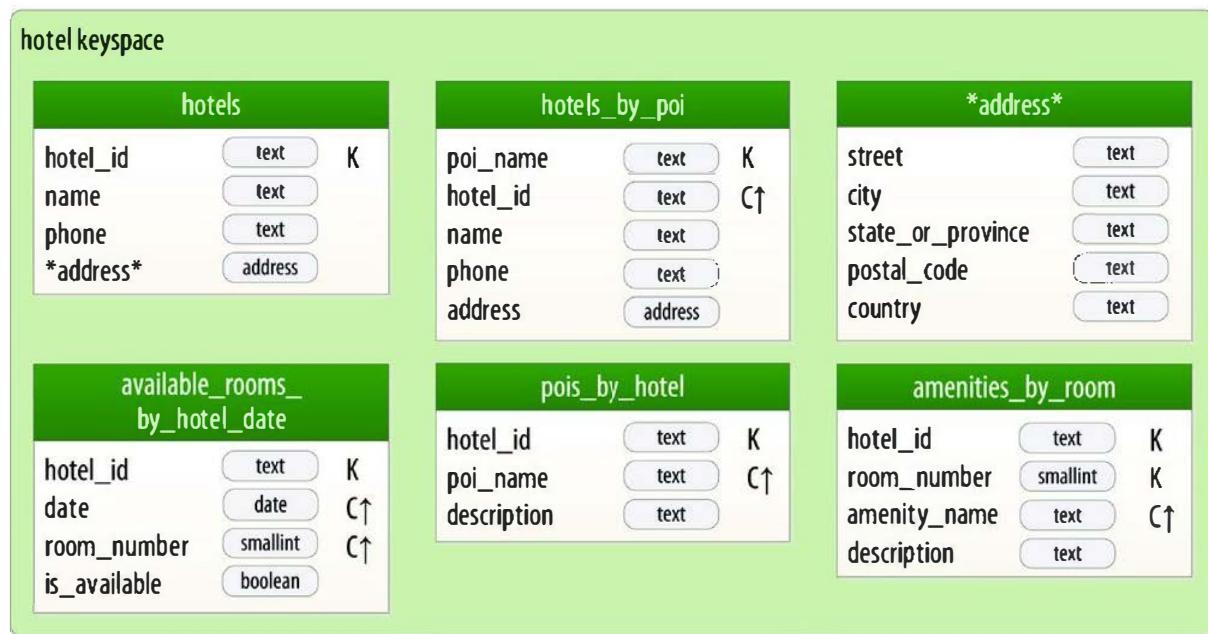


Рис. 5.8 ❖ Физическая модель отеля



Преимущества пользовательских типов

Пользовательские типы часто помогают уменьшить дублирование столбцов, не входящих в состав первичного ключа, что и продемонстрировано на примере типа address. А это, в свою очередь, уменьшает сложность проекта.

Напомним, что областью видимости UDT является пространство ключей, в котором он определен. Чтобы воспользоваться типом address в пространстве ключей reservation, нам придется объявить его еще раз. Это лишь один из многих компромиссов, на которые приходится идти при проектировании модели данных.

Физическая модель данных о бронировании

Теперь обратимся к таблицам с данными о бронировании. Напомним, что в логической модели имеются три денормализованные таблицы, поддерживающие запросы о поиске брони по номеру подтверждения, по имени гостя, а также по отелю и дате. Размышляя о реализации этих таблиц, мы можем выбрать один из двух вариантов: ручная денормализация или материализованные представления Cassandra.

На рис. 5.9 иллюстрируются оба подхода в применении к пространству ключей reservation. Мы решили реализовать reservations_by_hotel_date и reservations_by_guest как обычные таблицы, а reservations_by_confirmation – как материализованное представление таблицы reservations_by_hotel_date. Почему принято такое решение, будет сказано чуть ниже.

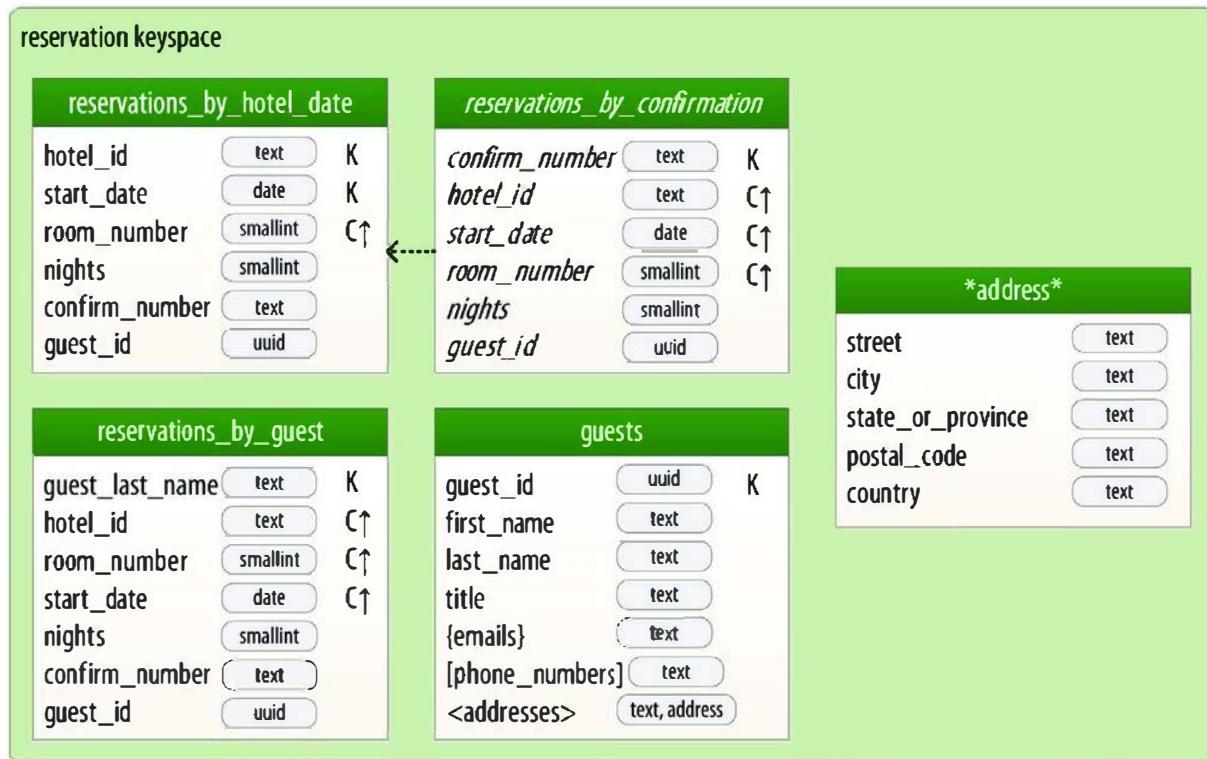


Рис. 5.9 ♦ Физическая модель бронирования

Обратите внимание, что мы повторили определение типа `address` в этом пространстве ключей и в качестве типа столбца `guest_id` во всех таблицах указали `uuid`.

Материализованные представления

Материализованные представления были введены для устранения некоторых недостатков вторичных индексов, о которых мы говорили в главе 4. Создание индекса по столбцу с большим числом различных значений ведет к снижению производительности, поскольку приходится опрашивать все или большинство узлов кольца.

Материализованные представления решают эту проблему за счет хранения заранее настроенных представлений, поддерживающих запросы по дополнительным столбцам, не являющимся частью исходного кластерного ключа. Материализованные представления упрощают разработку приложения: Cassandra берет на себя ответственность за своевременное обновление представлений синхронно с изменением базовой таблицы, освобождая приложение от бремени синхронизации нескольких денормализованных таблиц.

За поддержание согласованности приходится платить небольшим снижением производительности при записи. Но все равно материализованные представления работают быстрее, чем управление де-

нормализованными таблицами на стороне клиента. На внутреннем уровне обновление материализованного представления реализовано с помощью пакетов команд, о чем речь пойдет в главе 9.

Как и вторичные индексы, материализованные представления можно создавать над уже существующими таблицами.

Чтобы понять синтаксис материализованных представлений и налагаемые на них ограничения, рассмотрим команду CQL, которая создает таблицу `reservations_by_confirmation` из физической модели данных бронирования:

```
cqlsh> CREATE MATERIALIZED VIEW reservation.reservations_by_confirmation
  AS SELECT *
    FROM reservation.reservations_by_hotel_date
   WHERE confirm_number IS NOT NULL and hotel_id IS NOT NULL and
         start_date IS NOT NULL and room_number IS NOT NULL
 PRIMARY KEY (confirm_number, hotel_id, start_date, room_number);
```

Порядок фраз в команде `CREATE MATERIALIZED VIEW` может показаться странным, поэтому мы рассмотрим их в порядке, более удобном для понимания.

Первый параметр после названия команды – имя материализованного представления, в данном случае `reservations_by_confirmation`. Во фразе `FROM` указана базовая таблица, над которой строится представление, – `reservations_by_hotel_date`.

Во фразе `PRIMARY KEY` задается первичный ключ материализованного представления, он должен включать все столбцы первичного ключа базовой таблицы. Это ограничение не позволяет Cassandra свернуть несколько строк базовой таблицы в одну строку материализованного представления, что сильно усложнило бы управление обновлениями.

Для группировки столбцов первичного ключа применяется тот же синтаксис, что и в обычной таблице. Чаще всего вначале помещают дополнительный столбец, который становится ключом раздела, а за ним – столбцы, входящие в состав первичного ключа базовой таблицы, которые в материализованном представлении оказываются кластерными столбцами.

Фраза `WHERE` служит для фильтрации. Отметим, что фильтр необходимо задавать для каждого столбца первичного ключа материализованного представления, даже если это просто условие `IS NOT NULL`.

Во фразе `AS SELECT` определяется, какие столбцы базовой таблицы должны войти в материализованное представление. Можно указывать столбцы по отдельности, но в данном случае мы просто включили все столбцы, воспользовавшись метасимволом `*`.



Дополнительные возможности материализованных представлений

В первой реализации материализованных представлений в версии 3.0 были некоторые ограничения на выбор столбцов первичного ключа и фильтров. В системе JIRA зарегистрировано несколько находящихся в разработке предложений о добавлении возможностей, например: включение в первичный ключ материализованного представления нескольких столбцов, не входящих в состав первичного ключа базовой таблицы (<https://issues.apache.org/jira/browse/CASSANDRA-9928>), или использование агрегатов в материализованных представлениях (<https://issues.apache.org/jira/browse/CASSANDRA-9778>). Если вас эти возможности заинтересовали, следите за JIRA, чтобы узнать, когда они войдут в очередную версию.

Теперь, лучше понимая структуру и применение материализованных представлений, мы можем вернуться к обоснованию решения, принятого при проектировании физической модели бронирования. Конкретно, таблица `reservations_by_confirmation` является хорошим кандидатом на реализацию в виде материализованного представления из-за большого числа различных номеров подтверждений – ну действительно, у каждой брони уникальный номер, куда уж больше!

Альтернативно можно было бы сделать `reservations_by_confirmation` базовой таблицей, а `reservations_by_hotel_date` – материализованным представлением. Но поскольку мы не можем (по крайней мере, в ранних версиях из семейства 3.X) создавать материализованные представления, первичный ключ которых содержит более одного столбца, не входящего в состав первичного ключа базовой таблицы, для этого потребовалось бы сделать `hotel_id` или `date` кластерным столбцом таблицы `reservations_by_confirmation`. Оба решения приемлемы, но это проливает свет на компромиссы, которые приходится рассматривать при определении того, какую из нескольких денормализованных таблиц сделать базовой.

Оценка и уточнение

После создания физической модели нужно предпринять ряд шагов для оценки и уточнения структуры таблицы с целью добиться оптимальной производительности.

Вычисление размера раздела

Первым делом мы хотели бы понять, не будут ли разделы наших таблиц чрезмерно большими, или, иначе говоря, не будут ли широкие строки слишком широкими. Размер раздела измеряется числом ячеек (значений), хранящихся в разделе. Абсолютный предел в Cassandra составляет 2 миллиарда ячеек на раздел, но с проблемами в части производительности мы, скорее всего, столкнемся еще до достижения этого предела.

Размер раздела вычисляется по формуле

$$N_v = N_r(N_c - N_{pk} - N_s) + N_s.$$

Число значений (ячеек) в разделе (N_v) равно числу статических столбцов (N_s) плюс произведение числа строк (N_r) на число значений в одной строке. Число значений в строке определяется как общее число столбцов (N_c) за вычетом числа столбцов первичного ключа (N_{pk}) и числа статических столбцов (N_s).

Число столбцов обычно относительно неизменно, хотя мы видели, что изменять структуру таблицы во время выполнения вполне возможно. Поэтому основным фактором, определяющим размер раздела, является число строк в разделе. Именно его нужно оценивать, решая, не окажется ли раздел слишком большим. Два миллиарда значений – на первый взгляд, очень много, но в системе датчиков, где каждую миллисекунду измеряются десятки или сотни значений, объем данных растет весьма быстро.

Чтобы проанализировать размер раздела, рассмотрим одну из наших таблиц. Выберем таблицу `available_rooms_by_hotel_date`, поскольку в ней имеется широкая строка и под каждый отель отведен один раздел. Всего в таблице четыре столбца ($N_c = 4$), из них три входят в состав первичного ключа ($N_{pk} = 3$), а статических столбцов нет ($N_s = 0$). Подставляя в формулу, получаем:

$$N_v = N_r(4 - 3 - 0) + 0 = 1N_r.$$

Таким образом, число значений в этой таблице равно числу строк. Но надо еще определить число строк. Для этого сделаем оценки, исходя из природы проектируемого приложения. В таблице имеется по одной записи для каждого номера каждого отеля на каждую дату. Допустим, что в системе будут храниться данные за последние два года, что зарегистрировано 5000 отелей и среднее число номеров в отеле равно 100.

Тогда получается такая оценка числа строк:

$$N_r = 5000 \text{ отелей} \times 100 \text{ номеров/отель} \times 730 \text{ дней} = 365 \, 000 \, 000.$$

365 миллионов строк не создадут особых проблем, но если мы увеличим число отелей или не будем следить за сроком хранения данных с помощью TTL, то проблемы могут появиться. Имеет смысл подумать о том, как разбить этот большой раздел на части, и скоро мы этим займемся.



Оценка в худшем случае

При вычислении размеров возникает соблазн предполагать типичный или средний случай для таких переменных, как число строк. Но нужно рассмотреть и худший случай, потому что именно он и может стать явью.

Оценка места, занятого на диске

Помимо размера раздела, было бы замечательно оценить, сколько места на диске потребуется для каждой таблицы, хранящейся в кластере. Для этого воспользуемся следующей формулой вычисления размера таблицы S_t :

$$S_t = \sum_i \text{sizeOf}(c_{k_i}) + \sum_j \text{sizeOf}(c_{s_j}) + \\ + N_r \times \sum_k \left(\text{sizeOf}(c_{r_k}) + \sum_l \text{sizeOf}(c_{c_l}) \right) + 8 \times N_v.$$

Эта формула немного сложнее предыдущей, но мы рассмотрим ее по частям. Сначала введем обозначения.

- В этой формуле c_k – столбцы первичного ключа, c_s – статические столбцы, c_r – обычные столбцы, c_c – кластерные столбцы.
- N_r и N_v обозначают число строк и число значений, как и раньше.
- Функция `sizeOf()` возвращает размер в байтах типа данных CQL для указанного в аргументе столбца.

Первый член – это суммарный размер столбцов, образующих ключ раздела. В нашем примере в таблице `available_rooms_by_hotel_date` ключ раздела состоит всего из одного столбца, `hotel_id`, имеющего тип `text`. В предположении, что идентификаторы отелей – простые 5-значные коды, мы будем иметь 5-байтовое значение, так что суммарный размер столбцов, образующих ключ раздела, равен 5.

Второй член – сумма размеров статических столбцов. В нашей таблице таких нет, так что эта величина равна 0.

Третий член самый сложный и не без причины – в нем вычисляется размер ячеек в разделе. Мы суммируем размеры кластерных столбцов и прибавляем к этому значению размер обычного столбца (так что сумма размеров кластерных столбцов учитывается по одному разу для каждого обычного столбца). В нашем случае есть два кластерных столбца: `date` длиной 4 байта и `room_number` длиной 2-байта (короткое целое); в сумме получается 6 байтов. Обычный столбец всего один – `is_available` типа `boolean` длиной 1 байт. Вычисляя сумму размера обычного столбца (1 байт) и суммы размеров кластерных столбцов (6 байтов) для каждого обычного столбца, получаем 7 байтов. И в завершение умножаем эту величину на число строк (365 000 000) – получаем 2 555 000 000 байтов (2,56 ГБ).

Четвертый член служит для подсчета временных меток, которые Cassandra хранит для каждой ячейки. Мы умножаем на 8 величину, полученную в предыдущем расчете, что дает 2,92 ГБ.

Складывая все вместе, получаем окончательную оценку:

Размер раздела = 16 байтов + 0 байтов + 2,56 ГБ + 2,92 ГБ = 5,48 ГБ.

Эта аппроксимация истинного размера раздела на диске достаточно точна, чтобы быть полезной. Если вспомнить, что раздел должен целиком умещаться на одном узле, то получается, что спроектированная таблица предъявляет серьезные требования к дисковому хранилищу.

Следует также иметь в виду, что в этой оценке учтена только одна реплика данных. Ее нужно еще умножить на количество реплик, заданное в стратегии репликации пространства ключей, чтобы определить, сколько всего места на диске необходимо для таблицы. Это соображение пригодится, когда мы займемся планированием кластеров в главе 14.

Разбиение больших разделов

Как уже было сказано, наша цель – спроектировать таблицы, так чтобы для получения ответов на запросы нужно было в идеале обращаться всего к одному разделу, а если это невозможно, то хотя бы к минимальному числу разделов. Но, как показывают примеры, есть опасность спроектировать таблицы с широкими строками, приближающиеся к зашитым в Cassandra абсолютным ограничениям. Анализ размеров таблиц может выявить потенциально слишком большие разделы – по числу значений, по занимаемому месту на диске или по обоим критериям.

Техника разбиения большого раздела проста: добавить в ключи раздела еще один столбец. В большинстве случаев достаточно включить в состав ключа раздела один из уже имеющихся столбцов. Другой вариант – добавить столбец, который будет играть роль ключа сегментирования, но для этого потребуется изменить логику приложения.

Продолжим рассмотрение свободных номеров. Если добавить столбец `date` в состав ключа раздела для таблицы `available_rooms_by_hotel_date`, то каждый раздел будет представлять свободные номера в конкретном отеле на одну дату. Безусловно, это существенно уменьшит разделы, и, пожалуй, они станут слишком мелкими, потому что данные, относящиеся к соседним датам, вполне могут оказаться на разных узлах.

Для разбиения на разделы умеренного размера часто применяется другая техника – *раскладывание по корзинам* (*bucketing*). В случае таблицы `available_rooms_by_hotel_date` мы могли бы организовать корзины, добавив в состав ключа раздела столбец `month`. Он, конечно, частично дублирует столбец `date`, но зато позволяет сгруппировать данные в раздел не слишком большого размера.

Если хотим во что бы то ни стало сохранить структуру с широкими строками, то можем вместо этого включить в состав ключа раздела столбец `room_id`, так что каждый раздел будет представлять доступность одного номера на любую дату. Но поскольку анализ не выявил запросов, в которых требует узнать, свободен ли конкретный номер, первые два подхода лучше отвечают потребностям приложения.

Определение схемы базы данных

После оценки и уточнения физической модели можно приступить к реализации схемы на CQL. Ниже приведена схема пространства ключей `hotel`, причем мы используем комментарии CQL, чтобы документировать, какие запросы поддерживаются каждой таблицей.

```
CREATE KEYSPACE hotel
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};

CREATE TYPE hotel.address (
    street text,
    city text,
    state_or_province text,
    postal_code text,
    country text
);
```

```
CREATE TABLE hotel.hotels_by_poi (
    poi_name text,
    hotel_id text,
    name text,
    phone text,
    address frozen<address>,
    PRIMARY KEY ((poi_name), hotel_id)
) WITH comment = 'Q1. Find hotels near given poi'
AND CLUSTERING ORDER BY (hotel_id ASC) ;

CREATE TABLE hotel.hotels (
    id text PRIMARY KEY,
    name text,
    phone text,
    address frozen<address>,
    pois set<text>
) WITH comment = 'Q2. Find information about a hotel';

CREATE TABLE hotel.pois_by_hotel (
    poi_name text,
    hotel_id text,
    description text,
    PRIMARY KEY ((hotel_id), poi_name)
) WITH comment = 'Q3. Find pois near a hotel';

CREATE TABLE hotel.available_rooms_by_hotel_date (
    hotel_id text,
    date date,
    room_number smallint,
    is_available boolean,
    PRIMARY KEY ((hotel_id), date, room_number)
) WITH comment = 'Q4. Find available rooms by hotel / date';

CREATE TABLE hotel.amenities_by_room (
    hotel_id text,
    room_number smallint,
    amenity_name text,
    description text,
    PRIMARY KEY ((hotel_id, room_number), amenity_name)
) WITH comment = 'Q5. Find amenities for a room';
```



Явно выделяйте ключи разделов

В определениях таблиц мы заключили элементы ключа раздела в скобки даже в тех случаях, когда ключ раздела состоит всего из одного столбца, например `poi_name`. Так рекомендуется поступать, чтобы читателям CQL-схемы было сразу видно, как устроен ключ раздела.

А вот как выглядит схема пространства ключей reservation:

```

CREATE KEYSPACE reservation
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};

CREATE TYPE reservation.address (
    street text,
    city text,
    state_or_province text,
    postal_code text,
    country text
);

CREATE TABLE reservation.reservations_by_hotel_date (
    hotel_id text,
    start_date date,
    end_date date,
    room_number smallint,
    confirm_number text,
    guest_id uuid,
    PRIMARY KEY ((hotel_id, start_date), room_number)
) WITH comment = 'Q7. Find reservations by hotel and date';

CREATE MATERIALIZED VIEW reservation.reservations_by_confirmation AS
    SELECT * FROM reservation.reservations_by_hotel_date
    WHERE confirm_number IS NOT NULL and hotel_id IS NOT NULL and
        start_date IS NOT NULL and room_number IS NOT NULL
    PRIMARY KEY (confirm_number, hotel_id, start_date, room_number);

CREATE TABLE reservation.reservations_by_guest (
    guest_last_name text,
    hotel_id text,
    start_date date,
    end_date date,
    room_number smallint,
    confirm_number text,
    guest_id uuid,
    PRIMARY KEY ((guest_last_name), hotel_id)
) WITH comment = 'Q8. Find reservations by guest name';

CREATE TABLE reservation.guests (
    guest_id uuid PRIMARY KEY,
    first_name text,
    last_name text,
    title text,
    emails set<text>,
    phone_numbers list<text>,

```

```

addresses map<text, frozen<address>>,
confirm_number text
) WITH comment = 'Q9. Find guest by ID';

```

DataStax DevCenter

Мы научились создавать схемы с помощью `cqlsh`, но когда речь заходит о создании модели данных для приложения, в которой таблиц достаточно много, отслеживать CQL-код становится затруднительно.

По счастью, компания DataStax предлагает замечательное средство разработки – DevCenter. Его можно бесплатно скачать с сайта Академии DataStax (<https://academy.datastax.com/downloads>). На рис. 5.10 показано, как выглядит схема гостиничного приложения в DevCenter.

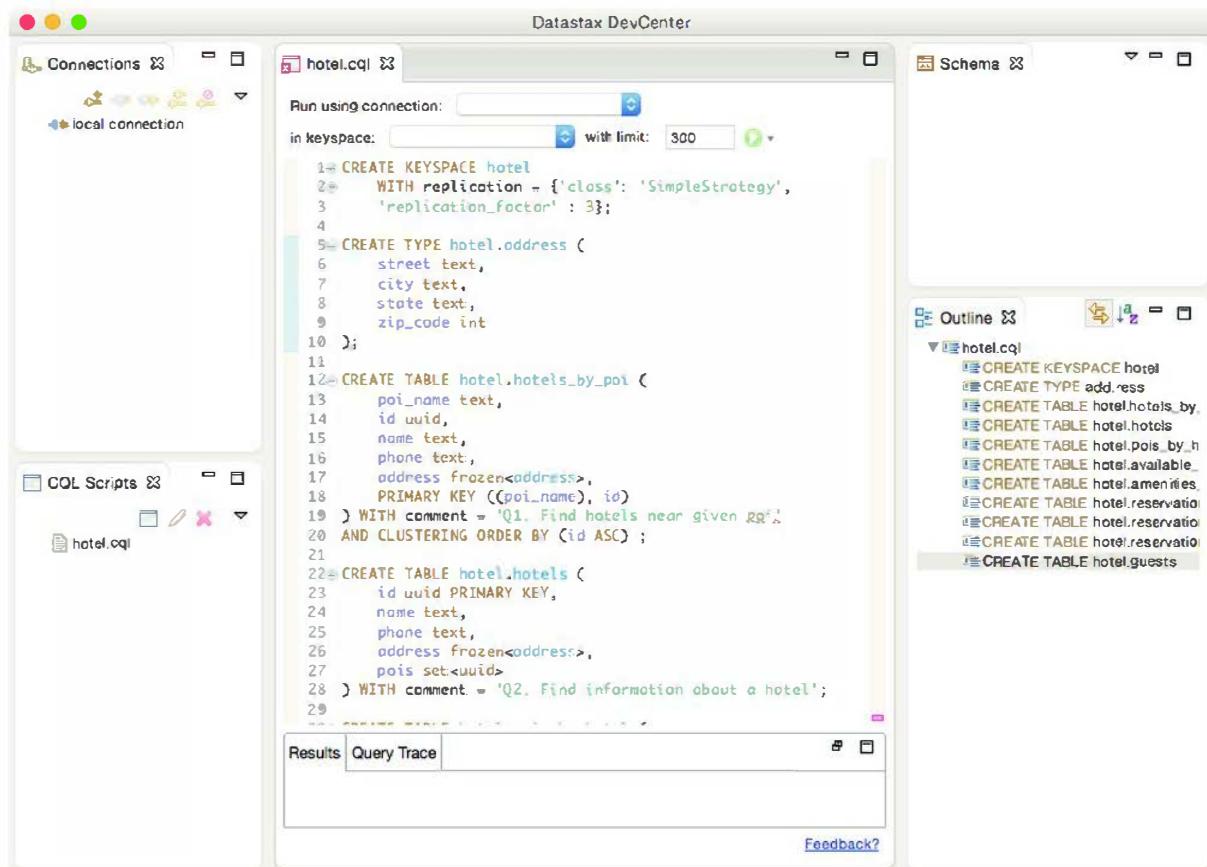


Рис. 5.10 ♦ Редактирование схемы Hotel в DataStax DevCenter

В средней панели мы видим выбранный в данный момент CQL-файл. Команды, типы данных и имена литералов выделены цветом. В DevCenter реализованы завершение CQL-команд и интерпретация команд в процессе ввода с указанием допущенных ошибок. Программа поддерживает несколько панелей для работы с несколькими CQL-

скриптами и подключение к нескольким кластерам. Подключения позволяют выполнять CQL-команды в работающих кластерах и просматривать результаты.

Резюме

В этой главе мы видели, как создать полную модель данных в Cassandra, и сравнили ее с эквивалентной реляционной моделью. Мы представили логическую и физическую модель и познакомились с новым инструментом для разработки моделей данных на CQL. В последующих главах мы продолжим работу над гостиничным приложением на основе созданной модели данных.

Глава 6

• • • • • • • • • • • • • • • • • • • • • • • • •

Архитектура Cassandra

3.2. Архитектура – принципиальная организация системы, воплощенная в ее элементах, их взаимоотношениях друг с другом и со средой, а также принципы, направляющие ее проектирование и эволюцию.

– ГОСТ Р ИСО/МЭК 15288

В этой главе мы рассмотрим некоторые аспекты архитектуры Cassandra, чтобы понять, как работает система. Мы объясним, как топологически устроен кластер и как его равноправные узлы взаимодействуют, имея целью обмен данными между собой и поддержку работоспособности кластера, для чего используются такие механизмы, как протокол распространения сплетен, антиэнтропия и вручение напоминаний. Мы заглянем внутрь узла, разберемся, как Cassandra поддерживает чтение, запись и удаление данных и как принятые решения отражаются на архитектурных аспектах: масштабируемости, долговечности, доступности, управляемости и т. д. Мы также обсудим включение в Cassandra многоступенчатой событийно-ориентированной архитектуры (Staged Event-Driven Architecture), используемой в качестве платформы делегирования запросов.

По ходу дела мы будем сообщать, где в исходном коде Cassandra находится соответствующая реализация.

Центры обработки данных и стойки

Cassandra зачастую используется в системах, находящихся в нескольких физических помещениях. Для описания топологии кластера

в Cassandra применяются два уровня группировки: центр обработки данных (ЦОД) и стойка. *Стойкой* (rack) называется логическая группа узлов, расположенных близко друг к другу, например на компьютерах, размещенных в одной физической стойке. *ЦОДом* называется логическая группа стоек, обычно находящихся в одном здании и соединенных между собой надежной сетью. Пример системы с несколькими ЦОДами и стойками показан на рис. 6.1.

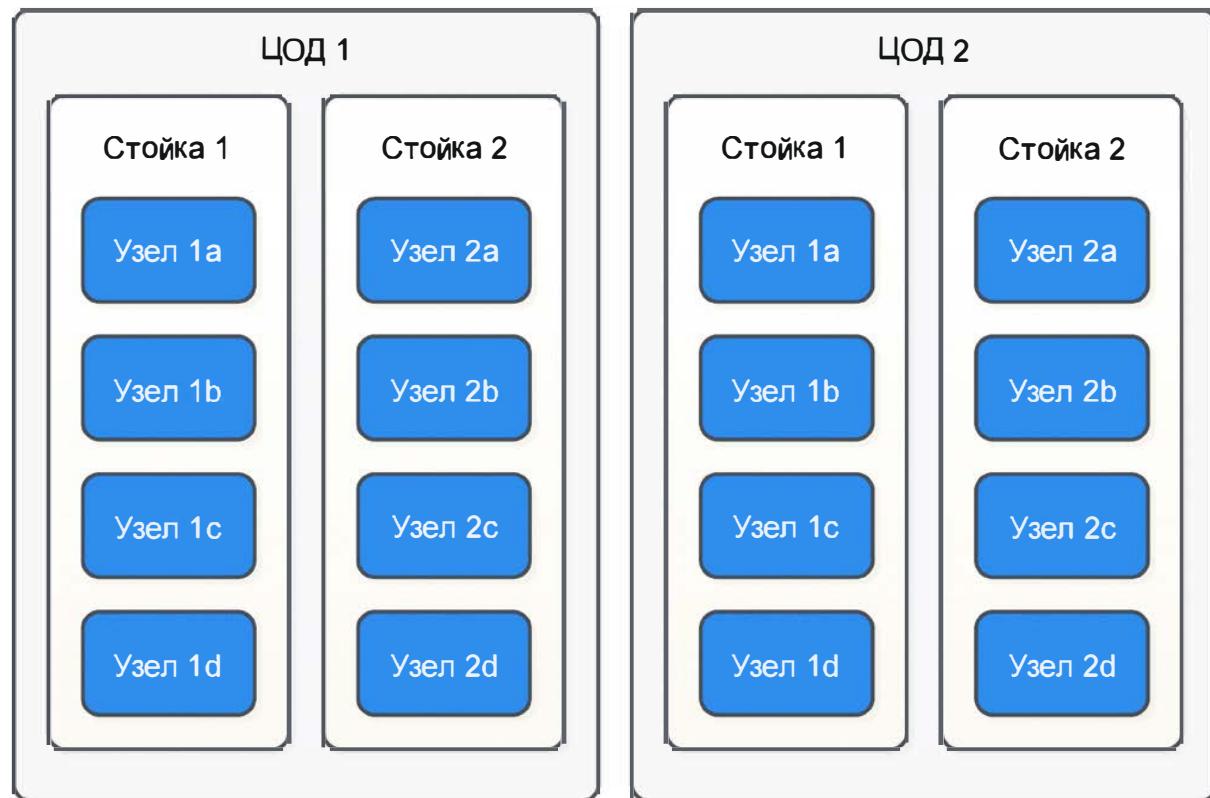


Рис. 6.1 ♦ Пример топологии кластера с несколько ЦОДами, стойками и узлами

Cassandra поставляется в стандартной конфигурации с одним ЦОДом ("DC1"), содержащим одну стойку ("RACK1"). В главе 7 мы узнаем, как построить более крупный кластер и определить его топологию.

Cassandra использует предоставленную нами информацию о топологии кластера, чтобы определить, где хранить данные и как эффективно маршрутизировать запросы. Cassandra стремится хранить копии данных в нескольких ЦОДах, чтобы максимизировать доступность и устойчивость к разделению, предпочитая вместе с тем направлять запросы узлам в локальном ЦОДе с целью максимизации производительности.

Сплетни и обнаружение отказов

Для обеспечения децентрализации и устойчивости к разделению в Cassandra используется протокол распространения сплетен, благодаря которому каждый узел располагает информацией о состоянии остальных узлов кластера. Процесс-сплетник активируется раз в секунду по таймеру.

В протоколах распространения сплетен (их также называют «протоколами распространения эпидемии») обычно предполагается, что сеть может отказывать. Они применяются в очень больших децентрализованных сетевых системах, в частности для автоматической репликации распределенных баз данных. Свое название они получили за сходство с распространением сплетен в человеческом обществе, когда равноправные участники сами выбирают, с кем поделиться информацией.



Происхождение термина «протокол распространения сплетен»

Термин «протокол распространения сплетен» (gossip protocol) в 1987 году придумал Алан Демерс (Alan Demers), который тогда работал исследователем в Исследовательском центре компании Xerox в Пало-Альто и занимался маршрутизацией информации в ненадежных сетях.

В Cassandra большая часть кода протокола распространения сплетен находится в классе org.apache.cassandra.gms.Gossiper, который отвечает за управление сплетнями в локальном узле. На этапе запуска серверный узел регистрируется в процессе-сплетнике, сообщая о желании получать информацию о состоянии конечных точек.

Поскольку в Cassandra сплетни используются для обнаружения отказов, класс Gossiper поддерживает список живых и мертвых узлов.

Процесс-сплетник работает следующим образом.

1. Раз в секунду сплетник случайным образом выбирает узел кластера и инициализирует сеанс передачи сплетни этому узлу. Каждый раунд распространения сплетни включает три сообщения.
2. Инициатор сплетни отправляет выбранному «знакомому» сообщение GossipDigestSynMessage.
3. Получив это сообщение, знакомый возвращает сообщение GossipDigestAckMessage.
4. Получив от знакомого сообщение ack, инициатор отправляет ему сообщение GossipDigestAck2Message, которое завершает раунд распространения сплетни.

Если сплетник определяет, что другой конец мертв, он «приговаривает» конечную точку, помечая ее мертвой в своем локальном списке и помещая сообщение об этом в журнал.

В Cassandra реализован надежный механизм обнаружения отказов, соответствующий популярному алгоритму распределенных вычислений под названием «обнаружение отказов с накоплением» (Phi Accrual Failure Detection). Этот алгоритм впервые был предложен в 2004 году в Японском институте науки и технологии (Advanced Institute of Science and Technology).

В основе обнаружения отказов с накоплением лежат две идеи. Во-первых, обнаружение отказов должно быть гибким, что достигается отделением механизма от обслуживаемого им приложения. Вторая, более новаторская идея заключается в отрицании традиционных детекторов отказов, которые реализованы с помощью простых контрольных сообщений-пульсов и принимают решение о работоспособности узла в зависимости от того, получен пульс или нет. В алгоритме обнаружения отказов с накоплением такой подход считается наивным, вместо дуализма «живой-мертвый» вводится понятие *уровня подозрения*.

Таким образом, система мониторинга отказов непрерывно пересчитывает уровень подозрения, показывающий степень уверенности в отказе узла. Это позволяет учесть флюктуации в сетевой среде и потому является желательным свойством. Так, сбой в одном подключении еще не означает, что узел окончательно вышел из строя. Поэтому идея подозрения дает более гибкий и проактивный индикатор вероятности отказа, основанный на интерпретации выборки пульсов, а не на простой бинарной оценке.

Порог ϕ и детекторы отказов с накоплением

Детектор отказов с накоплением вычисляет значение, ассоциированное с каждым процессом (или узлом). Оно называется ϕ . В алгоритм его вычисления изначально заложена приспособляемость к переменным условиям в сети, т. е. это не простое бинарное условие, показывающее, работает сервер или нет.

Параметр ϕ регулирует чувствительность детектора отказов. Чем ниже его значение, тем выше чувствительность, но зависимость нелинейная.

Значение ϕ описывает уровень *подозрения* в неработоспособности сервера. Приложения, в которых применяется алгоритм обнаружения отказов с накоплением, в частности Cassandra, могут задавать

переменные условия на вычисляемое значение ϕ . С помощью этого механизма Cassandra в общем случае может обнаружить отказ узла примерно за 10 секунд.

Интересующийся читатель может ознакомиться с оригинальной статьей Наохиро Хаясибара (Naohiro Hayashibara) об алгоритме обнаружения отказов с накоплением по адресу http://www.jaist.ac.jp/~defago/files/pdf/IS_RR_2004_010.pdf.

Код алгоритма обнаружения отказов в Cassandra находится в классе `org.apache.cassandra.gms.FailureDetector`, реализующем интерфейс `org.apache.cassandra.gms.IFailureDetector`. Класс предоставляет следующие методы:

`isAlive(InetAddress)`

Мнение детектора о работоспособности данного узла.

`interpret(InetAddress)`

Используется сплетником, чтобы решить, работает узел или нет, в зависимости от того, достигнут ли уровень подозрения. Для этого вычисляется величина ϕ (как описано в статье Хаясибара).

`report(InetAddress)`

Узел вызывает этот метод при получении контрольного сообщения-пульса.

Осведомители

Задача осведомителя (*snitch*) – определить относительную близость хоста для каждого узла кластера. Это необходимо для решения о том, с какого узла читать или на какой узел записывать данные. Осведомители собирают информацию о топологии сети, давая Cassandra возможность эффективно маршрутизировать запросы.

В качестве примера рассмотрим участие осведомителей в операции чтения. Производя чтение, Cassandra должна обратиться к нескольким репликам, число которых определяется уровнем согласованности. Чтобы обеспечить максимальную скорость чтения, Cassandra выбирает какую-то одну реплику, у которой запрашивает объект целиком, а остальные реплики просит вернуть только хэш-значения, подтверждающие, что получена последняя версия запрашиваемых данных. Роль осведомителя состоит в том, чтобы определить, какая реплика вернет ответ быстрее всех; именно у нее будут запрашиваться полные данные.

Осведомитель, подразумеваемый по умолчанию (`SimpleSnitch`), ничего не знает о топологии, т. е. о ЦОДах и стойках в кластере, так что для системы, развернутой в нескольких ЦОДах, он непригоден. Поэтому в состав Cassandra входит несколько осведомителей для различных облачных сред, в т. ч. Amazon EC2, Google Cloud и Apache Cloudstack.

Код осведомителей находится в пакете `org.apache.cassandra.locator`. Каждый класс осведомителя реализует интерфейс `IEndpointSnitch`. Как выбрать и настроить подходящий осведомитель, мы узнаем в главе 7.

Наряду со сменными механизмами статического описания топологии кластера Cassandra предлагает механизм *динамического осведомления*, который позволяет оптимизировать маршрутизацию операций чтения и записи с течением времени. Работает он следующим образом. Выбранный осведомитель обертывается другим классом осведомителя – `DynamicEndpointSnitch`. Базовое представление о топологии кластера динамический осведомитель получает от заданного в конфигурации. Затем он начинает следить за производительностью запросов к другим узлам, обращая внимание даже на то, производит ли узел уплотнение. Собранные данные о производительности используются для выбора наилучшей реплики для каждого запроса. Это позволяет Cassandra не направлять запросы репликам, которые отвечают медленно.

В реализации динамического осведомления используется модифицированная версия ф-детектора отказов, применяемого сплетником. «Порог негодности» – это настраиваемый параметр, который определяет, насколько предпочтительный узел должен работать хуже наилучшего, чтобы потерять статус предпочтительного. Оценки каждого узла периодически сбрасываются, чтобы дать плохо работавшему узлу возможность реабилитироваться – доказать, что он исправился, и восстановить статус предпочтительного.

Кольца и маркеры

До сих пор нас интересовало, как Cassandra следит за физическим расположением узлов кластера. А теперь взглянем на ситуацию под другим углом зрения и посмотрим, как Cassandra распределяет данные между узлами.

В Cassandra данные, управляемые кластером, представляются в виде *кольца*. Каждому узлу кольца назначается один или несколько

диапазонов данных, описываемых *маркером* (token), который определяет позицию в кольце. Маркер – это 64-разрядное число, идентифицирующее раздел. Следовательно, маркер может принимать значения от -2^{63} до $2^{63} - 1$.

Узел объявляет о владении диапазоном значений, меньших или равных маркеру и больших маркера предыдущего узла. Узел с наименьшим маркером владеет диапазоном, меньшим или равным своему маркеру, и диапазоном, большим наибольшего маркера, который называется еще «переходящим диапазоном». Таким образом, маркеры полностью описывают кольцо. На рис. 6.2 показано воображаемое кольцо, содержащее узлы из одного ЦОДа. В данном случае диапазоны маркеров распределены между узлами из разных стоек.

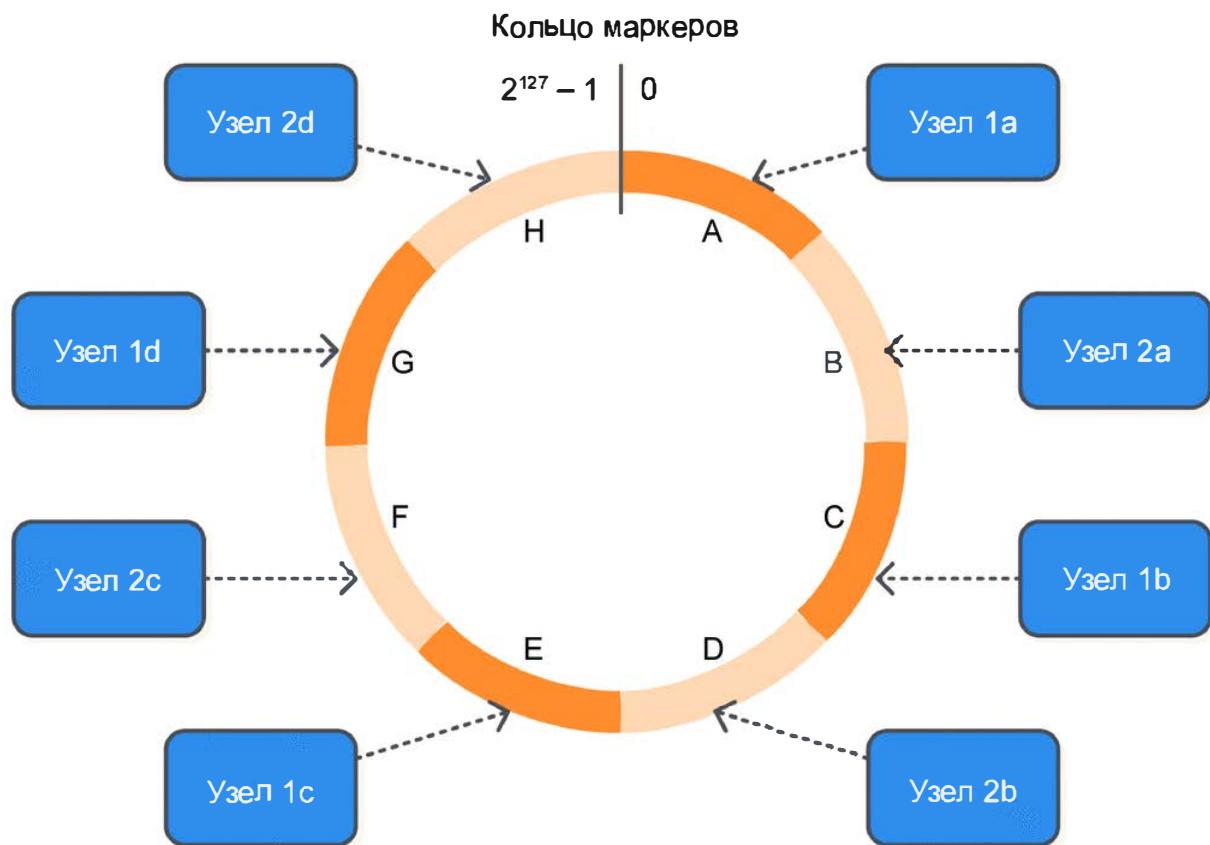


Рис. 6.2 ♦ Пример кольца узлов в одном ЦОДе

При решении вопроса о том, к какому узлу отнести данные, вычисляется хэш-функция ключа раздела, значением которой является маркер. Этот маркер сравнивается с маркерами различных узлов, и в итоге находится диапазон, а значит, и узел, владеющий данными.

Код диапазонов маркеров находится в классе `org.apache.cassandra.dht.Range`.

Виртуальные узлы

В ранних версиях Cassandra каждому узлу назначался только один маркер, причем это делалось статически, т. е. пользователь должен был вычислять маркеры узлов. И хотя имеются инструменты для вычисления маркеров при заданном числе узлов, все равно задавать свойство `initial_token` каждого узла в файле `cassandra.yaml` приходилось вручную. Добавление и замена узла оказывались дорогостоящей операцией, поскольку для перебалансировки кластера нужно было переместить большие объемы данных.

В версии Cassandra 1.2 введено понятие *виртуального узла*, или *v-узла*. Вместо того чтобы назначать узлу один маркер, диапазон маркеров разбивается на несколько меньших диапазонов. Затем каждому физическому узлу назначается несколько маркеров. По умолчанию каждому узлу назначается 256 таких маркеров, т. е. он содержит 256 виртуальных узлов. Начиная с версии 2.0 виртуальные узлы по умолчанию включены.

V-узлы упрощают обслуживание кластера, состоящего из неоднородных машин. Для узлов, располагающих большим объемом вычислительных ресурсов, можно увеличить количество v-узлов, задав свойство `num_tokens` в файле `cassandra.yaml`. Наоборот, для машин послабее значение `num_tokens` можно уменьшить.

Cassandra автоматически вычисляет диапазоны маркеров для каждого узла кластера пропорционально значениям `num_tokens`. Код назначения маркеров v-узлам находится в классе `org.apache.cassandra.dht.tokenallocator.ReplicationAwareTokenAllocator`.

Дополнительное преимущество виртуальных узлов состоит в том, что они ускоряют некоторые тяжелые операции Cassandra, в т. ч. инициализацию нового узла, вывод узла из эксплуатации и исправление узла. Связано это с тем, что нагрузка, ассоциированная с операциями в нескольких меньших диапазонах, более равномерно распределяется между узлами кластера.

Разделители

Разделитель (partitioner) определяет, как данные распределяются между узлами кластера. В главе 5 мы узнали, что Cassandra хранит данные в широких строках, или «разделах». У каждой строки имеется ключ раздела, который идентифицирует ее раздел. Таким образом, разделитель – это хэш-функция вычисления маркера ключа раздела.

Внутри кольца строка данных распределяется в соответствии со значением маркера ключа раздела.

Cassandra предоставляет несколько разделителей в пакете org.apache.cassandra.dht (DHT означает «distributed hash table» – распределенная хэш-таблица). Класс Murmur3Partitioner был добавлен в версии 1.2 и с тех пор является разделителем по умолчанию; это эффективная реализация на Java алгоритма murmur, изобретенного Остином Эпплби (Austin Appleby). Он генерирует 64-разрядные хэши. Ранее по умолчанию подразумевался класс RandomPartitioner.

Благодаря модульной структуре Cassandra вы можете создать собственный разделитель, реализовав интерфейс org.apache.cassandra.dht.IPartitioner и поместив свой класс в путь к классам Cassandra.

Стратегии репликации

Каждый узел играет роль *реплики* для различных диапазонов данных. Если узел выходит из строя, ответ на запросы к его диапазонам данных можно получить от других реплик. Cassandra реплицирует данные прозрачно для пользователей, а число узлов кластера, в которых хранятся копии (реплики) одних и тех же данных, определяется *коэффициентом репликации*. Если коэффициент репликации равен 3, то каждая строка хранится в трех узлах кластера.

Первая реплика всегда хранится в узле, которому принадлежит диапазон, куда попал маркер, а местоположение остальных реплик определяется *стратегией репликации* (или *стратегией размещения реплик*).

Для задания размещения реплик в Cassandra используется паттерн «Стратегия», описанный в книге «Банды четырех» и воплощенный в общем абстрактном классе org.apache.cassandra.locator.AbstractReplicationStrategy, который допускает различные реализации алгоритма (различные стратегии достижения одной цели). Каждая реализация инкапсулирована в классе, расширяющем AbstractReplicationStrategy.

В дистрибутиве Cassandra имеются две реализации этого интерфейса (расширения абстрактного класса): SimpleStrategy и NetworkTopologyStrategy. Класс SimpleStrategy помещает реплики в соседние узлы кольца, начиная с узла, указанного разделителем. Класс NetworkTopologyStrategy позволяет задать различные коэффициенты репликации для каждого ЦОДа. В пределах одного ЦОДа он распределяет реплики по разным стойкам, стремясь максимизировать доступность.



Унаследованные стратегии репликации

Третья стратегия, `OldNetworkTopologyStrategy`, оставлена ради обратной совместимости. Ранее она называлась `RackAwareStrategy`, тогда как класс `SimpleStrategy` назывался `RackUnawareStrategy`. Класс `NetworkTopologyStrategy` раньше назывался `DataCenterShardStrategy`. Все эти изменения произошли еще в версии 0.7.

Стратегии задаются независимо для каждого пространства ключей и являются обязательным параметром (см. главу 5).

Уровни согласованности

В главе мы обсуждали теорему САР Брюэра, в которой рассматривался компромисс между согласованностью, доступностью и устойчивостью к разделению. Cassandra предоставляет механизм настройки уровня согласованности, что позволяет более точно определять этот компромисс. Уровень согласованности задается в каждом запросе чтения или записи. Чем выше уровень согласованности, тем больше узлов должно ответить на запрос и тем, следовательно, выше уверенность, что во всех репликах хранятся одинаковые значения.

Для операций чтения уровень согласованности определяет, сколько узлов-реплик должно ответить на запрос, прежде чем система вернет данные, а для операций записи – сколько узлов-реплик должно ответить, чтобы операция считалась успешной. Поскольку Cassandra согласована в конечном счете, то обновление остальных реплик может продолжаться в фоновом режиме.

Определены уровни согласованности ONE, TWO и THREE; каждый из них задает абсолютное число ответивших узлов-реплик. Уровень согласованности QUORUM означает, что должно ответить большинство узлов-реплик (иногда эту идею выражают в виде «коэффициент репликации / 2 + 1»). В случае уровня согласованности ALL требуется, чтобы ответили все реплики. Эти и другие уровни согласованности подробно обсуждаются в главе 9.

Для операций чтения и записи уровни согласованности ANY, ONE, TWO и THREE считаются слабыми, а QUORUM и ALL – строгими. Согласованность в Cassandra считается настраиваемой, потому что клиент может задать желаемый уровень в операции чтения или записи. Для описания способа достижения строгой согласованности в Cassandra часто используют такое неравенство: $R + W > N$ = *строгая согласованность*. Здесь R , W и N – число реплик чтения, число реплик записи и коэффициент репликации соответственно; если это соотношение

соблюдено, то при любой операции чтения клиент увидит результат самой последней операции записи, поэтому имеет место строгая согласованность.



Различайте уровень согласованности и коэффициент репликации

Новички часто путают уровень согласованности с коэффициентом репликации. Коэффициент репликации задается один раз для всего пространства ключей. Уровень согласованности задается клиентом в каждом запросе. Коэффициент репликации определяет, на скольких узлах значение должно быть сохранено в процессе выполнения операции записи. Уровень согласованности определяет, сколько узлов должно ответить, чтобы клиент был уверен в успешном завершении операции чтения или записи. Путаница возникает из-за того, что уровень согласованности основан на коэффициенте репликации, а не на количестве узлов в системе.

Запросы и узлы-координаторы

Теперь соберем все вместе и обсудим, как узлы взаимодействуют для поддержки операций чтения и записи, инициированных клиентским приложением. На рис. 6.3 показан типичный путь взаимодействий в Cassandra.

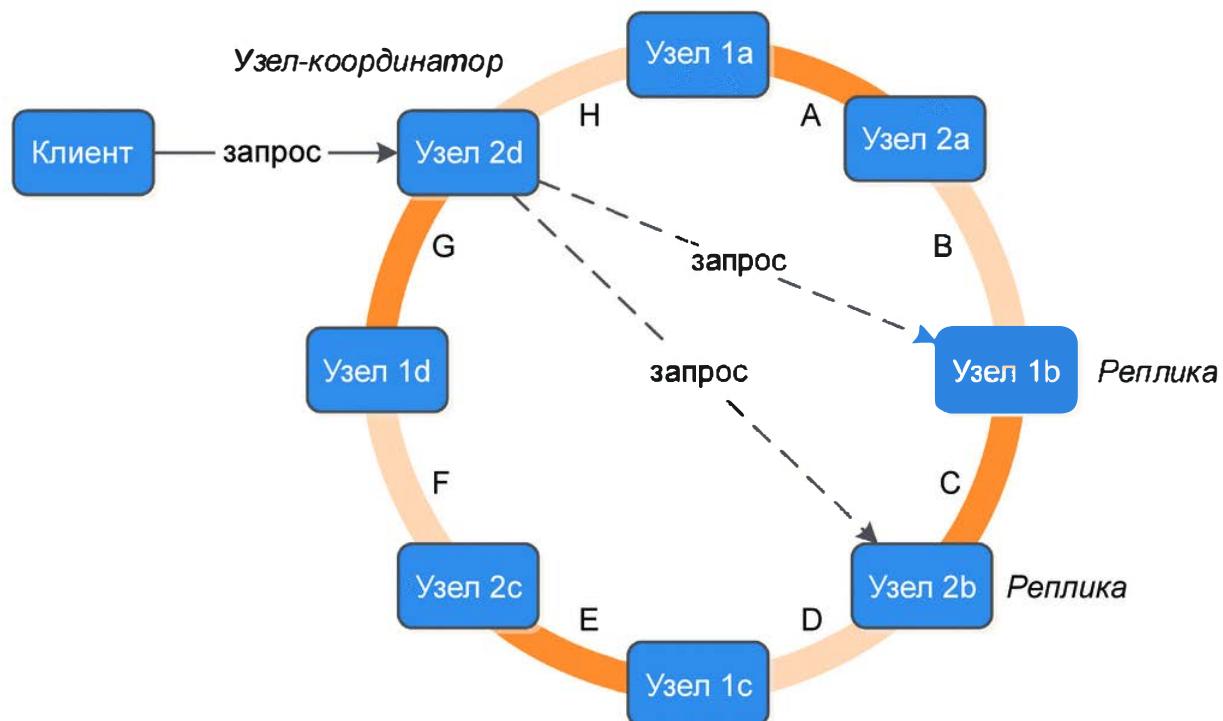


Рис. 6.3 ❖ Клиенты, узлы-координаторы и реплики

Клиент, желающий начать операцию чтения или записи, может подключиться к любому узлу кластера. Этот узел называется *узлом-координатором*. Координатор определяет, какие узлы содержат реплики читаемых или записываемых данных, и направляет им запрос.

В случае записи координатор связывается со всеми репликами, число которых определяется коэффициентом репликации, и считает операцию успешной, когда число полученных подтверждений сравняется с уровнем согласованности.

В случае чтения координатор обращается к такому числу реплик, чтобы можно было обеспечить требуемый уровень согласованности, и возвращает полученные данные клиенту.

Конечно, это описание работы Cassandra в случае, когда «все хорошо». Но вскоре мы обсудим некоторые механизмы обеспечения высокой доступности Cassandra, в т. ч. вручение напоминаний.

Таблицы в памяти, файлы SSTable и журналы фиксаций

Теперь рассмотрим некоторые внутренние структуры данных и файлы Cassandra, показанные на рис. 6.4. Cassandra хранит данные в памяти и на диске таким образом, чтобы обеспечить одновременно высокую производительность и долговечность. В этом разделе мы будем говорить о *таблицах в памяти*, *файлах SSTable* и *журналах фиксаций* (commit log), поддерживающих операции чтения и записи таблиц.



Рис. 6.4 ♦ Внутренние структуры данных и файлы в одном узле Cassandra

Любая операция записи сразу же заносится в *журнал фиксаций*. Это механизм восстановления после сбоев, необходимый для обеспечения долговечности. Запись не считается успешной, пока она не занесена в журнал фиксаций. Тем самым гарантируется, что даже если операция еще не отражена в хранилище в памяти, данные все равно можно будет восстановить. В случае останова или неожиданного краха базы данных журнал фиксаций – гарантия того, что данные не будут потеряны. Достигается это тем, что при следующем запуске узла журнал фиксаций воспроизводится (накатывается). И на самом деле это единственное место, где читается журнал фиксаций, клиенты никогда не читают из него данные.

После записи в журнал фиксаций значение записывается в структуру данных, которая называется *таблицей в памяти* (*memtable*). Каждая таблица в памяти содержит данные, относящиеся к одной таблице. В ранних реализациях Cassandra таблицы в памяти хранились в куче JVM, но благодаря улучшениям, начатым в версии 2.1, большинство таблиц в памяти было перемещено в память операционной системы. В результате Cassandra стала менее восприимчивой к флюктуациям производительности, связанным с работой сборщика мусора в Java.

Когда количество объектов, хранящихся в таблице в памяти, достигает некоторого порога, ее содержимое сбрасывается в файл *SSTable* на диске, после чего создается новая таблица в памяти. Операция сброса неблокирующая; в каждый момент времени может существовать несколько таблиц в памяти для одной логической таблицы: одна текущая, а остальные ожидают завершения сброса. Как правило, ждать приходится недолго, потому что узел выполняет сброс очень быстро, если только он не перегружен.

В каждом журнале фиксаций имеется внутренний битовый флаг, показывающий, нужен ли сброс на диск. Поступившая операция записи заносится в журнал фиксаций, и его флаг устанавливается в 1. Существует только один битовый флаг на таблицу, поскольку на сервере ведется только один журнал фиксаций. Любая запись в любую таблицу попадает в один и тот же журнал фиксаций, поэтому битовый флаг показывает, есть ли для данной таблицы в журнале что-то еще не сброшенное на диск. После того как таблица в памяти сброшена на диск, соответствующий флаг в журнале фиксаций устанавливается в 0, и это означает, что журнал фиксаций может больше не хранить эти данные для обеспечения долговечности. Как и для обычных

журналов, для журналов фиксаций можно задать порог смены; когда размер файла достигает этого порога, происходит замена текущего журнала новым, причем все существующие данные с несброшенным флагом переносятся из старого журнала в новый.

Идея SSTable заимствована из системы Google Bigtable. После того как таблица в памяти сброшена в файл SSTable на диск, она уже не может быть изменена приложением. Файлы SSTable уплотняются, но это отражается только на их представлении на диске; по существу, выполняется шаг слияния алгоритма сортировки слиянием, в результате чего создается новый файл, а старый удаляется, если все прошло успешно.



Откуда взялось название SSTable?

Исходный смысл акронима «SSTable» – Sorted String Table (отсортированная таблица строк) – для Cassandra не совсем точен, потому что данные не хранятся на диске в виде строк.

Начиная с версии 1.0 Cassandra поддерживает сжатие файлов SSTable, чтобы максимально эффективно использовать имеющееся место на диске. Сжатие настраивается на уровне таблиц. С каждым файлом SSTable ассоциирован фильтр Блума, который используется для дополнительного повышения производительности (см. раздел «Фильтры Блума» ниже).

Все операции записи последовательные, и это главная причина высочайшей производительности записи в Cassandra. Для того чтобы записать значение, не нужно ничего читать или искать, так как запись всегда производится в конец файла. Таким образом, единственное ограничение – это быстродействие диска. Задача уплотнения – реорганизация данных, но и в этом случае применяется последовательный ввод-вывод. Таким образом, выигрыш в производительности достигается благодаря расщеплению функций: операция записи – это чистое дописывание в конец, а уплотнение помогает организовать данные для повышения скорости чтения в будущем. Если бы Cassandra наивно вставляла значение сразу в нужное место, то пишущим клиентам пришлось бы безотлагательно платить за поиск в файле.

При чтении Cassandra ищет данные как в таблице в памяти, так и в файле SSTable, поскольку в памяти еще могут находиться данные, не сброшенные на диск. Код таблиц в памяти находится в классе org.apache.cassandra.db.Memtable.

Кэширование

На рис. 6.4 показано, что в Cassandra реализовано три вида кэширования:

- в *кэше ключей* хранится отображение ключей разделов на элементы индекса строк, что повышает скорость доступа к файлам SSTable на диске при выполнении чтения. Кэш ключей размещен в куче JVM;
- в *кэше строк* хранятся целые строки, что заметно увеличивает скорость доступа к часто читаемым строкам – ценой повышенного расхода памяти. Кэш строк размещен в памяти операционной системы;
- *кэш счетчиков* был добавлен в версии 2.1 с целью повышения производительности счетчиков путем уменьшения конкуренции за блокировки для наиболее часто используемых счетчиков.

По умолчанию кэширование ключей и счетчиков включено, а кэширование строк выключено, поскольку требует больше памяти. Cassandra периодически сбрасывает свои кэши на диск ради ускорения их прогрева при перезапуске узла. Настройку кэшей мы рассмотрим в главе 12.

Вручение напоминаний

Рассмотрим такой сценарий: Cassandra получает запрос на запись, но узел-реплика, на котором нужно сохранить данные, недоступен из-за разделения сети, отказа оборудования или по какой-то другой причине. Чтобы гарантировать общую доступность кольца в такой ситуации, в Cassandra реализован механизм *вручения напоминаний* (hinted handoff). Можете считать *напоминание* (hint) аналогом записи на бумажке с информацией о запрошенной операции записи. Если узел-реплика, которому предназначены данные, не работает, то узел-координатор А создает напоминание такого вида: «У меня есть информация, которую нужно записать на узел В. Я придержу эту информацию и буду наблюдать за доступностью узла В; как только он снова появится в сети, я отправлю ему запрос на запись». Таким образом, когда узел А с помощью протокола распространения сплетен обнаруживает, что узел В снова стал доступен, он «вручает» ему напоминание об операции записи. Cassandra хранит отдельное напоминание для каждого раздела, для которого есть задержанные операции записи.

В результате Cassandra всегда доступна для записи и, вообще говоря, позволяет кластеру выдерживать неизменную нагрузку записи даже в ситуации, когда некоторые узлы вышли из строя. Кроме того, сокращается время, в течение которого восстановившийся узел содержит несогласованные данные.

В общем случае напоминания не считаются операциями записи с точки зрения уровня согласованности. Исключение составляет уровень согласованности ANY, добавленный в версии 0.6. Он означает, что одного лишь наличия напоминания достаточно, чтобы операция записи считалась успешной. Иначе говоря, даже если удалось запомнить только напоминание, операция записи учитывается при подсчете успешных узлов. Отметим, что такая запись рассматривается как долговечная, но прочитать данные можно будет только после того, как напоминание доставлено целевой реплике.



Вручение напоминаний и гарантированная доставка

Вручение напоминаний используется в Amazon Dynamo и знакомо тем, кто наслышан о концепции гарантированной доставки в системах доставки сообщений типа Java Message Service (JMS). В долговечной очереди с гарантированной доставкой, если сообщение нельзя сразу доставить получателю, JMS будет ждать в течение заданного времени, а затем будет повторно отправлять запрос, пока сообщение не будет получено.

С вручением напоминания (да и с любой системой гарантированной доставки) связана одна практическая проблема: если узел в течение некоторого времени недоступен, то на других узлах может скопиться много напоминаний. И когда эти другие узлы обнаружат, что узел снова стал доступен, они затопят его запросами как раз в тот момент, когда он наиболее уязвим (изо всех сил старается вернуться в игру после отказа). Для решения этой проблемы в Cassandra имеется настраиваемый параметр, позволяющий ограничить время хранения напоминаний. Можно также вообще отключить механизм вручения напоминаний.

Код управления напоминаниями находится в классе `org.apache.cassandra.db.HintedHandOffManager`.

Хотя механизм вручения напоминаний повышает доступность Cassandra, он не в полной мере заменяет необходимость ручного исправления для обеспечения согласованности.

Облегченные транзакции и Paxos

В главе 2 мы говорили о том, что Cassandra предоставляет настраиваемую согласованность, в т. ч. возможность добиться строгой согласованности путем задания достаточно высоких уровней согласованности. Однако строгой согласованности недостаточно для предотвращения состояния гонки в случаях, когда клиент должен сначала прочитать, а затем записать данные.

Чтобы объяснить это на примере, вернемся к таблице `my_keyspace.user` из главы 5. Допустим, что мы разрабатываем клиент, являющийся частью приложения для управления учетными записями, который должен управлять записями о пользователях. При создании новой учетной записи мы должны проверить, что запись с таким именем пользователя не существует, иначе мы затрем данные старого пользователя. Итак, мы сначала выполняем чтение, чтобы проверить существование записи, а потом создаем запись, если такой еще не существует.

Мы хотели бы добиться *линеаризуемой согласованности*, т. е. гарантии, что никакой другой клиент не вклинился между нашими операциями чтения и записи и не произведет свою собственную модификацию. Начиная с версии 2.0 Cassandra поддерживает механизм *облегченных транзакций* (*lightweight transaction*, LWT), обеспечивающий линеаризуемую согласованность.

Реализация LWT в Cassandra основана на Paxos – алгоритме выработки консенсуса, который позволяет распределенным равноправным узлам получить согласованный результат без участия главного узла, координирующего транзакцию. Paxos и другие алгоритмы выработки консенсуса появились как альтернатива традиционному подходу к распределенным транзакциям на основе двухфазной фиксации.

Базовый алгоритм Paxos включает два этапа: подготовка/обещание и предложение/принятие. Для модификации данных узел-координатор может предложить новое значение узлам-репликам, взяв на себя роль лидера. Другие узлы могут одновременно выступать в роли лидеров для других модификаций. Каждый узел-реплика проверяет предложение, и если это последнее из предложений, которые он видел, то обещает не принимать предложения, ассоциированные с более ранними модификациями. Каждый узел-реплика также возвращает последнее полученное им предложение, которое еще находится в процессе рассмотрения. Если предложение одобрено большинством реплик, то лидер фиксирует предложение с той, однако, оговоркой, что

сначала он должен зафиксировать все находившиеся в процессе рассмотрения предложения, которые предшествовали его собственному.

Cassandra дополняет базовый алгоритм Paxos, стремясь поддержать желательную семантику «чтение раньше записи» (ее еще называют «проверить и установить») и разрешить сброс состояния между транзакциями. Для этого в алгоритм включаются два дополнительных этапа:

- 1) подготовка/обещание;
- 2) чтение/результаты;
- 3) предложение/принятие;
- 4) фиксация/подтверждение.

Таким образом, для успешного завершения транзакции необходимы четыре раунда коммуникации между координатором и репликами. Это обходится дороже обычной записи, поэтому, прежде чем применять облегченные транзакции в своем приложении, дважды подумайте.



Еще о протоколе Paxos

Протоколу Paxos посвящено несколько статей. Одно из лучших объяснений приведено в работе Лесли Лампорта «Paxos Made Simple» (<http://www.cs.utexas.edu/users/lorenzo/corsi/cs380d/past/03F/notes/paxos-simple.pdf>).

Облегченные транзакции в Cassandra ограничены одним разделом. На внутреннем уровне Cassandra хранит состояние Paxos для каждого раздела. Тем самым гарантируется, что транзакции в разных разделах не смогут помешать друг другу.

Код реализации алгоритма Paxos находится в классах из пакета `org.apache.cassandra.service.paxos`. Эти классы используются службой хранения, о которой мы скоро узнаем.

Надгробья

В реляционном мире часто встречается идея «мягкого удаления». Вместо того чтобы выполнять команду SQL `DELETE`, приложение выполняет команду `UPDATE`, которая изменяет значение в столбце с именем `deleted` или что-то в этом роде. Программисты часто применяют такой подход, например, для поддержки журнала аудита.

В Cassandra похожая идея называется *надгробьем* (*tombstone*). Именно так работают все операции удаления, причем делается это автоматически. Команда удаления не удаляет данные сразу, а тракту-

ется как операция обновления, которая возводит надгробье над «погившим» значением. Надгробье – это признак удаления, необходимый для пометки старых данных в файлах SSTable до тех пор, пока не будет выполнено уплотнение.

С этим механизмом связан параметр Garbage Collection Grace Seconds (время ожидания сборки мусора в секундах). Это время, на которое сервер откладывает уборку надгробий в мусор. По умолчанию оно равно 864 000 секунд, т. е. 10 суток. Cassandra следит за возрастом надгробья, и как только он достигает величины GCGraceSeconds, отправляет надгробье в мусор. Смысл этой задержки состоит в том, чтобы дать недоступному узлу время на восстановление; если узел не работает дольше, то он считается вышедшим из строя и заменяется.

Фильтры Блума

Фильтры Блума служат для повышения производительности чтения. Названы они по имени изобретателя, Бэртона Блума. Это очень быстрый недетерминированный алгоритм, который проверяет, является ли некий объект элементом множества. Недетерминированность связана с тем, что фильтр Блума может давать ложноположительные ответы, но никогда не дает ложноотрицательных. Принцип работы фильтра Блума заключается в отображении значений элементов множества в битовый массив и в сжатии большого множества данных в строку-дайджест с помощью хэш-функции. Дайджест, по определению, занимает гораздо меньше памяти, чем исходные данные. Фильтр сохраняется в памяти и позволяет повысить производительность, поскольку не каждая операция поиска ключа требует медленного доступа к диску. Таким образом, фильтр Блума является особым видом кэша. При выполнении запроса сначала проверяется фильтр Блума. Как мы уже сказали, ложноотрицательные ответы невозможны, поэтому если фильтр говорит, что объект отсутствует во множестве, значит, так оно и есть. Но если фильтр говорит, что объект является элементом множества, то необходимо обратиться к диску для проверки.

Код фильтра Блума находится в классе org.apache.cassandra.utils.BloomFilter. Cassandra позволяет повысить точность фильтра Блума (уменьшить число ложноположительных ответов) путем увеличения его размера, т. е. ценой потребления большего объема памяти. Вероятность ложноположительного ответа настраивается на уровне таблицы.



Другие применения фильтра Блума

Фильтры Блума используются и в других распределенных базах данных и технологиях кэширования, например: Apache Hadoop, Google Bigtable и Squid Proxy Cache.

Уплотнение

Как уже было сказано, файлы SSTable неизменяемы, что и позволяет Cassandra достигать столь высокой скорости записи. Но их необходимо периодически уплотнять, чтобы вычистить удаленные значения и обеспечить высокую скорость чтения. В ходе операции уплотнения файлы SSTable сливаются: производится объединение ключей и соответствующих им столбцов, выбрасывание надгробий и создание нового индекса.

Уплотнение – это процесс освобождения места путем слияния больших файлов данных. Можно провести грубую аналогию между уплотнением и перестройкой таблицы в реляционной базе. Но есть и существенное различие – в Cassandra уплотнение является прозрачной для пользователя операцией, которая производится в течение всего времени жизни сервера.

После уплотнения объединенные данные сортируются, над ними строится новый индекс, и только что объединенные, отсортированные и проиндексированные данные записываются в новый файл SSTable (каждый файл SSTable на самом деле состоит из нескольких файлов: *Данные*, *Индекс* и *Фильтр*). Всем этим занимается класс org.apache.cassandra.db.compaction.CompactionManager.

Еще одна важная функция уплотнения – повышение производительности путем сокращения числа операций поиска. Для нахождения столбца данных с указанным ключом нужно просмотреть ограниченное количество файлов SSTable. Если этот столбец часто изменяется, то вполне может оказаться, что все версии находятся в сброшенных на диск файлах SSTable. Уплотнение позволяет базе данных не просматривать каждый файл SSTable в поисках указанного ключа и не выбирать из них самое последнее значение каждого столбца.

В процессе уплотнения наблюдается кратковременный всплеск интенсивности ввода-вывода и занятого на диске места – это читаются старые и записываются новые файлы SSTable.

Cassandra поддерживает несколько алгоритмов уплотнения. Стратегия уплотнения задается для каждой таблицы и является расширением класса AbstractCompactionStrategy. Реализованы следующие стратегии:

- `SizeTieredCompactionStrategy` (**STCS**) подразумевается по умолчанию и рекомендуется для таблиц с высокой интенсивностью записи;
- `LeveledCompactionStrategy` (**LCS**) рекомендуется для таблиц с высокой интенсивностью чтения;
- `DateTieredCompactionStrategy` (**DTCS**) рекомендуется для временных рядов и других данных, основанных на времени.

Мы вернемся к этой теме в главе 12, когда будем обсуждать оптимальный выбор стратегии для таблицы.

Интересная особенность механизма уплотнения касается его пересечения с инкрементным исправлением. В версию 2.1 добавлена функция *антиуплотнения*. Как явствует из названия, антиуплотнение – операция, противоположная уплотнению в том смысле, что один файл SSTable разбивается на два: в первом находятся исправленные данные, а во втором – неисправленные.

При этом стратегии уплотнения модифицируются, так чтобы исправленные и неисправленные данные обрабатывались по отдельности и несливались в один файл SSTable.



Несколько слов о полном уплотнении

Пользователи, давно работающие с Cassandra, наверное, помнят об административной операции *полного уплотнения* (major compaction, или full compaction), которая объединяла несколько файлов SSTable в один. Эта возможность по-прежнему доступна, но со временем растеряла свою полезность. Более того, использовать ее в производственной среде даже не рекомендуется, потому что она ограничивает возможности Cassandra по удалению неактуальных данных. Подробнее об этой и других административных операциях над файлами SSTable, включенных в утилиту nodetool, мы узнаем в главе 11.

Антиэнтропия, исправление и деревья Меркла

В Cassandra применяется протокол *антиэнтропии* – частный случай протокола распространения сплетен для исправления реплицированных данных. Принцип работы протокола антиэнтропии заключается в сравнении реплик данных и согласовании различий между ними. Реализация антиэнтропии Cassandra устроена по образцу Amazon Dynamo (см. раздел 4.7 статьи о Dynamo).



Антиэнтропия в Cassandra

В Cassandra термин «антиэнтропия» употребляется в двух несколько отличающихся контекстах, но оба значения перекрываются.

- Как сокращение для обозначения механизма синхронизации реплик, призванного обеспечить обновление данных на разных узлах до последней версии.
- Иногда говорят, что Cassandra располагает средствами антиэнтропии, включая в них не только синхронизацию реплик, но и вручение напоминаний – антиэнтропийный механизм, работающий во время записи, о котором мы рассказывали выше.

Существуют два режима синхронизации реплик: *исправление на этапе чтения* и *антиэнтропийное исправление*. В режиме исправления во время чтения реплики синхронизируются в процессе чтения данных. Для достижения запрошенного уровня согласованности Cassandra читает данные из нескольких реплик и обнаруживает устаревшие данные. Если количество узлов, хранящих последнее значение, недостаточно, то сразу же производится исправление, в ходе которого устаревшие реплики обновляются. В противном случае исправление можно произвести в фоновом режиме, после того как операция чтения вернет данные. Этот принцип соблюдается как в Cassandra, так и в хранилищах ключей и значений Voldemort и Riak.

Антиэнтропийное исправление (иногда еще называемое *ручным исправлением*) запускается вручную и выполняется на узлах в процессе регулярного обслуживания. Такое исправление производится с помощью утилиты nodetool, о которой мы расскажем в главе 11. В этом случае Cassandra выполняет полное уплотнение (см. примечание выше). В процессе полного уплотнения сервер инициирует диалог TreeRequest/TreeReponse, чтобы обменяться деревьями Меркла с соседними узлами. Дерево Меркла – это хэш, представляющий данные в таблице. Если деревья, хранящиеся в разных узлах, не совпадают, их необходимо согласовать («исправить»), т. е. определить последние версии данных, которые должны храниться во всех деревьях. За сравнение и согласование деревьев отвечает класс org.apache.cassandra.service.AbstractReadExecutor.

Что такое дерево Меркла?

Дерево Меркла, названное по имени автора, Ральфа Меркла, называют еще «хэш-деревом». Эта структура данных, представленная двоичным деревом, полезна, потому что дает краткую сводку данных из

большего множества. Листьями хэш-дерева являются блоки данных (обычно файлы в файловой системе), подлежащие сокращению. Каждый родительский узел содержит хэш своих непосредственных потомков, так что в результате получается очень краткое представление. В Cassandra реализация дерева Меркля находится в классе `org.apache.cassandra.utils.MerkleTree`.

Деревья Меркля используются в Cassandra для того, чтобы гарантировать, что все узлы одноранговой сети получают блоки данных неизмененными и неповрежденными. Они применяются также в криптографии для проверки файлов и передаваемых данных.

И в Cassandra, и в Dynamo деревья Меркля применяются в механизме антиэнтропии, но реализации несколько различаются. В Cassandra у каждой таблицы имеется свое дерево Меркля; оно создается как мгновенный снимок в процессе полного уплотнения и хранится ровно столько времени, сколько необходимо для передачи соседним узлам в кольце. Достоинство такой реализации – в уменьшении объема сетевого ввода-вывода.

Многоступенчатая событийно-ориентированная архитектура (SEDA)

На дизайн Cassandra оказала влияние многоступенчатая событийно-ориентированная архитектура (Staged Event-Driven Architecture – SEDA). Эта общая архитектура интернет-служб с высокой степенью параллелизма первоначально была описана в статье 2001 года Matt Welsh, David Culler, Eric Brewer «SEDA: An Architecture for Well-Conditioned, Scalable Internet Services» (последний автор, как вы, наверное, помните, предложил также теорему CAP). Эта статья доступна по адресу <http://www.eecs.harvard.edu/~mdw/proj/seda>.

В типичном приложении единица работы часто выполняется в пределах одного потока. Так, операция записи начинается и заканчивается в одном и том же потоке. Но Cassandra устроена иначе: ее модель параллелизма основана на SEDA, поэтому операция может начаться в одном потоке, затем продолжиться в другом, который, в свою очередь, может передать работу еще каким-то потокам. Но не текущий поток отвечает за передачу работы другому потоку. Вместо этого работа разбивается на так называемые *ступени* (stage), и ход выполнения определяет пул потоков (на самом деле класс `java.util.concurrent.ExecutorService`), ассоциированный со ступенью.

Ступень – это основная единица работы, в процессе выполнения одной операции может производиться переход от предыдущей ступени к следующей. Поскольку каждая ступень может обрабатываться своим пулом потоков, производительность Cassandra резко возрастает. Кроме того, благодаря такому дизайну Cassandra может лучше управлять собственными ресурсами, потому что одни операции требуют дискового ввода-вывода, другие интенсивно обращаются к сети, третьи потребляют ресурсы процессора и т. д., а пулы способны управлять своей работой в соответствии с доступностью ресурсов.

Ступень состоит из входящей очереди событий, обработчика событий и ассоциированного пула потоков. Ступени управляются контроллером, который отвечает за планирование и выделение потоков; в Cassandra эта модель параллелизма реализована с помощью пула потоков в классе `java.util.concurrent.ExecutorService`. Если вам интересно, как это работает, загляните в код класса `org.apache.cassandra.concurrent.StageManager`. В виде ступеней представлены следующие операции Cassandra, многие из которых обсуждались в этой главе:

- чтение (локальное чтение);
- изменение (локальная запись);
- распространение сплетен;
- запрос-ответ (взаимодействие с другими узлами);
- антиэнтропия (исправление с помощью `nodetool`);
- исправление на этапе чтения;
- миграция (внесение изменений в схему);
- вручение напоминаний.

Увидеть пулы потоков, ассоциированные с каждой ступенью, можно с помощью команды `tpstats` утилиты `nodetool`, которую мы будем рассматривать в главе 10.

Есть и другие операции, реализованные в виде ступеней, например операции над таблицами в памяти, в т. ч. сброс данных в файлы `SSTable` и освобождение памяти. Ступени реализуют интерфейс `IVerbHandler`, поддерживая функциональность, описываемую некоторым глаголом. Поскольку идея изменения представлена ступенью, она может найти применение как в операциях вставки, так и в операциях удаления.



Прагматический подход к SEDA

Со временем разработчики Cassandra и других технологий на основе архитектуры SEDA столкнулись с проблемами производительности из-за того, что для каждой ступени нужен был отдельный пул потоков, а для передачи событий между ступенями – отдельная

очередь, пусть даже ступени живут очень короткое время. Эти проблемы были сформулированы Мэттом Уэлшем (Matt Welsh) в статье «A Retrospective on SEDA», которую он опубликовал в своем блоге (<http://matt-welsh.blogspot.ru/2010/07/retrospective-on-seda.html>). Разработчики Cassandra ослабили чрезмерно строгие соглашения SEDA, свернув некоторые ступени в общий пул потоков с целью повышения производительности. Но основные принципы – разделение работы на ступени и использование очередей и пулов потоков для управления ступенями – по-прежнему присутствуют в коде.

Диспетчеры и службы

Речь идет о наборе классов, реализующих внутренние механизмы управления в Cassandra. С некоторыми мы уже встречались в этой главе: HintedHandOffManager, CompactionManager и StageManager. Мы приведем краткий обзор еще нескольких классов, стремясь познакомить вас с наибольшими важными. Многие из них являются MBean-объектами в смысле технологии Java Management Extension (JMX), а значит, могут сообщать о состоянии и метриках и в некоторых случаях допускают конфигурирование и управление своей работой. Подробнее о том, как строится взаимодействие с такими MBean-объектами, мы поговорим в главе 10.

Демон Cassandra

Интерфейс org.apache.cassandra.service.CassandraDaemon представляет жизненный цикл службы Cassandra, работающей на одном узле. Он включает типичные операции жизненного цикла: start, stop, activate, deactivate и destroy.

Можно программно создать экземпляр Cassandra в памяти, воспользовавшись классом org.apache.cassandra.service.EmbeddedCassandraService. Создание такого экземпляра может быть полезно для автономного тестирования программ, работающих с Cassandra.

Движок хранения

Основную функциональность Cassandra, связанную с хранением данных, обычно называют движком хранения. Она реализована преимущественно в классах из пакета org.apache.cassandra.db. Главной точкой входа является класс ColumnFamilyStore, который отвечает за все аспекты хранения таблиц, в т. ч. за журналы фиксаций, таблицы в памяти, файлы SSTable и индексы.



Основные изменения в движке хранения

В версии 3.0 движок хранения был значительно переработан, чтобы привести представления данных в памяти и на диске в соответствие с CQL. Очень хорошая сводка этих изменений приведена в публикации JIRA с идентификатором CASSANDRA-8099 (<https://issues.apache.org/jira/browse/CASSANDRA-8099>).

Переписывание движка хранения стало предвестником многих других изменений и самого главного из них – поддержки материализованных представлений, реализация которой описана в публикации CASSANDRA-6477 (<https://issues.apache.org/jira/browse/CASSANDRA-6477>). Эти две публикации будет интересно почитать тем, кто хочет лучше понять, какие изменения пришлось реализовать «под капотом», чтобы открыть дорогу новым мощным возможностям.

Служба хранения

Cassandra обертыает движок хранения службой, представленной классом org.apache.cassandra.service.StorageService. Служба хранения содержит маркер узла, описывающий диапазон данных, за который отвечает данный узел.

Запуск сервера начинается с вызова метода initServer этого класса, после чего сервер регистрирует обработчики глаголов SEDA, принимает некоторые решения о своем состоянии (в частности, производилась ли при его запуске начальная загрузка или нет и какой используется разделитель) и регистрирует MBean-объект в сервере JMX.

Прокси хранения

Объект класса org.apache.cassandra.service.StorageProxy располагается перед StorageService и занимается ответом на запросы клиентов. Он координирует с другими узлами сохранения и извлечения данных, в том числе сохранения напоминаний, когда в этом возникает необходимость. Класс StorageProxy также участвует в управлении обложенными транзакциями.



Прямой вызов прокси хранения

В принципе, есть возможность программно создать экземпляр StorageProxy в памяти, но такая практика не считается частью официального API Cassandra, поэтому результат может изменяться от версии к версии.

Служба обмена сообщениями

Задача класса `org.apache.cassandra.net.MessagingService` – создать прослушиватели сокета для обмена сообщениями; через эту службу проходят входные и выходные сообщения, относящиеся к данному узлу. Метод `MessagingService.listen` создает поток. Каждое входящее подключение затем передается пулу потоков `ExecutorService`, где класс `org.apache.cassandra.net.IncomingTcpConnection` (расширяющий Thread) десериализует сообщение. Сообщение проверяется и направляется подходящему обработчику.

Поскольку в классе `MessagingService` вовсю используются ступени, а поддерживаемый им пул обернут MBean-объектом, JMX позволяет многое узнать о работе этой службы.

Диспетчер потоков данных

Потоковый режим (*streaming*) в Cassandra – это оптимизированный способ передачи секций файлов SSTable между узлами по постоянному TCP-соединению; все остальные коммуникации между узлами имеют вид сериализованных сообщений. За обработку этого потока сообщений отвечает класс `org.apache.cassandra.streaming.StreamManager`, который управляет соединением, занимается сжатием сообщений, отслеживанием хода выполнения и сбором статистики.

Сервер транспортного протокола CQL

CQL Native Protocol – это двоичный протокол, который клиенты используют для обмена данными с Cassandra. Классы, реализующие этот протокол, в т. ч. `Server`, находятся в пакете `org.apache.cassandra.transport`. Сервер транспортного протокола обслуживает запросы на подключение от клиентов и маршрутизирует входящие запросы, делегируя их обработку классу `StorageProxy`.

Существуют и другие классы, отвечающие за основные функции Cassandra. Ниже для интересующихся читателей перечислены некоторые из них.

Функциональная возможность	Класс
Исправление	<code>org.apache.cassandra.service.ActiveRepairService</code>
Кэширование	<code>org.apache.cassandra.service.CachingService</code>
Миграция	<code>org.apache.cassandra.service.MigrationManager</code>
Материализованные представления	<code>org.apache.cassandra.db.view.MaterializedViewManager</code>
Вторичные индексы	<code>org.apache.cassandra.db.index.SecondaryIndexManager</code>
Авторизация	<code>org.apache.cassandra.auth.CassandraRoleManager</code>

Системные пространства ключей

Следуя принципу самодостаточности, Cassandra пользуется собственным хранилищем для хранения метаданных о кластере и локальном узле. Та же идея заложена в базах метаданных master и tempdb в Microsoft SQL Server. В базе master хранится информация об использовании места на диске и системных параметрах, а также общие сведения об установке сервера, а база tempdb используется как рабочая область для хранения промежуточных результатов и выполнения задач общего характера. В Oracle тоже имеется табличное пространство SYSTEM, используемое для сходных целей. Ну а в Cassandra мы встречаем пространства ключей с префиксом system.

Вернемся к cqlsh и бегло познакомимся с таблицами в системных пространствах ключей.

```
cqlsh> DESCRIBE TABLES;

Keyspace system_traces
-----
events sessions

Keyspace system_schema
-----
materialized_views      functions      aggregates      types          columns
tables                  triggers       keyspaces      dropped_columns

Keyspace system_auth
-----
resource_role_permissions_index  role_permissions  role_members
roles

Keyspace system
-----
available_ranges          sstable_activity    local
range_xfers               peer_events        hints
materialized_views_builds_in_progress paxos
[IndexInfo]                batchlog
peers                      size_estimates
built_materialized_views   compaction_history

Keyspace system_distributed
-----
repair_history  parent_repair_history
```



Вы видите другие системные пространства ключей?

Если вы работаете с версией Cassandra, предшествующей 2.2, то некоторые из перечисленных выше пространств ключей отсутствуют. Основные системные пространства ключей существовали с самого начала, но `system_traces` keyspace появилось только в версии 1.2 для поддержки трассировки запросов. А `system_auth` и `system_distributed` были добавлены в версии 2.2 для поддержки ролевого управления доступом (RBAC) и запоминания данных об исправлениях соответственно. Наконец, таблицы, относящиеся к определению схем, при переходе на версию 3.0 были перемещены из пространства ключей `system` в `system_schema`.

Как видим, многие из этих таблиц связаны с концепциями, обсуждаемыми в этой главе.

- Информация о структуре кластера, передаваемая по протоколу распространения сплетен, хранится в таблицах `system.local` и `system.peers`. Сюда относятся данные о локальном и прочих узлах, включая IP-адреса, местоположение в терминах ЦОДа и стойки, номера версий CQL и протоколов.
- В таблицах `system.range_xfers` и `system.available_ranges` хранятся диапазоны маркеров, ассоциированные с каждым узлом, а также еще не назначенные диапазоны.
- В таблицах `system_schema.keyspaces`, `system_schema.tables` и `system_schema.columns` хранятся определения пространств ключей, таблиц и индексов для всего кластера.
- Сведения о построении материализованных представлений хранятся в таблицах `system.materialized_views_builds_in_progress` и `system.built_materialized_views`, а о доступных для работы представлениях – в таблице `system_schema.materialized_views`.
- В таблице `system_schema.types` находятся определения пользовательских типов, в таблице `system_schema.triggers` – определения триггеров для различных таблиц, в таблице `system_schema.functions` – определения пользовательских функций, а в таблице `system_schema.aggregates` – определения пользовательских агрегатов.
- В таблице `system.paxos` хранится состояние выполняемых транзакций, а в таблице `system.batchlog` – состояние атомарных пакетов команд.
- В таблице `system.size_estimates` хранится оценка числа разделов для каждой таблицы, эта информация нужна для интеграции с Hadoop.



Исключение таблицы system.hints

Напоминания традиционно хранились в таблице `system.hints`. Но как заметили вдумчивые разработчики, тот факт, что напоминания – это в действительности сообщения, которые нужно хранить в течение короткого времени, а затем удалять, означает, что мы имеем пример хорошо известного антипаттерна использования Cassandra в качестве очереди (см. обсуждение в главе 5). Поэтому начиная с версии 3.0 напоминания хранятся в плоских файлах.

Снова зайдем в `cqlsh` и взглянем на атрибуты пространства ключей `system`:

```
cqlsh> USE system;
cqlsh:system> DESCRIBE KEYSPACE;
CREATE KEYSPACE system WITH replication =
  {'class': 'LocalStrategy'} AND durable_writes = true;
...

```

Мы опустили часть вывода, так как полная структура всех таблиц нас не интересует. Из первой строки следует, что для пространства ключей `system` используется стратегия репликации `LocalStrategy`, т. е. эта информация предназначена только для внутренних целей и не реплицируется на другие узлы.



Неизменяемость системных пространств ключей

Команда `DESCRIBE` для системных пространств ключей выводит ту же информацию, что и для любого другого пространства ключей, т. е. таблицы описываются с помощью синтаксиса команды `CREATE TABLE`. Но не впадайте в заблуждение – модифицировать схему системных пространств ключей вам не удастся.

Резюме

В этой главе мы рассмотрели столпы архитектуры Cassandra, в т. ч. распространение сплетен, осведомители, разделители, репликацию, согласованность, антиэнтропию, вручение напоминаний, облегченные транзакции и применение многоступенчатой событийно-ориентированной архитектуры для максимизации производительности. Мы также обсудили некоторые внутренние структуры данных Cassandra: таблицы в памяти, файлы SSTable и журналы фиксаций, а равно выполнение таких операций, как создание надгробий и уплотнение. Наконец, мы упомянули некоторые важные классы и интерфейсы для тех, кому интересно покопаться в исходном коде.

Глава 7

Настройка Cassandra

В этой главе мы построим свой первый кластер и рассмотрим различные конфигурационные параметры Cassandra. В принципе, Cassandra будет работать и вообще без всякой настройки, достаточно просто скачать дистрибутив, распаковать его и запустить сервер в конфигурации по умолчанию. Однако Cassandra является такой мощной технологией не в последнюю очередь потому, что уделяет много внимания возможности настройки и написания дополнительных модулей. Правда, количество параметров может поначалу вогнать в ступор.

Мы сосредоточимся на тех аспектах Cassandra, которые влияют на поведение узла кластера, а также на метаоперациях: разделении, осведомлении и репликации. Настройка производительности и обеспечение безопасности – это дополнительные темы, которые будут рассмотрены в главах 12 и 13 соответственно.

Диспетчер кластера Cassandra

Чтобы попрактиковаться в создании и настройке кластера, мы воспользуемся диспетчером кластера Cassandra (Cassandra Cluster Manager) – см. Это набор скриптов на Python, разработанный Сильвеном Лебреном (Sylvain Lebresne) и другими; он позволяет запустить кластер из нескольких узлов на одной машине, а следовательно, быстро настроить кластер, не приобретая дополнительного оборудования.

Программа распространяется через GitHub (<https://github.com/rctmanus/ccm>). Проще всего начать работу с ней, клонировав репози-

торий Git. Откройте окно терминала, перейдите в каталог, где хотите создать клон, и выполните такую команду:

```
$ git clone https://github.com/pcmanus/ccm.git
```

После этого выполните скрипт установки с правами администратора:

```
$ sudo ./setup.py install
```



Актуальная процедура установки ccm

Мы приводим упрощенные инструкции по работе с ccm. Рекомендуем заглянуть на сайт, где описаны зависимости и особенности установки для платформ Windows и MacOS X. Поскольку ccm активно разрабатывается, детали могут со временем изменяться.

После установки ccm должен оказаться в списке путей к каталогам с исполняемыми файлами. Для получения перечня доступных команд введите ccm или ccm -help. Чтобы получить дополнительные сведения о конкретной команде, введите ccm <command> -h. С некоторыми командами мы познакомимся в следующих разделах, когда будем создавать и настраивать кластер.

Чтобы больше узнать о том, как работает ccm, можете заглянуть в код скриптов на Python. Можно также запускать скрипты из автоматизированных комплектов тестов.

Создание кластера

Cassandra можно запустить на одной машине, и для начала этого достаточно, чтобы понять, как читать и записывать данные. Но вообще-то Cassandra задумывалась для работы в кластере из многих машин, между которыми можно распределять нагрузку в случае очень плотного потока запросов. В этом разделе мы узнаем о конфигурации, необходимой для создания кольца общающихся между собой экземпляров Cassandra. На каждом узле кластера имеется конфигурационный файл *cassandra.yaml*, который находится в подкаталоге *conf* установочного каталога Cassandra.

Для настройки кластера нужно задать прежде всего имя кластера, разделитель, осведомитель и узлы-распространители (seed node). Имя кластера, разделитель и осведомитель должны быть одинаковы на всех узлах кластера. Узлы-распространители могут различаться, но лучше бы тоже делать их одинаковыми; чуть ниже мы приведем более конкретные рекомендации по настройке.

Кластерам Cassandra присваиваются имена, для того чтобы машины из одного кластера не могли присоединиться к другому, где они совершенно не к месту. По умолчанию в файле *cassandra.yaml* сконфигурирован кластер с именем «Test Cluster». Чтобы задать другое имя, измените свойство *cluster_name* – только не забудьте сделать это на всех компьютерах, которые предполагается сделать узлами кластера.



Изменение имени кластера

Если вы уже записали данные в кластер Cassandra, а затем изменили его имя, то Cassandra предупредит о несоответствии имени кластера при попытке прочитать файлы данных на этапе запуска, после чего остановится.

Давайте создадим кластер с помощью программы *ccm*:

```
$ ccm create -v 3.0.0 -n 3 my_cluster --vnodes
Downloading http://archive.apache.org/dist/cassandra/3.0.0/
apache-cassandra-3.0.0-src.tar.gz to
/var/folders/63/6h7dm1k51bd6phvm7fbngskc0000gt/T/
    ccm-z2kHp0.tar.gz (22.934MB)
24048379 [100.00%]
Extracting /var/folders/63/6h7dm1k51bd6phvm7fbngskc0000gt/T/
    ccm-z2kHp0.tar.gz as version 3.0.0 ...
Compiling Cassandra 3.0.0 ...
Current cluster is now: my_cluster
```

Эта команда создает кластер, запуская указанную нами версию Cassandra – в данном случае 3.0.0. Кластер называется *my_cluster* и состоит из трех узлов. Мы указали, что хотим использовать виртуальные узлы, потому что по умолчанию *ccm* создает узлы с единственным маркером. *ccm* делает наш кластер текущим, т. е. будет использовать его в последующих командах. Вы наверняка заметили, что *ccm* скачивает исходный код запрошенной версии и компилирует его. Объясняется это тем, что *ccm* немного модифицирует исходный код Cassandra, чтобы можно было поддержать запуск нескольких узлов на одной машине. Можно было бы также использовать исходный код, который мы уже скачали в главе 3. Чтобы узнать о других параметрах создания кластера, выполните команду *ccm create -h*.

Создав кластер, мы можем убедиться, что это единственный кластер в списке (помеченный к тому же как кластер по умолчанию), и узнать его состояние:

```
$ ccm list
*my_cluster
```

```
$ ccm status
Cluster: 'my_cluster'
-----
node1: DOWN (Not initialized)
node3: DOWN (Not initialized)
node2: DOWN (Not initialized)
```

Сейчас ни один узел еще не инициализирован. Запустим кластер и снова проверим состояние:

```
$ ccm start
$ ccm status
Cluster: 'my_cluster'
-----
node1: UP
node3: UP
node2: UP
```

Того же эффекта можно было бы достичь, запустив каждый узел по отдельности с помощью скрипта *bin/cassandra* (или *service start cassandra* для установок из пакета). Чтобы подробнее узнать о состоянии узла, введем такую команду:

```
$ ccm nodetool status
Datacenter: datacenter1
=====
Status=Up/Down
| / State=Normal/Leaving/Joining/Moving
-- Address      Load      Tokens  Owns    Host ID      Rack
UN 127.0.0.1   193.2 KB   256     ?        e5a6b739-... rack1
UN 127.0.0.2   68.45 KB   256     ?        48843ab4-... rack1
UN 127.0.0.3   68.5  KB   256     ?        dd728f0b-... rack1
```

Это эквивалентно выполнению команды *nodetool status* для отдельного узла. Из распечатки видно, что все узлы работают и сообщают о нормальном состоянии (UN). У каждого узла 256 маркеров, но никаких данных, поскольку данные мы еще не добавляли. (Мы немного сократили идентификатор хоста.)

Для получения списка маркеров, принадлежащих каждому узлу, служит команда *nodetool ring*. В *ccm* то же самое можно сделать так:

```
$ ccm nodetool ring
Datacenter: datacenter1
=====
```

```
Address   Rack Status State ... Token
                                9205346612887953633
127.0.0.1 rack1 Up      Normal ... -9211073930147845649
127.0.0.3 rack1 Up      Normal ... -9114803904447515108
127.0.0.3 rack1 Up      Normal ... -9091620194155459357
127.0.0.1 rack1 Up      Normal ... -9068215598443754923
127.0.0.2 rack1 Up      Normal ... -9063205907969085747
```

Эта команда требует указания узла. На вывод это никак не влияет, а нужно только для того, чтобы знать, к какому узлу должна подключиться утилита nodetool, чтобы получить информацию о кольце. Как видим, маркеры случайным образом распределены между тремя узлами. (Мы снова сократили вывод и опустили столбцы Owns и Load.)

Подробнее о конфигурации кластера

Весьма интересно заглянуть под капот и посмотреть, какие изменения в конфигурации производит `ccm`, чтобы получить кластер, работающий на локальной машине. По умолчанию `ccm` хранит метаданные и конфигурационные файлы кластера в подкаталоге `.ccm` домашнего каталога; там же хранятся файлы с исходным кодом выполняемой версии Cassandra. Познакомимся с содержимым этого каталога:

```
$ cd ~/.ccm; ls
CURRENT          my_cluster    repository
```

Каталог `repository` содержит исходный код, который скачал `ccm`. Зайдя внутрь каталога `my_cluster`, мы увидим, что для каждого узла имеется свой подкаталог:

```
$ cd my_cluster; ls
cluster.conf     node1         node2         node3
```

Файл `cluster.conf` содержит параметры, заданные при создании кластера. Чтобы понять, в каких параметрах узлы различаются, воспользуемся командой `diff` для сравнения содержимого каталогов. Например:

```
$ cd ~/.ccm/my_cluster
$ diff node1/conf/ node2/conf/
```

Команда покажет, что конфигурационные файлы отличаются именами каталогов для хранения данных, журналов фиксаций и выходных журналов, адресами прослушивания и RPC, используемыми для сетевых коммуникаций, и номерами портов JMX для удаленного управления. Ниже в этой главе мы рассмотрим все эти параметры подробнее.

Узлы-распространители

Для нового узла кластера нужен так называемый *узел-распространитель* (seed node). Этот узел используется как точка контакта с другими узлами, чтобы Cassandra могла узнать о топологии кластера, т. е. о том, какие хосты какими диапазонами владеют. Например, если узел А играет роль распространителя для узла В, то всякий раз, как В входит в сеть, он будет использовать А как справочник, от которого можно получить данные. Этот процесс называется *начальной загрузкой* (bootstrapping), а иногда *автоматической начальной загрузкой* (auto bootstrapping), потому что Cassandra все это проделывает автоматически. Узлы-распространители не выполняют автоматическую начальную загрузку, поскольку предполагается, что они будут первыми узлами кластера.

По умолчанию в файле *cassandra.yaml* имеется только одна запись seeds, в которой указан локальный хост:

```
- seeds: "127.0.0.1"
```

Для добавления узлов-распространителей в кластер нужно просто включить в строку seeds дополнительные элементы: IP-адреса или имена хостов узлов. Например, заглянув в файл *cassandra.yaml* для одного из созданных сим узлов, мы увидим такую строку:

```
- seeds: 127.0.0.1, 127.0.0.2, 127.0.0.3
```

В производственном кластере будут указаны IP-адреса других хостов, а не возвратные адреса. Чтобы обеспечить высокую доступность процесса начальной загрузки Cassandra, рекомендуется иметь по меньшей мере два узла-распространителя на один ЦОД. Тогда возрастает вероятность, что хотя бы один локальный распространитель останется доступным, если другой временно выйдет из строя вследствие потери связности сети между ЦОДами.

Как вы, наверное, заметили, изучая файл *cassandra.yaml*, список распространителей на самом деле является частью объемлющего определения поставщика распространителей (seed provider). Интерфейс *org.apache.cassandra.locator.SeedProvider* определяет контракт, который должен быть реализован. Cassandra предоставляет простую реализацию в виде класса *SimpleSeedProvider*, который загружает IP-адреса узлов-распространителей из файла *cassandra.yaml*.

Разделители

Задача разделителя – указать порядок сортировки ключей разделов. От этого зависит, как данные будут распределяться между узлами. Разделитель также определяет, какие возможности будут доступны для запроса диапазонов строк. Имя класса разделителя задается в свойстве `partitioner` в файле `cassandra.yaml`. В нашем распоряжении несколько разделителей, которые рассматриваются ниже.



Изменение разделителя

После того как в кластер добавлены какие-то данные, изменять разделитель уже нельзя. Поэтому дважды подумайте, прежде чем изменять значение по умолчанию!

Разделитель Murmur3Partitioner

По умолчанию подразумевается разделитель `org.apache.cassandra.dht.Murmur3Partitioner`. В нем для генерации маркеров применяется алгоритм хэширования `murmur`. Его достоинством является случайное равномерное распределение ключей между узлами кластера, а недостатком – неэффективность запросов по диапазону, поскольку ключи, принадлежащие заданному диапазону, могут находиться в отдаленных узлах кольца; кроме того, в ответ на запрос по диапазону данные будут возвращены в случайному порядке.

В общем случае в новых кластерах следует использовать разделитель `Murmur3Partitioner`. Но в целях обратной совместимости Cassandra предлагает несколько старых разделителей.

Разделитель RandomPartitioner

Этот разделитель, реализованный в классе `org.apache.cassandra.dht.RandomPartitioner`, ранее использовался в Cassandra по умолчанию. Для решения о том, какому узлу кольца принадлежит данный ключ, в нем используется объект `BigIntegerToken`, хэшированный с помощью криптографического алгоритма MD5. Хотя и `RandomPartitioner`, и `Murmur3Partitioner` основаны на случайных функциях хэширования, криптографический алгоритм в `RandomPartitioner` работает гораздо медленнее, потому-то теперь по умолчанию подразумевается `Murmur3Partitioner`.

Разделитель OrderPreservingPartitioner

Этот разделитель реализован в классе `org.apache.cassandra.dht.OrderPreservingPartitioner`. Маркером является строка в кодировке

UTF-8, основанная на ключе. Поэтому строки сортируются в порядке ключей, так что физическая структура данных согласована с заданным пользователем порядком сортировки. Настроив семейство столбцов на разделение с сохранением порядка (*order-preserving partitioning – OPP*), мы сможем выполнять запросы с групповыми условиями (*range slice*).

Стоит отметить, что при выполнении запросов по диапазону OPP не дает преимуществ в эффективности, по сравнению со случайным разделением, а лишь обеспечивает упорядочение результатов. К его недостаткам можно отнести тот факт, что кольцо может получиться «кособоким», потому что реальные данные обычно распределены неравномерно. Например, рассмотрим ценность, назначаемую различным буквам в игре Скрабл. Буквы Q и Z встречаются редко, поэтому их ценность велика. В случае распределителя OPP в одних узлах может оказаться много данных, а в других – гораздо меньше. Узлы с большим количеством данных, из-за которых кольцо становится несимметричным, часто называют *горячими*. Сохранение порядка многим пользователям кажется привлекательным свойством. Но на практике применение OPP заставляет эксплуатационников чаще вручную перебалансировать узлы с помощью команд `nodetool loadbalance` или `move`. Из-за этого мы не рекомендуем использовать разделители, сохраняющие порядок. Лучше пользуйтесь индексами.

Разделитель **ByteOrderedPartitioner**

Этот разделитель (ВОР) сохраняет порядок, но вместо преобразования данных в строки, как *OrderPreservingPartitioner* или *CollatingOrderPreservingPartitioner*, он рассматривает их просто как последовательность байтов. Если вам нужен сохраняющий порядок разделитель, который не проверяет, что ключи являются корректными строками, то для повышения производительности рекомендуем использовать ВОР.



Избегайте горячих узлов

Хотя разделитель *Murmur3Partitioner* формирует маркеры случайнym образом, он все-таки может приводить к появлению горячих узлов, но, по сравнению с разделителями, сохраняющими порядок, проблема далеко не так остра. Чтобы снизить вероятность образования горячих узлов, нужны дополнительные знания о топологии кольца. В версии 3.0 алгоритм выбора маркера улучшен в этом направлении. Если присвоить свойству `allocate_tokens_keyspace`

в файле *cassandra.yaml* имя конкретного пространства ключей, то разделитель оптимизирует выбор маркера с учетом стратегии репликации, заданной для этого пространства ключей. Это особенно полезно, когда в кластере размещено только одно пространство ключей или для всех пространств ключей задана одна и та же стратегия репликации. В версии 3.0 эта возможность поддерживается только для разделителя *Murmur3Partitioner*.

Осведомители

Задача осведомителя состоит в том, чтобы предоставить сведения об относительной близости хостов. Осведомители собирают информацию о топологии сети, чтобы Cassandra могла эффективно маршрутизировать запросы. Осведомитель определяет, как каждый узел расположен относительно других. Делать выводы о строении ЦОДов – задача стратегии репликации. Какой именно осведомитель использовать, задает свойство *endpoint_snitch* в файле *cassandra.yaml*.

Простой осведомитель

По умолчанию в Cassandra используется осведомитель *org.apache.cassandra.locator.SimpleSnitch*. Он ничего не знает о стойках, поэтому не годится для развертывания в нескольких ЦОДах. Выбирая этот осведомитель, вы должны использовать стратегию репликации *SimpleStrategy* для всех пространств ключей.

Осведомитель на основе файла свойств

Осведомитель *org.apache.cassandra.locator.PropertyFileSnitch* знает о стойках, т. е. использует информацию о топологии кластера, предоставленную вами в виде стандартного для Java файла свойств в формате ключ-значение *cassandra-topology.properties*. Конфигурация по умолчанию в файле *cassandra-topology.properties* выглядит так:

```
# IP-адрес узла Cassandra=Data Center:Rack
192.168.1.100=DC1:RAC1
192.168.2.200=DC2:RAC2

10.0.0.10=DC1:RAC1
10.0.0.11=DC1:RAC1
10.0.0.12=DC1:RAC2

10.20.114.10=DC2:RAC1
10.20.114.11=DC2:RAC1
```

```

10.21.119.13=DC3:RAC1
10.21.119.10=DC3:RAC1

10.0.0.13=DC1:RAC2
10.21.119.14=DC3:RAC2
10.20.114.15=DC2:RAC2

# умолчание для неизвестных узлов
default=DC1:r1

```

Здесь мы видим три ЦОДа (DC1, DC2, DC3), с двумя стойками каждый (RAC1 и RAC2). Предполагается, что все не упомянутые явно узлы находятся в ЦОДе по умолчанию и стойке (DC1, r1).

Если вы решите выбрать этот или другой знающий о стойках осведомитель, то имейте в виду, что это те же имена ЦОДов и стоец, что используются при задании параметров стратегии репликации `NetworkTopologyStrategy`.

Зарегистрируйте в этом файле все узлы своего кластера, указав для каждого узла IP-адрес ЦОДа и имя стойки, в которой узел находится. Может показаться, что этот файл трудно поддерживать, если узлы добавляются и удаляются достаточно часто, но помните, что это лишь одна из возможностей: мы жертвуем гибкостью и простотой обслуживания ради более точного контроля и повышения производительности, поскольку Cassandra в этом случае не приходится вычислять, где находятся узлы, – вы ей об этом уже сообщили.

Сплетничающий осведомитель с файлом свойств

Еще один осведомитель, знающий о стойках, реализован в классе `org.apache.cassandra.locator.GossipingPropertyFileSnitch`. Каждый узел обменивается с другими информацией о своем ЦОДе и стойке по протоколу распространения сплетен. Сведения о местоположении ЦОДа и стойки хранятся в файле `cassandra-rackdc.properties`. Осведомитель `GossipingPropertyFileSnitch` пользуется также файлом `cassandra-topology.properties`, если таковой имеется.

Осведомитель, догадывающийся о стойках

Осведомитель `org.apache.cassandra.locator.RackInferringSnitch` предполагает, что реализована определенная схема назначения IP-адресов узлам кластера. Он просто сравнивает октеты IP-адресов узлов. Если вторые октеты адресов двух узлов совпадают, считается, что они находятся в одном ЦОДе. Если совпадают третий октеты, считается, что узлы находятся в одной стойке. Слово «считается» означает, что до-

гадки Cassandra основаны на предположениях о структуре виртуальной локальной сети или подсетей.

Облачные осведомители

В состав Cassandra включено несколько осведомителей, рассчитанных на работу в облаке.

- Осведомители `org.apache.cassandra.locator.Ec2Snitch` и `Ec2MultiRegionSnitch` предназначены для работы в облаке Amazon Elastic Compute Cloud (EC2), являющемся частью комплекса служб Amazon Web Services (AWS). `Ec2Snitch` полезен при развертывании в одном регионе AWS или в нескольких регионах, принадлежащих одной и той же виртуальной сети. `Ec2MultiRegionSnitch` рассчитан на развертывание в нескольких регионах, соединенных публичной сетью Интернет.
- Осведомитель `org.apache.cassandra.locator.GoogleCloudSnitch` можно использовать в одном или нескольких регионах на облачной платформе Google Cloud Platform.
- Осведомитель `org.apache.cassandra.locator.CloudstackSnitch` предназначен для развертывания в публичном или частном облаке Apache Cloudstack.

Осведомители для платформ EC2 и Google Cloud пользуются файлом `cassandra-rackdc.properties`, причем соглашения об именовании ЦОДов и стоек зависят от среды. Мы еще вернемся к этим осведомителям в главе 14.

Динамический осведомитель

Как было сказано в главе 6, Cassandra обертывает выбранный вами осведомитель классом `org.apache.cassandra.locator.DynamicEndpointSnitch`, чтобы найти узлы, показывающие наиболее высокую производительность выполнения запросов. Свойство `dynamic_snitch_badness_threshold` задает порог изменения предпочтительного узла. Значение по умолчанию 0,1 означает, что если предпочтительный узел работает на 10% хуже самого быстрого, то он теряет статус предпочтительного. Динамический осведомитель обновляет статус с частотой, определяемой свойством `dynamic_snitch_update_interval_in_ms`, и сбрасывает предыдущие результаты вычислений через промежуток времени, равный значению свойства `dynamic_snitch_reset_interval_in_ms`. Интервал сброса должен быть намного больше интервала обновления, поскольку это более накладная операция, зато она позволяет узлу

восстановить утраченный статус предпочтительного, не демонстрируя производительность, превышающую порог негодности.

Конфигурация узлов

Помимо относящихся к кластеру параметров, в файле *cassandra.yaml* можно настроить много других свойств. В этой главе мы рассмотрим некоторые свойства, касающиеся использования сети и диска, а обсуждение остальных отложим до глав 12 и 13.



Справочник по файлу *cassandra.yaml*

Мы рекомендуем ознакомиться с документацией DataStax для вашей версии, поскольку в ней имеется полезное руководство по настройке различных параметров в файле *cassandra.yaml*. Авторы начинают с самых употребительных параметров и постепенно переходят к более сложным.

Маркеры и виртуальные узлы

По умолчанию в Cassandra используются виртуальные узлы (*v-узлы*). Количество маркеров, обслуживаемых данным узлом, задается с помощью свойства `num_tokens`. Вообще говоря, лучше оставить значение по умолчанию (сейчас оно равно 256, однако см. примечание ниже), но можно и увеличить его, если машина достаточно мощная, чтобы справиться с большим числом маркеров, или уменьшить, если машина слабенькая.



Сколько задавать *v-узлов*?

В последнее время многие эксперты рекомендуют изменить значение `num_tokens` по умолчанию с 256 до 32. Обосновывается это тем, что 32 маркера на один узел обеспечивают приемлемый баланс между диапазонами маркеров и в то же время требуют значительно меньшей пропускной способности сети для обслуживания. Вероятно, в будущих версиях значение по умолчанию будет изменено.

Чтобы отключить *v-узлы* и вернуться к традиционным диапазонам маркеров, нужно первым делом установить `num_tokens` в 1 или вообще закомментировать это свойство. Затем нужно задать свойство `initial_token`, показывающее, каким диапазоном маркеров будет владеть узел. Для каждого узла кластера задается свой диапазон.

В версиях Cassandra до 3.0 имеется программа `token-generator`, помогающая вычислить значения свойства `initial_token` для всех уз-

лов кластера. Запустим ее, к примеру, для кластера с тремя узлами в одном ЦОДе:

```
$ cd $CASSANDRA_HOME/tools/bin  
$ ./token-generator 3  
DC #1:  
Node #1: -9223372036854775808  
Node #2: -3074457345618258603  
Node #3: 3074457345618258602
```

Для конфигураций с несколькими ЦОДами нужно при вызове указать столько целых чисел, сколько имеется ЦОДов: каждое число равно количеству узлов в соответственном ЦОДе. По умолчанию token-generator генерирует начальные маркеры для разделителя Murmur3Partitioner, но может генерировать их и для RandomPartitioner, если указать флаг `--random`. Если вам нужно сгенерировать начальные маркеры, а в дистрибутиве нет программы token-generator, то можете воспользоваться удобным калькулятором по адресу <http://www.geroba.com/cassandra/cassandra-tokencalculator>.

Вообще говоря, настоятельно рекомендуется использовать v-узлы, чтобы не иметь дополнительных проблем с вычислением маркеров и ручным изменением конфигурации, которые возникают при добавлении или удалении узлов, владеющих единственным маркером, и последующей перебалансировке кластера.

Сетевые интерфейсы

В файле *cassandra.yaml* есть несколько свойств, относящихся к номенклатуре портов и сетевым протоколам, которые используются для взаимодействия узлов с клиентами и другими узлами.

```
$ cd ~/.ccm  
$ find . -name cassandra.yaml -exec grep -H 'listen_address' {} \\;  
.node1/conf/cassandra.yaml:listen_address: 127.0.0.1  
.node2/conf/cassandra.yaml:listen_address: 127.0.0.2  
.node3/conf/cassandra.yaml:listen_address: 127.0.0.3
```

Если вы предпочитаете привязку по имени интерфейса, то можете использовать свойство `listen_interface`, а не `listen_address`, например: `listen_interface=eth0`. Задавать оба свойства нельзя.

Свойство `storage_port` задает номер порта для межузловых коммуникаций, обычно 7000. Если вы планируете использовать Cassandra в среде, включающей публичные сети или несколько регионов в случае облачного развертывания, то необходимо еще задать свой-

ство `ssl_storage_port` (обычно 7001). Для настройки защищенного порта нужно еще задать параметры межузлового шифрования, но об этом – в главе 14.

Исторически Cassandra поддерживала два разных клиентских интерфейса: оригинальный Thrift API, известный также под названием «удаленный вызов процедур» (Remote Procedure Call – RPC), и интерфейс на основе CQL, впервые добавленный в версии 0.8 и называемый также *внутренним транспортным протоколом* (native transport). До версии 2.2 включительно поддерживались и по умолчанию были включены оба интерфейса. Начиная с версии 3.0, Thrift по умолчанию выключен, и в будущей версии его поддержка прекратится.

Для включения и выключения внутреннего транспортного протокола служит свойство `start_native_transport`, по умолчанию равное `true`. Внутренний транспортный протокол работает через порт 9042, который может быть изменен с помощью свойства `native_transport_port`.

В файле `cassandra.yaml` имеется аналогичный набор свойств для настройки интерфейса RPC. По умолчанию для RPC используется порт 9160, но его можно изменить с помощью свойства `rpc_port`. Если существуют клиенты, работающие через Thrift, то этот интерфейс придется включить. Но, учитывая, что CQL в его текущем виде (CQL3) доступен начиная с версии 1.1, давно пора перевести клиентов на этот интерфейс.

Свойство `rpc_keepalive` используется в обоих интерфейсах. Подразумеваемое по умолчанию значение `true` означает, что Cassandra позволяет клиентам передавать несколько запросов по одному соединению. Прочие свойства, относящиеся к количеству потоков, подключений и размеру кадра, будут рассмотрены в главе 12.

Хранение данных

Cassandra позволяет указывать, где и как должны храниться различные файлы, в т. ч. файлы данных, журналы фиксаций и сохраненные кэши. По умолчанию это подкаталог `data` установочного каталога (`$CASSANDRA_HOME/data` или `%CASSANDRA_HOME%/data`). В старых версиях и в некоторых дистрибутивных пакетах для Linux указан каталог `/var/lib/cassandra/data`.

Напомним (см. главу 6), что *журнал фиксаций* используется для краткосрочного хранения записываемых данных. Каждое записанное значение сразу же дописывается в конец журнала фиксаций. В случае останова или неожиданного краха базы данных журнал фиксаций гарантирует, что данные не будут потеряны, поскольку при следую-

щем запуске узла журнал фиксаций накатывается. На самом деле это единственное место, где производится чтение из журнала фиксаций; клиенты его никогда не читают. Но операция записи в журнал фиксаций требует блокировки, т. е. клиенты должны ждать, пока запись завершится. Журналы фиксаций хранятся в каталоге, определяемом свойством `commitlog_directory`.

Файлы данных хранятся в виде файлов SSTable (Sorted String Tables – отсортированные таблицы строк). В отличие от записи в журнал фиксаций, запись в эти файлы производится асинхронно. Файлы SSTable периодически сливаются в процессе полного уплотнения, чтобы освободить место на диске. Для этого Cassandra объединяет столбцы с одинаковыми ключами и удаляет надгробья.

Файлы данных хранятся в каталоге, определяемом свойством `data_file_directories`. Разрешается задать несколько каталогов, тогда Cassandra будет равномерно распределять данные между ними. Так Cassandra поддерживает развертывание типа JBOD (just a bunch of disks – просто пачка дисков), когда каждый каталог представляет отдельную точку монтирования.



Места для хранения файлов в Windows

При работе в Windows необязательно изменять заданные по умолчанию пути к каталогам, потому что Windows автоматически скорректирует разделитель компонентов пути и будет считать, что каталоги находятся на диске C:\. Разумеется, в реальной системе лучше бы задать отдельные каталоги для различных файлов, как описано выше.

Для тестирования изменять подразумеваемые каталоги, может быть, и не имеет смысла. Но в производственной среде, где используются вращающиеся диски, рекомендуется размещать файлы данных и журналы фиксаций на разных дисках для повышения производительности и доступности.

Cassandra может перенести потерю одного или нескольких дисков, не останавливая узел целиком, но предоставляет возможность задать желаемое поведение узлов в случае отказа дисков. Что должно происходить при отказе диска с файлами данных, задается свойством `disk_failure_policy`, а при отказе диска с журналом фиксаций – свойством `commit_failure_policy`. По умолчанию подразумевается поведение `stop` – т. е. выключить клиентские интерфейсы, но оставить включенными интерфейс для просмотра через JMX. Имеются также варианты `die` – остановить весь узел (путем завершения JVM) – и `ignore` – про-

токолировать и игнорировать ошибки файловой системы. Поведение `ignore` не рекомендуется. Еще один вариант, `best_effort`, применяется для файлов данных и означает, что разрешены операции с файлами SSTable, которые хранятся на еще работающих дисках.

Параметры JVM и протоколирования

В этой главе мы по большей части рассматривали параметры в файле `cassandra.yaml`, но существуют и другие конфигурационные файлы.

В скриптах запуска Cassandra немало места уделено логике оптимизации различных параметров виртуальной машины Java (JVM). Эти знания были добыты тяжким трудом. Интерес представляет прежде всего скрипт настройки среды `conf/cassandra.env.sh` (или скрипт для PowerShell `conf/cassandra.env.ps1` в Windows). В нем задаются версия JVM (если в вашей системе установлено несколько версий), размер кучи и другие параметры JVM. Большинство из них приходится изменять редко, исключение составляют только параметры JMX. В этом скрипте задаются порт JMX и параметры безопасности для удаленного доступа.

В файле `conf/logback.xml` задаются параметры протоколирования: уровень, формат сообщений, местоположение файлов журналов, их максимальные размеры и параметры ротации. В Cassandra для протоколирования используется библиотека Logback, о которой можно прочитать на сайте <http://logback.qos.ch>. Переход от Log4J к Logback случился в версии 2.1.

Более подробно протоколирование и настройка JMX будут рассмотрены в главе 10, а настройка памяти JVM – в главе 12.

Добавление узлов в кластер

Теперь, понимая, как настраиваются узлы кластера Cassandra, можно перейти к вопросу о добавлении узлов. Как уже было сказано, для ручного добавления узла нужно прописать новый узел в файле `cassandra.yaml`, задав для него узлы-распространители, разделитель, осведомитель и сетевые порты. Если вы собираетесь создавать узлы с одним маркером, то нужно еще вычислить диапазон маркеров для нового узла и соответственно изменить диапазоны других узлов.

Поскольку мы используем ccm, процесс добавления узла сильно упрощается. Выполним такую команду:

```
$ ccm add node4 -i 127.0.0.4 -j 7400
```

Она создает новый узел node4 с очередным возвратным адресом и номером порта JMX, равным 7400. Чтобы узнать о дополнительных параметрах этой команды, введите `ccm add -h`. Итак, узел добавлен, теперь проверим состояние кластера:

```
$ ccm status
Cluster: 'my_cluster'
-----
node1: UP
node3: UP
node2: UP
node4: DOWN (Not initialized)
```

Новый узел появился, но он еще не запущен. Еще раз выполнив команду `nodetool ring`, мы увидим, что маркеры никак не изменились. Теперь мы готовы запустить новый узел командой `ccm node4 start` (на всякий случай еще раз проверьте, что дополнительный возвратный адрес активирован). После этого повторное выполнение команды `nodetool ring` дает такой результат:

```
Datacenter: datacenter1
=====
Address   Rack    Status   State ...   Token
                                                9218701579919475223
127.0.0.1  rack1    Up      Normal ...   -9211073930147845649
127.0.0.4  rack1    Up      Normal ...   -9190530381068170163
...
```

Сравнив с предыдущим результатом, мы заметим две вещи. Во-первых, распределение маркеров по узлам изменилось, поскольку включен новый узел. Во-вторых, изменились значения маркеров – диапазоны стали уже. Чтобы можно было выделить новому узлу 256 маркеров (`num_tokens`), нам пришлось создать для кластера 1024 маркера.

Что означает для других узлов запуск node4, можно понять, заглянув в журнал. На автономном узле это файл `system.log` в каталоге `/var/log/cassandra` (или `$CASSANDRA_HOME/logs` – в зависимости от конфигурации). Но `ccm` предлагает удобную команду, позволяющую просматривать файлы журналов с любого узла. Откроем журнал узла node1, выполнив команду `ccm node1 showlog`. Просмотр выглядит так же, как при работе со стандартной командой Unix `more`, т. е. мы можем листать файл и искать в нем. Поискав ближе к концу файла строки, относящиеся к `gossip`, найдем:

```
INFO [GossipStage:1] 2015-08-24 20:02:24,377 Gossiper.java:1005 -
  Node /127.0.0.4 is now part of the cluster
INFO [HANDSHAKE-/127.0.0.4] 2015-08-24 20:02:24,380
  OutboundTcpConnection.java:494 - Handshaking version with /127.0.0.4
INFO [SharedPool-Worker-1] 2015-08-24 20:02:24,383
  Gossiper.java:970 - InetAddress /127.0.0.4 is now UP
```

Отсюда видно, что node1 успешно посплетничал с node4 и что node4 теперь считается работающим и входящим в состав кластера. В этот момент процесс начальной загрузки начинает выделять маркер узла node4 и переправлять ему все данные, ассоциированные с этими маркерами.

Динамическое присоединение к кольцу

Узлы кластера Cassandra можно останавливать и снова запускать, не прерывая работу кластера в целом (в предположении, что коэффициент репликации и уровень согласованности заданы разумно). Предположим, что мы запустили кластер с двумя узлами, как описано выше в разделе «Создание кластера». Попробуем вызвать такую ошибку, которая «положит» один узел, а затем убедимся, что кластер по-прежнему работает.

Смоделировать такую ситуацию позволит команда `ccm node4 stop`. Проверим, что узел остановился, командой `ccm status`, а затем просмотрим журнал командой `ccm node1 showlog`. В нем обнаружатся такие строки:

```
INFO [GossipStage:1] 2015-08-27 19:31:24,196 Gossiper.java:984 -
  InetAddress /127.0.0.4 is now DOWN
INFO [HANDSHAKE-/127.0.0.4] 2015-08-27 19:31:24,745
  OutboundTcpConnection.java:494 - Handshaking version with /127.0.0.4
```

Теперь вернем node4 к жизни и снова проверим журнал на узле node1. Как и следовало ожидать, Cassandra автоматически обнаружила, что пропавший был узел вернулся в кластер и готов к работе:

```
INFO [HANDSHAKE-/127.0.0.4] 2015-08-27 19:32:56,733 OutboundTcpConnection
  .java:494 - Handshaking version with /127.0.0.4
INFO [GossipStage:1] 2015-08-27 19:32:57,574 Gossiper.java:1003 -
  Node /127.0.0.4 has restarted, now UP
INFO [SharedPool-Worker-1] 2015-08-27 19:32:57,652 Gossiper.java:970 -
  InetAddress /127.0.0.4 is now UP
INFO [GossipStage:1] 2015-08-27 19:32:58,115 StorageService.java:1886 -
  Node /127.0.0.4 state jump to normal
```

Строка «state jump to normal» для узла node4 показывает, что он снова является частью кластера. На всякий случай еще раз выполним команду status и убедимся, что узел работает:

```
$ ccm status
Cluster: 'my_cluster'
-----
node1: UP
node2: UP
node3: UP
node4: UP
```

Стратегии репликации

Мы потратили немало времени на изучение различных конфигурационных параметров кластера и узлов, но Cassandra предлагает также гибкие средства настройки пространств ключей и таблиц. Эти параметры доступны как из cqlsh, так и из клиента, пользующегося подходящим драйвером, о чем мы поговорим с главе 8.

```
cqlsh> DESCRIBE KEYSPACE my_keyspace ;
CREATE KEYSPACE my_keyspace WITH replication =
  {'class': 'SimpleStrategy',
   'replication_factor': '1'} AND durable_writes = true;
```



Что такое долговечная запись?

Свойство `durable_writes` позволяет обойти запись в журнал фиксаций для указанного пространства ключей. По умолчанию оно равно `true`, т. е. запись в журнал производится при любой модификации. Установка его в `false` повышает скорость записи, но ценой потери данных в случае, если узел «грохнется», не успев сбросить содержимое таблиц в памяти в файлы SSTable.

Правильный выбор стратегии репликации важен, потому что она определяет, какие узлы за какие диапазоны ключей отвечают. Отсюда вытекает, что одновременно мы определяем, каким узлам будут направляться те или иные операции записи, а это может оказать существенное влияние на эффективность. Если кластер настроен так, что все операции записи направляются двум ЦОДам, один в Австралии, а другой в Рестоне, штат Вирджиния, то падение производительности будет заметно. Возможность выбора взаимозаменяемых стратегий увеличивает гибкость, так как позволяет настроить Cassandra в соответствии с топологией сети и вашими потребностями.

Первая реплика всегда размещается на узле, владеющем диапазоном, в который попадает маркер, а остальные – в узлах, определяемых стратегией репликации.

Как мы помним из главы 6, Cassandra предоставляет две стратегии репликации: SimpleStrategy и NetworkTopologyStrategy.

Стратегия SimpleStrategy

Эта стратегия размещает реплики в пределах одного ЦОДа, ничего не зная о том, в каких стойках находятся узлы. Следовательно, теоретически реализация быстрая, но не в том случае, когда следующий узел, владеющий данным ключом, окажется не в той же стойке, что все остальные. Это иллюстрируется на рис. 7.1.

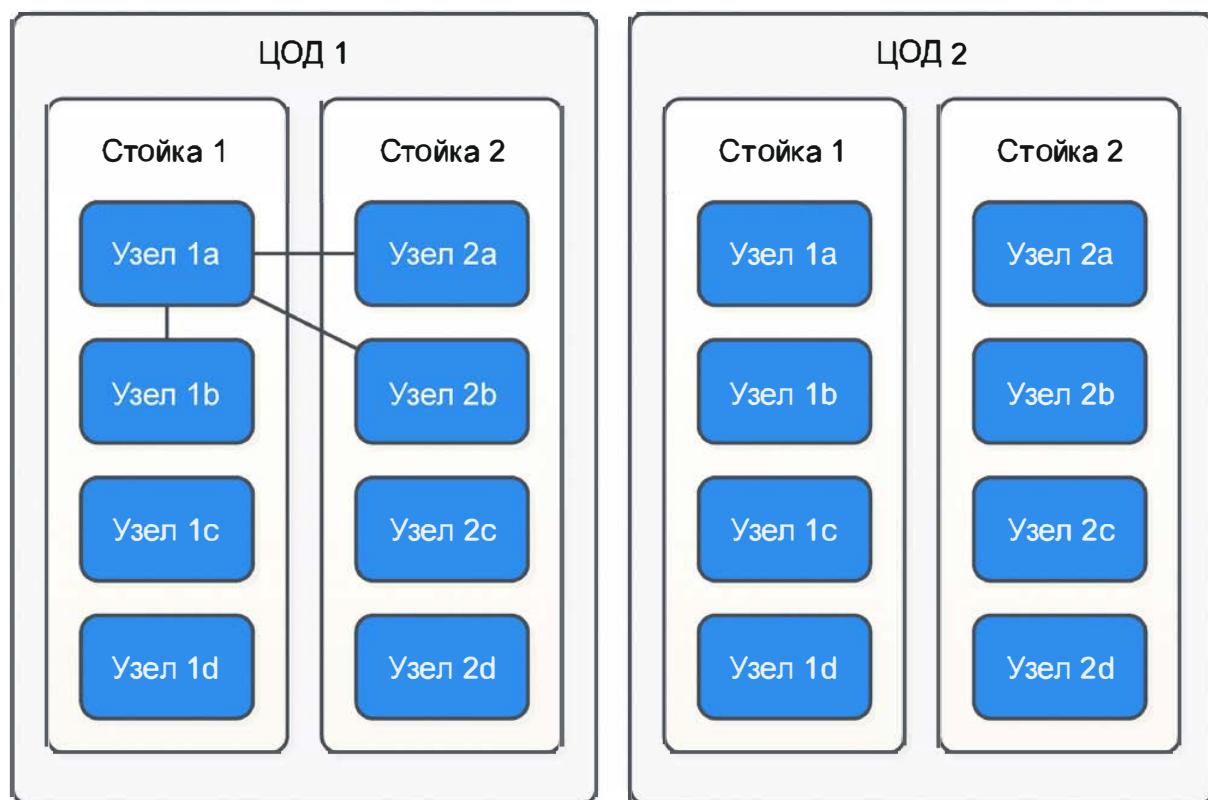


Рис. 7.1 ♦ Стратегия SimpleStrategy размещает реплики в пределах одного ЦОДа, не обращая внимания на топологию

Здесь для хранения реплик выбираются следующие N узлов кольца, и стратегия ничего не знает о центрах обработки данных. Второй ЦОД показан, чтобы наглядно продемонстрировать, что стратегия о нем не знает.

Стратегия NetworkTopologyStrategy

Теперь предположим, что мы хотим распределить реплики по нескольким ЦОДам на случай, если один полностью выйдет из строя из-за катастрофы или окажется недоступен по сети. Стратегия NetworkTopologyStrategy позволяет указать, что часть реплик должна размещаться в ЦОД1, а часть – в ЦОД2. В каждом ЦОДе реплики распределяются между разными стойками, поскольку узлы, находящиеся в одной стойке (или физически сгруппированные похожим образом), часто выходят из строя одновременно – из-за проблем с электропитанием, охлаждением или сетью.

Стратегия NetworkTopologyStrategy распределяет реплики следующим образом: первая реплика размещается в соответствии с указаниями выбранного разделителя, а последующие – путем обхода узлов кольца с пропуском узлов в той же стойке, что и предыдущий. После того как все стойки задействованы, для размещения реплик можно использовать ранее пропущенные узлы – пока не будет размещено столько реплик, сколько требуется в соответствии с коэффициентом репликации.

Стратегия NetworkTopologyStrategy позволяет задавать различные коэффициенты репликации для разных ЦОДов. Следовательно, общее число реплик равно сумме коэффициентов репликации для всех ЦОДов. Результат работы этой стратегии показан на рис. 7.2.



Другие стратегии репликации

Внимательный читатель обратит внимание, что в составе Cassandra есть еще две стратегии репликации: OldNetworkTopologyStrategy и LocalStrategy.

OldNetworkTopologyStrategy напоминает NetworkTopologyStrategy тем, что размещает реплики в нескольких ЦОДах, но ее алгоритм проще. Она размещает вторую реплику не в том ЦОДе, что первую, третью – в другой стойке первого ЦОДа, а остальные – путем перебора последующих узлов кольца.

LocalStrategy зарезервирована для внутренних нужд Cassandra. Как следует из названия, эта стратегия размещает данные только в локальном узле, не реплицируя их на другие узлы. Cassandra использует ее для системных пространств ключей, в которых хранятся метаданные о локальном и других узлах кластера.

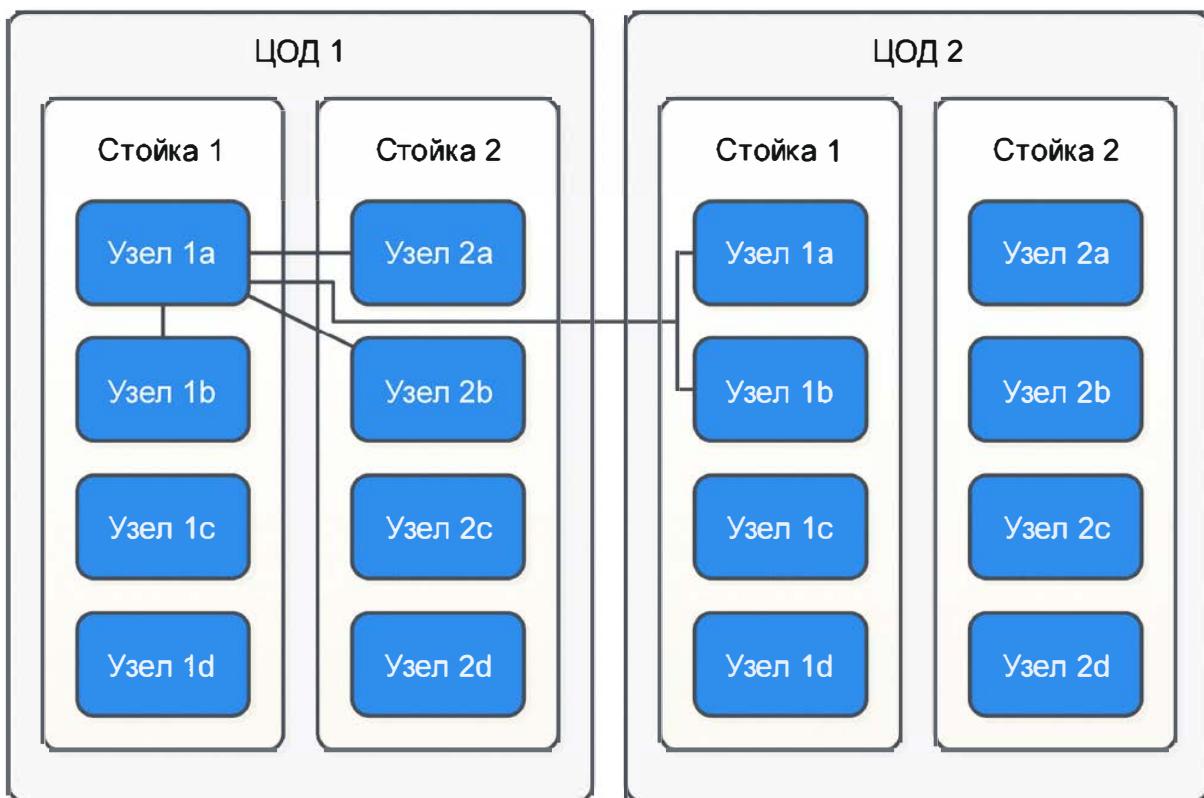


Рис. 7.2 ♦ Стратегия NetworkTopologyStrategy размещает реплики в нескольких ЦОДах в соответствии с коэффициентами репликации, заданными для каждого ЦОДа

Изменение коэффициента репликации

Изменить коэффициент репликации существующего пространства ключей можно с помощью cqlsh или другого клиента. Чтобы изменение возымело эффект, необходимо выполнить команду nodetool на каждом узле, который оно затрагивает. Если коэффициент репликации увеличивается для кластера (или ЦОДа), то выполните команду nodetool repair на каждом узле кластера (или ЦОДа), чтобы Cassandra сгенерировала дополнительные реплики. Пока исправление не завершится, клиенты могут получать уведомление о том, что данных не существует, если попытаются подключиться к реплике, еще не наполненной данными.

Если коэффициент репликации уменьшается для кластера (или ЦОДа), то выполните команду nodetool clean на каждом узле кластера (или ЦОДа), чтобы Cassandra освободила место, занятое уже ненужными репликами. О командах repair, clean и других речь пойдет в главе 11.

Можно дать общую рекомендацию: пропускная способность пропорциональна числу узлов, поделенному на коэффициент репликации. Так, если в кластере с 10 узлами при коэффициенте репликации 1 она составляет 10 000 операций записи в секунду, то при коэффициенте репликации 2 можно ожидать снижения до 5000 операций записи в секунду.

Резюме

В этой главе мы рассмотрели создание кластеров Cassandra и добавление в них узлов. Мы узнали, как с помощью файла *cassandra.yaml* настроить параметры узлов Cassandra, в том числе узлы-распространители, разделитель, осведомитель и др. Мы также научились настраивать репликацию пространства ключей и выбирать подходящую стратегию репликации.

Глава 8

Клиенты

Мы привыкли, что для подключения к реляционной базе данных нужны драйверы. Так, в Java стандарт JDBC представляет собой API, абстрагирующий конкретную реализацию базы данных и предлагающий единый способ сохранения и извлечения данных с помощью объектов Statement, PreparedStatement, ResultSet и т. д. Для работы с конкретной базой данных, например Oracle, SQL Server или MySQL, необходим драйвер для нее; детали реализации взаимодействия от разработчика скрыты. Существуют драйверы, позволяющие обращаться к различным базам данных из программ на разных языках программирования.

Для Cassandra тоже существует целый ряд клиентских драйверов для большинства популярных языков. Преимуществом таких клиентов является то, что их можно легко встроить в собственное приложение (мы увидим, как это делается) и что они зачастую предлагают больше возможностей, чем интерфейс CQL, в том числе организацию пула соединений, интеграцию с JMX и мониторинг. В следующих разделах мы узнаем о том, какие есть клиенты и что они предлагают.

Hector, Astyanax и другие устаревшие клиенты

На заре существования Cassandra сообществом было написано много клиентских драйверов для разных языков. Именно благодаря им Cassandra и добилась признания. Из самых известных ранних драйверов стоит отметить Hector и Astyanax.

Названный в честь брата Кассандры, троянского героя Гектора, Hector стал одним из самых первых клиентов Cassandra. Он предлагал простой интерфейс на Java, который помог многим ранним разра-

ботчикам избежать сложностей, связанных с использованием Thrift API, и послужил источником вдохновения для нескольких других драйверов. Сейчас проект уже не развивается, но доступен по адресу <https://github.com/hector-client/hector>.

Astyanaх – это Java-клиент, первоначально написанный компанией Netflix поверх Thrift API как логическое продолжение драйвера Hector (Астианаксом звали сына Гектора). После выпуска компанией DataStax драйвера для Java Netflix адаптировала Astyanax, включив поддержку Java-драйвера в дополнение к оригинальной реализации на основе Thrift. Это помогло многим пользователям перейти от Thrift к CQL. Но по мере того, как Java-драйвер набирал популярность, развитие Astyanax сильно замедлилось, и в феврале 2016 года проект был прекращен. Однако он все еще доступен по адресу <https://github.com/Netflix/astyanax>.

Среди других клиентов упомянем Russassa для Python, Perlcassa для Perl, Helenus для Node.js и Cassandra-Sharp для платформы Microsoft .NET и языка C#. Большинство этих клиентов сейчас не сопровождается, так как основано на интерфейсе Thrift, который объявлен нерекомендуемым. Полный перечень актуальных и устаревших драйверов можно найти по адресу <http://www.planetcassandra.org/client-drivers-tools>.

Драйвер DataStax для Java

Появление CQL послужило толчком к кардинальному изменению состояния дел с клиентскими драйверами для Cassandra. Простота и знакомый синтаксис CQL сделали разработку клиентских программ очень похожей на создание приложений для традиционных реляционных баз данных. DataStax пошла на стратегическое капиталовложение, раскрыв исходный код драйверов для Java и ряда других языков, чтобы подкинуть дров в огонь распространения Cassandra. Эти драйверы стали стандартом де facto для новых проектов разработки. Драйверы, а также дополнительные соединители и инструменты можно скачать с сайта <https://github.com/datastax>.



Дополнительная информация о драйверах DataStax

На странице матрицы драйверов (<http://docs.datastax.com/en/developer/developer-matrix/doc/common/developerMatrix.html>) имеются ссылки на документацию и указаны версии драйверов, совместимые с версией вашего сервера.

Java-драйвер DataStax – самый старый и самый зрелый из всех. Поэтому мы сосредоточимся на работе с ним и воспользуемся этой возможностью для изучения возможностей, включенных в драйверы DataStax для разных языков.

Настройка среды разработки

Прежде всего необходим доступ к драйверу из среды разработки. Можно было бы скачать драйвер прямо с указанной выше страницы и управлять зависимостями самостоятельно, но современные методики разработки на Java предполагают использование какого-нибудь инструмента для этой цели, например Maven. Для работы с Maven необходимо добавить в файл *pom.xml* для проекта примерно такие строки:

```
<dependency>
    <groupId>com.datastax.cassandra</groupId>
    <artifactId>cassandra-driver-core</artifactId>
    <version>3.0.0</version>
</dependency>
```

Найти документацию по Java-драйверу в формате Javadoc можно на странице <http://docs.datastax.com/en/drivers/java/3.0/index.html> или в исходном дистрибутиве.

Все драйверы DataStax управляются через GitHub как проекты с открытым исходным кодом. Если вам интересен исходный код драйвера, можете скачать к себе столовую версию, доступную только для чтения, командой

```
$ git clone https://github.com/datastax/java-driver.git
```

Кластеры и точки контакта

Настроив среду, можно приступить к кодированию. Создадим клиентское приложение на основе модели данных отеля, построенной в главе 5. Весь исходный код, встречающийся в этой и других главах, доступен на странице <https://github.com/jeffreyscarpenter/cassandra-guide>.

Прежде всего воспользуемся API драйвера для подключения к кластеру. В Java-драйвере для этого служат классы `com.datastax.driver.core.Cluster` и `Session`.

Класс `Cluster` – главная точка входа в драйвер. Он поддерживает текущий API, продиктованный паттерном «Построитель». Так, в следующих двух строках создается подключение к узлу, работающему на локальном хосте:

```
Cluster cluster = Cluster.builder().  
addContactPoint("127.0.0.1").build();
```

В этом предложении задана минимально необходимая информация для создания кластера: одна точка контакта и ничего больше. Можно было бы задать несколько точек контакта. По существу, точки контакта похожи на узлы-распространители, с помощью которых Cassandra подключается к другим узлам в том же кластере.



Создание специального инициализатора кластера

Класс Cluster.Builder реализует интерфейс Cluster.Initializer, что позволяет подменить его каким-нибудь другим механизмом инициализации кластера, воспользовавшись статическим методом Cluster.buildFrom(Initializer initializer). Это может быть полезно, например, если нам понадобится загружать информацию о подключении из конфигурационного файла.

Можно задать еще несколько параметров объекта Cluster, например метрики, параметры запросов по умолчанию, политики повторного подключения, повторения попыток и упреждающего выполнения. Все это мы рассмотрим ниже, но сначала поговорим о других связанных с подключением вещах: версии протокола, сжатии и аутентификации.

Версия протокола

Драйвер поддерживает несколько версий внутреннего протокола CQL. Версия 3.0 поддерживает протокол версии 4, мы упоминали об этом в историческом обзоре в главе 2.

По умолчанию драйвер использует версию протокола, поддерживаемую первым узлом, к которому он подключился. В большинстве случаев этого достаточно, но если вы работаете с кластером, основанном на более ранней версии Cassandra, то такое поведение нужно переопределить. Чтобы задать версию протокола, передайте подходящее значение перечисления com.datastax.driver.core.ProtocolVersion методу Cluster.Builder.withProtocolVersion().

Сжатие

Драйвер позволяет сжимать сообщения, передаваемые между клиентом и узлами Cassandra, пользуясь средствами сжатия, определенными во внутреннем протоколе CQL. Сжатие снижает потребление драйвером сетевых ресурсов ценой повышенного потребления ресурсов процессора как клиентом, так и сервером.

В настоящее время поддерживаются два алгоритма сжатия: LZ4 и SNAPPY, они определены перечислением `com.datastax.driver.core.ProtocolOptions.Compression`. По умолчанию подразумевается режим `NONE`, но его можно переопределить, вызвав метод `Cluster.Builder.withCompression()`.

Аутентификация и шифрование

Драйвер предоставляет сменный механизм аутентификации, допускающий как простой вход по имени и паролю, так и интеграцию с другими системами аутентификации. По умолчанию не производится никакой аутентификации. Для выбора поставщика аутентификации нужно передать методу `Cluster.Builder.withAuthProvider()` какую-нибудь реализацию интерфейса `com.datastax.driver.core.AuthProvider`, например `PlainTextAuthProvider`.

Драйвер умеет также шифровать весь трафик с сервером. Параметры шифрования задаются на каждом узле в файле `cassandra.yaml`. Поведение драйвера согласовано с этими параметрами.

Подробнее аутентификация, авторизация и шифрование с точки зрения клиента и сервера обсуждаются в главе 13.

Сеансы и пулы соединений

Сразу после создания экземпляр класса `Cluster` не подключен ни к одному узлу `Cassandra`. Для подключения нужно вызвать метод `init()`:

```
cluster.init();
```

При вызове этого метода драйвер подключается к одной из сконфигурированных точек контакта для получения метаданных о кластере. Если все точки контакта недоступны, то возбуждается исключение `NoHostAvailableException`, а в случае ошибки аутентификации – исключение `AuthenticationException`. Подробнее об аутентификации мы расскажем в главе 13.

После инициализации объекта `Cluster` необходимо открыть сеанс для отправки запросов. Получить объект `com.datastax.driver.core.Session` можно, вызвав один из набора перегруженных методов `Cluster.connect()`. Этому методу можно передать имя пространства ключей, к которому необходимо подключиться, например:

```
Session session = cluster.connect("hotel");
```

Метод `connect()` без параметров создает объект `Session`, который можно использовать для работы с несколькими пространствами ключей.

чей. В таком случае любое имя таблицы в запросе должно быть квалифицировано именем соответствующего пространства ключей. Отметим, что вызывать метод `Cluster.init()`, строго говоря, не обязательно, потому что он автоматически вызывается при вызове `connect()`.

Объект `Session` управляет подключениями к кластеру `Cassandra`, используемыми для выполнения запросов и управляющих операций по внутреннему протоколу. В сеансе хранится пул TCP-соединений с хостами.



Сеансы обходятся дорого

Поскольку в объекте сеанса хранится пул соединений с несколькими узлами, он получается довольно тяжеловесным. В большинстве случаев достаточно создать один объект `Session` на все приложение, а не создавать и уничтожать сеансы многократно. Другой вариант – создавать по одному `Session` на каждое пространство ключей, если приложение работает с несколькими пространствами ключей.

Поскольку внутренний протокол CQL асинхронный, он допускает передачу одновременно нескольких запросов по одному соединению, в версии 2 их может быть не более 128, а в версиях 3 и 4 – до 32 768. Благодаря этому требуется меньше соединений с каждым узлом. На самом деле, по умолчанию создается всего одно соединение на каждый узел.

Драйвер поддерживает возможность автоматического увеличения или уменьшения количества соединений в зависимости от числа запросов на соединение. Параметры пула соединений настраиваются с помощью класса `PoolingOptions`, который позволяет задать минимальное (базовое) и максимальное число соединений для локального и удаленных хостов. Если базовое и максимальное значения различаются, то драйвер динамически регулирует размер пула соединений для каждого узла в зависимости от количества отправленных клиентом запросов. Минимальное и максимальное число одновременных запросов, а также условие, при котором пристаивающие соединения можно удалить, задаются при создании соединения. Существует также буферный период для предотвращения постоянного создания и уничтожения соединения.

Объект `PoolingOptions` можно настроить на этапе создания объекта `Cluster` методом `ClusterBuilder.withPoolingOptions()` или после его создания методом `Cluster.getConfiguration().getPoolingOptions()`. Ниже приведен пример создания объекта `Cluster`, в котором максимальное число соединений с удаленными узлами равно 1:

```
PoolingOptions poolingOptions = new PoolingOptions().
    setMaxConnectionsPerHost(HostDistance.REMOTE, 1);

Cluster cluster = Cluster.builder().
    addContactPoint("127.0.0.1").
    withPoolingOptions(poolingOptions).build();
```

Драйвер осуществляет периодическую отправку контрольных сообщений (пульсов), чтобы предотвратить преждевременное закрытие соединений промежуточными сетевыми устройствами. По умолчанию период равен 30 с, но это значение можно переопределить методом `PoolingOptions.setHeartbeatIntervalSeconds()`. Однако новое значение распространяется только на соединения, созданные после его установки, поэтому лучше задавать период при создании объекта `Cluster`.

Объекты `Statement`

До сих пор мы рассматривали только настройку подключения к кластеру, но еще не выполняли операций чтения и записи. Чтобы приложение сделало что-то полезное, нужно создать и выполнить команды – объекты подклассов класса `com.datastax.driver.core.Statement`. `Statement` – это абстрактный класс, имеющий несколько реализаций: `SimpleStatement`, `PreparedStatement`, `BoundStatement`, `BatchStatement` и `BuiltStatement`. Самый простой способ создать и выполнить команду – обратиться к методу `Session.execute()`, передав ему строку запроса. Команда, показанная в следующем примере, возвращает все содержимое таблицы `hotels`:

```
session.execute("SELECT * from hotel.hotels");
```

В этой строке кода мы сразу создаем и выполняем запрос. На практике, если база данных велика, такая операция может оказаться очень накладной, но это полезный пример очень простого запроса. Большинство запросов сложнее, поскольку нужно задать критерий поиска или вставить конкретные записи. Конечно, для построения запроса можно было использовать различные имеющиеся в Java средства работы со строками, но это чревато ошибками. А если подставлять в запрос строки, полученные от пользователей, без тщательного контроля, то приложение может оказаться уязвимым для атак внедрением кода.

Класс `SimpleStatement`

К счастью, таких титанических усилий от нас никто не требует. Java-драйвер предоставляет класс `SimpleStatement`, помогающий

конструировать параметризованные команды. А показанный выше метод `execute()` – это просто вспомогательная операция для создания объекта `SimpleStatement`.

Давайте построим запрос, попросив объект `Session` создать `SimpleStatement`. Следующая команда вставляет строку в таблицу `hotels`:

```
SimpleStatement hotelInsert = session.newSimpleStatement(  
    "INSERT INTO hotels (hotel_id, name, phone) VALUES (?, ?, ?)",  
    "AZ123", "Super Hotel at WestWorld", "1-888-999-9999");  
session.execute(hotelInsert);
```

Первый параметр – это заготовка строки запроса, в которой указаны имя таблицы и интересующие нас столбцы. Вопросительные знаки обозначают места, в которые нужно будет подставить следующие параметры. Эти параметры – идентификатор, название и телефон отеля – заданы с помощью строк.

Если команда вставки создана правильно, то метод `execute()` отработает успешно (и ничего не вернет). Теперь создадим команду, которая прочитает только что вставленную строку:

```
SimpleStatement hotelSelect = session.newSimpleStatement(  
    "SELECT * FROM hotels WHERE id=?", "AZ123");  
ResultSet hotelSelectResult = session.execute(hotelSelect);
```

И снова мы пользуемся параметризацией для задания искомого идентификатора. Но на этот раз запоминаем объект `ResultSet`, возвращенный методом `execute()`. Далее можно перебрать строки в полученном объекте `ResultSet`:

```
for (Row row : hotelSelectResult) {  
    System.out.format("hotel_id: %s, name: %s, phone: %s\n",  
        row.getString("hotel_id"), row.getString("name"), row.getString("phone"));  
}
```

Здесь мы получаем от метода `ResultSet.iterator()` объект `Iterator`, который позволяет в цикле перебрать все строки результирующего набора, и печатаем значения интересующих нас столбцов. Обратите внимание, что для получения значения столбца используется аксессор, зависящий от его типа, – в данном случае `Row.getString()`. Как и следовало ожидать, печатается такой результат:

```
hotel_id: AZ123, name: Super Hotel at WestWorld, phone: 1-888-999-9999
```

Пользовательские кодеки

Как уже отмечалось, при работе со строками, хранящимися в результирующем наборе, необходимо знать типы столбцов. Попытка запросить столбец `id` методом `Row.getString()` привела бы к исключению `CodecNotFoundException`, означающему, что драйвер не знает, как преобразовать тип CQL `uuid` в тип `java.lang.String`.

На самом деле драйвер хранит подразумеваемый по умолчанию список отображений между типами Java и CQL, который называется **кодеком** и используется для преобразования данных между приложением и Cassandra. Драйвер позволяет добавлять новые отображения путем создания класса, расширяющего `com.datastax.driver.core.TypeCodec<T>`, и регистрации его в объекте `CodecRegistry`, которым управляет `Cluster`:

```
cluster.getConfiguration().getCodecRegistry().register(myCustomCodec)
```

Механизм пользовательских кодеков очень гибкий, в доказательство чего приведем несколько примеров:

- отображение на альтернативные форматы даты и времени (например, классы Joda для пользователей, еще не перешедших на Java 8);
- преобразование между строками и форматами XML или JSON;
- отображение списков, множеств и словарей на различные типы коллекций Java.

Примеры работы с классом `SimpleStatement` можно найти в исходном коде класса `com.cassandraguide.clients.SimpleStatementExample`.

Асинхронное выполнение

Метод `Session.execute()` синхронный, т. е. блокирует выполнение программы до получения результата или возникновения ошибки, например сетевого тайм-аута. Драйвер предоставляет также асинхронный метод `executeAsync()`, поддерживающий неблокирующий доступ к Cassandra. Это упрощает параллельную отправку нескольких запросов с целью повысить скорость работы клиентского приложения.

Превратим рассмотренную выше операцию в асинхронную:

```
ResultSetFuture result = session.executeAsync(statement);
```

В результате возвращается объект класса `ResultSetFuture`, реализующего интерфейс `java.util.concurrent.Future`. `Future` – это универсальный тип Java, который служит для хранения результата асинхронной операции. У объекта типа `Future` можно спросить, завершилась ли операция, и если да, то запросить ее результат в виде объекта па-

метрического типа. Существуют также блокирующие методы `wait()`, позволяющие дождаться результата. Объект `Future` можно отменить, если вызывающую сторону результат операции уже не интересует. Класс `Future` – полезное средство реализации паттернов асинхронного программирования, но при работе с ним приходится либо блокировать выполнение, либо заниматься периодическим опросом.

Для решения этой проблемы Java-драйвер пользуется интерфейсом `ListenableFuture` из библиотеки Google Guava. Этот интерфейс расширяет `Future`, добавляя операцию `addListener()`, которая позволяет клиенту зарегистрировать метод, который должен быть вызван, когда `Future` получит результат. Этот метод вызывается в потоке самого драйвера, поэтому очень важно, чтобы он завершался быстро, не захватывая надолго ресурсы драйвера. Класс `ResultSetFuture` параметризован типом `ResultSet`.



Другие асинхронные операции

Помимо `Session.executeAsync()`, драйвер поддерживает еще несколько асинхронных операций, в т. ч. `Cluster.closeAsync()`, `Session.prepareAsync()` и ряд операций класса преобразователя объектов.

Класс `PreparedStatement`

Класс `SimpleStatement` очень полезен для создания одноразовых запросов, но в большинстве приложений обычно многократно выполняются одни и те же запросы с разными параметрами. Для эффективной работы в такой ситуации предназначен класс `PreparedStatement`. Узлам один раз отправляется заготовка команды для подготовки, а в ответ драйвер получает описатель команды. В дальнейшем для использования подготовленной команды нужно будет послать только этот описатель и параметры.

В приложении класс `PreparedStatement`, как правило, применяется для чтения данных в соответствии с видами доступа, выявленными при проектировании модели данных, а также для записи данных в таблицы, поддерживающие эти виды доступа.

Ниже показано, как с помощью метода `Session.prepare()` создать объекты `PreparedStatement` для тех же запросов, что и раньше:

```
PreparedStatement hotelInsertPrepared = session.prepare(  
    "INSERT INTO hotels (hotel_id, name, phone) VALUES (?, ?, ?)");  
  
PreparedStatement hotelSelectPrepared = session.prepare(  
    "SELECT * FROM hotels WHERE hotel_id=?");
```

Отметим, что в PreparedStatement используется такой же синтаксис параметров, что и в SimpleStatement. Главное же различие состоит в том, что PreparedStatement не является подтипов Statement. Поэтому невозможно по ошибке передать несвязанный объект типа PreparedStatement сеансу для выполнения.

Но прежде чем углубляться в эту тему, вернемся на шаг назад и рассмотрим, что происходит за кулисами при выполнении метода Session.prepare(). Драйвер передает заготовку запроса, указанную в качестве параметра, узлу Cassandra и получает в ответ уникальный идентификатор команды. Этот идентификатор используется при создании объекта BoundStatement. Если интересно, приложение может получить идентификатор, вызвав метод PreparedStatement.getPreparedID().

Можно считать PreparedStatement шаблоном для создания запросов. Помимо самой формы запроса, мы можем задать и другие атрибуты объекта PreparedStatement, которые будут использоваться по умолчанию в создаваемых командах, например: уровень согласованности, политика повторения и трассировка.

Класс PreparedStatement повышает не только эффективность, но и безопасность, поскольку отделяет логику запроса CQL от данных. Это защищает от атак внедрением кода, идея которых заключается в том, чтобы вставить команды в поля данных и тем самым получить несанкционированный доступ.

Класс BoundStatement

Итак, мы имеем подготовленную команду, которую можем использовать для создания запросов. Чтобы воспользоваться объектом PreparedStatement, с ним нужно связать фактические значения параметров, вызвав метод bind(). Вот, например, как связать созданную ранее команду SELECT:

```
BoundStatement hotelSelectBound = hotelSelectPrepared.bind("AZ123");
```

Метод bind() сопоставляет значение с каждым вопросительным знаком в PreparedStatement. Можно связать первые *n* значений, но тогда оставшиеся нужно будет связывать отдельно перед выполнением команды. Существует также вариант bind(), в этом случае все параметры нужно связывать отдельно. В классе BoundStatement есть несколько методов set(), позволяющих связывать параметры со значениями разных типов. Например, название и телефон в подготовленной ранее команде INSERT можно связать, вызвав метод setString():

```
BoundStatement hotelInsertBound = hotelInsertPrepared.bind("AZ123");
hotelInsertBound.setString("name", "Super Hotel at WestWorld");
hotelInsertBound.setString("phone", "1-888-999-9999");
```

Связав все параметры, мы исполняем связанную команду методом `Session.execute()`. Параметры, оставшиеся несвязанными, сервер проигнорирует, если используется протокол версии 4 (Cassandra 3.0 или более поздняя версия). Для предыдущих версий протокола драйвер в такой ситуации возбуждает исключение `IllegalStateException`.

Примеры работы с `PreparedStatement` и `BoundStatement` имеются в исходном коде класса `com.cassandraguide.clients.PreparedStatementExample`.

Классы BuiltStatement и QueryBuilder

Драйвер предоставляет также класс `com.datastax.driver.core.querybuilder.QueryBuilder` с текущим API для построения запросов. Он удобен, когда структура запросов не постоянна (например, имеются необязательные параметры), вследствие чего работать с подготовленными командами затруднительно. Как и `PreparedStatement`, этот класс обеспечивает защиту от атак внедрением кода.

Для создания объекта `QueryBuilder` воспользуемся простым конструктором, принимающим объект `Cluster`:

```
QueryBuilder queryBuilder = new QueryBuilder(cluster);
```

`QueryBuilder` порождает запросы, представленные классом `BuiltStatement` и его подклассами. Методы каждого такого класса возвращают экземпляры `BuiltStatement`, соответствующие уже построенной части запроса. Среда IDE наверняка поможет узнать, какие операции допустимы на каждой стадии построения запроса.

Повторим построение рассмотренных выше запросов с помощью `QueryBuilder`, чтобы понять, как он работает. Сначала строим CQL-команду `INSERT`:

```
BuiltStatement hotelInsertBuilt =
queryBuilder.insertInto("hotels")
.value("hotel_id", "AZ123")
.value("name", "Super Hotel at WestWorld")
.value("phone", "1-888-999-9999");
```

На первом шаге вызывается метод `QueryBuilder.insertInto()`, который создает команду `Insert` для вставки в таблицу `hotels`. При желании можно было бы затем добавить в команду фразу `USING` методом

`Insert.using()`, но мы просто начинаем задавать значения параметров. Последовательные вызовы метода `Insert.value()` возвращают команды `Insert` со связанными параметрами. Конечную команду `Insert` можно выполнить, как обычно, вызвав метод `Session.execute()` или `executeAsync()`.

Команда CQL `SELECT` строится аналогично:

```
BuiltStatement hotelSelectBuilt = queryBuilder.select()
    .all()
    .from("hotels")
    .where(eq("hotel_id", "AZ123"));
```

В этом случае мы начинаем с вызова метода `QueryBuilder.select()`, который создает команду `Select`. Метод `Select.all()` означает, что будут выбраны все столбцы, но можно было бы воспользоваться методом `column()` для выборки конкретных столбцов. Фраза CQL `WHERE` добавляется методом `Select.where()`, который принимает объект класса `Clause`. Такие объекты создаются статическими методами класса `QueryBuilder`. В данном случае мы используем метод `eq()`, чтобы сравнить столбец с указанным идентификатором.

Для доступа к этим статическим методам необходимо включить в исходный код на Java дополнительные предложения импорта, например:

```
import static com.datastax.driver.core.querybuilder.QueryBuilder.eq;
```

Полный код примера использования `QueryBuilder` и `BuiltStatement` можно найти в классе `com.cassandraguide.clients.QueryBuilderExample`.

Преобразователь объектов

Мы рассмотрели несколько способов создания и выполнения запросов с помощью драйвера. Но есть еще один способ, находящийся на несколько более высоком уровне абстракции. Java-драйвер предоставляет преобразователь объектов, позволяющий сконцентрироваться на разработке и взаимодействии с моделями данных (или типами данных, определенными в API). Преобразователь основывается на аннотациях в исходном коде, которые используются для преобразования между классами Java и таблицами или пользовательскими типами (UDT).

API преобразования объектов содержится в отдельной библиотеке в файле `cassandra-driver-mapping.jar`, поэтому для работы с ним нужно включить дополнительную зависимость в конфигурационный файл Maven:

```
<dependency>
  <groupId>com.datastax.cassandra</groupId>
  <artifactId>cassandra-driver-mapping</artifactId>
  <version>3.0.0</version>
</dependency>
```

Давайте, например, создадим и аннотируем класс модели предметной области Hotel, соответствующий таблице hotels:

```
import com.datastax.driver.mapping.annotations.Column;
import com.datastax.driver.mapping.annotations.PartitionKey;
import com.datastax.driver.mapping.annotations.Table;

@Table(keyspace = "hotel", name = "hotels")
public class Hotel {

    @PartitionKey
    private String id;

    @Column (name = "name")
    private String name;

    @Column (name = "phone")
    private String phone;

    @Column (name = "address")
    private String address;

    @Column (name = "pois")
    private Set<String> pointsOfInterest;

    // конструкторы, методы get/set, вычисление хэш-кода, метод equals
}
```

Теперь создадим объект класса com.datastax.driver.mapping.MappingManager, присоединим его к сеансу и получим от него объект-преобразователь Mapper для аннотированного класса:

```
MappingManager mappingManager = new MappingManager(session);
Mapper<Hotel> hotelMapper = MappingManager.mapper(Hotel.class);
```

Предположим, что в классе Hotel определен простой конструктор, который принимает только UUID, название и номер телефона, и воспользуемся им для создания отеля, который сможем сохранить с помощью преобразователя объектов:

```
Hotel hotel = new Hotel("AZ123", "Super Hotel at WestWorld",
    "1-888-999-9999");
hotelMapper.save(hotel);
```

Метод Mapper.save() – все, что необходимо для выполнения команды CQL INSERT или UPDATE, поскольку в Cassandra это на самом деле одна и та же операция. Объект Mapper строит и выполняет команду за нас.

Для выборки объекта служит метод Mapper.get(), которому передается список аргументов, соответствующих элементам ключа раздела:

```
Hotel retrievedHotel = hotelMapper.get(hotelId);
```

Синтаксис удаления объекта аналогичен:

```
hotelMapper.delete(hotelId);
```

Как и save(), методы get() и delete() берут на себя все детали выполнения команд с помощью драйвера. Существуют также методы saveAsync(), getAsync() и deleteAsync(), поддерживающие асинхронное выполнение запросов с помощью рассмотренного выше интерфейса ListenableFuture.

Если вам нужно дополнительно настроить запросы перед выполнением, то в классе Mapper имеются методы, возвращающие объекты Statement: saveQuery(), getQuery() и deleteQuery().

Преобразователь объектов – полезное средство абстрагирования деталей взаимодействия с драйвером, особенно если уже имеется модель предметной области. Если модель содержит классы, ссылающиеся на другие классы, то класс, на который ведет ссылка, можно аннотировать как пользовательский тип с помощью аннотации @UDT. Преобразователь обрабатывает объекты рекурсивно, ориентируясь на аннотированные типы.



Achilles: более развитый преобразователь объектов

ДуйХайДоан (DuyHai Doan) разработал более развитый преобразователь объектов для Java под названием Achilles. Он дополнитель но поддерживает отображение составных ключей, облегченные транзакции, пользовательские функции и многое другое. Скачать его можно по адресу <https://github.com/doanduyhai/Achilles>.

ПОЛИТИКИ

Java-драйвер предлагает несколько интерфейсов политик, которые можно использовать для настройки поведения драйвера, а именно: политики балансировки нагрузки, повторения попыток выполнения запросов и управления подключениями к узлам кластера.

Политика балансировки нагрузки

В главе 6 мы узнали, что запрос можно адресовать любому узлу кластера, который становится координатором данного запроса. В за-

висимости от характера запроса координатор может обращаться к другим узлам для его удовлетворения. Если бы клиент адресовал все свои запросы одному и тому же узлу, то нагрузка на кластер оказалась бы несбалансированной, особенно если и другие клиенты поступят точно так же.

Для решения этой проблемы драйвер предоставляет сменный механизм балансировки нагрузки на узлы. Достигается это путем реализации интерфейса `com.datastax.driver.core.policies.LoadBalancingPolicy`.

Любая политика балансировки нагрузки должна предоставлять метод `distance()`, который классифицирует каждый узел кластера как локальный, удаленный или игнорируемый, возвращая один из элементов перечисления `HostDistance`. Драйвер предпочитает взаимодействовать с локальными узлами и устанавливает больше соединений с локальными узлами, чем с удаленными. Другая важная операция, `newQueryPlan()`, возвращает список узлов в том порядке, в каком им следует посыпать запросы. В интерфейсе `LoadBalancingPolicy` определены также операции, используемые для того, чтобы информировать политику о добавлении или удалении узлов, а также об их остановке и запуске. Эти операции помогают политике избегать включения остановленных или удаленных узлов в план рассылки запросов.

Драйвер предлагает две базовые реализации балансировки нагрузки: `RoundRobinPolicy` (подразумевается по умолчанию) и `DCAwareRoundRobinPolicy`.

Политика `RoundRobinPolicy` раздает запрос узлам по кругу, чтобы распределить между ними нагрузку. Политика `DCAwareRoundRobinPolicy` похожа, но в своих планах рассылки отдает предпочтение узлам в локальном ЦОДе. Эта политика может включать в план настраиваемое число узлов из удаленных ЦОДов, но их приоритет всегда будет меньше, чем у локальных узлов. Локальный ЦОД можно задать явно или позволить драйверу определить его автоматически.

Еще один режим – принятие во внимание маркеров, в этом случае для выбора узла, содержащего реплику требуемых данных, используется значение маркера ключа раздела, что позволяет минимизировать число опрашиваемых узлов. Это достигается обертыванием выбранной политики классом `TokenAwarePolicy`.

Политика балансировки нагрузки задается при создании объекта `Cluster`. Так, в следующем предложении для объекта `Cluster` указана политика предпочтения локальных узлов с учетом маркера:

```
Cluster.builder().withLoadBalancingPolicy(
    new TokenAwarePolicy(new DCAwareRoundRobinPolicy.Builder().build()));
```

Политика повторения попыток выполнения

Когда узел Cassandra выходит из строя или становится недосягаем по сети, драйвер автоматически и прозрачно пытается обратиться к другим узлам и планирует повторное подключение к «мертвым» узлам в фоновом режиме. Поскольку узлы могут показаться недоступными из-за краткосрочного изменения обстановки в сети, драйвер также предоставляет механизм попытки повторного выполнения запросов, не выполненных вследствие сетевой ошибки. Это устраняет необходимость кодировать логику повторения попыток в клиентском коде.

Политика повторения попыток выполнения определяется реализацией интерфейса `com.datastax.driver.core.RetryPolicy`. Методы `onReadTimeout()`, `onWriteTimeout()` и `onUnavailable()` определяют, что нужно делать в случае, когда при выполнении запроса возбуждается соответственно исключение `ReadTimeoutException`, `WriteTimeoutException` или `UnavailableException`, связанное с ошибками сети.



Исключения, возбуждаемые драйвером DataStax для Java

Различные исключения, возбуждаемые Java-драйвером, собраны в пакете `com.datastax.driver.core.exceptions`.

Методы интерфейса `RetryPolicy` возвращают значение типа `RetryDecision`, которое говорит, нужно ли пытаться повторить запрос, и если да, то с каким уровнем согласованности. Если запрос повторять не нужно, то исключение можно возбудить заново или проигнорировать – в последнем случае запрос вернет пустой объект `ResultSet`.

Java-драйвер предоставляет несколько реализаций интерфейса `RetryPolicy`.

- `DefaultRetryPolicy` – консервативная реализация, которая пытается повторно выполнить запрос лишь при весьма специфических условиях.
- Политика `FallthroughRetryPolicy` никогда не рекомендует повторную попытку, предпочитая заново возбудить исключение.
- `DowngradingConsistencyRetryPolicy` – более агрессивная политика, она понижает уровень согласованности, пытаясь хотя бы таким способом добиться выполнения запроса.



Замечание о политике DowngradingConsistencyRetryPolicy

В связи с этой политикой возникает закономерный вопрос: если вы готовы смириться с пониженным уровнем согласованности

при некоторых условиях, то нужен ли вам более высокий уровень в общем случае?

Политику повторения попыток выполнения можно задать при создании объекта Cluster, как показано в следующем примере, где выбранная политика DowngradingConsistencyRetryPolicy обернута классом LoggingRetryPolicy, так что все повторные попытки протоколируются:

```
Cluster.builder().withRetryPolicy(new LoggingRetryPolicy(  
    DowngradingConsistencyRetryPolicy.INSTANCE));
```

Политика, заданная для кластера, применяется ко всем запросам, выполняемым в этом кластере, если только она не переопределена в конкретном запросе путем вызова метода Statement.setRetryPolicy().

Политика упреждающего выполнения

Механизм повторов, автоматизирующий реакцию на сетевые тайм-ауты, – это прекрасно, но часто мы не можем позволить себе роскошь дожидаться не то что окончания тайм-аута, но даже длительных задержек на сборку мусора. Для ускорения драйвер предлагает механизм *упреждающего выполнения* (speculative execution). На случай если исходный узел-координатор запроса не ответит в течение предопределенного времени, драйвер заблаговременно начинает выполнение того же запроса с другим координатором. Как только на любой запрос будет получен ответ, драйвер возвращает его и отменяет остальные, не завершившиеся запросы.

Это поведение задается для объекта Cluster путем выбора реализации интерфейса com.datastax.driver.core.policies.SpeculativeExecutionPolicy.

По умолчанию подразумевается политика NoSpeculativeExecutionPolicy, при которой упреждающее выполнение вообще не планируется. Имеется также политика ConstantSpeculativeExecutionPolicy, которая планирует не более заданного количества повторных попыток с фиксированной задержкой в миллисекундах. Политика Percentile SpeculativeExecutionPolicy появилась недавно, и в версии 3.0 все еще имеет статус бета. Она запускает упреждающее выполнение с задержкой, величина которой зависит от наблюдаемого запаздывания исходного узла-координатора.

Эта политика задается при построении объекта Cluster:

```
Cluster.builder().withSpeculativeExecutionPolicy(  
    new ConstantSpeculativeExecutionPolicy (
```

```

200, // задержка в мс
3     // макс число упреждающих выполнений
);

```

Заданную политику впоследствии нельзя ни изменить, ни определить для отдельных команд.

Транслятор адресов

Во всех рассмотренных до сих пор примерах узлы идентифицировались IP-адресом, прописанным в свойстве узла `rpc_address` в расположеннном на нем файле `cassandra.yaml`. Бывает, что этот адрес для клиента недостижим. На такой случай драйвер предлагает сменный механизм трансляции адресов – интерфейс `com.datastax.driver.core.policies.AddressTranslator` (в версиях драйвера младше 3.0 слово «`translator`» писалось с ошибкой – «`translater`»).

В составе самого драйвера имеются две реализации: подразумеваемая по умолчанию `IdentityTranslator`, которая не изменяет IP-адреса, и `EC2MultiRegionAddressTranslator`, рассчитанная на среду Amazon EC2. Второй транслятор полезен в случаях, когда клиенту может понадобиться доступ к узлу из другого ЦОДа по публичному IP-адресу. Подробнее развертывание в среде EC2 рассматривается в главе 14.

Метаданные

Для доступа к метаданным кластера служит метод `Cluster.getMetadata()`. Класс `com.datastax.driver.core.Metadata` предоставляет различную информацию о кластере, в т. ч. его имя, схему, включающую пространства ключей и таблицы, и список известных хостов. Ниже показано, как получить имя кластера:

```

Metadata metadata = cluster.getMetadata();
System.out.printf("Подключение к кластеру: %s\n",
    metadata.getClusterName(), cluster.getClusterName());

```



Назначение имени кластеру

Как ни странно, класс `Cluster.Builder` позволяет назначить имя объекту `Cluster` в процессе его построения. Но оно предназначено только для того, чтобы клиент мог различать несколько объектов `Cluster`, и может отличаться от истинного имени, под которым кластер известен узлам. Вот это второе имя мы и получаем с помощью класса `Metadata`.

Если не задавать имя объекта `Cluster` при конструировании, то по умолчанию присваиваются имена «`cluster1`», «`cluster2`» и т. д. Что-

бы увидеть это имя, замените в примере выше метод `metadata.getClusterName()` на `cluster.getClusterName()`.

Обнаружение узлов

В объекте `Cluster` хранится постоянное соединение с одной из точек контакта, нужное для того, чтобы быть в курсе состояния и топологии кластера. С помощью этого соединения драйвер обнаруживает все узлы, находящиеся в кластере в данный момент. Драйвер представляет узел классом `com.datastax.driver.core.Host`. В примере ниже мы обходим все узлы и печатаем сведения о них:

```
for (Host host : cluster.getAllHosts())
{
    System.out.printf("ЦОД: %s; Стойка: %s; Хост: %s\n",
        host.getDatacenter(), host.getRack(), host.getAddress());
}
```

Полностью этот код приведен в классе `com.cassandraguide.clients.SimpleConnectionExample`.

При запуске в кластере с несколькими узлами, например созданном в главе 7 с помощью диспетчера кластеров Cassandra (`csm`), программа напечатает строки такого вида:

```
Подключение к кластеру: my_cluster
ЦОД: datacenter1; Стойка: rack1; Хост: /127.0.0.1
ЦОД: datacenter1; Стойка: rack1; Хост: /127.0.0.2
ЦОД: datacenter1; Стойка: rack1; Хост: /127.0.0.3
```

Пользуясь этим соединением, драйвер может обнаружить все узлы, входящие в состав кластера, а также узнавать о добавлении в кластер новых узлов. Для этого нужно зарегистрировать прослушиватель, реализовав интерфейс `Host.StateListener`, который содержит методы `onAdd()` и `onRemove()`, вызываемые соответственно при добавлении и удалении узла, и методы `onUp()` и `onDown()`, вызываемые, когда узел запущен или остановлен. Рассмотрим часть класса для регистрации прослушивателя кластера:

```
public class ConnectionListenerExample implements Host.StateListener {

    public String getHostString(Host host) {
        return new StringBuilder("ЦОД: " + host.getDatacenter() +
            " Стойка: " + host.getRack() +
            " Хост: " + host.getAddress().toString() +
            " Version: " + host.getCassandraVersion() +
            " Состояние " + host.getState());
```

```

    }

    public void onUp(Host host) {
        System.out.printf("Узел запущен: %s\n", getHostString(host));
    }

    public void onDown(Host host) {
        System.out.printf("Узел остановлен: %s\n", getHostString(host));
    }

    // прочие обязательные методы опущены...
    public static void main(String[] args) {

        List<Host.StateListener> list =
            ArrayList<Host.StateListener>();
        list.add(new ConnectionListenerExample());

        Cluster cluster = Cluster.builder().
            addContactPoint("127.0.0.1").
            withInitialListeners(list).
            build();

        cluster.init();
    }
}

```

Здесь мы просто печатаем сообщение, когда узел запускается или останавливается. Отметим, что выводится чуть больше информации об узлах, чем раньше, – включена также версия Cassandra, работающая на каждом узле. Полный код этого примера имеется в классе com.cassandraguide.clients.ConnectionListenerExample.

Выполним эту программу. Поскольку прослушиватель был добавлен до вызова init(), то сразу печатается такой результат:

```

Добавлен узел: ЦОД: datacenter1 Стойка: rack1
Хост: /127.0.0.1 Версия: 3.0.0 Состояние: UP
Добавлен узел: ЦОД: datacenter1 Стойка: rack1
Хост: /127.0.0.2 Версия: 3.0.0 Состояние: UP
Добавлен узел: ЦОД: datacenter1 Стойка: rack1
Хост: /127.0.0.3 Версия: 3.0.0 Состояние: UP

```

Если теперь остановить один узел командой ccm stop, то мы увидим:

```

Остановлен узел: ЦОД: datacenter1 Стойка: rack1
Хост: /127.0.0.2 Версия: 3.0.0 Состояние: DOWN

```

Если снова запустить этот узел, то программа напечатает:

```

Запущен узел: ЦОД: datacenter1 Стойка: rack1
Хост: /127.0.0.2 Версия: 3.0.0 Состояние: UP

```

Доступ к схеме

Класс Metadata также позволяет клиенту узнать о схеме кластера. Метод `exportSchemaAsString()` создает строку, содержащую описание всех пространств ключей и таблиц в кластере, включая системные. Эта строка эквивалентна результату команды `cqlsh DESCRIBE FULL SCHEMA`. Существуют также дополнительные методы для просмотра определения отдельного пространства ключей или таблицы.

В главе 2 мы подробно обсуждали, как Cassandra поддерживает согласованность в конечном счете. Поскольку информация о схеме хранится в базе данных Cassandra, то она тоже согласована в конечном счете, поэтому на разных узлах могут оказаться разные версии схемы. Начиная с версии 3.0, сам Java-драйвер не возвращает версию схемы, но ее можно узнать, выполнив команду `nodetool describecluster`:

```
$ ccm nodetool describecluster

Cluster Information:
  Name: test_cluster
  Snitch: org.apache.cassandra.locator.DynamicEndpointSnitch
  Partitioner: org.apache.cassandra.dht.Murmur3Partitioner
  Schema versions:
    ea46580a-4ab4-3e70-b68f-5e57dal89ac5:
      [127.0.0.1, 127.0.0.2, 127.0.0.3]
```

Здесь следует обратить внимание на две вещи. Во-первых, версия схемы представлена как UUID. Это значение вычисляется путем хэширования определений всех пространств ключей и таблиц, о которых известно узлу. Из того, что на всех трех узлах версия схемы одинакова, можно сделать вывод, что на них хранится одна и та же схема.

Разумеется, версия схемы со временем может меняться в результате создания, изменения и удаления пространств ключей и таблиц. Драйвер предоставляет клиентам механизм уведомления, позволяющий узнавать о таких изменениях; для этого нужно реализовать интерфейс `com.datastax.driver.core.SchemaChangeListener` и зарегистрировать его в объекте Cluster.

Примеры можно найти в классе `com.cassandraguide.clients.SimpleSchemaExample`.

Помимо средств доступа к схеме с помощью класса `Metadata`, Java-драйвер предлагает механизм управления схемой в виде пакета `com.datastax.driver.core.schemabuilder`. Класс `SchemaBuilder` предоставляет текущий API для создания объектов `SchemaStatement`, соответствую-

щих операциям CREATE, ALTER и DROP для пространств ключей, таблиц, индексов и пользовательских типов (UDT).

Так, для создания нашей таблицы `hotels` можно было бы написать такой код:

```
SchemaStatement hotelSchemaStatement = SchemaBuilder.createTable("hotels") .
    addPartitionKey("id", DataType.text()) .
    addColumn("name", DataType.text()) .
    addColumn("phone", DataType.text()) .
    addColumn("address", DataType.text()) .
    addColumn("pois", DataType.set(DataType.text())));
session.execute(hotelSchemaStatement);
```

Мы импортировали также класс `com.datastax.driver.core.DataType`, чтобы можно было воспользоваться его статическими методами для задания типа данных в каждом столбце.



Избегайте конфликтов при программном определении схемы

Многие разработчики обратили внимание, что такой механизм управления схемой из программы можно использовать в качестве техники «ленивой инициализации», позволяющей упростить развертывание приложения: если схема приложения еще не существует, то мы просто создаем ее программно. Однако поступать так не рекомендуется, если существует возможность, что одновременно будет запущено несколько клиентов. Не поможет даже фраза `IF NOT EXISTS`. Команды `CREATE TABLE` или `ALTER TABLE`, одновременно выполненные несколькими клиентами, могут привести к рассогласованию состояния узлов, что потребует ручного исправления.

Отладка и мониторинг

Драйвер предоставляет средства мониторинга и отладки работы клиентов с Cassandra, в т. ч. протоколирование и метрики. Существует также возможность трассировки запросов, которую мы рассмотрим в главе 12.

Протоколирование

В главе 10 мы расскажем об API Simple Logging Facade for Java (SLF4J), используемой в Cassandra. Java-драйвер пользуется тем же API. Чтобы включить протоколирование в клиентском приложении на Java, нужно поместить в путь к классам реализацию, совместимую с SLF4J.

Ниже показано, как включить в POM-файл Maven зависимость от проекта Logback, используемого в этом качестве:

```
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.1.3</version>
</dependency>
```

Дополнительные сведения о проекте Logback имеются на сайте <http://logback.qos.ch/>.

По умолчанию в Java-драйвере установлен уровень протоколирования DEBUG, при этом выдается довольно подробная информация. Мы можем воспользоваться удобным механизмом конфигурирования Logback, который позволяет отдельно задавать параметры протоколирования для тестовой и производственной среды. Logback сначала ищет в пути к классам файл *logback-test.xml*, описывающий тестовую конфигурацию, а если таковой не найден, то ищет файл *logback.xml*.

Дополнительные сведения о настройке Logback, включая примеры конфигурационных файлов для тестовой и производственной среды, см. на странице <http://logback.qos.ch/manual/configuration.html>.

Метрики

Иногда полезно наблюдать за поведением клиентских приложений на протяжении длительного времени с целью обнаружения аномальных ситуаций и отладочных сообщений. Java-драйвер собирает сведения о своей работе и предоставляет их в распоряжение клиента в виде метрик с помощью библиотеки Dropwizard Metrics (<https://github.com/dropwizard/metrics>). Собираются сведения о подключениях, очередях задач, запросах и ошибках, например ошибках подключения, тайм-аутах при чтении и записи, повторах и упреждающих выполнениях.

Получить локальные метрики Java-драйвера позволяет метод `Cluster.getMetrics()`. Библиотека Metrics также интегрируется с Java Management Extensions (JMX) и позволяет вести мониторинг метрик удаленно. По умолчанию отчеты JMX включены, но это можно отменить в объекте Configuration, который доступен при построении объекта Cluster.

Драйвер DataStax для Python

Самым популярным из ранних клиентов для Python был Pycassa, в котором использовался интерфейс Thrift. Но сейчас проект Pycassa уже не сопровождается, а во всех новых разработках рекомендуется использовать драйвер DataStax для Python.

Первая полная версия драйвера DataStax для Python была представлена в январе 2014 года, в ней были реализованы управление сеансами, обнаружение узлов, пул соединений, синхронные и асинхронные запросы, балансировка нагрузки, трассировка запросов, метрики (с использованием библиотеки Scales), протоколирование, аутентификация и поддержка SSL. Поддержка разбиения на страницы и облеченных транзакций, появившихся в версии Cassandra 2.1, а также поддержка Python 3 добавлена в версию драйвера 2.0 в мае 2014 г. Python-драйвер совместим с версиями Cassandra начиная с 1.2 и может работать с Python 2.6, 2.7, 3.3 и 3.4. Версии драйвера существуют для платформ Linux, Mac OS и Windows.

Официальная документация по драйверу размещена на сайте DataStax (<https://datastax.github.io/python-driver/index.html>), а исходный код доступен по адресу <https://github.com/datastax/python-driver>. Установить драйвер можно с помощью установщика *pip* программ на Python:

```
$ pip install cassandra-driver
```



Установка Python и PIP

Для выполнения примера понадобятся совместимая версия Python для вашей платформы (см. выше) и программа *pip*. Для установки *pip* нужно скачать скрипт <https://bootstrap.pypa.io/get-pip.py> и выполнить его командой `python get-pip.py`. В системах на базе Unix это, возможно, придется делать через `sudo`.

Приведенная ниже программа подключается к кластеру и вставляет одну строку в таблицу `hotels`:

```
from cassandra.cluster import Cluster
cluster = Cluster(['127.0.0.1'])
session = cluster.connect('hotel')
session.execute("""
    insert into hotels (id, name, phone)
    values (%s, %s, %s)
    """
    ('AZ123', 'Super Hotel at WestWorld', '1-888-999-9999')
)
```

Python-драйвер включает преобразователь объектов *cqlengine*, доступ к которому осуществляется с помощью класса *cassandra.cqlengine.models.Model*. В драйвере используются сторонние библиотеки для повышения производительности, сжатия и сбора метрик. Для ускорения работы часть кода написана на С с применением Cython. Драйвер может также работать в альтернативной среде исполнения PyPy, где используется JIT-компилятор. Благодаря меньшему потреблению ресурсов процессора в этой среде достигается более высокая производительность, иногда вдвое выше, чем в стандартном интерпретаторе Python. Для сжатия необходимо установить одну из библиотек *lz4* или *python-snappy* – в зависимости от предпочтительного алгоритма.

Драйвер DataStax для Node.js

Первоначальный основанный на Thrift клиентский интерфейс к Node.js назывался *Helenus*, впоследствии Йорге Бэй (*Jorge Bay*) разработал клиент на основе CQL, *node-cassandra-cql*.

Драйвер DataStax для Node.js, официально выпущенный в октябре 2014 года, основан на *node-cassandra-cql* с добавлением стандартных возможностей, имеющихся во всех драйверах DataStax для Apache Cassandra. Скачать драйвер можно со страницы <https://github.com/datastax/nodejs-driver>.

Драйвер для Node.js устанавливается с помощью менеджера пакетов NPM:

```
$ npm install cassandra-driver
```



Установка среды исполнения Node.js и менеджера пакетов

Скачать дистрибутив Node для различных платформ можно с сайта <https://nodejs.org>. Он включает как Node.js, так и NPM. В системах на базе Unix они обычно устанавливаются в */usr/local/bin/node* и */usr/local/bin/npm*.

Основное отличие в том, что мы работаем с объектом *Client*, а не *Cluster*, как в драйверах для других языков. Остальные конструкции очень похожи:

```
var cassandra = require('cassandra-driver');
var client = new cassandra.Client({ contactPoints: ['127.0.0.1'],
    keyspace: 'hotel' });
```

Построение и выполнение параметризованного запроса выглядит так:

```
var query = 'SELECT * FROM hotels WHERE id=?';
client.execute(query, ['AZ123'], function(err, result) {
  assert.ifError(err);
  console.log('получен отель с названием ' + result.rows[0].name);
});
```

Драйвер DataStax для Ruby

Первый основанный на Thrift клиент для Ruby был создан компанией Twitter. В начале 2013 года Тео Гультберг (Theo Hultberg) возглавил разработку gem-пакета `cql-rb`, который лег в основу драйвера DataStax для Ruby, выпущенного в ноябре 2014 г. Скачать его можно со страницы <https://github.com/datastax/ruby-driver>.

Для установки драйвера нужна система Ruby Gems:

```
$ gem install cassandra-driver
```

Программа ниже создает кластер и сеанс, а затем выполняет простой асинхронный запрос, который перебирает строки таблицы `hotels`:

```
require 'cassandra'

cluster = Cassandra.cluster(hosts: ['127.0.0.1'])
session = cluster.connect('hotel')

future = session.execute_async('SELECT * FROM hotels')
future.on_success do |rows|
  rows.each do |row|
    puts "Отель: #{row['id']} Название: #{row['name']}"
  end
end
future.join
```

Ruby-драйвер работает со стандартным Ruby, а также под управлением интерпретатора JRuby версии 1.7 или старше, который обеспечивает более высокую производительность. Существуют версии драйвера для Linux и Mac OS, но Windows не поддерживается.

Драйвер DataStax для C#

Первая версия драйвера DataStax для C# была выпущена в июле 2013 года. Драйвер поддерживает Windows-клиентов на платформе .NET, поэтому его часто называют «драйвером для .NET».

C#-драйвер устанавливается с помощью NuGet, менеджера пакетов для платформы разработки Microsoft. Из PowerShell выполните следующую команду на консоли менеджера пакетов:

```
PM> Install-Package CassandraCSharpDriver
```

Чтобы воспользоваться драйвером, создайте новый проект в Visual Studio и добавьте директиву `using`, содержащую ссылку на пространство имен `Cassandra`. Следующая программа подключается к пространству ключей `hotel` и вставляет новую запись в таблицу `hotels`:

```
Cluster Cluster = Cluster.Builder()
    .AddContactPoint("127.0.0.1") .Build();

ISession Session = Cluster.Connect("hotel");
Session.Execute(
    "INSERT INTO hotels (id, name, phone) " +
    "VALUES (" +
    "'AZ123', " +
    "'Super Hotel at WestWorld', " +
    "'1-888-999-9999', " +
    ");");
```

C#-драйвер интегрирован с языком Language Integrated Query (LINQ), компонентом Microsoft .NET Framework, добавляющим средства выполнения запросов в .NET-совместимые языки; имеется также отдельный преобразователь объектов.



Пример приложения: KillrVideo

Люк Тиллман (Luke Tillman), Патрик Макфейдин (Patrick McFadin) и другие разработали приложение для обмена видео KillrVideo. Это приложение для .NET с открытым исходным кодом, написанное с использованием драйвера DataStax для C# и развернутое в облаке Microsoft Azure. В нем также применяются возможности DataStax Enterprise: интеграция с Apache Spark и Apache SOLR. Найти исходный код можно на GitHub.

Драйвер DataStax для C/C++

Драйвер DataStax для C/C++ вышел в феврале 2014 года. Скачать исходный код можно со страницы <https://github.com/datastax/cpp-driver>, а документация опубликована по адресу <http://datastax.github.io/cpp-driver>.

Драйвер для C/C++ отличается от других драйверов тем, что в API присутствуют только асинхронные операции, а синхронные полно-

стью исключены. Например, создание сеанса – асинхронная операция, которая возвращает будущий объект:

```
#include <cassandra.h>
#include <stdio.h>

int main() {
    CassFuture* connect_future = NULL;
    CassCluster* cluster = cass_cluster_new();
    CassSession* session = cass_session_new();

    cass_cluster_set_contact_points(cluster, "127.0.0.1");
    connect_future = cass_session_connect(session, cluster);
    if (cass_future_error_code(connect_future) == CASS_OK) {
        /* продолжить работу... */
    }
}
```

Но, как легко видеть, синхронную семантику легко поддержать, если сразу же начать ожидание будущего объекта. Построение и выполнение простого запроса выглядят так:

```
CassStatement* select_statement
= cass_statement_new("SELECT * FROM hotel.hotels", 0);

CassFuture* hotel_select_future =
    cass_session_execute(session, select_statement);

if(cass_future_error_code(result_future) == CASS_OK) {
    const CassResult* result = cass_future_get_result(result_future);
    CassIterator* rows = cass_iterator_from_result(result);

    while(cass_iterator_next(rows)) {
        const CassRow* row = cass_iterator_get_row(rows);
        const CassValue* value = cass_row_get_column_by_name(row, "name");
        const char* name;
        size_t name_length;
        cass_value_get_string(value, &name, &name_length);
        printf("Название отеля: '%.*s'\n", (int)name_length, name);
    }
}
```

Напомним, что в программах на C/C++ очень важно правильно управлять памятью; для краткости мы опустили освобождение различных объектов: кластеров, сеансов, будущих объектов и результатов.

В драйвере для C/C++ используются библиотека *libuv* для реализации асинхронных операций ввода-вывода и – факультативно – библиотека OpenSSL для шифрования трафика между клиентом

и узлом. Инструкции по компиляции и компоновке зависят от платформы, детали смотрите в документации по драйверу.

Драйвер DataStax для PHP

Драйвер DataStax для PHP поддерживает написание серверных PHP-скриптов. Он был выпущен в 2015 году, является оберткой драйвера DataStax для C/C++ и работает как в Unix, так и в Windows.

Есть несколько способов установить драйвер, но самый простой – из репозитория PECL:

```
pecl install cassandra
```

Показанный ниже код выбирает строки из таблицы `hotels` и печатает их, применяя асинхронный API:

```
<?php  
  
$keyspace = 'hotel';  
$session = $cluster->connect($keyspace);  
$statement = new Cassandra\SimpleStatement(  
    'SELECT * FROM hotels'  
);  
  
$future = $session->executeAsync($statement);  
$result = $future->get();  
foreach ($result as $row) {  
    printf("ид: %s, название: %s, телефон: %s\n",  
        $row['id'], $row['name'], $row['phone']);  
}
```

Документация по PHP-драйверу размещена по адресу <https://github.com/datastax/phpdriver>, а исходный код – по адресу <https://datastax.github.io/php-driver>.

Резюме

Теперь вы в курсе различных клиентских интерфейсов для Cassandra, их возможностей, а также порядка установки и использования. Мы уделили особое внимание драйверу DataStax для Java, чтобы вы могли немного попрактиковаться. Этот опыт будет полезен, если вы решите поработать с другими драйверами DataStax. Мы продолжим использовать драйвер DataStax для Java и в следующих главах.

Глава 9

Чтение

и запись данных

Понимая, как устроена модель данных и как работать с простым клиентом, мы теперь более глубоко обсудим различные виды запросов для чтения и записи данных в Cassandra. Мы также заглянем за кулисы и посмотрим, как Cassandra обрабатывает такие запросы.

Как и в предыдущей главе, для иллюстрации практической работы мы включили примеры кода с использованием драйвера DataStax для Java.

Запись

Для начала остановимся на некоторых важнейших свойствах записи данных в Cassandra. Прежде всего, запись производится очень быстро, потому что не требуется ни чтение с диска, ни поиск в файлах на диске. Таблицы в памяти и файлы SSTable избавляют Cassandra от необходимости выполнять при записи действия, которые замедляют работу многих других баз данных. В Cassandra любая запись сводится к дописыванию в конец файла.

Благодаря журналу фиксаций и вручению напоминаний запись в базу данных всегда возможна, а в пределах семейства столбцов операции записи атомарны.



Вставка, обновление и обновление-вставка

Поскольку в Cassandra применяется модель записи с дописыванием, между операциями вставки и обновления нет принципиальной разницы. Если вставить строку с таким же первичным ключом, как у существующей, то существующая строка будет заменена. При

попытке обновить строку с несуществующим первичным ключом Cassandra создаст ее.

Поэтому часто говорят, что Cassandra поддерживает операцию обновления-вставки (*upsert*), понимая под этим, что вставка и обновление обрабатываются одинаково с одним небольшим исключением, которое проявляется в облегченных транзакциях.

Уровни согласованности при записи

Наличие в Cassandra механизма настраиваемой согласованности означает, что мы можем указать в запросе, какой уровень согласованности требуется при записи. Чем выше уровень согласованности, тем больше узлов-реплик должно ответить, чтобы операция считалась успешно завершенной. Увеличение уровня согласованности сопровождается снижением доступности, поскольку для успеха требуется больше работающих узлов. В табл. 9.1 описаны различные уровни согласованности и их интерпретация.

Таблица 9.1. Уровни согласованности записи

Уровень согласованности	Следствие
ANY	Гарантируется, что перед возвратом управления клиенту значение записано как минимум в одну реплику, при этом напоминания считаются как запись
ONE, TWO, THREE	Гарантируется, что перед возвратом управления клиенту значение записано в журнал фиксаций и в таблицу в памяти по меньшей мере на одном, двух или трех узлах
LOCAL_ONE	То же, что ONE, но дополнительно требуется, чтобы ответивший узел находился в локальном центре обработки данных
QUORUM	Гарантируется, что запись произошла в большинство реплик ((коэффициент репликации / 2) + 1)
LOCAL_QUORUM	То же, что QUORUM, но ответившие узлы находятся в локальном центре обработки данных
EACH_QUORUM	Гарантируется, что QUORUM узлов ответили в каждом центре обработки данных
ALL	Гарантируется, что перед возвратом управления клиенту значение записано на узлах, число которых равно коэффициенту репликации. Если хотя бы одна реплика не отозвалась, операция завершается с ошибкой

ANY – самый примечательный уровень согласованности. Он означает, что запись гарантированно произведена хотя бы на одном узле, но позволяет считать напоминания успешной записью. То есть если узел, на который должно быть записано значение, не работает, то сервер

сделает для себя заметку об этом – *напоминание* – и будет хранить ее до тех пор, пока узел не вернется в строй. Как только узел заработает, сервер обнаружит это, посмотрит, есть ли для этого узла какие-то несостоявшиеся операции записи в форме напоминаний, и, если да, отправит запомненное значение ожившему узлу. Во многих случаях напоминание хранится не на том узле, который первоначально осознал его необходимость, поскольку тот передает напоминание одному из соседей неработающего узла.

Уровень согласованности ONE означает, что значение должно быть записано в журнал фиксаций и в таблицу в памяти. Таким образом, запись при уровне ONE долговечна, и, следовательно, это минимальный уровень, обеспечивающий высокую производительность и долговечность. Если узел выйдет из строя сразу после записи, то значение останется в журнале фиксаций, и сервер накатит его в процессе следующего запуска.

Уровни согласованности по умолчанию

Типичные клиенты Cassandra поддерживают задание уровня согласованности по умолчанию, действующего для всех запросов, а также специального уровня для отдельных запросов. Например, в cqlsh существует команда `CONSISTENCY`, позволяющая проверить и установить уровень согласованности по умолчанию:

```
cqlsh> CONSISTENCY;
Current consistency level is ONE.
cqlsh> CONSISTENCY LOCAL_ONE;
Consistency level set to LOCAL_ONE.
```

В драйвере DataStax для Java Driver уровень согласованности по умолчанию можно задать, передав `Cluster.Builder` объект `com.datastax.driver.core.QueryOptions`:

```
QueryOptions queryOptions = new QueryOptions();
queryOptions.setConsistencyLevel(ConsistencyLevel.LOCAL_ONE);

Cluster cluster = Cluster.builder().addContactPoint("127.0.0.1").
withQueryOptions(queryOptions).build();
```

Уровень согласованности по умолчанию можно переопределить в конкретной команде:

```
Statement statement = ...
statement.setConsistencyLevel(ConsistencyLevel.LOCAL_ONE);
```

Путь записи в Cassandra

Путь записи описывает способ обработки запросов модификации данных клиентами, конечным итогом чего является сохранение данных на диске. Мы рассмотрим путь записи с двух точек зрения: взаимодействие между узлами и внутренний процесс сохранения данных на отдельном узле. Общая картина взаимодействия между узлами кластера показана на рис. 9.1.

Путь начинается, когда клиент отправляет запрос записи некоторому узлу Cassandra, который становится координатором данного запроса. Узел-координатор обращается к разделителю, чтобы узнать, какие узлы кластера являются репликами согласно коэффициенту репликации, заданному для указанного пространства ключей. Координатор сам может оказаться репликой, особенно если клиент пользуется драйвером, знающим о маркерах. Если координатор понимает, что для достижения указанного в запросе уровня согласованности реплик недостаточно, то он сразу же возвращает код ошибки.

Затем координатор посыпает одновременно всем репликам запросы о записи данных. Тем самым гарантируется, что все работающие узлы-реплики получат данные для записи. Данные на неработающих узлах будут не согласованы, но впоследствии это будет исправлено одним из механизмов антиэнтропии: вручение напоминаний, исправление на этапе чтения или антиэнтропийное исправление.

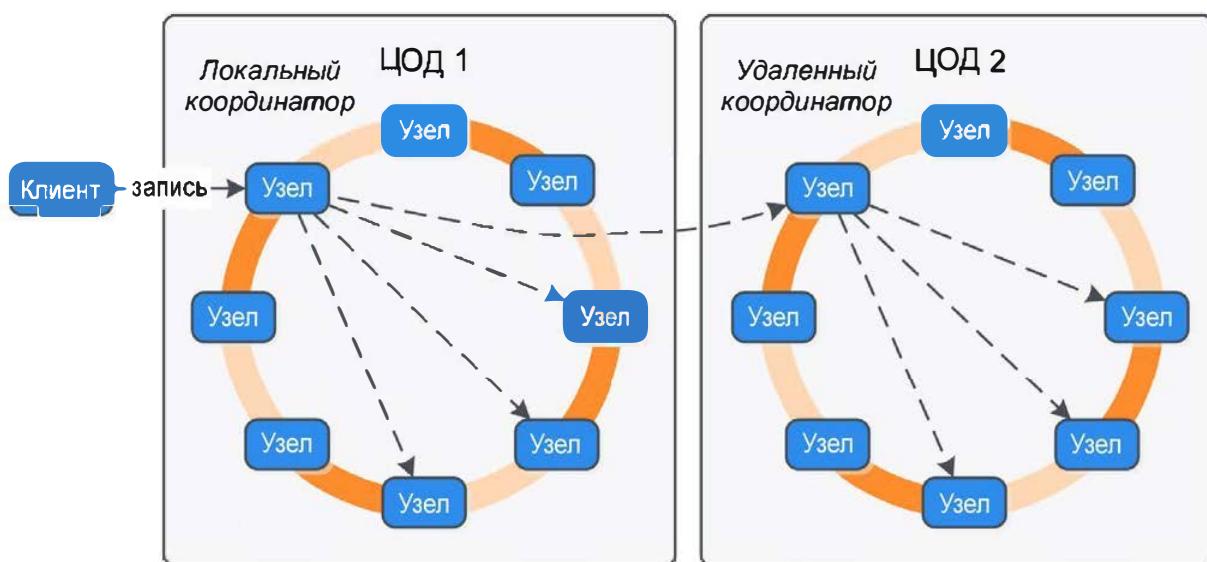


Рис. 9.1 ♦ Взаимодействие между узлами на пути записи

Если кластер охватывает несколько ЦОДов, то *локальный координатор* выбирает в каждом из остальных ЦОДов *удаленного координатора*

тора, который будет координировать запись в реплики, находящиеся в этом ЦОДе. Каждый удаленный узел-реплика отвечает непосредственно исходному узлу-координатору.

Координатор ждет ответов от реплик. Как только будет получено достаточно ответов, чтобы удовлетворить заданный уровень согласованности, координатор подтверждает успешность записи клиенту. Если реплика не ответила в течение времени тайм-аута, то она считается неработающей и для нее сохраняется напоминание. Напоминание считается успешной записью, только если задан уровень согласованности ANY.

На рис. 9.2 изображены взаимодействия, имеющие место на каждом узле-реплике в процессе обработки запроса на запись.



Рис. 9.2 ♦ Взаимодействия внутри узла на пути записи

Как только узел-реплика получает запрос на запись, он первым делом записывает данные в журнал фиксаций. Затем данные записываются в таблицу в памяти. Если включен режим кэширования строк и строка находится в кэше, то эта строка объявляется недействительной. Более подробно мы обсудим кэширование при рассмотрении пути чтения.

Если в результате записи оказывается превышен порог, заданный для журнала фиксаций или таблицы в памяти, то планируется выполнение сброса. Как настраивать эти пороги, мы обсудим в главе 12.

В этот момент запись считается успешной, и узел может ответить координатору или клиенту.

Ответив, узел выполняет сброс, если таковой был запланирован. Содержимое всех таблиц в памяти сохраняется в файлах SSTable на диске, а журнал фиксаций очищается. По завершении сброса плани-

руются дополнительные задачи, которые проверят, нужно ли произвести уплотнение, и в случае необходимости займутся этим.



Дополнительные сведения о пути записи

Разумеется, это лишь упрощенное описание пути записи, в нем опущены такие детали, как модификация счетчиков и обработка материализованных представлений. Запись в таблицы, над которыми построены материализованные представления, сложнее, потому что разделы необходимо заблокировать. Для обслуживания материализованных представлений Cassandra использует протоколируемые пакеты команд.

Более подробное обсуждение пути записи можно найти в блестящей статье Майкла Эджа (Michael Edge) на сайте Apache Cassandra Wiki по адресу <https://wiki.apache.org/cassandra/WritePathFor-Users>.

Запись файлов на диск

Теперь рассмотрим некоторые детали записи на диск файлов Cassandra, в т. ч. журналы фиксаций и файлы SSTable.

Журналы фиксаций

Журналы фиксаций представляют собой двоичные файлы и хранятся в каталоге `$CASSANDRA_HOME/data/commitlog`.

Имена журналов фиксаций имеют вид `CommitLog-<версия>-<временная метка>.log`, например: `CommitLog-6-1451831400468.log`. *Версия* – целое число, обозначающее формат журнала фиксаций. Так, в версии Cassandra 3.0 номер версии журнала равен 6. Узнать, какие есть версии, можно, заглянув в исходный код класса `org.apache.cassandra.db.commitlog.CommitLogDescriptor`.

Файлы SSTable

При сбросе данных в файлы SSTable на диске фактически производится запись в несколько файлов, ассоциированных с одной логической таблицей SSTable. Рассмотрим, как организованы файлы в каталоге `$CASSANDRA_HOME/data/data` на диске.



Принудительная запись в файлы SSTable на диске

Если вы прорабатываете приведенные в книге примеры на реальном узле Cassandra, то попробуйте сейчас выполнить команду `nodetool flush`, поскольку может статься, что вы ввели еще недостаточно данных, чтобы Cassandra произвела их сброс на диск автоматически. Подробнее об этой команде мы поговорим в главе 11.

В каталоге *data* вы найдете отдельные подкаталоги для каждого пространства ключей, а в них подкаталоги для каждой таблицы, имена которых образованы из имени таблицы и UUID. Уникальный идентификатор UUID позволяет различить версии схемы, поскольку схема таблицы со временем может изменяться.

Каждый каталог содержит файлы SSTable с данными. Вот пример пути к каталогу: *hotel/hotels-3677bbb0155811e5899aa9fac1d00bce*.

Каждая таблица SSTable представлена несколькими файлами с общей схемой именования: <версия>-<поколение>-<реализация>-<компонент>.db, где:

- **версия** – два знака, описывающих главную и дополнительную версии формата SSTable. Например, в Cassandra 3.0 используется версия формата *ta*. Дополнительные сведения о версиях можно найти в исходном коде класса org.apache.cassandra.io.sstable.Descriptor;
- **поколение** – число, которое увеличивается на 1 при создании каждого нового файла SSTable для данной таблицы;
- **реализация** – ссылка на используемую реализацию интерфейса org.apache.cassandra.io.sstable.format.SSTableWriter. В Cassandra 3.0 значение равно «big», что означает «формат Bigtable», т. е. класс org.apache.cassandra.io.sstable.format.big.BigFormat.

Каждая таблица SSTable разбита на несколько файлов, или *компонентов*. В Cassandra 3.0 есть такие компоненты.

**-Data.db*

Файлы, в которых находятся собственно данные. Только эти файлы сохраняются механизмами резервного копирования Cassandra, о которых мы узнаем в главе 11.

**-CompressionInfo.db*

Метаданные о сжатии файла **-Data.db*.

**-Digest.adler32*

Контрольная сумма файла **-Data.db* (до выхода версии 3.0 использовался алгоритм вычислений контрольной суммы CRC 32, поэтому у файлов было расширение *crc32*).

**-Filter.db*

Фильтр Блума для этой таблицы SSTable.

**-Index.db*

Смещения строк и столбцов для соответственного файла **-Data.db*.

Summary.db

Выборка из индекса для ускорения чтения.

Statistics.db

Статистика для таблицы SSTable, используемая при выполнении команды nodetool tablehistograms.

TOC.txt

Список компонентов этой таблицы SSTable.

В прежних версиях номера версий и схема именования файлов отличались. До версии 2.2 в начале имени каждого файла указывались имена пространства ключей и таблицы, а начиная с версии 2.2 этих частей нет, потому что их можно вывести из имени каталога.

В главе 11 мы познакомимся с некоторыми инструментами для работы с файлами SSTable.

Облегченные транзакции

В главе 1 мы сказали, что Cassandra, как и многие другие базы данных NoSQL, не поддерживает в полной мере семантику ACID-транзакций, принятую в реляционных базах данных. Однако Cassandra все же предоставляет два механизма с некоторыми транзакционными чертами: *облегченные транзакции и пакеты*.

Механизм облегченных транзакций (*lightweight transaction* – LWT) в Cassandra основан на алгоритме Paxos, описанном в главе 6. LWT появились в версии 2.0 и поддерживают следующую семантику:

- область видимости транзакции ограничена одним разделом;
- транзакция состоит из операций чтения и записи, называемых также операцией «сравнения и установки». Установка производится, только если сравнение оказалось успешным;
- если транзакция завершается с ошибкой, потому что фактические значения отличаются от ожидаемых, то Cassandra включит в ответ текущие значения, чтобы пользователь мог решить, нужно ли повторить или отменить операцию, без дополнительного запроса;
- фраза `USING TIMESTAMP` не поддерживается.

Допустим, мы хотим создать запись о новом отеле, применяя модель данных, описанную в главе 5. Мы хотим быть уверены, что не перезапишем запись об отеле с таким же идентификатором, поэтому добавили в команду вставки фразу `IF NOT EXISTS`:

```
cqlsh> INSERT INTO hotel.hotels (id, name, phone) VALUES (
  'AZ123', 'Super Hotel at WestWorld', '1-888-999-9999') IF NOT EXISTS;
```

[applied]

True

Команда проверяет, есть ли запись с таким же ключом раздела; для данной таблицы это поле `hotel_id`. Посмотрим, что произойдет при повторном выполнении этой команды:

```
cqlsh> INSERT INTO hotel.hotels (id, name, phone) VALUES (
  'AZ123', 'Super Hotel at WestWorld', '1-888-999-9999') IF NOT EXISTS;
```

[applied]	id	address	name	phone	pois
False	AZ123	null	Super Hotel at WestWorld	1-888-999-9999	null

В данном случае транзакция завершается с ошибкой, потому что уже существует отель с идентификатором «AZ123», и `cqlsh` умудрился возвращает строку с признаком ошибки и значениями, которые мы пробовали вставить.

Точно так же обстоит дело с обновлением. Например, можно было выполнить такую команду, изменяющую название отеля:

```
cqlsh> UPDATE hotel.hotels SET name='Super Hotel Suites at WestWorld'
... WHERE id='AZ123' IF name='Super Hotel at WestWorld';
```

[applied]

True

```
cqlsh> UPDATE hotel.hotels SET name='Super Hotel Suites at WestWorld'
... WHERE id='AZ123' IF name='Super Hotel at WestWorld';
```

[applied] | name

False | Super Hotel Suites at WestWorld

Вторая команда `UPDATE` тоже завершается с ошибкой, потому что в первой название уже было изменено. Из-за того, что в Cassandra принята модель обновления-вставки, единственное различие между этими операциями заключается в том, что в `INSERT` для проверки используется конструкция `IF NOT EXISTS`, а в `UPDATE` – `IF x=y`.



Применение транзакций при создании схемы

CQL поддерживает фразу IF NOT EXISTS также при создании пространств ключей и таблиц. Это особенно полезно при запуске нескольких скриптов обновления схемы.

Давайте теперь реализуем создание отеля с помощью драйвера DataStax для Java. При выполнении условной команды объект ResultSet будет содержать единственную строку Row с столбцом applied типа boolean, который показывает, завершилась команда успешно или неудачно. Можно также воспользоваться методом wasApplied() объекта команды:

```
SimpleStatement hotelInsert = session.newSimpleStatement(
    "INSERT INTO hotels (id, name, phone) VALUES (?, ?, ?) IF NOT EXISTS",
    "AZ123", "Super Hotel at WestWorld", "1-888-999-9999");

ResultSet hotelInsertResult = session.execute(hotelInsert);

boolean wasApplied = hotelInsertResult.wasApplied();

if (wasApplied) {
    Row row = hotelInsertResult.one();
    row.getBool("applied");
}
```

Для условных команд записи можно задавать не только обычный уровень согласованности, но и *последовательный уровень согласованности* (serial consistency level). Эта величина определяет, сколько узлов должно ответить на стадии записи алгоритма Paxos, когда узлы-участники договариваются о предложенной операции записи. В табл. 9.2 описаны два возможных значения.

Таблица 9.2. Последовательные уровни согласованности

Уровень согласованности	Следствие
SERIAL	Подразумевается по умолчанию и означает, что должен быть полный кворум ответивших узлов
LOCAL_SERIAL	Аналогичен SERIAL и дополнительно означает, что в транзакции участвуют только узлы в локальном ЦОДе

Последовательный уровень согласованности применим также к операциям чтения. Если Cassandra обнаруживает, что запрос читает данные, являющиеся частью незафиксированной транзакции, то она фиксирует транзакцию в процессе чтения, соблюдая заданный последовательный уровень согласованности.

С помощью команды SERIAL CONSISTENCY можно задать уровень согласованности по умолчанию для всех команд, исполняемых в cqlsh. В драйвере DataStax для Java для той же цели применяется метод Options.setSerialConsistencyLevel().

Пакеты

Облегченные транзакции ограничены одним разделом, но Cassandra предоставляет механизм *пакетов* (batch), позволяющий производить групповые модификации нескольких разделов одной командой.

Ниже описывается семантика пакетной операции.

- Разрешено включать в пакет только команды модификации (INSERT, UPDATE, DELETE).
- Пакеты атомарны, т. е. если пакет принят, то все входящие в него команды в конечном счете будут выполнены. Поэтому пакеты в Cassandra иногда называют *атомарными пакетами*, или *протоколируемыми пакетами*.
- Все операции обновления, входящие в пакет и относящиеся к одному разделу, выполняются изолированно, но гарантий изоляции между разделами нет. Это означает, что модификации, произведенные в другом разделе, могут быть прочитаны до завершения пакета.
- Пакеты не являются транзакционным механизмом, но в них можно включать облегченные транзакции. Если в пакете несколько команд облегченных транзакций, то все они должны относиться к одному и тому же разделу.
- Модификации счетчиков разрешены только в пакетах специального вида – *пакетах модификаций счетчиков* (counter batch). Такие пакеты могут содержать только команды модификации счетчиков.



Использовать непротоколируемые пакеты не рекомендуется

В версиях Cassandra младше 3.0 поддерживались непротоколируемые пакеты, в которых пропускались шаги, связанные с журналом пакетов. Но они не давали гарантии успешного завершения пакета, поэтому могли оставить базу данных в несогласованном состоянии.

Пакеты уменьшают объем трафика между клиентом и узлом-координатором, поскольку клиент может собрать несколько команд в один запрос. Но в то же время они возлагают дополнительную работу на координатора, который вынужден управлять выполнением нескольких команд.

В некоторых случаях пакеты Cassandra – хорошее средство, например когда нужно выполнить несколько обновлений в одном разделе или поддерживать синхронизацию нескольких таблиц. Типичный пример – модификация денормализованных таблиц, в которых хранятся одни и те же данные для различных способов доступа.



Пакеты не предназначены для массовой загрузки

Начинающие пользователи часто ошибочно считают, что пакет позволяет повысить производительность массовых обновлений. Это совершенно не так – на самом деле пакеты приводят к снижению производительности и могут повлечь за собой заметные накладные расходы на сборку мусора.

Рассмотрим пример пакета, который был бы полезен для вставки нового отеля в наши денормализованные таблицы. Начало и конец пакета отмечаются командами CQL BEGIN BATCH и APPLY BATCH:

```
cqlsh> BEGIN BATCH
    INSERT INTO hotel.hotels (id, name, phone)
        VALUES ('AZ123', 'Super Hotel at WestWorld', '1-888-999-9999');
    INSERT INTO hotel.hotels_by_poi (poi_name, id, name, phone)
        VALUES ('West World', 'AZ123', 'Super Hotel at WestWorld',
        '1-888-999-9999');
APPLY BATCH;
```

Драйвер DataStax для Java поддерживает пакеты с помощью класса com.datastax.driver.core.BatchStatement. Вот как тот же пример выглядел бы в клиентской программе на Java:

```
SimpleStatement hotelInsert = session.newSimpleStatement(
    "INSERT INTO hotels (id, name, phone) VALUES (?, ?, ?)",
    "AZ123", "Super Hotel at WestWorld", "1-888-999-9999");
SimpleStatement hotelsByPoiInsert = session.newSimpleStatement(
    "INSERT INTO hotels_by_poi (poi_name, id, name, phone)
VALUES (?, ?, ?, ?)", "WestWorld", "AZ123",
"Super Hotel at WestWorld", "1-888-999-9999");

BatchStatement hotelBatch = new BatchStatement();
hotelBatch.add(hotelsByPoiInsert);
hotelBatch.add(hotelInsert);

ResultSet hotelInsertResult = session.execute(hotelBatch);
```

Для создания пакета можно также вызвать метод QueryBuilder.batch(), передав ему несколько объектов Statement. Примеры работы с BatchStatement можно найти в исходном коде класса com.cassandra-guide.readwrite.BatchStatementExample.



Создание пакетов модификаций счетчиков в драйверах DataStax

Драйверы DataStax не содержат отдельного механизма для пакетов модификаций счетчиков. Нужно просто помнить, что пакет должен содержать либо только команды модификаций счетчиков, либо только команды модификации, не относящиеся к счетчикам.

Рассмотрим, как работает механизм пакетов. Координатор отправляет копию пакета, называемую *журналом пакета* (batchlog), двум другим узлам, где она сохраняется в таблице system.batchlog. Затем координатор выполняет все команды, включенные в пакет, и, когда они завершатся, удаляет журнал пакета с других узлов.

Если координатору не удастся успешно завершить пакет, то другие узлы, хранящие журнал пакета, смогут его воспроизвести. Каждый узел раз в минуту проверяет, есть ли требующие завершения журналы пакетов в его таблице batchlog. Чтобы дать координатору достаточно времени для завершения начатых пакетов, Cassandra не предпринимает никаких действий с момента, указанного во временной метке команды начала пакета, пока не истечет количество миллисекунд, заданное в свойстве write_request_timeout_in_ms. Пакеты, хранящиеся дольше этого времени, воспроизводятся, а затем удаляются на втором узле. Второй узел хранения журналов пакетов обеспечивает дополнительный уровень избыточности и тем самым высокую надежность механизма пакетов.

Cassandra ограничивает размер пакетов, чтобы предотвратить снижение производительности и стабильности кластера. Этим управляют два свойства в файле *cassandra.yaml*: batch_size_warn_threshold_in_kb определяет, при каком размере в журнал записывается предупреждение о получении большого пакета, а если размер превышает batch_size_fail_threshold_in_kb, то пакет вообще отвергается, и клиент получает сообщение об ошибке. Размер пакета измеряется в терминах длины команды CQL. Порог предупреждения по умолчанию равен 5 КБ, а порог ошибки – 50 КБ.

Чтение

Что касается чтения, то стоит обратить внимание на несколько важных свойств. Прежде всего данные читать легко, потому что клиент может подключиться к любому узлу кластера, даже не зная, хранится на этом узле реплика нужных ему данных или нет. Если на узле, к которому подключился клиент, нет данных, то узел становится коор-

динатором чтения с узла, где эти данные хранятся, а что это за узел, определяется путем анализа диапазонов ключей.

В Cassandra операция чтения обычно медленнее, чем запись. Для выполнения чтения, как правило, требуется произвести поиск на диске, но можно хранить в памяти больше данных, если добавить узлы, использовать вычислительные экземпляры с большим объемом памяти и включить в Cassandra режим кэширования. Кроме того, Cassandra должна ждать результата чтения синхронно (в соответствии с заданными уровнем согласованности и коэффициентом репликации) и при необходимости производить исправление на этапе чтения.

Уровни согласованности при чтении

Уровни согласованности при чтении аналогичны уровням согласованности при записи, но их семантика несколько отличается. Чем выше уровень согласованности, тем больше узлов должно ответить на запрос, и это повышает уверенность в том, что значения, хранящиеся во всех репликах, одинаковы. Если два узла возвращают значения с разными временными метками, то побеждает – и возвращается клиенту – более позднее. После этого Cassandra в фоновом режиме производит так называемое *исправление на этапе чтения* (read repair): заметив, что одна или несколько реплик вернули устаревшее значение, она записывает в них самое актуальное значение, чтобы восстановить согласованность.

В табл. 9.3 показаны уровни согласованности для чтения и их интерпретация.

Как видно из таблицы, для операций чтения уровень согласованности ANY не поддерживается. Отметим, что уровень ONE означает, что клиент получит значение, возвращенное первым ответившим узлом, даже если оно не актуально. Но после возврата данных выполняется исправление на этапе чтения, поэтому все последующие операции чтения вернут согласованное значение вне зависимости от того, какой узел ответит.

Стоит также сказать несколько слов об уровне согласованности ALL. Задав его, вы требуете, что ответили все реплики, поэтому если какой-то узел не работает или не отвечает по любой другой причине, то операция чтения завершается с ошибкой. Узел считается не ответившим, если ответ не получен до истечения тайм-аута, указанного в свойстве `rpc_timeout_in_ms` в конфигурационном файле. По умолчанию тайм-аут равен 10 секундам.

Таблица 9.3. Уровни согласованности чтения

Уровень согласованности	Следствие
ONE, TWO, THREE	Немедленно вернуть клиенту данные, полученные от первого узла (или узлов), ответившего на запрос. Создается фоновый поток, который сверяет данные с другими репликами. Если где-то данные устарели, то производится <i>исправление на этапе чтения</i> для восстановления синхронизации с самым последним значением
LOCAL_ONE	То же, что ONE, но дополнительно требуется, чтобы ответивший узел находился в локальном центре обработки данных
QUORUM	Отправить запрос всем узлам. После того как большинство реплик ($(\text{коэффициент_репликации} / 2) + 1$) ответит, вернуть клиенту значение с последней временной меткой. Затем при необходимости произвести <i>исправление на этапе чтения</i> в остальных репликах
LOCAL_QUORUM	То же, что QUORUM, но ответившие узлы находятся в локальном центре обработки данных
EACH_QUORUM	Гарантируется, что QUORUM узлов ответили в каждом центре обработки данных
ALL	Отправить запрос всем узлам. Дождаться всех ответов и вернуть клиенту значение с последней временной меткой. Затем при необходимости произвести <i>исправление на этапе чтения</i> в фоновом режиме. Если хотя бы один узел не ответил, считается, что при чтении произошла ошибка

Настройка уровней согласованности чтения и записи

Задаваемые в приложении уровни согласованности чтения и записи – пример гибкости Cassandra в выборе компромисса между согласованностью, доступностью и производительностью.

Как было сказано в главе 6, Cassandra может гарантировать строгую согласованность при чтении, если сумма уровней согласованности чтения и записи больше коэффициента репликации. Простой способ добиться этого – задавать для чтения и записи уровень QUORUM. Например, в пространстве ключей с коэффициентом репликации 3 уровень QUORUM означает, что нужен ответ от двух из трех узлов. Поскольку $2 + 2 > 3$, то гарантируется строгая согласованность.

Если вы готовы пожертвовать строгой согласованностью, чтобы увеличить пропускную способность и устойчивость к отказам узлов, то можно указывать меньшие уровни согласованности. Например, если использовать уровень QUORUM для записи и ONE для чтения, то строгая согласованность не гарантируется, поскольку $2 + 1$ всего лишь равно 3.

Действительно, если гарантируется запись только на две из трех реплик, то, безусловно, есть шанс, что одна реплика не записала данные и еще не была исправлена, а при уровне согласованности ONE запрос чтения мог попасть именно этой реплике.

Путь чтения в Cassandra

Теперь посмотрим, что происходит, когда клиент запрашивает данные, т. е. *путь чтения*. Мы опишем путь чтения для запроса единственного ключа раздела и начнем со взаимодействия между узлами, показанного на рис. 9.3.

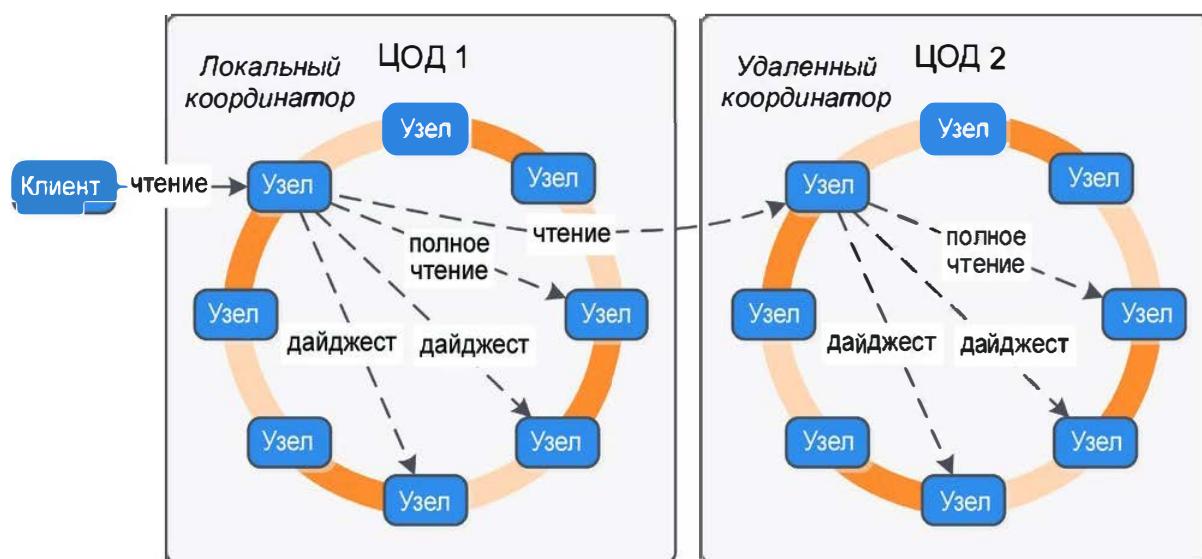


Рис. 9.3 ❖ Взаимодействие между узлами на путях чтения

Путь чтения начинается, когда клиент отправляет запрос узлу-координатору. Как и в случае пути записи, координатор обращается к разделителю для получения информации о репликах и проверяет, достаточно ли реплик для удовлетворения указанного уровня согласованности. На этом сходство с путем записи не заканчивается – если запрос на чтение относится к нескольким ЦОДам, то в каждом ЦОДе выбирается удаленный координатор.

Если координатор сам не является узлом-репликой, то он отправляет запрос реплике, которую динамический осведомитель счител самой быстрой. А остальным репликам отправляется *запрос дайджеста*, отличающийся от обычного запроса на чтение тем, что реплики возвращают не сами данные, а их дайджест, т. е. хэшированное значение.

Координатор вычисляет дайджест данных, возвращенных самой быстрой репликой, и сравнивает его с дайджестами, полученными

от других реплик. Если дайджесты совпадают и требуемый уровень согласованности удовлетворен, то можно вернуть данные от самой быстрой реплики. Если же дайджесты не совпадают, то координатор должен произвести исправление на этапе чтения, как описано в следующем разделе.

На рис. 9.4 показаны взаимодействия, имеющие место внутри каждого узла-реплики при обработке запросов на чтение.

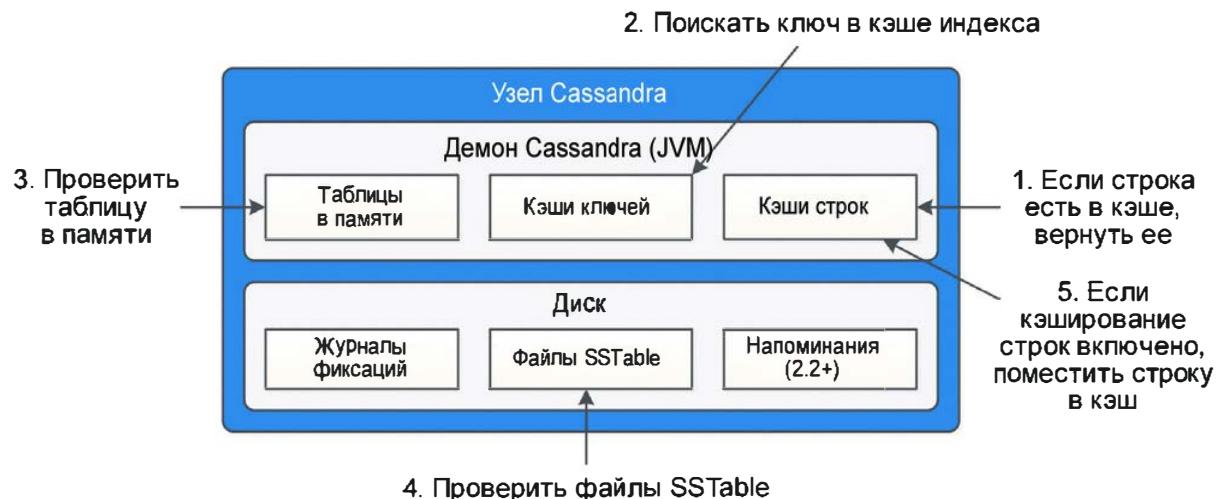


Рис. 9.4 ♦ Взаимодействие внутри узла на пути чтения

Получив запрос на чтение, узел-реплика первым делом заглядывает в кэш строк. Если данные находятся в кэше, их можно вернуть немедленно. Кэш помогает ускорить чтения строк, к которым часто обращаются. Плюсы и минусы кэширования строк мы обсудим в главе 12.

Если данных в кэше нет, то узел-реплика просматривает таблицы в памяти и файлы SSTable. Для любой таблицы существует только одна таблица в памяти, так что эта часть поиска не вызывает затруднений. Однако физических файлов SSTable для одной таблицы Cassandra может быть много, и в каждом из них может храниться часть запрошенных данных.

В Cassandra реализовано несколько механизмов оптимизации поиска в файлах SSTable: кэширование ключей, фильтры Блума, индексы SSTable и сводные индексы.

Для поиска в файлах SSTable на диске сначала используется фильтр Блума, с его помощью определяется, может ли существовать запрошенный раздел в данном файле. Если нет, то и просматривать этот файл SSTable не нужно.



Настройка фильтров Блума

Cassandra хранит фильтры Блума в памяти, хотя, как было упомянуто выше, они хранятся также на диске вместе с файлами данных SSTable, чтобы не пересчитывать их при каждом запуске узла.

Фильтр Блума не гарантирует, что файл SSTable содержит раздел, он говорит лишь, что это не исключено. Для каждой таблицы можно с помощью свойства `bloom_filter_fp_chance` задать процентную долю ложноположительных ответов фильтра Блума. Но за повышение точности приходится платить дополнительным расходом памяти.

Если файл SSTable проходит фильтр Блума, то Cassandra ищет в кэше ключей смещение заданного ключа раздела в SSTable. Кэш ключей реализован в виде словаря, в котором ключом является комбинация дескриптора файла SSTable и ключа раздела, а значением – смещение от начала файла SSTable. Кэш ключей позволяет исключить поиск часто читаемых данных в файлах SSTable, заменив его прямой выборкой.

Если в кэше ключей смещение не найдено, то для поиска смещения Cassandra пользуется хранящимся на диске двухуровневым индексом. Первый уровень индекса называется *сводкой раздела* (*partition summary*) и служит для получения смещения, используемого для поиска ключа раздела в индексе второго уровня – *индексе раздела*. Именно в индексе раздела хранятся смещения ключей раздела в файле SSTable.

Если смещение ключа раздела найдено, то Cassandra читает данные из файла SSTable, начиная с этого смещения.

Получив данные из всех файлов SSTable, Cassandra объединяет их с данными из таблиц в памяти, выбирая для каждого из запрошенных столбцов значение с самой последней временной меткой. Встретившиеся надгробья игнорируются.

Наконец, объединенные данные можно поместить в кэш строк (если режим кэширования включен) и вернуть клиенту или узлу-координатору. Запрос дайджеста обрабатывается почти так же, как обычный запрос на чтение, только в самом конце вычисляется и возвращается дайджест данных, а не сами данные.



Дополнительные сведения о пути чтения

Для получения дополнительных сведений обратитесь к вики-сайту проекта Apache Cassandra по адресу <https://wiki.apache.org/cassandra/ReadPathForUsers>.

Исправление на этапе чтения

Исправление на этапе чтения работает следующим образом: координатор отправляет запрос на чтение всем узлам-репликам. Затем координатор объединяет данные, выбирая правильное значение для каждого запрошенного столбца. Для этого он сравнивает значения, полученные от разных реплик, и возвращает значение с последней временной меткой. Если Cassandra обнаруживает несколько различных значений с одинаковыми временными метками, то сравнивает их лексикографически и оставляет большее. Такая ситуация если и встречается, то исключительно редко. Объединенные данные возвращаются клиенту.

Тем временем координатор асинхронно определяет, какие реплики вернули устаревшие данные, и посыпает им запрос на исправление, включая в него актуальные данные, которые нужно записать взамен устаревших.

Исправление на этапе чтения может производиться как до, так и после возврата данных клиенту. Если указан один из двух уровней строгой согласованности (QUORUM или ALL), то исправление производится до возврата данных. Если же клиент задал уровень слабой согласованности (например, ONE), то исправление производится факультативно в фоновом режиме после возврата данных клиенту. Процентная доля операций чтения, по достижении которой выполняется фоновое исправление, задается на уровне таблицы с помощью свойств `read_repair_chance` и `dc_local_read_repair_chance`.

Запросы по диапазону, упорядочение и фильтрация

До сих пор мы рассматривали только очень простые примеры запросов на чтение. Обратимся теперь к дополнительным фразам в команде `SELECT`, а именно: `WHERE` и `ORDER BY`.

Сначала обсудим использование фразы `WHERE`, с помощью которой Cassandra читает диапазоны данных внутри раздела, иногда называемые *срезками* (*slice*).

Но для демонстрации запросов по диапазону сначала надо бы иметь какие-то данные. Сейчас у нас данных немного, но этот недостаток легко исправить, воспользовавшись одним из средств массовой загрузки для Cassandra.

Способы массовой загрузки

При работе с Cassandra часто возникает необходимость загрузить в кластер тестовые или эталонные данные. По счастью, существуют два способа быстрой загрузки и выгрузки данных.

cqlsh поддерживает загрузку и выгрузку файлов в формате CSV с помощью команды COPY.

Например, следующая команда сохраняет в файле содержимое таблицы hotels:

```
cqlsh:hotel> COPY hotels TO 'hotels.csv' WITH HEADER=TRUE;
```

Параметр TO задает имя выходного файла, а если параметр HEADER равен TRUE, то в первой строке выходного файла будут перечислены имена столбцов. Созданный файл можно отредактировать и загрузить обратно такой командой:

```
cqlsh:hotel> COPY hotels FROM 'hotels.csv' WITH HEADER=true;
```

В команде COPY можно задавать и другие параметры форматирования, описывающие обработку кавычек, экранирование и представление даты и времени.

Брайан Хесс (Brian Hess) написал командную утилиту Cassandra Loader (<https://github.com/brianmhess/cassandra-loader>) для загрузки и выгрузки файлов в формате CSV и с другими разделителями. Она, кстати, понимает десятичные числа с запятой, а не точкой в качестве разделителя.

Воспользуемся cqlsh для загрузки в кластер тестовых данных об отелях. CSV-файл *available_rooms.csv* можно скачать из репозитория книги на GitHub (<https://github.com/jeffreyscarpenter/cassandra-guide>). Он содержит данные за месяц для двух небольших отелей по пять номеров в каждом. Загрузим их в кластер:

```
cqlsh:hotel> COPY available_rooms_by_hotel_date FROM  
'available_rooms.csv' WITH HEADER=true;
```

```
310 rows imported in 0.789 seconds.
```

Прочитав данные, вы обнаружите два отеля: «AZ123» и «NY229».

Теперь посмотрим, как выполнить запрос, который в главе 5 мы обозначили «Q4. Найти свободный номер в заданном диапазоне дат». Напомним, что для поддержки этого запроса мы создали таблицу *available_rooms_by_hotel_date* с первичным ключом

```
PRIMARY KEY (hotel_id, date, room_number)
```

Это означает, что `hotel_id` – ключ раздела, а `date` и `room_number` – кластерные столбцы.

Следующая команда CQL ищет номера по идентификатору отеля и диапазону дат:

```
cqlsh:hotel> SELECT * FROM available_rooms_by_hotel_date
  WHERE hotel_id='AZ123' and date>'2016-01-05' and date<'2016-01-12';

  hotel_id | date      | room_number | is_available
-----+-----+-----+-----+
    AZ123 | 2016-01-06 |      101 |      True
    AZ123 | 2016-01-06 |      102 |      True
    AZ123 | 2016-01-06 |      103 |      True
    AZ123 | 2016-01-06 |      104 |      True
    AZ123 | 2016-01-06 |      105 |      True
...
(60 rows)
```

Заметим, что в запросе участвуют ключ раздела `hotel_id` и диапазон, задающий границы кластерного ключа `date`.

Чтобы найти записи для номера 101 в отеле AZ123, можно было бы попытаться выполнить такой запрос:

```
cqlsh:hotel> SELECT * FROM available_rooms_by_hotel_date
  WHERE hotel_id='AZ123' and room_number=101;
InvalidRequest: code=2200 [Invalid query] message="PRIMARY KEY column
  "room_number" cannot be restricted as preceding column "date" is not
  restricted"
```

Но запрос завершается с ошибкой, потому что мы попытались ограничить значение второго кластерного ключа, не наложив ограничения на значение первого.

Фраза `WHERE` должна строиться с соблюдением следующих правил:

- должны быть заданы все элементы ключа раздела;
- если на какой-то кластерный ключ наложено ограничение, то должны быть ограничения и на все предшествующие ему кластерные ключи.

Эти ограничения объясняются тем, как Cassandra хранит данные на диске: в том порядке кластерных столбцов, который был задан в команде `CREATE TABLE`. На кластерные столбцы можно накладывать лишь такие ограничения, которые позволяют Cassandra выбирать только соседние строки.

Исключением из этого правила является фраза `ALLOW FILTERING`, которая позволяет опускать элемент ключа раздела. Например, можно поискать свободные номера в любом отеле на указанную дату:

```
cqlsh:hotel> SELECT * FROM available_rooms_by_hotel_date
  WHERE date='2016-01-25' ALLOW FILTERING;

  hotel_id | date      | room_number | is_available
-----+-----+-----+-----+
    AZ123 | 2016-01-25 |       101 |      True
    AZ123 | 2016-01-25 |       102 |      True
    AZ123 | 2016-01-25 |       103 |      True
    AZ123 | 2016-01-25 |       104 |      True
    AZ123 | 2016-01-25 |       105 |      True
    NY229 | 2016-01-25 |       101 |      True
    NY229 | 2016-01-25 |       102 |      True
    NY229 | 2016-01-25 |       103 |      True
    NY229 | 2016-01-25 |       104 |      True
    NY229 | 2016-01-25 |       105 |      True

(10 rows)
```

Однако использовать ALLOW FILTERING не рекомендуется, потому что такие запросы могут исполняться очень долго. Обнаружив, что по-другому не получается, пересмотрите модель данных и подумайте о том, как спроектировать таблицы для поддержки соответствующих запросов.

Оператор IN можно использовать для сравнения столбца с несколькими значениями. Вот, например, как найти свободные номера на две даты, отстоящие друг от друга на неделю:

```
cqlsh:hotel> SELECT * FROM available_rooms_by_hotel_date
  WHERE hotel_id='AZ123' AND date IN ('2016-01-05', '2016-01-12');
```

Отметим, что запросы с оператором IN могут выполняться медленнее, потому что заданные значения могут соответствовать несмежным областям в строке.

Наконец, команда SELECT позволяет переопределить порядок сортировки столбцов, заданный при создании таблицы. Так, в любом из показанных выше запросов можно было бы вернуть номера в порядке убывания дат, добавив фразу ORDER BY:

```
cqlsh:hotel> SELECT * FROM available_rooms_by_hotel_date
  WHERE hotel_id='AZ123' and date>'2016-01-05' and date<'2016-01-12'
  ORDER BY date DESC;
```

Функции и агрегаты

В версии Cassandra 2.2 появились две возможности, позволяющие клиентам переложить часть работы на узел-координатор: пользова-

тельские функции (user-defined functions – UDF) и пользовательские агрегаты (UDA). В некоторых ситуациях они повышают производительность за счет уменьшения количества возвращаемых клиенту данных и сокращения объема обработки на стороне клиента (правда, ценой увеличения нагрузки на сервер).

Пользовательские функции

Пользовательские функции применяются к хранимым данным прямо на узлах Cassandra в процессе обработки запроса. Но предварительно необходимо разрешить использование пользовательских функций в кластере, добавив в файл *cassandra.yaml* на каждом узле такую строку:

```
enable_user_defined_functions: true
```

Принцип простой: мы создаем пользовательскую функцию с помощью команды CQL CREATE FUNCTION, которая распространяет ее на все узлы кластера. При выполнении запроса, в котором эта функция встречается, она применяется к каждой строке результата.

Для примера создадим пользовательскую функцию для подсчета числа свободных номеров в таблице *available_rooms_by_hotel_date* table:

```
cqlsh:hotel> CREATE FUNCTION count_if_true(input boolean)
    RETURNS NULL ON NULL INPUT
    RETURNS int
    LANGUAGE java AS 'if (input) return 1; else return 0;';
```

Разберемся, что здесь написано. Мы создали пользовательскую функцию *count_if_true*, которая получает на входе параметр типа *boolean* и возвращает целое число. Мы включили проверку на *null*, чтобы функция работала корректно, если входное значение не определено. Отметим, что если пользовательская функция завершается с ошибкой, то выполнение запроса прерывается, так что эта проверка далеко не лишняя.



Безопасность пользовательских функций

Начиная с версии 3.0 код пользовательских функций выполняется в песочнице, чтобы усложнить вредоносным функциям получение неавторизованного доступа к среде исполнения Java.

Отметим далее, что во фразе LANGUAGE указано, что функция написана на Java. Cassandra изначально поддерживает функции и агрегаты, написанные на Java или JavaScript. Но вообще они могут быть реализованы на любом языке, поддерживающем Java Scripting API, опре-

деленный в документе JSR 223, в т. ч. на Python, Ruby и Scala. Только нужно включить дополнительный JAR-файл, содержащий реализацией дополнительного скриптового движка, в переменную CLASSPATH.

Наконец, после ключевого слова AS идет собственно код функции на Java. Сама по себе функция тривиальна – она просто подсчитывает значения, равные true. Но скоро мы напишем функцию посложнее.

Однако сначала применим эту функцию к таблице available_rooms_by_hotel_date и посмотрим на результат:

```
cqlsh:hotel> SELECT room_number, count_if_true(is_available)
  FROM available_rooms_by_hotel_date
 WHERE hotel_id='AZ123' and date='2016-01-05';

  room_number | hotel.count_if_true(is_available)
-----+-----
    101 |           1
    102 |           1
    103 |           1
    104 |           1
    105 |           1

(5 rows)
```

Как видим, столбец, содержащий результат нашей функции, квалифицирован пространством ключей hotel. Объясняется это тем, что любая пользовательская функция ассоциирована с конкретным пространством ключей. Если бы мы выполнили такой запрос с помощью драйвера DataStax для Java, то в каждой строке Row обнаружили бы столбец Column с именем hotel_count_if_true_is_available.

Пользовательские агрегаты

Как мы только что узнали, пользовательские функции применяются к одной строке. Для выполнения операций над несколькими строками служат пользовательские агрегаты. Пользовательский агрегат состоит из двух пользовательских функций: функции состояния и необязательной финальной функции. Функция состояния выполняется для каждой строки, а финальная функция, если она присутствует, применяется к результату, вычисленному функцией состояния.

На простом примере рассмотрим, как это работает. Сначала определим функцию состояния. Функция count_if_true – почти то, что нужно, но внесем в нее небольшое изменение, которое позволит суммировать значения по нескольким строкам. Новая функция принимает накопительный итог, увеличивает его и возвращает результат.

```
cqlsh:hotel> CREATE FUNCTION state_count_if_true(total int, input boolean)
    RETURNS NULL ON NULL INPUT
    RETURNS int
    LANGUAGE java AS 'if (input) return total+1; else return total;';
```

Отметим, что первым передается параметр `total`, а его тип совпадает с типом возвращаемого значения функции (`int`). Чтобы пользовательская функция могла выступать в роли функции состояния, типы первого параметра и возвращаемого значения должны совпадать. Второй параметр типа `boolean` интерпретируется так же, как в исходной функции `count_if_true`.

Теперь можно создать агрегат на основе этой функции состояния.

```
cqlsh:hotel> CREATE AGGREGATE total_available (boolean)
    SFUNC state_count_if_true
    STYPE int
    INITCOND 0;
```

Рассмотрим эту команду по частям. В первой строке объявлен пользовательский агрегат с именем `total_available`, применяемый к столбцам типа `boolean`.

После ключевого слова `SFUNC` задается функция состояния – в данном случае `state_count_if_true`.

Затем – после ключевого слова `STYPE` – мы указываем тип переменной, в которой функция состояния будет накапливать результат. Cassandra запоминает эту переменную и передает ее функции состояния при обработке каждой строки. Тип `STYPE` должен совпадать с типом первого параметра и возвращаемого значения функции состояния. После ключевого слова `INITCOND` задается начальное значение результата, в данном случае оно равно нулю.

В этом примере финальной функции нет, но если она задана, то должна принимать аргумент типа `STYPE` и возвращать значение любого типа, например принимать целое число и возвращать значение типа `boolean`, показывающее, что уровень запасов на складе слишком низкий и пора отправлять оповещение.

Теперь воспользуемся этим агрегатом для подсчета числа свободных номеров. Отметим, что запрос должен включать только пользовательский агрегат и никаких других столбцов или функций.

```
cqlsh:hotel> SELECT total_available(is_available)
    FROM available_rooms_by_hotel_date
    WHERE hotel_id='AZ123' and date='2016-01-05';
```

```
hotel.total_available(is_available)
-----
      5
(1 rows)
```

Как видим, для указанного отеля на указанную дату есть пять свободных номеров.



Дополнительные команды для работы с UDF/UDA

При создании пользовательских функций и агрегатов можно включать фразу IF NOT EXISTS, чтобы избежать ошибок при создании объектов с такой же сигнатурой. Можно вместо этого воспользоваться командой CREATE OR REPLACE, которая перезаписывает функцию или агрегат, если таковые уже существуют.

Команды DESCRIBE FUNCTIONS и DESCRIBE AGGREGATES позволяют узнать, какие функции и агрегаты уже определены. Это особенно полезно, если существует функция с таким же именем, но другой сигнатурой. Наконец, для удаления пользовательских функций и агрегатов предназначены команды DROP FUNCTION и DROP AGGREGATE соответственно.

Встроенные функции и агрегаты

Помимо пользовательских функций и агрегатов, Cassandra предоставляет ряд встроенных.

COUNT

Функция COUNT подсчитывает число строк, возвращенных запросом. Вот, например, как подсчитать число отелей в нашей базе данных:

```
SELECT COUNT(*) FROM hotel.hotels;
```

Эту команду можно также использовать для подсчета числа значений, отличных от null, в указанном столбце. Например, следующий запрос возвращает количество гостей, указавших свой адрес электронной почты:

```
SELECT COUNT(emails) FROM reservation.guests;
```

MIN и MAX

Эти функции служат для вычисления минимального и максимального значений столбца, указанного в запросе. Так, следующий запрос вычисляет минимальную и максимальную продолжительности пребывания (в ноках) для номеров, забронированных в указанном отеле на указанную дату:

```
SELECT MIN(nights), MAX(nights) FROM reservations_by_hotel_date
  WHERE hotel_id='AZ123' AND start_date='2016-09-09';
```

`sum`

Используется для суммирования значений столбца, указанного в запросе. Можно было бы вычислить, на сколько всего ночей забронированы номера:

```
SELECT SUM(nights) FROM reservations_by_hotel_date
  WHERE hotel_id='AZ123' AND start_date='2016-09-09';
```

`avg`

Вычисляет среднее всех значений столбца, указанного в запросе. Чтобы получить среднюю длительность пребывания, можно написать такой запрос:

```
SELECT AVG(nights) FROM reservations_by_hotel_date
  WHERE hotel_id='AZ123' AND start_date='2016-09-09';
```

Технически встроенные агрегаты являются частью пространства ключей `system`. Поэтому имя столбца, содержащего результаты, имеет вид `system_avg_nights`.

Разбиение на страницы

В ранних версиях Cassandra клиент должен был ограничивать объем данных, возвращаемых за один раз. Слишком большой результирующий набор мог бы довести узлы и клиентов до исчерпания всей доступной памяти.

По счастью, Cassandra предоставляет механизм разбиения на страницы, который позволяет извлекать результат по частям. Пример ниже иллюстрирует использование ключевого слова CQL `LIMIT`. Следующая команда вернет не более 100 отелей:

```
cqlsh> SELECT * FROM hotel.hotels LIMIT 100;
```

Разумеется, у `LIMIT` есть недостаток – невозможность получить строки сверх запрошенного числа.

В версии Cassandra 2.0 появился механизм *автоматического разбиения на страницы*. Он позволяет клиенту указать, какое подмножество результата следует вернуть в ответ на запрос. Сервер сам разбивает результат на страницы и возвращает их по мере поступления запросов от клиента.

В `cqlsh` узнать состояние режима разбиения на страницы можно с помощью команды `PAGING`. Ниже приведена последовательность

команд: проверить, включено ли разбиение на страницы, изменить размер порции (размер страницы), выключить и снова включить разбиение на страницы:

```
cqlsh> PAGING;
Query paging is currently enabled. Use PAGING OFF to disable
Page size: 100
cqlsh> PAGING 1000;
Page size: 1000
cqlsh> PAGING OFF;
Disabled Query paging.
cqlsh> PAGING ON;
Now Query paging is enabled
```

Теперь посмотрим, как разбиение на страницы работает в драйвере DataStax для Java. Размер порции по умолчанию можно задать глобально для экземпляра Cluster:

```
Cluster cluster = Cluster.builder().addContactPoint("127.0.0.1").
    withQueryOptions(new QueryOptions().setFetchSize(2000)).build();
```

Можно установить размер порции и для отдельной команды, переопределив значение по умолчанию:

```
Statement statement = new SimpleStatement("...");
statement.setFetchSize(2000);
```

Размер порции, заданный для команды, имеет приоритет над размером для всего кластера (по умолчанию 5000). Отметим, что задание размера порции не означает, что Cassandra всегда будет возвращать в точности такое число строк, иногда их может быть чуть больше или чуть меньше.

Драйвер автоматически управляет автоматическим разбиением на страницы от нашего имени, т. е. позволяет обходить результирующий набор ResultSet, ничего не зная о том, что он разбит на страницы. В следующем примере перебираются все возвращенные отели:

```
SimpleStatement hotelSelect = session.newSimpleStatement(
    "SELECT * FROM hotels");

ResultSet resultSet = session.execute(hotelSelect);

for (Row row : resultSet) {
    // обработать строку
}
```

За кулисами же происходит вот что: когда приложение вызывает метод `session.execute()`, драйвер обращается к Cassandra, запрашивая первую страницу результатов. Приложение обходит полученные строки в цикле `for`, а драйвер, заметив, что текущая страница закончилась, запрашивает следующую.

Может случиться, что небольшая задержка, связанная с запросом следующей страницы, раздражает пользователей, поэтому класс `ResultSet` предоставляет дополнительные средства более точного контроля над разбиением на страницы. Ниже показано, как можно осуществить заблаговременную выборку строк:

```
for (Row row : resultSet) {
    if (resultSet.getAvailableWithoutFetching() < 100 &&
        !resultSet.isFullyFetched())
        resultSet.fetchMoreResults();
    // обработать строку
}
```

Метод `getAvailableWithoutFetching()` проверяет, сколько строк осталось на текущей странице. Если меньше 100 и есть еще страницы (это определяет метод `isFullyFetched()`), то вызывается асинхронный метод выборки дополнительных строк `fetchMoreResults()`.

Драйвер также предоставляет возможность получить состояние разбиения на страницы напрямую, чтобы его можно было сохранить и использовать позже. Это полезно, если приложение является веб-службой без сохранения состояния, не поддерживающей многократных обращений в одном сеансе.

Получить состояние разбиения на страницы можно от объекта `ExecutionInfo`, являющегося атрибутом `ResultSet`:

```
PagingState nextPage = resultSet.getExecutionInfo().getPagingState();
```

Это состояние можно сохранить в приложении или вернуть клиенту. Объект `PagingState` можно преобразовать в строку методом `toString()` или в массив байтов методом `toBytes()`.

Отметим, что состоянием в форме строки или массива байтов не следует манипулировать или пытаться использовать в другом методе. Это приведет к исключению `PagingStateException`.

Чтобы возобновить запрос в заданном состоянии `PagingState`, мы передаем его объекту команды:

```
SimpleStatement hotelSelect = session.newSimpleStatement(
    "SELECT * FROM hotels");
hotelSelect.setPagingState(pagingState);
```

Упреждающее выполнение

В главе 8 мы уже упоминали предоставляемый драйвером DataStax для Java класс SpeculativeExecutionPolicy, который заранее повторяет запросы на чтение на разных узлах, если от исходного узла не получен ответ в течение настраиваемого промежутка времени.

Любой узел можно сконфигурировать так, что, выступая в роли координатора, он сможет инициировать упреждающие запросы другим узлам. Это поведение настраивается на уровне таблицы с помощью свойства speculative_retry, принимающего следующие значения:

ALWAYS

Повторять запросы на чтение на всех репликах.

<X>PERCENTILE

Инициировать упреждающие запросы, если ответ не получен в течение X-го процентиля отведенного времени.

<Y>ms

Повторять, если ответ не получен в течение Y миллисекунд.

NONE

Не повторять запросов на чтение.

По умолчанию подразумевается значение 99.0PERCENTILE. При этом достигается хороший баланс за счет ускорения аномально медленных запросов, но без затопления кластера огромным количеством повторных запросов на чтение.

Эта функциональность, известная также под названием *защита от быстрого чтения* (rapid read protection), была впервые введена в версии 2.0.2. Отметим, что она никак не отражается на запросах с уровнем согласованности ALL, поскольку в этом случае дополнительных узлов, на которых можно было бы повторить запрос, просто нет.

Удаление

Удаление данных в Cassandra производится не так, как в реляционных базах данных. В РСУБД мы просто выполняем команду удаления, указывая, какие строки удалить. В Cassandra удаление не приводит к немедленному стиранию данных. И тому существует простое объяснение: Cassandra – распределенная, согласованная в конечном счете база. Если бы удаление производилось, как обычно, то узлы, которые не работали в момент выполнения команды удаления, о ней никогда

бы не узнали, поэтому данные на них остались. Когда такой узел «оживает», он ошибочно считает, что узлы, выполнившие запрос на удаление, на самом деле пропустили команду записи, и начнет исправлять данные на них. Поэтому Cassandra нуждается в более сложном механизме поддержки удаления. Он основан на *надгробьях* (*tombstone*).

Надгробье – это специальный маркер, который перезаписывает удаленные значения. Если какая-то реплика не получила запроса на удаление, то впоследствии, когда она снова станет доступной, маркер будет записан и на нее. Таким образом, сразу после удаления объем хранимых данных не уменьшается. Каждый узел следит за возрастом своих надгробий. По истечении *gc_grace_seconds* секунд (по умолчанию 10 суток) запускается процедура уплотнения, надгробья становятся жертвой сборщика мусора, и место на диске освобождается.

Поскольку файлы SSTable неизменяемы, данные из них не удаляются. Процедура уплотнения учитывает надгробья, сортирует объединенные данные, создает новый индекс над отсортированными данными, после чего записывает только что объединенные, отсортированные и проиндексированные данные в новый файл. Предполагается, что десяти дней достаточно, чтобы восстановить работоспособность отказавшего узла. При желании можно уменьшить период ожидания, чтобы место на диске освобождалось быстрее.

В драйвере DataStax для Java простая команда удаления всей строки выглядит так:

```
SimpleStatement hotelDelete = session.newSimpleStatement(
    "DELETE * FROM hotels WHERE id=?",
    "AZ123");

ResultSet hotelDeleteResult = session.execute(hotelDelete);
```

Можно удалять и отдельные столбцы, не входящие в состав первичного ключа, указав в запросе их имена.

Удаление можно производить также с помощью объектов PreparedStatement, QueryBuilder и MappingManager.

Вот пример удаления всей строки с помощью QueryBuilder:

```
BuiltStatement hotelDeleteBuilt = queryBuilder.delete().all().
    from("hotels").where(eq("id", "AZ123"));

session.execute(hotelDeleteBuilt);
```



Уровни согласованности при удалении

Поскольку удаление – разновидность записи, то для операций удаления определены такие же уровни согласованности, как для записи.

Резюме

В этой главе мы видели, как читать, записывать и удалять данные в cqlsh и с помощью клиентских драйверов. Мы также заглянули за кулисы и обсудили, как эти операции реализованы в Cassandra. Знать это полезно для принятия обоснованных решений при проектировании, реализации, развертывании и обслуживании приложений.

Глава 10

Мониторинг

В этой главе мы научимся использовать различные инструменты мониторинга, которые позволяют наблюдать за важными событиями в жизненном цикле кластера Cassandra. Мы рассмотрим некоторые простые способы узнать, что происходит, например, при изменении уровня протоколирования и анализе информации в журнале.

В Cassandra также встроена поддержка технологии Java Management Extensions (JMX), что позволяет получать больше информации об узлах Cassandra и работающей на них среде исполнения Java. Благодаря JMX мы можем наблюдать за состоянием базы данных и текущими событиями, а также удаленно взаимодействовать с ней для оптимизации некоторых параметров. JMX – важная часть Cassandra, и мы потратим время на то, чтобы понять, как эта технология работает и что именно Cassandra раскрывает для мониторинга и управления средствами JMX. Итак, приступим!

Протоколирование

Самый простой способ составить представление о происходящем в базе данных – изменить уровень протоколирования, чтобы выводилось больше информации. Это очень полезно для разработки и изучения того, что происходит внутри Cassandra.

Для протоколирования в Cassandra используются библиотека Simple Logging Facade для Java (SLF4J) и Logback в качестве конкретной реализации. SLF4J представляет собой фасад, за которым могут стоять различные библиотеки протоколирования, включая Logback, Log4J и встроенное в Java средство протоколирования (`java.util.logging`). Дополнительные сведения о Logback можно найти на сайте <http://logback.qos.ch/>.

По умолчанию на сервере Cassandra установлен уровень протоколирования INFO, который не дает подробной информации о том, что делала Cassandra в каждый момент времени. Выводятся лишь основные сообщения об изменении состояния:

```
INFO [main] 2015-09-19 09:40:20,215 CassandraDaemon.java:149 -  
    Hostname: Carp-iMac27.local  
INFO [main] 2015-09-19 09:40:20,233 YamlConfigurationLoader.java:92 -  
    Loading settings from file:/Users/jeff/Cassandra/  
        apache-cassandra-2.1.8/conf/cassandra.yaml  
INFO [main] 2015-09-19 09:40:20,333 YamlConfigurationLoader.java:135 -  
    Node configuration  
...
```

При запуске Cassandra в терминале с флагом -f (в приоритетном режиме) вся эта информация будет отображаться в окне терминала. И одновременно она записывается в файл журнала для последующего просмотра.

Если задать уровень протоколирования DEBUG, то сведения о работе сервера будут гораздо подробнее.

Чтобы изменить уровень протоколирования, откройте файл *<cassandra-home>/conf/logback.xml* и найдите в нем такую секцию:

```
<root level="INFO">  
    <appender-ref ref="FILE" />  
    <appender-ref ref="STDOUT" />  
</root>
```

Измените первую строчку:

```
<root level="DEBUG">
```

Вскоре после сохранения файла Cassandra начнет протоколирование с уровнем DEBUG, поскольку по умолчанию система протоколирования просматривает конфигурационный файл раз в минуту. Этот режим просмотра задается в строке

```
<configuration scan="true">
```

Теперь Cassandra сообщает о своей работе гораздо больше. Мы точно знаем, что и когда делала Cassandra, и это очень помогает при отладке, да и просто для того, чтобы лучше понимать, как живет система.



Настройка протоколирования в производственной среде

Конечно, в производственной среде следует установить уровень протоколирования WARN или ERROR, поскольку подробная диагностика сильно замедляет работу.

По умолчанию журналы Cassandra находятся в подкаталоге *logs* установочного каталога.

Чтобы изменить место их расположения, найдите в файле *logback.xml* строку

```
<file>${cassandra.logdir}/system.log</file>
```

и введите другое имя файла.



Если отсутствуют файлы журналов

Не увидев журналов в указанном месте, проверьте, что вы являетесь владельцем каталога или хотя бы имеете права на его чтение и запись. Cassandra ничего не скажет, если запись в журнал невозможна, она просто ничего не будет писать. К файлам данных это тоже относится.

В файле *logback.xml* имеются параметры, описывающие смену журналов. По умолчанию файл *system.log* архивируется, когда достигнет размера 20 МБ. Архив хранится в сжатом виде и получает имя вида *system.log.1.zip*, *system.log.2.zip* и т. д.

Динамическое наблюдение за журналом

Не обязательно запускать Cassandra в приоритетном режиме, чтобы видеть журнал по мере его формирования. Можно запустить ее и без флага *-f*, а потом просматривать журнал с помощью утилиты *tail*. Эта программа не является частью Cassandra, она имеется во всех дистрибутивах Linux и показывает новые строки файла по мере их добавления.

Для динамического наблюдения за журналом запустите Cassandra командой

```
$ bin/cassandra
```

Затем откройте второе окно терминала и введите команду *tail*, передав ей путь к интересующему вас файлу:

```
$ tail -f $CASSANDRA_HOME/logs/system.log
```

Флаг *-f* означает «follow» – как только Cassandra записывает что-то в файл журнала, *tail* сразу же выводит это на экран. Для завершения программы просто нажмите **Ctrl+C**.

В Windows это тоже возможно, но программы *tail* в штатной поставке Windows нет. Поэтому придется скачать и установить Cygwin (<http://www.cygwin.com/>) – бесплатный и открытый эмулятор обо-

лочки Bash. Cygwin предоставляет интерфейс в стиле Linux и целый ряд программ Linux, работающих в Windows.

После этого можете запустить Cassandra, как обычно, и следить за журналом с помощью tail:

```
$ tail -f %CASSANDRA_HOME%\logs\system.log
```

Содержимое журнала будет отображаться на консоли так, будто сервер запущен в приоритетном режиме.

Изучение журналов

Если сервер запущен с отладочным протоколированием, то в журнал выводится огромный объем информации о том, что происходит; для отладки это весьма полезно. Например, вот что мы увидим после записи в базу простого значения в cqlsh:

```
cqlsh> INSERT INTO hotel.hotels (id, name, phone, address)
... VALUES ( 'AZ123', 'Comfort Suites Old Town Scottsdale',
... '(480) 946-1111', { street : '3275 N. Drinkwater Blvd.',
... city : 'Scottsdale', state : 'AZ', zip_code : 85251 });

DEBUG [SharedPool-Worker-1] 2015-09-30 06:21:41,410 Message.java:506 -
Received: OPTIONS, v=4
DEBUG [SharedPool-Worker-1] 2015-09-30 06:21:41,410 Message.java:525 -
Responding: SUPPORTED {COMPRESSION=[snappy, lz4],
CQL_VERSION=[3.3.1]}, v=4
DEBUG [SharedPool-Worker-1] 2015-09-30 06:21:42,082 Message.java:506 -
Received: QUERY INSERT INTO hotel.hotels (id, name, phone, address)
VALUES ( 'AZ123', 'Comfort Suites Old Town Scottsdale',
'(480) 946-1111', { street : '3275 N. Drinkwater Blvd.',
city : 'Scottsdale', state : 'AZ', zip_code : 85251 });
[pageSize = 100], v=4
DEBUG [SharedPool-Worker-1] 2015-09-30 06:21:42,086
AbstractReplicationStrategy.java:87 - clearing cached endpoints
DEBUG [SharedPool-Worker-1] 2015-09-30 06:21:42,087 Tracing.java:155 -
request complete
DEBUG [SharedPool-Worker-1] 2015-09-30 06:21:42,087 Message.java:525 -
Responding: EMPTY RESULT, v=4
```

Эта информация не так интересна, как могла бы быть, потому что система работает в кластере из одного узла.

Теперь прочитаем строку:

```
cqlsh> SELECT * from hotel.hotels;
```

Сервер протоколирует запрос в журнале:

```
DEBUG [SharedPool-Worker-1] 2015-09-30 06:27:27,392 Message.java:506 -
    Received: QUERY SELECT * from hotel.hotels;[pageSize = 100], v=4
DEBUG [SharedPool-Worker-1] 2015-09-30 06:27:27,395
    StorageProxy.java:2021 - Estimated result rows per range: 0.0;
    requested rows: 100, ranges.size(): 257; concurrent range requests: 1
DEBUG [SharedPool-Worker-1] 2015-09-30 06:27:27,401
    ReadCallback.java:141 - Read: 0 ms.
DEBUG [SharedPool-Worker-1] 2015-09-30 06:27:27,401 Tracing.java:155 -
    request complete
DEBUG [SharedPool-Worker-1] 2015-09-30 06:27:27,401 Message.java:525 -
    Responding: ROWS [id(hotel, hotels),
    org.apache.cassandra.db.marshall.UUIDType] [address(hotel, hotels),
    org.apache.cassandra.db.marshall.UserType(hotel, 61646472657373,
    737472656574:org.apache.cassandra.db.marshall.UTF8Type,
    63697479:org.apache.cassandra.db.marshall.UTF8Type, 7374617465:
    org.apache.cassandra.db.marshall.UTF8Type, 7a69705f636f6465:
    org.apache.cassandra.db.marshall.Int32Type)] [name(hotel, hotels),
    org.apache.cassandra.db.marshall.UTF8Type] [phone(hotel, hotels),
    org.apache.cassandra.db.marshall.UTF8Type] [pois(hotel, hotels),
    org.apache.cassandra.db.marshall.SetType(org.apache.cassandra.db.
    marshal.UUIDType)]
| 452d27e1-804e-479b-aeaf-61dlfa31090f | 3275 N. Drinkwater Blvd.:
Scottsdale:AZ:85251 | Comfort Suites Old Town Scottsdale | (480) 946-1111 | null
```

Как видим, сервер загружает запрошенные столбцы с помощью класса, отвечающего за преобразование данных из дискового формата.

При уровне протоколирования DEBUG выводится достаточно информации, чтобы следить за тем, что делает сервер в ответ на ваши действия.

Мониторинг Cassandra средствами JMX

В этом разделе мы рассмотрим, как в Cassandra используется технология Java Management Extensions (JMX) для удаленного управления серверами. JMX впервые была специфицирована в документе Java Specification Request (JSR) 160 и окончательно вошла в состав Java начиная с версии 5.0.



Дополнительные сведения о JMX

Ознакомиться с реализацией JMX в Java можно, изучив пакет `java.lang.management`.

JMX – это Java API, предоставляющий средства для управления приложениями двумя основными способами. Во-первых, JMX позволяет оценить состояние приложения и общую производительность в терминах потребляемой памяти, числа потоков и использования ресурсов процессора – это вещи, относящиеся к любому Java-приложению. Во-вторых, JMX позволяет работать с конкретными аспектами приложения, оснащенного средствами контроля.

Под *оснащением* понимается обертывание кода приложения специальным кодом взаимодействия с JVM, который позволяет JVM собирать сведения о программе и предоставлять их внешним инструментальным средствам: агентам мониторинга, средствам анализа данных, профилировщикам и т. д. JMX позволяет не только просматривать такие данные, но и – если приложение разрешает – управлять приложением во время его работы, изменяя значения параметров.

JMX нередко используется для различных операций управления приложением, в том числе:

- обнаружения приближающегося исчерпания памяти, в частности размера каждого поколения в куче;
- обнаружения взаимоблокировок потоков, определения числа потоков в пиковые периоды и текущего числа активных потоков;
- подробной трассировки работы загрузчика классов;
- управления уровнем протоколирования;
- получения общей информации, например времени работы приложения с момента запуска и активного списка путей к классам.

Средствами контроля оснащены многие популярные Java-приложения, в т. ч. сервер приложений Tomcat и Cassandra. Архитектура JMX изображена на рис. 10.1.

Архитектура JMX проста. JVM получает информацию от операционной системы. Сама JVM оснащена средствами контроля, поэтому многие аспекты ее функциональности доступны для управления, как описано выше. Оснащенное Java-приложение (например, Cassandra) работает поверх JVM и также раскрывает свою функциональность в виде управляемых объектов. В состав JDK входит MBean-сервер, который делает оснащенную функциональность доступной управляющему приложению на основе JMX по удаленному протоколу. JVM также предоставляет средства управления по протоколу Simple Network Monitoring Protocol (SNMP), это может быть полезно, если вы работаете со средствами мониторинга на основе SNMP, например Nagios или Zenoss.

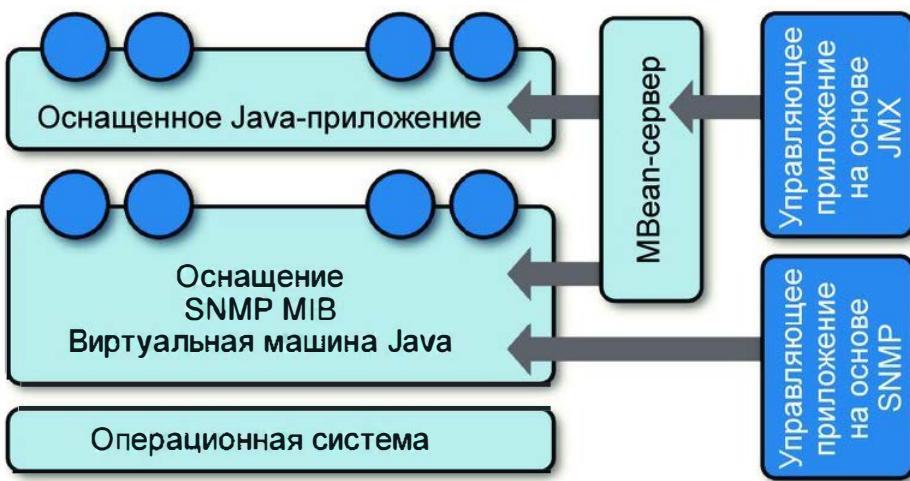


Рис. 10.1 ♦ Архитектура JMX

Но управлять можно лишь тем, что открыли разработчики приложения. К счастью, разработчики Cassandra оснастили средствами контроля многие части движка базы данных, так что управлять им с помощью JMX довольно просто.

Для оснащения Java-приложения его части обертываются управляемыми bean-объектами (MBean).

Подключение к Cassandra через JConsole

Программа jconsole входит в стандартный комплект средств разработчика Java (JDK). Она предлагает графический интерфейс для работы с MBean-объектами и может применяться как для локального, так и для удаленного управления. Давайте подключимся к JMX-порту Cassandra с помощью JConsole. Для этого откройте новое окно терминала и введите команду:

```
>jconsole
```

Появится экран, показанный на рис. 10.2.

Если вы занимаетесь мониторингом узла на локальной машине, то просто дважды щелкните мышью по строке `org.apache.cassandra.service.CassandraDaemon` в разделе Local Process. Если же вы собираетесь следить за узлом на другой машине, то выберите переключатель Remote Process, а затем введите имя хоста и номер порта, к которому подключаетесь. По умолчанию Cassandra JMX прослушивает порт 7199, так что можете ввести строку вида

```
>lucky:7199
```

и нажать кнопку **Connect**.



Рис. 10.2 ❖ Окно входа в jconsole



Удаленное подключение через JMX

По умолчанию Cassandra включает JMX только для локального доступа. Чтобы разрешить удаленный доступ, откройте файл `<cassandra-home>/cassandraenv.sh` (или `cassandra.ps1` в Windows) и поищите строку «JMX». Вы найдете раздел, где находятся параметры, задающие порт JMX, и другие параметры для локального и удаленного подключения.

После подключения к серверу вы увидите главное представление, в котором есть четыре постоянно обновляемых раздела с информацией о состоянии сервера.

Использование кучи

Общий размер памяти, доступной Cassandra, а также занятой в данный момент части.

Потоки

Количество активных потоков, используемых Cassandra.

Классы

Число загруженных Cassandra классов. Для такой мощной программы оно сравнительно невелико – без дополнительных модулей Cassandra обычно загружает менее 3000 классов. Сравните с Oracle WebLogic, в которой количество классов где-то в районе 24 000.

Использование ЦП

Доля ресурсов процессора, потребляемая Cassandra в данный момент.

Селектор позволяет отрегулировать временной диапазон, отображаемый на графиках.

Чтобы получить детальное представление об использовании Cassandra памяти – в куче Java и вне нее, перейдите на вкладку **Memory**. Раскрывающийся список **Chart** позволяет детально и в разных ракурсах увидеть, как Cassandra использует память. Можете также попытаться принудительно запустить сборку мусора, если считаете это необходимым.

Можно подключаться сразу к нескольким агентам JMX. Выберите из меню команду **File → New Connection...** и подключитесь к другому узлу Cassandra, чтобы видеть одновременно несколько серверов.

Другие JMX-клиенты

Когда нужен JMX-клиент, JConsole – очевидный выбор, потому что программа входит в состав JDK и проста в использовании. Но, помимо нее, существует еще много JMX-клиентов. Перечислим лишь несколько из них, решайте сами, какой вам больше подходит.

Oracle Java Mission Control и Visual VM

Эти программы входят в состав Oracle JDK и предоставляют более надежные метрики, средства диагностики и визуализации таких аспектов работы, как потребление памяти, потоки, сборка мусора и т. п. Основное отличие между ними состоит в том, что Visual VM – проект с открытым исходным кодом, распространяемый по лицензии GNU, а Mission Control предлагает более глубокую интеграцию с Oracle JVM с помощью каркаса Flight Control.

Java Mission Control запускается командой `$JAVA_HOME/bin/jmc`, а Visual VM – командой `$JAVA_HOME/bin/jvisualvm`. Та и другая программы могут использоваться как на этапе разработки, так и в производственной среде.

MX4J

Проект Management Extensions для Java (MX4J) представляет собой открытую реализацию JMX, включающую различные инструменты, в т. ч. встроенный веб-интерфейс к JMX по протоколу HTTP/HTML. Это позволяет взаимодействовать с JMX с помощью стандартного браузера.

Для интеграции MX4J с Cassandra скачайте библиотеку `mx4j-tools.jar` (<http://mx4j.sourceforge.net/>), сохраните JAR-файл в подкатало-

те *lib* установочного каталога Cassandra и задайте значения переменных MX4J_ADDRESS и MX4J_PORT в файле *conf/cassandra-env.sh*.

Jmxterm

Jmxterm – это командный JMX-клиент, позволяющий обращаться к JMX-серверу без графического интерфейса. Это особенно полезно при работе в облачной среде, потому что графические инструменты обычно потребляют больше ресурсов. *Jmxterm* – проект на Java с открытым исходным кодом, разработанный фондом Cyclops Group (<http://wiki.cyclosgroup.org/jmxterm/>).

Интеграция с IDE

Можно также найти JMX-клиентов, интегрируемых с популярными средами IDE, например *eclipse-jmx* (<https://code.google.com/archive/p/eclipse-jmx>).

Краткий обзор MBean-объектов

Управляемый *bean-объект*, или MBean-объект, – это специальный тип bean-объекта Java, представляющий один управляемый ресурс внутри JVM. MBean-объекты взаимодействуют с MBean-сервером, чтобы сделать свои функции доступными удаленно.

На рис. 10.3 показано, как выглядит окно **JConsole**.

На этом рисунке мы видим окно с вкладками, в которых отображаются представления, относящиеся к потокам, памяти и процессору, актуальные для любого приложения, а также вкладка **MBeans**, демонстрирующая возможность взаимодействия с MBean-объектами, раскрываемыми приложением. Конкретно здесь выбран показ пикового значения числа потоков. Как видите, есть еще много оснащенных аспектов функциональности приложения.

У приложения или JVM также есть много аспектов, которые оснащены средствами контроля, но могут быть выключены. Примером потенциально полезного MBean-объекта является Thread Contention (Конкуренция потоков), который по умолчанию выключен в JVM. Такие объекты могут оказаться очень полезны на этапе отладки, поэтому, встретив MBean-объект, который, как вам кажется, может помочь в решении проблемы, не колеблясь, включайте его. Но имейте в виду, что бесплатных завтраков не бывает, поэтому разумно было бы почитать документацию по этому объекту и понять, как его включение отразится на производительности. К примеру, измерение потребления ЦП на уровне потока – полезный, но накладный MBean-объект.

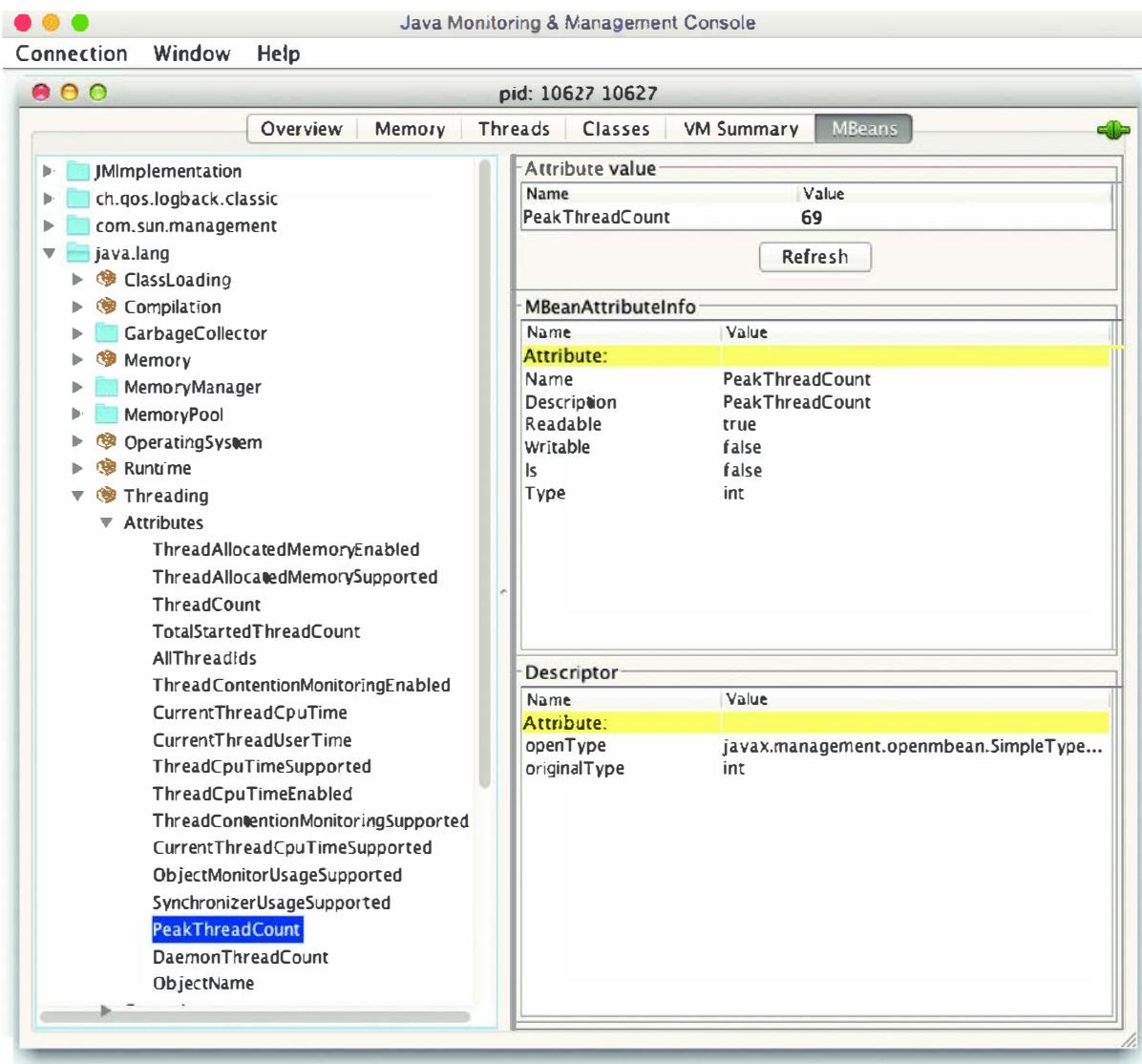


Рис. 10.3 ♦ Отображение пикового числа потоков демона Cassandra в JConsole



Соглашение об именовании MBean-объектов

Регистрируясь в MBean-сервере, MBean-объект указывает имя, под которым будет известен JMX-клиентам. Имя объекта состоит из имени домена, за которым следует список пар ключ-значение, по крайней мере одна из которых должна определять тип. Принято имя домена строить по образцу имен Java-пакетов, а имя типа образовывать из имени интерфейса MBean-объекта (за вычетом слова «MBean»), но это не обязательно.

Например, атрибуты потоков, показанные на рисунке выше, находятся в разделе `java.lang.Threading` и раскрываются классом, реализующим интерфейс `java.lang.management.ThreadMXBean`, поэтому MBean-объект регистрируется под именем `java.lang:type=Threading`. При обсуждении различных MBean-объектов в этой главе мы будем

дем указывать как имя объекта, так и интерфейс, это поможет вам переходить от JMX-клиентов к исходному коду Cassandra.

Некоторые простые переменные приложения раскрываются в виде *атрибутов*. Примером может служить атрибут Threading > PeakThreadCount, который сообщает хранящееся в MBean-объекте наибольшее значение потоков, когда-либо одновременно работавших в приложении. Мы можем просмотреть последнее текущее значение этого показателя, но и только. Поскольку оно хранится внутри JVM, устанавливать его извне не имеет смысла (оно является следствием реальных событий и настройке не подлежит).

Но бывают и настраиваемые MBean-объекты. Они предоставляют JMX-агенту *операции*, позволяющие получать и устанавливать значения. Чтобы узнать, позволяет ли MBean-объект устанавливать значение, посмотрите на атрибут writable. Если он равен false, то значение доступно только для чтения, в противном случае будет присутствовать одно или несколько полей, в которые можно ввести новые значения, и кнопка для обновления. Примером является объект ch.qos.logback.classic.jmx.JMXConfigurator, показанный на рис. 10.4.

Отметим, что имена параметров JMX-агенту неизвестны; они просто обозначаются p0, p1 и т. д. Причина в том, что компилятор Java «стирает» имена параметров во время компиляции. Поэтому, чтобы узнать, какие параметры устанавливает операция, придется заглянуть в JavaDoc-документацию по интересующему вас MBean-объекту.

Конечно, класс JMXConfigurator реализует интерфейс JMXConfiguratorMBean, который оснащает его средствами контроля. Чтобы узнать правильные параметры операции setLoggerLevel, обращаемся к документации по этому интерфейсу по адресу <http://logback.qos.ch/apidocs/ch/qos/logback/classic/jmx/JMXConfiguratorMBean.html>. Там мы находим, что p0 – это имя объекта протоколирования, который мы собираемся изменить, а p1 – новый уровень протоколирования.

Некоторые MBean-объекты возвращают значение атрибута типа javax.management.openmbean.CompositeDataSupport. Это означает, что мы имеем не простое значение, которое можно показать в одном поле, например LoadedClassCount, а многозначный атрибут. Примером может служить атрибут Memory > HeapMemoryUsage, который содержит несколько элементов данных и потому заслуживает собственного представления.

У MBean-объектов могут быть также операции, которые не просто показывают или устанавливают значение, а позволяют выполнить полезное действие, например: dumpAllThreads и resetPeakThreadCount.

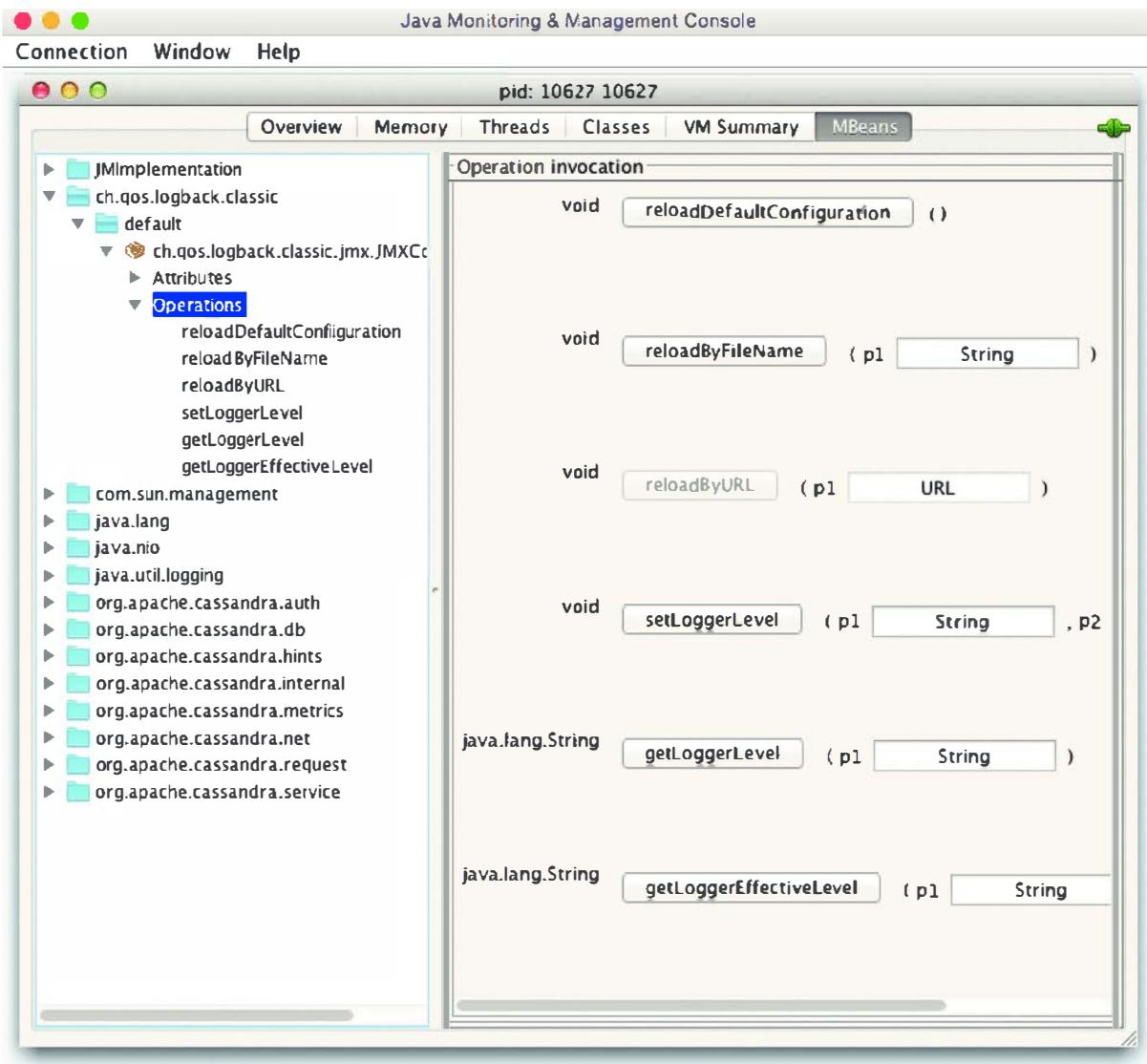


Рис. 10.4 ❖ MBean-объект JMXConfigurator позволяет установить уровень протоколирования

А теперь займемся вопросом о мониторинге и управлении собственно Cassandra.

MBean-объекты Cassandra

Подключившись к серверу Cassandra с помощью JMX-агента типа JConsole, вы можете управлять MBean-объектами, которые Cassandra раскрывает. Для этого перейдите на вкладку **MBeans**. Помимо стандартных объектов Java, доступных любому агенту, существует несколько пакетов Cassandra, содержащих управляемые bean-объекты. Их имена начинаются с `org.apache.cassandra.` Не будем рассматривать все подряд, а остановимся на нескольких особенно интересных.

Многие классы Cassandra раскрываются в виде MBean-объектов, на практике это означает, что они реализуют специальный интерфейс, который описывает определенные операции и для которого JMX-агент предоставит точки подключения. Основные шаги работы с MBean-объектами одинаковы. Если вы хотите раскрыть через JMX что-то, еще не раскрытое, модифицируйте исходный код, следуя общему шаблону, – и наслаждайтесь жизнью.

Рассмотрим, к примеру, как используются MBean-объекты в классе `Cassandra CompactionManager` из пакета `org.apache.cassandra.db.compaction`. Ниже приведено определение интерфейса `CompactionManagerMBean` с опущенными для краткости комментариями.

```
public interface CompactionManagerMBean
{
    public List<Map<String, String>> getCompactions();
    public List<String> getCompactionSummary();
    public TabularData getCompactionHistory();

    public void forceUserDefinedCompaction(String dataFiles);
    public void stopCompaction(String type);
    public void stopCompactionById(String compactionId);

    public int getCoreCompactorThreads();
    public void setCoreCompactorThreads(int number);

    public int getMaximumCompactorThreads();
    public void setMaximumCompactorThreads(int number);

    public int getCoreValidationThreads();
    public void setCoreValidationThreads(int number);

    public int getMaximumValidatorThreads();
    public void setMaximumValidatorThreads(int number);
}
```

Как видим, в определении интерфейса MBean-объекта нет никакой тайны. Это обычный интерфейс, в котором определяется набор раскрываемых через JMX операций, которые должен предоставить класс `CompactionManager`. Как правило, для этого придется включить дополнительные метаданные и обеспечить их поддержку в коде обычных методов класса.

Класс `CompactionManager` реализует этот интерфейс и должен напрямую поддержать JMX. Этот класс регистрирует себя в MBean-сервере:

```

public static final String MBEAN_OBJECT_NAME =
    "org.apache.cassandra.db:type=CompactionManager";
// ...
static
{
    instance = new CompactionManager();
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
    try
    {
        mbs.registerMBean(instance, new ObjectName(MBEAN_OBJECT_NAME));
    }
    catch (Exception e)
    {
        throw new RuntimeException(e);
    }
}
}

```

Отметим, что MBean-объект зарегистрирован в домене org.apache.cassandra.db с типом CompactionManager. Атрибуты и операции, раскрываемые этим объектом, появятся в JMX-клиентах в разделе org.apache.cassandra.db > CompactionManager. Класс выполняет все содержательные действия, а также реализует методы, необходимые только для общения с MBean-сервером. Вот, например, как в CompactionManager реализована операция stopCompaction():

```

public void stopCompaction(String type)
{
    OperationType operation = OperationType.valueOf(type);
    for (Holder holder : CompactionMetrics.getCompactions())
    {
        if (holder.getCompactionInfo().getTaskType() == operation)
            holder.stop();
    }
}

```

CompactionManager перебирает все выполняемые в данный момент операции уплотнения и останавливает те, которые имеют указанный тип. В документации перечислены допустимые типы: COMPACTION, VALIDATION, CLEANUP, SCRUB и INDEX_BUILD.

Найдя объект CompactionManagerMBean в JConsole, мы можем открыть список его операций и найти в нем stopCompaction(), как показано на рис. 10.5. Затем можно ввести один из вышеперечисленных типов и потребовать, чтобы все операции уплотнения такого типа были остановлены.

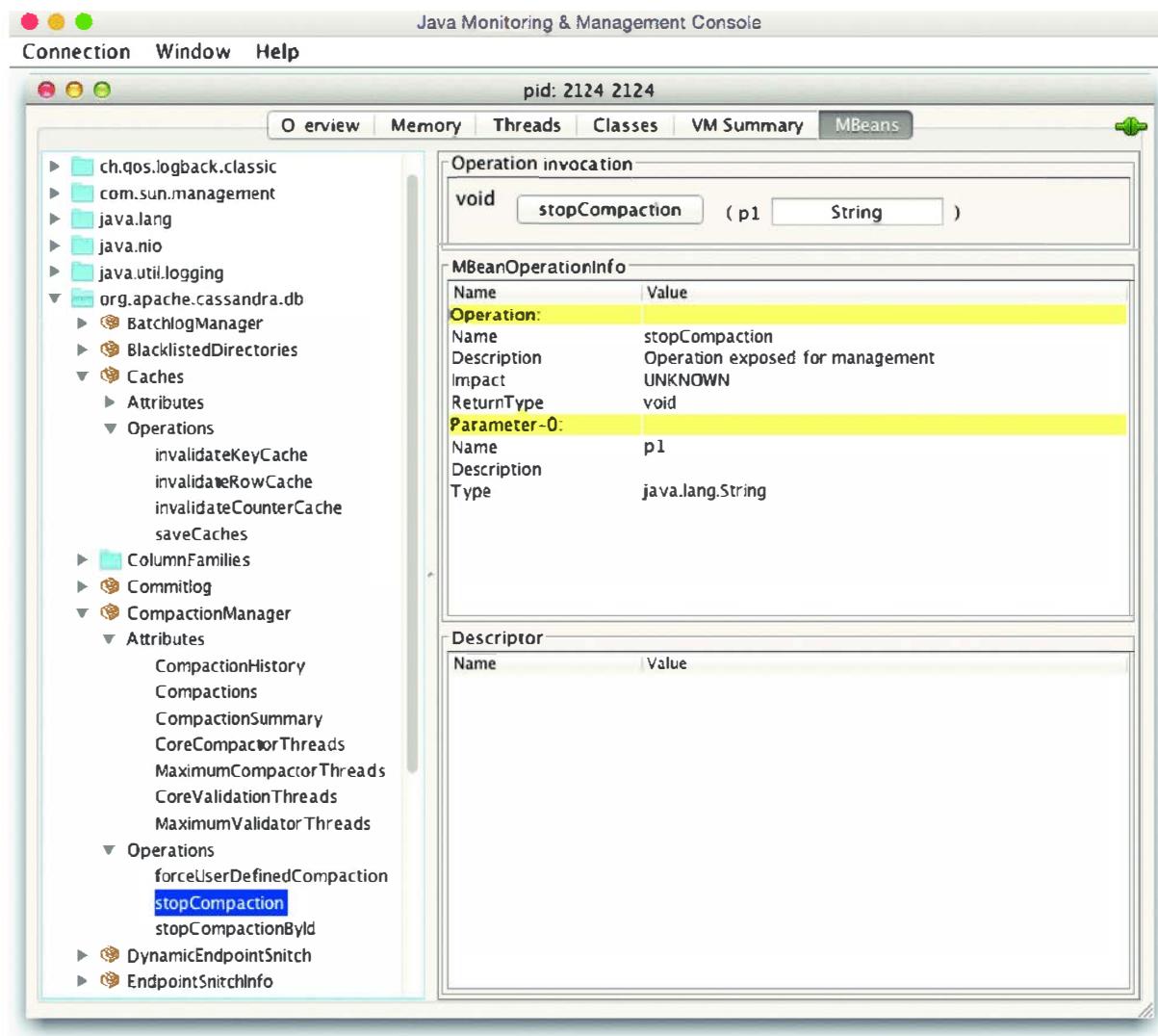


Рис. 10.5 ❖ Операция stopCompaction() в классе CompactionManager

В следующих разделах мы узнаем, какие функции доступны для мониторинга и управления через JMX.

MBean-объекты, относящиеся к базе данных

Эти классы относятся к самой базе данных, клиенты видят их в домене org.apache.cassandra.db. MBean-объектов в этом домене много, но мы ограничимся только несколькими – касающимися хранилища, кэширования, журнала фиксаций и хранения данных в таблицах.

MBean-объект службы хранилища

Являясь базой данных, Cassandra представляет собой, по сути дела, очень сложную программу организации хранилища; поэтому при возникновении проблем первым делом нужно смотреть на MBean-объект org.apache.cassandra.service.StorageServiceMBean. В нем есть

атрибут `OperationMode`, равный `normal`, если все хорошо (другие возможные значения: `leaving`, `joining`, `decommissioned` и `client`).

Можно также посмотреть на текущий список работающих узлов, а также на список недостижимых узлов кластера. IP-адреса недостижимых узлов перечислены в атрибуте `UnreachableNodes`.

Чтобы изменить уровень протоколирования, не прерывая работу `Cassandra` (как мы делали выше), вызовите метод `setLoggingLevel(String classQualifier, String level)`. Допустим, к примеру, что на время поиска причин неисправности был установлен уровень `DEBUG`. Воспользовавшись какими-то из описанных ниже методов, вы устранили проблему и теперь хотите установить уровень протоколирования, который не так сильно нагружает систему. Для этого найдите в JMX-клиенте, например `JConsole`, MBean-объект `StorageService`. Давайте изменим уровень протоколирования для особенно болтливого класса `Gossiper`. Первым параметром метода является имя класса, для которого устанавливается уровень, вторым – новое значение уровня. Введите соответственно `org.apache.cassandra.gms.Gossiper` и `INFO` и нажмите кнопку `setLoggingLevel`. В журнале появится следующее сообщение (в предположении, что до этого был установлен уровень протоколирования `DEBUG`):

```
INFO 03:08:20 set log level to INFO for classes under  
'org.apache.cassandra.gms.Gossiper' (if the level doesn't look  
like 'INFO' then the logger couldn't parse 'INFO')
```

После выполнения метода `setLoggingLevel` будет установлен уровень `INFO`, и сообщений уровня `DEBUG` мы больше не увидим.

Чтобы узнать, сколько данных хранится в каждом узле, воспользуйтесь методом `getLoadMap()`, который возвращает объект `Map`, в котором ключами являются IP-адреса узлов, а значениями – объем данных, хранящихся в узле. Есть также метод `effectiveOwnership(String keyspace)`, который показывает, какой процент данных из указанного пространства ключей хранится в каждом узле.

Если вы ищете определенный ключ, вам поможет операция `getNaturalEndpoints(String table, byte[] key)`. Передайте имя таблицы и значение ключа – в ответ получите список IP-адресов узлов, отвечающих за хранение этого ключа.

Метод `getRangeToEndPointMap` возвращает словарь, отображающий диапазоны на конечные точки. Этот словарь описывает топологию кольца.

Если вы ничего не боитесь, можете вызвать операцию `truncate()`, указав имя таблицы и пространства ключей. Если все узлы доступны, то эта операция удалит из таблицы все данные, но оставит определение.

Объект `StorageServiceMBean` предоставляет в распоряжение администратора много стандартных операций обслуживания, в т. ч. `resumeBootstrap()`, `joinRing()`, `repairAsync()`, `drain()`, `removeNode()`, `decommission()`, а также операций запуска и остановки распространения сплетен, внутреннего транспортного протокола и Thrift (до тех пор, пока Thrift не будет окончательно удален). Знать назначение имеющихся операций обслуживания важно для поддержания кластера в работоспособном состоянии, мы еще вернемся к этой теме в главе 11.

MBean-объект прокси хранилища

В главе 6 упоминалось, что класс `org.apache.cassandra.service.StorageProxy` организует слой поверх `StorageService`, предназначенный для обработки клиентских запросов и межузловых коммуникаций. Объект `StorageProxyMBean` позволяет просмотреть и установить таймауты различных операций, в т. ч. чтения и записи.

Он же предоставляет доступ к параметрам механизма вручения напоминаний, таким как время хранения напоминаний. Методы `getTotalHints()` и `getHintsInProgress()` возвращают статистические сведения о напоминаниях. Можно отключить сохранение напоминаний для конкретного узла, вызвав метод `disableHintsForDC()`.

Включить или выключить участие узла в механизме вручения напоминаний позволяет метод `setHintedHandoffEnabled()`, а проверить текущее состояние – метод `getHintedHandoffEnabled()`. Они используются соответственно командами `enablehandoff`, `disablehandoff` и `statushandoff` утилиты `nodetool`.

Метод `reloadTriggerClasses()` устанавливает новый триггер без перезагрузки узла.

MBean-объект `ColumnFamilyStoreMBean`

Cassandra регистрирует экземпляр MBean-объекта `org.apache.cassandra.db.ColumnFamilyStoreMBean` для каждой таблицы. Все они расположены в разделе `org.apache.cassandra.db > Tables` (раньше он назывался `ColumnFamilies`).

Класс `ColumnFamilyStoreMBean` дает доступ к параметрам уплотнения каждой таблицы и позволяет временно переопределить их на отдельном узле. После перезапуска узла будут восстановлены значения, заданные в схеме таблицы.

MBean-объект также раскрывает детальную информацию об особенностях хранения данных в таблице на диске. Метод `getSSTableCountPerLevel()` возвращает список, показывающий, сколько файлов

SSTable находится на каждом уровне. Метод `estimateKeys()` возвращает оценку числа разделов на этом узле. То и другое вместе может подсказать, поможет ли вызов `forceMajorCompaction()` освободить место на диске и повысить скорость чтения.

Метод `trueSnapshotsSize()` позволяет узнать размер уже неактивных снимков файлов SSTable. Если значение велико, то следует удалить эти снимки, быть может, сделав предварительно архивную копию.

Поскольку Cassandra хранит индексы как таблицы, для каждого индексированного столбца также существует экземпляр `ColumnFamilyStoreMBean` с такими же атрибутами и операциями, и все они находятся в разделе `org.apache.cassandra.db > IndexTables` (раньше он назывался `IndexColumnFamilies`).

MBean-объект *CacheServiceMBean*

Класс `org.apache.cassandra.service.CacheServiceMBean` предоставляет доступ к кэшу ключей, кэшу строк и кэшу счетчиков, он находится в разделе `org.apache.cassandra.db > Caches`. Для каждого кэша MBean-объект сообщает максимальный размер и время хранения в нем элементов, а также дает возможность сделать кэш недействительным.

MBean-объект *CommitLogMBean*

В классе `org.apache.cassandra.db.commitlog.CommitLogMBean` определены атрибуты и операции, позволяющие узнать о текущем состоянии журналов фиксаций. Метод `recover()` восстанавливает состояние базы данных из архивированных журналов фиксаций.

Параметры по умолчанию, управляющие восстановлением, задаются в файле `conf/commitlog_archiving.properties`, но могут быть переопределены с помощью этого MBean-объекта. Дополнительные сведения о восстановлении приведены в главе 11.

MBean-объект *CompactionManagerMBean*

Мы уже заглядывали в исходный код класса `org.apache.cassandra.db.compaction.CompactionManagerMBean`, чтобы посмотреть, как он взаимодействует с JMX, но ничего не сказали о его назначении. Этот MBean-объект сообщает статистику произведенных в прошлом операций уплотнения, а также позволяет принудительно запустить уплотнение указанных файлов SSTable методом `forceUserDefinedCompaction` класса `CompactionManager`. Объект используется в командах `compact`, `compactionhistory` и `compactionstats` утилиты `nodetool`.

MBean-объекты, связанные с осведомителями

Cassandra предоставляет два MBean-объекта для мониторинга и настройки поведения осведомителя. Класс org.apache.cassandra.locator.EndpointSnitchInfoMBean возвращает имя стойки и ЦОДа для указанного хоста, а также имя используемого осведомителя.

Осведомитель DynamicEndpointSnitch регистрирует MBean-объект org.apache.cassandra.locator.DynamicEndpointSnitchMBean, который умеет сбрасывать порог негодности, при превышении которого осведомитель помечает узел как неработающий. Кроме того, объект позволяет просмотреть оценки качества различных узлов.

MBean-объект HintedHandoffManagerMBean

Помимо относящихся к вручению напоминаний операций, которые уже упоминались при рассмотрении класса StorageServiceMBean, Cassandra предлагает средства более точного управления этим механизмом с помощью класса org.apache.cassandra.db.HintedHandOffManagerMBean. Метод listEndpointsPendingHints() этого MBean-объекта возвращает список узлов, для которых хранятся напоминания. Получив список, можно затем принудительно инициировать доставку напоминаний методом scheduleHintDelivery() или удалить напоминания, хранящиеся для конкретного узла методом deleteHintsForEndpoint().

Можно также приостановить или возобновить доставку напоминаний всем узлам методом pauseHintDelivery() или удалить напоминания сразу для всех узлов методом truncateAllHints(). Эти операции используются соответственно в командах pausehandoff, resumehandoff и truncatehints утилиты nodetool.



Дублирование средств управления вручением напоминаний

Класс org.apache.cassandra.hints.HintsService дает доступ к MBean-объекту HintsServiceMBean в разделе org.apache.cassandra.hints > HintsService. Этот объект предоставляет методы для приостановки и возобновления вручения напоминаний, а также для удаления хранящихся напоминаний для всех узлов или только для одного узла, заданного своим IP-адресом.

Поскольку функциональность StorageServiceMBean, HintedHandOffManagerMBean и HintsServiceMBean в значительной мере перекрываются, в будущих версиях они, вероятно, будут объединены.

MBean-объекты, относящиеся к сети

В домене org.apache.cassandra.net находятся MBean-объекты для управления аспектами работы Cassandra, связанными с сетью, в т. ч. ф-детектором отказов, распространением сплетен, службой обмена сообщениями (Messaging Service) и диспетчером потоков (Stream Manager).

MBean-объект FailureDetectorMBean

В классе org.apache.cassandra.gms.FailureDetectorMBean имеются атрибуты, описывающие состояния и ф-оценки других узлов, а также порог признания отказа ф-детектором.

MBean-объект GossiperMBean

Класс org.apache.cassandra.gms.GossiperMBean предоставляет доступ к управлению работой протокола распространения сплетен.

Мы уже обсуждали объект StorageServiceMBean, который сообщает, какие узлы недостижимы. Имея этот список, мы можем вызвать метод getEndpointDowntime() объекта GossiperMBean и узнать, сколько времени данный узел «лежит». Времяостоя измеряется с точки зрения узла, на котором мы опрашиваем MBean-объект, и сбрасывается в нуль, когда узел «оживает». Cassandra сама вызывает этот метод, чтобы узнать, сколько времени еще ждать, перед тем как отбросить напоминания.

Метод getCurrentGenerationNumber() возвращает номер поколения, ассоциированный с указанным узлом. Этот номер включается в сплетни, которыми обмениваются узлы, и служит для того, чтобы отличить текущее состояние узла от состояния, предшествующего перезапуску. Номер поколения не меняется, пока узел работает, и увеличивается при каждом перезапуске. Класс Gossiper хранит его в виде временной метки.

Метод assassinateEndpoint() пытается удалить узел из кольца, сообщая другим узлам о том, что узел удален навсегда, – по аналогии с подрывом репутации, практикуемым обычными сплетниками. Подрыв репутации узла (assassination) – административное действие, применяемое в крайнем случае, когда узел не удается удалить из кластера нормальным путем. Эта операция применяется в команде assassinate утилиты nodetool.

MBean-объект StreamManagerMBean

Класс `org.apache.cassandra.streaming.StreamManagerMBean` позволяет узнать, какие потоковые операции происходят между узлом и его соседями. У потока есть источник и назначение. Любой узел может передавать свои данные потоком другому узлу для балансировки нагрузки, а управляет этими операциями класс `StreamManager`. MBean-объект `StreamManagerMBean` позволяет получить сведения о перемещении данных между узлами кластера.

`StreamManagerMBean` поддерживает два режима работы. Метод `getCurrentStreams()` дает мгновенный снимок входных и выходных потоков, а кроме того, MBean-объект публикует уведомления об изменениях состояния потоков: инициализации, завершении или сбое. JMX-клиент может подписаться на эти уведомления и наблюдать за потоковыми операциями в реальном времени.

Если вы полагаете, что узел не получает данных, как положено, или что он не сбалансирован или даже вообще не работает, то воспользуйтесь двумя MBean-объектами: `StorageServiceMBean` и `StreamManagerMBean` – в совокупности они могут дать очень детальные сведения о происходящем в кластере.

MBean-объекты, относящиеся к метрикам

Средства для доступа к метрикам, отражающим производительность приложения, его состояние и ключевые действия, стали необходимым инструментом обслуживания приложений масштаба веба. К счастью, Cassandra поддерживает широкий спектр показателей собственной работы, чтобы мы могли лучше понять ее поведение.

В состав метрик Cassandra входят:

- *метрики буферного пула*, описывающие использование памяти;
- *метрики CQL*, в т. ч. количество выполнений подготовленных и обычных команд;
- *метрики кэшей* – ключей, строк и счетчиков, в т. ч. количество элементов в сравнении с максимальной емкостью, а также коэффициенты попадания и промаха;
- *метрики клиентов*, в т. ч. количество подключенных клиентов и сведения о клиентских запросах: задержка, ошибки, тайм-ауты;
- *метрики журнала фиксаций*, в т. ч. размер журнала и статистика ожидающих и завершенных задач;
- *метрики уплотнения*, в т. ч. общее число уплотненных байтов и статистика ожидающих и завершенных операций уплотнения;

- *метрики соединений* с узлами кластера, включая распространение сплетен;
- *метрики отброшенных сообщений*, которые показываются командой nodetool tpstats;
- *метрики исправлений на этапе чтения*: сравнение количества блокирующих и фоновых операций исправления;
- *метрики хранения*, в т. ч. число хранящихся в данный момент напоминаний и общее число напоминаний;
- *метрики пула потоков*, в т. ч. количество активных, завершенных и блокированных задач для каждого пула;
- *табличные метрики*, в т. ч. сведения об использовании кэшей, таблиц в памяти, файлов SSTable и фильтров Блума, о задержке различных операций чтения и записи, собираемые с интервалом в одну, пять и пятнадцать минут;
- *метрики пространств ключей*, агрегирующие табличные метрики для каждого пространства ключей.

Чтобы сделать эти метрики доступными через JMX, Cassandra пользуется Java-библиотекой Dropwizard Metrics (<http://metrics.dropwizard.io/3.1.0/>) с открытым исходным кодом. Cassandra регистрирует свои метрики в библиотеке Metrics, а та, в свою очередь, раскрывает их в виде MBean-объектов в домене org.apache.cassandra.metrics.

Многие из этих метрик используются такими командами утилиты nodetool, как tpstats, tablehistograms и proxyhistograms. Например, tpstats просто выводит метрики пула потоков и отброшенных сообщений.

MBean-объекты, относящиеся к потокам

В домене org.apache.cassandra.internal находятся MBean-объекты для настройки пулов потоков, ассоциированных с каждой ступенью многоступенчатой событийно-ориентированной архитектуры (SEDA) Cassandra. В состав ступеней входят AntiEntropyStage, GossipStage, InternalResponseStage, MigrationStage и др.



MBean-объект исправления на этапе чтения

В силу исторических причин MBean-объект ReadRepairStage находится в домене org.apache.cassandra.request, а не org.apache.cassandra.internal.

Пулы потоков реализованы в классах JMXEnabledThreadPoolExecutor и JMXEnabledScheduledThreadPoolExecutor из пакета org.apache.cassandra.

concurrent. MBean-объекты для каждой ступени реализуют интерфейс `JMXEnabledScheduledThreadPoolExecutorMBean`, который позволяет просматривать и настраивать количество потоков в каждом пуле, а также максимальное число потоков.

MBean-объекты, относящиеся к службам

Класс `GCInspectorMXBean` содержит единственный метод `getAndResetStats()`, который извлекает и сразу же переустанавливает метрики сборки мусора, которые Cassandra получает от JVM. Этот MBean-объект находится в домене `org.apache.cassandra.service` и используется командой `nodetool gcstats`.

MBean-объекты, относящиеся к безопасности

В домене `org.apache.cassandra.auth` находятся MBean-объекты, относящиеся к безопасности. В версии 3.0 эта группа состоит из одного объекта, `PermissionsCacheMBean`, который клиентам известен под именем `org.apache.cassandra.auth.PermissionsCache`. Мы обсудим его в главе 13.

Мониторинг с помощью nodetool

В предыдущих главах мы уже встречали несколько команд утилиты `nodetool`, но теперь представим их официально.

Утилита `nodetool` входит в комплект поставки Cassandra и находится в каталоге `<cassandra-home>/bin`. Эта командная программа, предоставляющая массу способов взглянуть на кластер, изучить его работу и модифицировать его. `nodetool` позволяет получить ограниченную статистику работы кластера, узнать, какие диапазоны ключей назначены каждому узлу, переместить данные с одного узла на другой, вывести узел из эксплуатации и даже исправить узел, испытывающий проблемы.



Перекрытие функциональности nodetool и JMX

Многие команды `nodetool` перекрываются с функциями в интерфейсе к JMX. Объясняется это тем, что за кулисами `nodetool` вызывает JMX, пользуясь вспомогательным классом `org.apache.cassandra.tools.NodeProbe`. Таким образом, JMX делает реальную работу, класс `NodeProbe` устанавливает соединение с агентом JMX и запрашивает данные, а класс `NodeCmd` выводит данные в интерактивном командном интерфейсе.

nodetool, как и демон Cassandra, пользуется параметрами среды, прописанными в файлах *bin/cassandra.in.sh* и *conf/cassandra-env.sh* в Unix (или *bin/cassandra.in.bat* и *conf/cassandra-env.ps1* в Windows). Настройки протоколирования находятся в файле *conf/logback-tools.xml* и устроены так же, как настройки протоколирования для демона Cassandra в файле *conf/logback.xml*.

Для запуска nodetool нужно просто открыть окно терминала, перейти в каталог *<cassandra-home>* и ввести команду:

```
$ bin/nodetool help
```

Программа в ответ напечатает список доступных команд, часть которых мы скоро рассмотрим. Запуск nodetool без аргументов эквивалентен запуску с аргументом *help*. Можно после слова *help* указать имя конкретной команды для получения подробной информации о ней.



Подключение к конкретному узлу

При выполнении любой команды, кроме *help*, nodetool должна подключиться к какому-то узлу Cassandra для получения информации об этом узле или кластере в целом.

IP-адрес узла задается с помощью флага *-h*. Если адрес не задан, то nodetool пытается подключиться к порту по умолчанию на локальной машине, и именно так мы будем поступать в примерах из этой главы.

Получение информации о кластере

О кластере и его узлах можно получить много разной информации, кое-что мы рассмотрим в этом разделе. Можно запросить базовые сведения об одном узле или обо всех узлах кольца.

describecluster

Команда *describecluster* выводит основную информацию о кластере: имя, осведомитель и разделитель.

```
$ bin/nodetool describecluster
Cluster Information:
  Name: Test Cluster
  Snitch: org.apache.cassandra.locator.DynamicEndpointSnitch
  Partitioner: org.apache.cassandra.dht.Murmur3Partitioner
  Schema versions:
    2d4043cb-2124-3589-b2d0-375759b9dd0a: [127.0.0.1, 127.0.0.2, 127.0.0.3]
```

Последняя часть особенно важна для выявления рассогласований между узлами в определениях, или «схеме» таблиц. Когда Cassandra

распространяет изменения схемы по кластеру, расхождения обычно устраняются очень быстро, поэтому различия, существующие длительное время, могут указывать на то, что узел не работает или недостижим и должен быть перезапущен.

status

Более прямой способ узнать об узлах кластера и их состоянии дает команда `status`:

```
$ bin/nodetool status
Datacenter: datacenter1
=====
Status=Up/Down
| / State=Normal/Leaving/Joining/Moving
-- Address      Load      Tokens  Owns    Host ID
UN 127.0.0.1   103.82 KB  256      ?      31d9042b-6603-4040-8aac-fef0a235570b
UN 127.0.0.2   110.9  KB  256      ?      caad1573-4157-43d2-a9fa-88f79344683d
UN 127.0.0.3   109.6  KB  256      ?      e78529c8-ee9f-46a4-8bc1-3479f99a1860
```

Статусы узлов собраны в группы по ЦОДам и стойкам. Статус обозначается двузначным кодом. Первый знак равен U, если узел работает (доступен для приема запросов), или D, если не работает. Второй знак описывает состояние, или режим работы узла. В столбце `Load` показано число байтов данных, хранящихся в узле.

В столбце `Owns` показана эффективная процентная доля диапазонов маркеров, которой владеет данный узел, с учетом репликации. Поскольку мы не задали пространство ключей, а у разных пространств ключей в кластере могут быть разные стратегии репликации, `nodetool` не может вычислить осмысленного значения этого показателя.

info

Команда `info` означает, что `nodetool` должна подключиться к одному узлу и получить основную информацию о его текущем состоянии. Достаточно задать адрес узла.

```
$ bin/nodetool -h 192.168.2.7 info
ID : 197efa22-ecaa-40dc-a010-6c105819bf5e
Gossip active : true
Thrift active : false
Native Transport active : true
Load : 301.17 MB
Generation No : 1447444152
Uptime (seconds) : 1901668
Heap Memory (MB) : 395.03 / 989.88
```

```

Off Heap Memory (MB) : 2.94
Data Center : datacenter1
Rack : rack1
Exceptions : 0
Key Cache : entries 85, size 8.38 KB, capacity 49 MB,
47958 hits, 48038 requests, 0.998 recent hit rate, 14400 save
period in seconds
Row Cache : entries 0, size 0 bytes, capacity 0 bytes,
0 hits, 0 requests, NaN recent hit rate, 0 save period in seconds
Counter Cache : entries 0, size 0 bytes, capacity 24 MB,
0 hits, 0 requests, NaN recent hit rate, 7200 save period in seconds
Token : (invoke with -T/--tokens to see all 256 tokens)

```

Программа сообщает об использовании памяти и диска («Load») в узле и статусе различных служб Cassandra. Можно также опросить статусы отдельных служб с помощью команд statusgossip, status-thrift, statusbinary и statushandoff (отметим, что info не выдает статуса службы вручения напоминаний).

ring

Чтобы получить перечень узлов, входящих в кольцо, и узнать их состояние, воспользуйтесь командой *ring*:

```

$ bin/nodetool ring
Datacenter: datacenter1
=====
Address      Rack    Status   State    Load       Owns     Token
192.168.2.5  rack1   Up       Normal   243.6 KB  ?        -9203905334627395805
192.168.2.6  rack1   Up       Normal   243.6 KB  ?        -9145503818225306830
192.168.2.7  rack1   Up       Normal   243.6 KB  ?        -9091015424710319286
...

```

Распечатка организована по виртуальным узлам. Здесь мы видим IP-адреса всех узлов кольца. В данном случае узлов три, и все они работают (доступны для приема запросов). В столбце Load показано число байтов данных, хранящихся в узле. Команда *describering* похожа, но организует вывод по диапазонам маркеров.

Упомянем другие полезные команды вывода статуса, предлагаемые *nodetool*.

- Команды *getLoggingLevels* и *setLoggingLevels* позволяют динамически изменять уровни протоколирования, обращаясь для этого к объекту *JMXConfiguratorMBean*, который мы уже рассматривали.

- Команда `gossipinfo` выводит параметры, которые данный узел сообщает о себе другим узлам по протоколу распространения сплетен.
- Команда `version` печатает номер версии сервера Cassandra, работающего на данном узле.

Получение статистики

Утилита `nodetool` позволяет также собрать агрегированную статистику о состоянии сервера и статистику на уровне отдельных пространств ключей и таблиц. Чаще всего используются команды `tpstats` и `tablestats`.

Команда `tpstats`

Команда `tpstats` выдает информацию о пулах потоков. Cassandra – программа с высоким уровнем параллелизма, оптимизированная для работы на многопроцессорных и многоядерных компьютерах. К тому же, в ней используется многоступенчатая событийно-ориентированная архитектура SEDA, поэтому для обслуживания системы важно знать о поведении и состоянии пулов потоков.

Для получения статистики работы пулов потоков служит команда `nodetool tpstats`:

```
$ bin/nodetool tpstats
Pool Name          Active Pending Completed Blocked  All time
                                         blocked

ReadStage    0      0      216      0      0
MutationStage 1      0      3637      0      0
CounterMutationStage 0      0      0      0      0
ViewMutationStage 0      0      0      0      0
GossipStage   0      0      0      0      0
RequestResponseStage 0      0      0      0      0
AntiEntropyStage 0      0      0      0      0
MigrationStage 0      0      2      0      0
MiscStage     0      0      0      0      0
InternalResponseStage 0      0      2      0      0
ReadRepairStage 0      0      0      0      0

Message type        Dropped
READ      0
RANGE_SLICE 0
_TRACE    0
HINT      0
MUTATION   0
```

```
COUNTER_MUTATION 0
BATCH_STORE 0
BATCH_REMOVE 0
REQUEST_RESPONSE 0
PAGED_RANGE 0
READ_REPAIR 0
```

В верхней части приведены данные о заданиях в каждом пуле потоков Cassandra. Видно, сколько операций на каждой ступени, и являются ли они активными, ожидающими или завершенными. Эта распечатка была получена во время операции записи, поэтому на ступени MutationStage имеется активное задание.

В нижней части показано количество отброшенных узлом сообщений. Механизм отбрасывания сообщений применяется в Cassandra для ограничения нагрузки, с его помощью узел защищает себя от получения большего числа запросов, чем способен обработать. Например, межузловые сообщения, полученные узлом, но не обработанные в течение промежутка времени `rpc_timeout`, отбрасываются, так как узел-координатор дольше ждать ответа не будет.

Если в столбце `Blocked` и в разделе отброшенных сообщений много нулей, значит, либо сервер почти ничем не занят, либо Cassandra великолепно справляется с нагрузкой. Если же много ненулевых значений, значит, Cassandra испытывает трудности и, возможно, пора применить некоторые методы, описанные в главе 12.

Команда `tablestats`

Команда `tablestats` показывает сводную статистику пространств ключей и таблиц. Раньше она называлась `cfstats`. Ниже приведен пример выдачи для пространства ключей `hotel`:

```
$ bin/nodetool tablestats hotel
Keyspace: hotel
  Read Count: 8
  Read Latency: 0.617 ms.
  Write Count: 13
  Write Latency: 0.13330769230769232 ms.
  Pending Flushes: 0
    Table: hotels
    SSTable count: 3
    Space used (live): 16601
    Space used (total): 16601
    Space used by snapshots (total): 0
    Off heap memory used (total): 140
```

```
SSTable Compression Ratio: 0.6277372262773723
Number of keys (estimate): 19
Memtable cell count: 8
Memtable data size: 792
Memtable off heap memory used: 0
Memtable switch count: 1
Local read count: 8
Local read latency: 0.680 ms
Local write count: 13
Local write latency: 0.148 ms
Pending flushes: 0
Bloom filter false positives: 0
Bloom filter false ratio: 0.00000
Bloom filter space used: 56
Bloom filter off heap memory used: 32
Index summary off heap memory used: 84
Compression metadata off heap memory used: 24
Compacted partition minimum bytes: 30
Compacted partition maximum bytes: 149
Compacted partition mean bytes: 87
Average live cells per slice (last five minutes): 1.0
Maximum live cells per slice (last five minutes): 1
Average tombstones per slice (last five minutes): 1.0
Maximum tombstones per slice (last five minutes): 1
```

Мы опустили части, относящиеся к другим таблицам в этом пространстве ключей, поскольку они аналогичны. Показаны задержки чтения и записи, а также общее число операций чтения и записи на уровне таблицы и всего пространства ключей. Также имеется подробная информация о внутренних структурах Cassandra для каждой таблицы: таблицах в памяти, фильтрах Блума и файлах SSTable.

Резюме

В этой главе мы познакомились с различными способами мониторинга и управления кластером Cassandra. В частности, было рассказано о деталях технологии JMX и о многообразных операциях, которые Cassandra предоставляет через MBean-сервер. Мы видели, как с помощью JConsole и nodetool посмотреть, что происходит с кластером Cassandra. Теперь можно переходить к изучению повседневных задач обслуживания для поддержания кластера в работоспособном состоянии.

Глава 11

Обслуживание

В этой главе мы рассмотрим, что можно сделать для поддержания кластера Cassandra в исправном состоянии. Наша цель – дать обзор различных задач обслуживания. Поскольку детали немного меняются от версии к версии, не забывайте обращаться к документации DataStax по той версии, с которой работаете, – это поможет не пропустить новые шаги.

Итак, примеряем роль эксплуатационника – и вперед!

Проверка исправности

Чтобы убедиться в исправности узлов кластера, нужно выполнить несколько простых операций.

- С помощью команды nodetool status убедиться, что все узлы работают и их статус – нормальный. Проверить нагрузку на каждый узел на предмет сбалансированности кластера. Неодинаковое количество узлов в разных стойках может стать причиной несбалансированности.
- С помощью команды nodetool tpstats проверить, нет ли отброшенных сообщений, особенно в строке MUTATION (изменение), поскольку это свидетельствует о возможном отбрасывании запросов па запись. Растущее число блокированных потоковброса говорит о том, что узел помещает данные в память быстрее, чем они сбрасываются на диск. То и другое – признак того, что Cassandra не справляется с нагрузкой. Как часто бывает с базами данных, стоит проблемам появиться, как они нарастают. Улучшить ситуацию помогут три вещи: уменьшение нагрузки, вертикальное масштабирование (добавление оборудования) или горизонтальное масштабирование (добавление еще одного узла и пербалансировка).

Если эти проверки показывают наличие проблем, нужно продолжить исследование и разобраться, что происходит.

- Проверьте, нет ли в журналах сообщений типа ERROR или WARN (например, `OutOfMemoryError` – нехватка памяти). Cassandra выводит предупреждения, если обнаруживает неправильные или устаревшие конфигурационные параметры, операции, которые не завершились успешно, или проблемы с памятью либо хранилищем данных.
- Проверьте конфигурационные файлы `cassandra.yaml` и `cassandra-env.sh` и убедитесь, что узлы Cassandra настроены именно так, как вы хотели. Например, что куча имеет рекомендованный размер 8 ГБ.
- Проверьте параметры пространств ключей и таблиц. Например, часто по ошибке забывают изменить стратегии репликации пространства ключей после добавления нового центра обработки данных.
- Помимо состояния узлов Cassandra, стоит обратить внимание на состояние и конфигурацию системы в целом, в т. ч. на связность сети и правильную работу серверов протокола Network Time Protocol (NTP).

Это лишь малая часть важных вещей, на которые привычно смотрят опытные администраторы Cassandra. По мере накопления опыта этот список будет дополнен проверками, актуальными для вашей конкретной среды.

Базовое обслуживание

Некоторые операции следует производить до или после действий, оказывающих заметное влияние на систему. Например, делать мгновенный снимок имеет смысл после сброса данных на диск. В этом разделе мы рассмотрим базовые задачи обслуживания.

Многие обсуждаемые в этой главе задачи различаются в зависимости от типа узла: с одним маркером или виртуальный. Поскольку по умолчанию подразумеваются v-узлы, мы в основном обсуждаем их, но приводим ссылки для тех, кто пользуется узлами с одним маркером.

Сброс на диск

Чтобы заставить Cassandra записать данные из таблиц в памяти в файлы SSTable на диске, используется команда `flush` утилиты `nodetool`:

```
$ nodetool flush
```

Заглянув в журнал сервера, вы увидите серию таких сообщений, по одному для каждой хранящейся в узле таблицы:

```
DEBUG [RMI TCP Connection(297)-127.0.0.1] 2015-12-21 19:20:50,794  
StorageService.java:2847 - Forcing flush on keyspace hotel,  
CF reservations_by_hotel_date
```

Можно избирательно сбрасывать отдельные пространства ключей или даже отдельные таблицы. Для этого нужно задать имя в командной строке:

```
$ nodetool flush hotel  
$ nodetool flush hotel reservations_by_hotel_date hotels_by_poi
```

После выполнения команды flush Cassandra может очистить некоторые участки журнала фиксаций, потому что данные уже записаны в файлы SSTable.

Команда nodetool drain похожа на flush. Она сначала производит сброс, а потом инструктирует Cassandra прекратить прием команд от клиентов и других узлов. Обычно команда drain применяется как часть процедуры упорядоченной остановки узла Cassandra и способствует ускорению его последующего запуска, поскольку не нужно будет накатывать журналы фиксаций.

Очистка

Команда cleanup просматривает все хранящиеся на узле данные и отбрасывает те, которыми узел больше не владеет. Возникает вопрос: каким образом на узле оказались данные, которыми он не владеет?

Допустим, что кластер некоторое время проработал, после чего вы решили изменить коэффициент или стратегию репликации. Если уменьшить число реплик в каком-то ЦОДе, то в нем окажутся узлы, которые уже не являются репликами для вторичных диапазонов.

Или предположим, что вы добавили в кластер новый узел и уменьшили размер диапазонов маркеров, хранящихся в каждом узле. Тогда на каждом узле останутся части диапазонов, которыми этот узел больше не владеет.

В обоих случаях Cassandra не бросается немедленно удалять ненужные данные, поскольку узел может быть остановлен на техническое обслуживание. Рано или поздно стандартный процесс уплотнения избавится от этих данных.

Однако может понадобиться поскорее освободить место, занятое лишними данными, чтобы уменьшить нагрузку на кластер. Для этого и предназначена команда nodetool cleanup.

Как и в случае flush, можно указать, какие пространства ключей и таблицы очищать. Не стоит автоматизировать выполнение очистки, потому что она нужна только после выполнения одного из описанных выше действий.

Исправление

Как было сказано в главе 6, наличие в Cassandra механизма настраиваемой согласованности означает, что со временем узлы кластера могут рассинхронизоваться. Например, операция записи при любом уровне согласованности, кроме ALL, может закончиться успешно, даже если некоторые узлы не отвечают, особенно когда кластер испытывает высокую нагрузку. Узел может пропустить запрос на обновление и в том случае, когда он не работает или недостижим в течение времени, большего, чем срок хранения напоминаний. В результате в разных репликах одного раздела могут оказаться различные версии данных.

Это особенно неприятно, когда пропущен запрос на удаление. Узел, не работавший дольше времени gc_grace_seconds, заданного для некоторой таблицы, может «воскресить» удаленные данные, когда вернется в состав кластера.

По счастью, в Cassandra имеются различные антиэнтропийные механизмы для борьбы с рассогласованием. Мы уже обсуждали, как можно использовать для этой цели исправление на этапе чтения и повышение уровня согласованности чтения. И последнее оружие в арсенале Cassandra – ручное исправление, которое запускается командой nodetool repair:

```
$ nodetool repair
[2016-01-01 14:47:59,010] Nothing to repair for keyspace 'hotel'
[2016-01-01 14:47:59,018] Nothing to repair for keyspace 'system_auth'
[2016-01-01 14:47:59,129] Starting repair command #1, repairing
  keyspace system_traces with repair options (parallelism: parallel,
  primary range: false, incremental: true, job threads: 1,
  ColumnFamilies: [], dataCenters: [], hosts: [], # of ranges: 510)
...
...
```

Результат, конечно, зависит от текущего состояния кластера. Эта команда обходит и исправляет все пространства ключей и таблицы в кластере. Можно также задать конкретные пространства ключей или таблицы в виде nodetool repair <пространство ключей> {<таблицы>}, например: nodetool repair hotel hotels_by.poi.



Ограничение области исправления

Флаг `-local` (или его длинная форма `--in-local-dc`) позволяет указать, что команда исправления должна работать только в локальном ЦОДе, а флаг `-dc <name>` (или `--in-dc <name>`) – что в ЦОДе с указанным именем.

Посмотрим, что происходит за кулисами, когда выполняется команда `nodetool repair` на некотором узле. Узел, на котором запущена команда, становится координатором запроса. Класс `org.apache.cassandra.service.ActiveRepairService` отвечает за исправление на узле-координаторе и обработку входящих запросов. Сначала `ActiveRepairService` выполняет вариант полного уплотнения в режиме чтения, называемый *контрольным уплотнением* (*validation compaction*). Во время контрольного уплотнения узел анализирует свое локальное хранилище данных и создает деревья Меркла, которые содержат хэш-значения, представляющие данные в одной из исправляемых таблиц. Этот процесс, вообще говоря, потребляет значительные ресурсы ввода-вывода и памяти.

Затем узел инициирует диалог `TreeRequest-TreeResponse`, чтобы обменяться деревьями Меркла с соседними узлами. Если деревья из разных узлов не совпадают, необходимо провести согласование узлов и определить самые поздние версии данных, которые должны быть записаны в каждом узле. При обнаружении расхождений узлы потоком передают друг другу данные из несовпавших диапазонов. Полученные в процессе исправления данные узел сохраняет в файлах `SSTable`.

Отметим, что если в таблице много данных, то деревья Меркла не опускаются до уровня отдельного раздела. Так, в узле, содержащем миллион разделов, каждый листовый узел дерева Меркла соответствует примерно 30 разделам. Потоком передаются все эти разделы, даже если в исправлении нуждается только один из них. Такое поведение называется *избыточной потоковой передачей* (*overstreaming*). Поэтому потоковая передача обычно оказывается дорогостоящей частью процесса с точки зрения потребления сетевых ресурсов и может привести к сохранению дубликатов данных, не нуждающихся в исправлении.

Этот процесс повторяется на каждом узле для всех указанных пространств ключей и таблиц, до тех пор пока не будут исправлены все диапазоны маркеров в кластере.

Поскольку исправление может оказаться накладной операцией, Cassandra предоставляет гибкие средства распределения нагрузки во времени.

Полное исправление, инкрементное исправление и антиуплотнение

В версиях Cassandra до 2.1 при исправлении проверялись все файлы SSTable на узле, теперь такой режим называется *полным исправлением*. В версии 2.1 появилось *инкрементное исправление*, когда уже исправленные данные отделяются от еще не исправленных. Этот процесс называется *антиуплотнением*.

Инкрементная процедура повышает производительность исправления, поскольку на каждом раунде исправления нужно просматривать меньше файлов SSTable. Кроме того, сокращение области просмотра означает, что участвует меньше разделов, а следовательно, деревья Меркля получаются меньше, а вместе с ними уменьшается избыточность потоковой передачи.

Cassandra включает в каждый файл SSTable дополнительные метаданные для отслеживания состояния исправления. Утилита sstablemetadata позволяет получить время исправления. Например, запустив ее для файла SSTable с данными об отелях, мы увидим, что эти данные не подвергались исправлению.

```
$ tools/bin/sstablemetadata data/data/hotel/
hotels-d089fec0677411e59f0ba9fac1d00bce/ma-5-big>Data.db
SSTable: data/data/hotel/hotels-d089fec0677411e59f0ba9fac1d00bce/ma-5-big
Partitioner: org.apache.cassandra.dht.Murmur3Partitioner
Bloom Filter FP chance: 0.010000
Minimum timestamp: 1443619302081305
Maximum timestamp: 1448201891587000
SSTable max local deletion time: 2147483647
Compression ratio: -1.0
Estimated droppable tombstones: 0.0
SSTable Level: 0
Repaired at: 0
ReplayPosition(segmentId=1449353006197, position=326)
Estimated tombstone drop times:
2147483647: 37
...
...
```



Переход на инкрементное исправление

Инкрементное исправление стало режимом по умолчанию в версии 2.2, а чтобы выполнить полное исправление, нужно указывать флаг `-full`. Если вы пользуетесь версией Cassandra младше 2.2, то обязательно почитайте в документации о дополнительных шагах, необходимых для подготовки кластера к инкрементному исправлению.

Последовательное и параллельное исправление

Последовательное исправление означает, что в каждый момент времени исправляется только один узел, а параллельное – что одновременно исправляется несколько узлов. Последовательное исправление было режимом по умолчанию до версии 2.1 включительно, а параллельное стало таковым в версии 2.2.

Если с помощью флага `-seq` запускается последовательное исправление, то на узле-координаторе и на каждом узле-реплике делается снимок данных, и по этим снимкам строятся деревья Меркла. Исправления производятся последовательно между координатором и каждой репликой. Во время последовательного исправления динамический осведомитель Cassandra участвует в повышении производительности. Поскольку узлы-реплики активно не участвуют в текущем исправлении и могут быстро отвечать на запросы, динамический осведомитель старается направлять запросы именно этим узлам.

Параллельное исправление запускается при задании флага `-rag`. В этом случае все реплики исправляются одновременно, и никакие снимки не нужны. При параллельном исправлении кластер испытывает большую нагрузку, чем при последовательном, зато и завершается оно быстрее.

Исправление диапазона разделителя

Когда на некотором узле запускается исправление, Cassandra по умолчанию исправляет все диапазоны маркеров, для которых этот узел является репликой. Это хорошо в ситуации, когда в исправлении нуждается один узел – например, тот, который некоторое время не работал, а теперь готовится к возврату в кластер.

Но если вы, как рекомендовано, регулярно запускаете исправления в целях профилактического обслуживания, то исправление всех диапазонов маркеров на каждом узле означает, что каждый диапазон будет подвергаться исправлению многократно. Поэтому у команды `nodetool repair` имеется флаг `-pr`, который позволяет исправлять только главный диапазон маркеров, или *диапазон разделителя*. Если исправлять главный диапазон каждого узла, то в конечном итоге будет исправлено все кольцо.

Исправление поддиапазона

Даже с флагом `-pr` исправление все еще может оказаться накладной операцией, поскольку главному диапазону узла может соответствовать большой объем данных. Поэтому поддерживается возмож-

ность исправления в режиме разбиения диапазона маркеров на более мелкие порции. Это процесс называется *исправлением поддиапазона*.

Исправление поддиапазона также в какой-то мере решает проблему избыточной потоковой передачи. Поскольку максимальная разрешающая способность дерева Меркля теперь означает меньший диапазон, то Cassandra может точно определить отдельные строки, нуждающиеся в исправлении.

Для запуска исправления поддиапазона нужно указать его начальный (-st) и конечный (-et) маркеры:

```
$ nodetool repair -st <start token> -et <end token>
```

Получить эти маркеры можно программно с помощью драйверов DataStax. Например, драйвер для Java предоставляет методы для получения диапазонов маркеров для заданного хоста и пространства ключей и для разбиения диапазона на поддиапазоны. Этими методами можно воспользоваться для автоматизации исправления поддиапазонов или просто распечатать значения, как мы и сделали ниже:

```
for (Host host : cluster.getMetadata().getAllHosts())
{
    for (TokenRange tokenRange :
        cluster.getMetadata().getTokenRanges("hotel", host))
    {
        for (TokenRange splitRange : tokenRange.splitEvenly(SPLIT_SIZE))
        {
            System.out.println("Start: " + splitRange.getStart().getValue() +
                               ", End: " + splitRange.getEnd().getValue());
        }
    }
}
```

Похожий алгоритм исправления диапазонов реализован в службе OpsCenter Repair Service, которую мы скоро рассмотрим.

Рекомендации по исправлению

На практике выбор подходящей стратегии исправления и следование ей – одна из наиболее сложных задач обслуживания кластера Cassandra. Контрольный список ниже призван помочь вам в принятии решения.

Частоты исправлений

Напомним, что наблюдаемая приложением согласованность данных зависит не только от используемых уровней согласованности

чтения и записи, но и от выбранной стратегии исправления. Если вы предпочитаете уровни, не гарантирующие немедленной согласованности, то исправления надо производить чаще.

Планирование исправлений

Минимизируйте влияние исправлений на работу приложения, планируя их на время наименьшей загруженности. Либо растягивайте процесс во времени: используйте исправление поддиапазонов или запускайте исправление различных пространств ключей и таблиц в разное время.

Операции, требующие исправления

Не забывайте, что некоторые операции, например изменение осведомителя в кластере или задание другого коэффициента репликации для пространства ключей, требуют запуска полного исправления.

Избегайте конфликтующих исправлений

Cassandra запрещает выполнять несколько одновременных исправлений одного и того же диапазона маркеров, так как исправление, по определению, подразумевает взаимодействие между узлами. Поэтому лучше всегда запускать исправление из одного места, внешнего по отношению к кластеру, чем пытаться реализовать на каждом узле автоматизированные процедуры для исправления диапазонов, которыми владеет данный узел.

Дотехпор, пока не будет внедрен более надежный механизм получения статуса (см., например, предложение CASSANDRA-10302 в JIRA по адресу <https://issues.apache.org/jira/browse/CASSANDRA-10302>), наблюдать за текущими исправлениями можно с помощью команды nodetool netstats.

Переиндексирование

Вторичные индексы могут рассинхронизироваться, как и любые другие данные. Но хотя Cassandra хранит вторичные индексы как таблицы, эти таблицы ссылаются только на данные, хранящиеся в локальном узле. Поэтому описанные выше механизмы исправления не способны поддерживать индексы в актуальном состоянии.

Поскольку вторичные индексы нельзя исправить и не существует простого способа проверить их корректность, Cassandra предоставляет возможность перестроить их заново с помощью команды nodetool rebuild_index. Рекомендуется перестраивать индекс после исправления таблицы, над которой он построен, поскольку значения в индексированных столбцах могли быть исправлены. Помните, что переин-

дексирование, как и исправление, – дорогая операция с точки зрения использования ресурсов процессора и ввода-вывода.

Перемещение маркеров

Если кластер настроен на использование v-узлов (это подразумевается по умолчанию, начиная с версии 2.0), то Cassandra сама управляет назначением диапазонов маркеров каждому узлу кластера. Это касается и изменения назначенных диапазонов после добавления или удаления узлов. Однако если вы пользуетесь узлами с одним маркером, то переконфигурировать их придется вручную.

Для этого нужно сначала пересчитать диапазоны маркеров для каждого узла, как описано в главе 7. Затем выполните команду `nodetool move` для назначения диапазонов. Эта команда принимает один аргумент – новый начальный маркер для узла:

```
$ nodetool move 3074457345618258600
```

Изменив назначение диапазонов для всех узлов, завершите процесс, выполнив на каждом узле команду `nodetool cleanup`.

Добавление узлов

В главе 7 было сказано несколько слов о том, как добавить узел с помощью диспетчера кластера Cassandra (`csm`), тогда нам этого вполне хватило, чтобы приступить к работе. Но теперь копнем глубже и обсудим причины и процедуры добавления новых узлов и центров обработки данных.

Добавление узлов в существующий центр обработки данных

Если приложение пользуется успехом, то рано или поздно наступает момент, когда в кластер нужно добавлять узлы. Это может быть запланированное наращивание вычислительных мощностей или реакция на ситуацию, замеченную при проверке исправности кластера, например близящееся исчерпание места на дисках, или высокое потребление памяти и процессора некоторыми узлами, или увеличение задержки чтения и записи.

Но какова бы не была причина расширения, сначала нужно установить и настроить Cassandra на машинах, где будут развернуты новые узлы. Процесс похож на описанный в главе 7, однако нужно помнить о нескольких моментах.

- Версия Cassandra должна быть такой же, как на уже существующих узлах. Если вы планируете перейти на новую версию, то сначала сделайте это на имеющихся узлах, а потом добавляйте новые.
- В файлах *cassandra.yaml* и *cassandra-env.sh* задавайте такие же конфигурационные параметры, как для других узлов. Это относится, в частности, к параметрам *cluster_name*, *dynamic_snitch*, *partitioner* и *listen_address*.
- Задавайте те же узлы-распространители, что и на других узлах. Обычно новые узлы не делают распространителями, поэтому нет нужды добавлять их в списки распространителей на старых узлах.
- Если в ЦОДе имеется несколько стоек, то рекомендуется добавлять одинаковое число узлов во все стойки, чтобы сохранить сбалансированность. Это напоминает мне классическую настольную игру «Монополия», в которой дома на принадлежащих вам улицах нужно строить равномерно.
- Если используются узлы с одним маркером, то необходимо вручную пересчитывать диапазоны маркеров, назначаемых каждому узлу, как описано в разделе «Перемещение маркеров» выше в этой главе. Простой и эффективный способ поддержать сбалансированность кластера – поделить каждый диапазон пополам и удвоить число узлов в кластере.
- В большинстве случаев новые узлы конфигурируются, так чтобы они сразу же приняли участие в начальной загрузке, т. е. объявили о своих диапазонах, после чего начинается потоковая перекачка данных с других узлов. Этим поведением управляет свойство *autobootstrap*, по умолчанию равное *true*. Вы можете добавить его в файл *cassandra.yaml*, чтобы явно разрешить или запретить автоматическую начальную загрузку.

После того как все узлы настроены, их можно запустить и с помощью команды *nodetool status* проверить, корректно ли произошла инициализация.

Можно также последить за процессом начальной загрузки узла, выполнив команду *nodetool bootstrap*. Если автоматическая начальная загрузка узла выключена, то ее можно инициировать удаленно в удобный момент с помощью команды *nodetool bootstrap resume*.

Когда все новые узлы войдут в стационарный режим работы, не забудьте выполнить команду *nodetool cleanup* на каждом из старых узлов, чтобы вычистить данные, которыми узел больше не владеет.

Добавление центра обработки данных в кластер

Для добавления целого нового ЦОДа в кластер могут быть разные причины. Допустим, к примеру, что вы развертываете свое приложение в новом ЦОДе, чтобы уменьшить сетевую задержку, испытываемую клиентами на новом рынке. Или требуется конфигурация активный–активный для обеспечения требований к аварийному восстановлению. Еще одна распространенная причина для создания отдельного ЦОДа – использование его для решения аналитических задач без помех оперативной обработке транзакций.

В главе 14 мы рассмотрим некоторые из описанных ситуаций подробнее, а пока займемся технической проблемой включения в кластер нового ЦОДа.

К новому ЦОДу применимы те же основные шаги, что и в случае добавления нового узла в существующий ЦОД. Но при редактировании файла *cassandra.yaml* на каждом новом узле нужно помнить о нескольких моментах.

- Не забудьте сконфигурировать подходящий осведомитель, задав свойство `endpoint_snitch`, и отредактировать все конфигурационные файлы, относящиеся к осведомителю. Хочется надеяться, что вы планировали это, когда только настраивали кластер, но если нет, то придется менять осведомитель в первоначальном ЦОДе. Если это так, то сначала нужно изменить его в узлах существующего ЦОДа, произвести исправление и только потом добавлять новый ЦОД.
- Выберите в новом ЦОДе два узла, которые станут распространителями, и соответственно настройте свойство `seeds` в остальных узлах. В каждом ЦОДе должны быть свои распространители, не зависящие от других ЦОДов.
- Конфигурация диапазонов маркеров в новом ЦОДе не обязана быть такой же, как в существующих ЦОДах. Можете задать другое число v-узлов или использовать узлы с одним маркером.
- Выключите автоматическую начальную загрузку, для чего найдите (или добавьте) свойство `autobootstrap` и присвойте ему значение `false`. Тогда новые узлы не будут пытаться начать потоковую перекачку данных раньше времени.

После того как все узлы в новом ЦОДе «поднимутся», настройте параметры репликации, выбрав стратегию `NetworkTopologyStrategy` для всех пространств ключей, которые должны реплицироваться в новый ЦОД.

Например, чтобы распространить наше пространство ключей `hotel` еще на один ЦОД, можно выполнить такую команду:

```
cqlsh> ALTER KEYSPACE hotel WITH REPLICATION =
  {'class' : 'NetworkTopologyStrategy', 'dcl' : 3, 'dc2' : 2};
```

Отметим, что стратегия `NetworkTopologyStrategy` позволяет задавать разное число реплик в разных ЦОДах.

Далее выполните команду `nodetool rebuild` на каждом узле нового ЦОДа. Так, следующая команда заставляет узел перестроить свои данные, скачав их потоком из ЦОДа `dcl`:

```
$ nodetool rebuild -- dcl
```

При желании можно параллельно перестраивать несколько узлов; не забудьте только предварительно проанализировать, как это отразится на работе кластера.

После завершения перестройки новый ЦОД готов к работе. Не забудьте отредактировать файл `cassandra.yaml` на каждом узле нового ЦОДа, удалив свойство `autobootstrap: false`, чтобы узлы нормально восстанавливались после перезапуска.



Не забывайте своих клиентов

Нужно также помнить о том, как добавление еще одного ЦОДа отразится на существующих клиентах, точнее на использовании ими уровней согласованности `LOCAL_*` и `EACH_*`. Например, если некоторые клиенты указывают уровень согласованности `QUORUM` для чтения и записи, то запросы, выполнение которых раньше ограничивалось одним ЦОДом, теперь будут выполняться в нескольких. Быть может, для ограничения задержки стоит перейти на уровень `LOCAL_QUORUM` или, наоборот, указать `EACH_QUORUM`, требуя строгой согласованности во всех ЦОДах.

Обработка отказа узла

Иногда узлы Cassandra отказывают. Причины могут быть разными: отказ оборудования, крах процесса Cassandra process, остановка или уничтожение виртуальной машины. Узел, недоступный сети, может быть помечен как отказавший в результате распространения сплетен, и команда `nodetool status` покажет, что он не работает, хотя после восстановления связности сети узел может вернуться.

В этом разделе мы поговорим о том, как починить или заменить отказавший узел, а также о том, как корректно исключать узлы из кластера.

Ремонт узлов

Обнаружив отказавший узел, нужно прежде всего попытаться определить, как долго он не работает. Ниже приведены эвристические правила, позволяющие решить, что делать: чинить или заменять узел.

- Если узел не работает меньше, чем срок хранения напоминаний, заданный свойством `max_hint_window_in_ms`, то механизм вручения напоминаний сможет его восстановить. Перезапустите узел и посмотрите, восстановится ли он. Можно заглянуть в журнал узла или понаблюдать за ходом восстановления с помощью команды `nodetool status`.
- Если узел не работает меньше, чем срок исправления, определяемый как минимум из значений свойства `gc_grace_seconds` по всем хранящимся на нем таблицам, то перезапустите узел. Если он успешно поднимется, выполните команду `nodetool repair`.
- Если узел не работает дольше, чем срок исправления, то его следует заменить во избежание воскрешения надгробий.

Восстановление после отказа диска

Отказ диска – один из видов отказа оборудования, после которого узел еще можно восстановить. Если узел настроен на использование нескольких дисков (JBOD), то свойство `disk_failure_policy` определяет, что делать в случае отказа дисков и как обнаружить такой отказ.

- Если задана политика по умолчанию (`stop`), то узел прекратит участвовать в распространении сплетен, в результате чего команда `nodetool status` будет показывать его как неработающий. Но к узлу все еще можно подключиться по JMX.
- Если задана политика `die`, то JVM завершает работу, и `nodetool status` будет показывать его как неработающий.
- Если задана политика `ignore`, то сразу обнаружить отказ невозможно.
- Если задана политика `best_effort`, то Cassandra будет продолжать работу с другими дисками, но запишет в журнал предупреждение, которое можно увидеть, если используется какое-нибудь средство агрегирования журналов. Можно вместо этого воспользоваться средством мониторинга на основе JMX для наблюдения за состоянием MBean-объекта `org.apache.cassandra.db.BlacklistedDirectoriesMBean`, перечисляющим каталоги, для которых были зарегистрированы отказы.

- Если отказ диска обнаружен на этапе запуска узла и задана любая политика, кроме `best_effort`, то в журнал записывается сообщение об ошибке и узел завершает работу.

Обнаружив отказ диска, можно попытаться перезапустить процесс Cassandra или перезагрузить узел. Но если отказ повторяется, то придется заменить диск и удалить содержимое каталога `data/system` на оставшихся дисках, чтобы после перезапуска узел оказался в согласованном состоянии. Когда узел поднимается, запустите процедуру исправления.

Замена узлов

Придя к выводу, что узел отремонтировать невозможно, вы, скорее всего, захотите заменить его, чтобы кластер остался сбалансированным и сохранил прежнюю емкость.

Конечно, можно было бы исключить старый узел (как описано в следующем разделе) и добавить новый, но это не самый эффективный способ замены узла. Исключение и последующее добавление узлов порождает излишнюю потоковую передачу данных по сети – сначала чтобы убрать данные со старого узла, а затем чтобы переместить их на новый узел.

Более эффективно – добавить узел, который принимает на себя все диапазоны маркеров, за которые отвечает существующий узел. Для этого нужно выполнить описанную выше процедуру добавления узла с одним дополнением. Добавьте в файл `cassandra-env.sh` на новом узле следующую строку параметров JVM (здесь `<address>` – IP-адрес или имя хоста заменяемого узла):

```
JVM_OPTS="$JVM_OPTS -Dcassandra.replace_address=<address>"
```

После того как новый узел закончит процедуру начальной загрузки, эту строку можно удалить, поскольку больше она во время перезагрузки не понадобится.

Если используется осведомитель, которому нужен файл свойств для определения топологии кластера, например `GossipingPropertyFileSnitch` или `PropertyFileSnitch`, то надо будет добавить адрес нового узла в файл свойств на каждом узле и произвести поочередный перезапуск узлов кластера. Рекомендуется подождать 72 часа, перед тем как удалять адрес старого узла, чтобы не сбить работу протокола распространения сплетен.



Замена узла-распространителя

Если заменяется узел-распространитель, то назначьте на эту роль какой-нибудь из существующих узлов. Выбранный узел нужно будет добавить в свойство `seeds` в файле `cassandra.yaml` на всех существующих узлах.

Обычно распространителя выбирают из числа узлов в том же ЦОДе, согласно рекомендации использовать разные списки распространителей в каждом ЦОДе. Таким образом, новый созданный узел не будет распространителем и сможет нормально выполнить начальную загрузку.

При использовании пакетного дистрибутива Cassandra имеются дополнительные тонкости, ознакомьтесь с документацией по конкретной версии.

Исключение узлов

Если вы решили не заменять вышедший из строя узел немедленно или просто хотите уменьшить кластер, то необходимо исключить узел или вывести его из эксплуатации. Метод исключения зависит от того, поднят ли узел или может ли он подняться в будущем. Мы рассмотрим три метода в порядке предпочтительности: вывод из эксплуатации, исключение и уничтожение.

Вывод узла из эксплуатации

Если узел показывается как работающий, то его исключение называется *выводом из эксплуатации* (*decommission*). Команда `nodetool decommission` инициирует вызов соответствующего метода класса `StorageService`. В ходе этой операции диапазоны маркеров, за которые отвечает данный узел, распределяются между другими узлами, после чего данные потоком перекачиваются на эти узлы. По существу, это противоположность начальной загрузке.

Во время работы процедуры вывода из эксплуатации команда `nodetool status` показывает статус `UL` (`up, leaving`), т. е. узел еще работает, но находится в процессе выхода из кластера.

```
$ nodetool status
Datacenter: datacenter1
=====
Status=Up/Down
| / State=Normal/Leaving/Joining/Moving
-- Address      Load      Tokens  Owns    Host ID            Rack
UN 127.0.0.1  340.25 KB  256      ?      31d9042b-6603-4040-8aac-...  rack1
```

```
UN 127.0.0.2 254.31 KB 256 ? caad1573-4157-43d2-a9fa-... rack1
UN 127.0.0.3 259.45 KB 256 ? e78529c8-ee9f-46a4-8bcl-... rack1
UL 127.0.0.4 116.33 KB 256 ? 706a2d42-32b8-4a3a-85b7-... rack1
```

За ходом вывода из эксплуатации можно следить по журналу сервера. Так, в сообщениях показанного ниже вида для каждого диапазона маркеров указывается его новый владелец:

```
DEBUG [RMI TCP Connection(4)-127.0.0.1] 2016-01-07 06:00:20,923
StorageService.java:2369 - Range {-1517961367820069851,-1490120499577273657}
will be responsibility of /127.0.0.3
```

Затем будет присутствовать сообщение о начале потоковой перекачки данных на другие узлы.

```
INFO [RMI TCP Connection(4)-127.0.0.1] 2016-01-07 06:00:21,274
StorageService.java:1191 - LEAVING: replaying batch log and
streaming data to other nodes
```

Для наблюдения за перекачкой данных на новые узлы-реплики можно также использовать команду nodetool netstats.

По окончании перекачки узел в течение 30 секунд рассыпает объявления о своем выходе из кластера, а затем останавливается.

```
INFO [RMI TCP Connection(4)-127.0.0.1] 2016-01-07 06:00:22,526
StorageService.java:3425 - Announcing that I have left the ring for 30000ms
...
WARN [RMI TCP Connection(4)-127.0.0.1] 2016-01-07 06:00:52,529
Gossiper.java:1461 - No local state or state is in silent shutdown, not
announcing shutdown
```

И наконец, сообщение о завершении вывода из эксплуатации:

```
INFO [RMI TCP Connection(4)-127.0.0.1] 2016-01-07 06:00:52,662
StorageService.java:1191 - DECOMMISSIONED
```

При попытке выполнить команду decommission для узла, который нельзя вывести из эксплуатации (потому что он еще не стал частью кольца или является единственным доступным узлом), будет выдано соответствующее сообщение.



Вывод из эксплуатации не означает удаления файлов

Помните, что данные, хранившиеся на выведенном из эксплуатации узле, автоматически не удаляются. Если впоследствии вы решите снова ввести такой узел в состав кластера, но уже с другим диапазоном, то предварительно удалите данные вручную.

Исключение узла

Если узел не работает, то вместо команды `decommission` нужно использовать команду `nodetool removenode`. Если в кластере используются v-узлы, то по команде `removenode` Cassandra пересчитает диапазоны маркеров для оставшихся узлов и перекачает данные из текущих реплик новому владельцу каждого диапазона.

Если в кластере нет v-узлов, то нужно будет вручную переназначить диапазоны маркеров оставшимся узлам (как описано в разделе «Перемещение маркеров» выше), перед тем как выполнять команду `removenode`, которая произведет перекачку данных. У команды `removenode` имеется флаг `-status`, позволяющий следить за ходом перекачки.

Большинство команд `nodetool` работает непосредственно с узлом, заданным флагом `-h`. Но команда `removenode` устроена несколько иначе, потому что должна работать не на том узле, который удаляется. Целевой узел задается не IP-адресом, а идентификатором хоста, который можно получить от команды `nodetool status`:

```
$ nodetool status
Datacenter: datacenter1
=====
Status=Up/Down
| / State=Normal/Leaving/Joining/Moving
--  Address      Load      Tokens  Owns    Host ID
UN  127.0.0.1   244.71 KB  256      ?      31d9042b-6603-4040-8aac-fef0a235570b
UN  127.0.0.2   224.96 KB  256      ?      caad1573-4157-43d2-a9fa-88f79344683d
DN  127.0.0.3   230.37 KB  256      ?      e78529c8-ee9f-46a4-8bc1-3479f99a1860

$ nodetool removenode e78529c8-ee9f-46a4-8bc1-3479f99a1860
```

Уничтожение узла

Если операция `nodetool removenode` завершается ошибкой, то остается последняя надежда – команда `nodetool assassinate`. Команда `assassinate` отличается от `removenode` тем, что не перераспределяет данные исключенного узла между другими репликами. В результате кластер оказывается в состоянии, нуждающемся в исправлении.

Еще одно важное отличие от `removenode` заключается в том, что `assassinate` принимает IP-адрес уничтожаемого узла, а не идентификатор его хоста:

```
$ nodetool assassinate 127.0.0.3
```



Не забудьте поправить узел-распространитель

Если удаляется узел-распространитель, то нужно обязательно поправить файл `cassandra.yaml` на всех оставшихся узлах, убрав IP-адрес удаленного узла. Кроме того, нужно следить за количеством распространителей в кластере (как минимум, два в каждом ЦОДе).

Переход на новую версию Cassandra

Поскольку Cassandra продолжает жить и развиваться, регулярно выходят новые версии с новой функциональностью, исправленными ошибками и улучшенной производительностью. Сообщество Apache установило периодичность выхода версий, а вам стоит соответственно планировать переход на них.

Какая версия Cassandra «самая подходящая»?

При выборе версии Cassandra следует избегать крайностей и держаться подальше от «слишком горячего» и «слишком холодного».

Так, не стоит развертывать версию с новым главным номером сразу после выпуска. Так уж повелось, что новая функциональность в главной версии поначалу нестабильна, и вряд ли стоит записываться в волонтеры по ее тестированию в боевых условиях.

С другой стороны, слишком сильно отставать тоже нехорошо, поскольку чем дольше вы будете выжидать, тем сложнее окажется процедура перехода на новую версию. Архивы списка рассылки Cassandra изобилиуют вопросами и сетованиями по поводу старых версий, отправленными, когда ошибка давно уже исправлена.

Решив, что время модернизации настало, внимательно прочитайте файл `NEWS.txt` в корневом каталоге новой версии и следуйте содержащимся в нем инструкциям по переходу с вашей текущей версии на новую. Переход может оказаться сложным процессом, и пренебрежение инструкциями может привести к причинению масштабного вреда кластеру.

Ознакомьтесь также с общим руководством по переходу на новую версию на сайте DataStax. В нем приведены прекрасный обзор процедуры перехода и список промежуточных этапов, которые необходимо пройти для перехода от исходной версии к конечной.

Кластер Cassandra модернизируется с помощью процесса *поочередной модернизации* (*rolling upgrade*), когда за один шаг обновляется

один узел. Для выполнения поочередной модернизации нужно выполнить следующие действия.

1. Сначала выполните на узле команду `nodetool drain`, чтобы сбросить на диск все еще не сброшенные данные и прекратить прием новых запросов на запись.
2. Остановите узел.
3. Сделайте резервные копии конфигурационных файлов `cassandra.yaml`, `cassandraenv.sh` и прочих, чтобы не затереть их.
4. Установите новую версию.
5. Модифицируйте конфигурационные файлы в соответствии со своими требованиями.

Рекомендации по обновлению конфигурационных файлов

При переходе на новую версию Cassandra может возникнуть искушение использовать файл `cassandra.yaml` из предыдущей версии. Проблема в том, что в большинстве версий добавляются новые конфигурационные параметры, а значения по умолчанию старых параметров иногда изменяются. И, взяв прежний файл, вы лишите себя возможности воспользоваться преимуществами того и другого. А если вы переходите сразу на несколько версий вперед, то не исключено, что со старыми параметрами Cassandra будет вообще работать неправильно. Поэтому рекомендуется взять за основу файл `cassandra.yaml` из новой версии и включить в него изменения конфигурации, произведенные ранее, в т. ч. указать правильное имя кластера, список узлов-репликаторов и разделитель. Напомним, что если в кластере есть данные, то менять разделитель не разрешается.

Если вы работаете на платформе Unix, то объединение файлов можно произвести вручную, найдя различия с помощью команды `diff`. Или воспользоваться средствами объединения, предоставляемыми системой управления версиями или интегрированной средой разработки.

При переходе на версию с другим главным номером (а иногда и при смене дополнительного номера) необходимо выполнить команду `nodetool upgradesstables` для преобразования формата файлов данных. Как и в случае ранее рассмотренных команд `nodetool`, можно указать пространство ключей или даже имена отдельных таблиц, но, вообще говоря, требуется модернизировать все таблицы, хранящиеся в узле. Обновить таблицы можно и тогда, когда узел отключен от кластера; для этого служит скрипт `bin/sstableupgrade`.

Эти действия следует повторить на каждом узле кластера. Во время поочередной модернизации кластер продолжает работать, но следует тщательно спланировать переход на новую версию, принимая во внимание размер кластера. Пока кластер модернируется, не следует вносить изменения в схему или запускать процедуру исправления.

Резервное копирование и восстановление

Благодаря настраиваемой репликации и нескольким ЦОДам Cassandra очень устойчива к отказам. Тем не менее есть ряд причин для резервного копирования данных.

- Дефекты в логике приложения могут привести к затиранию правильных данных, и такие ошибочные данные будут реплицированы на все узлы раньше, чем проблема будет осознана.
- Возможно повреждение файлов SSTable.
- Сбой, распространяющийся на несколько ЦОДов, может свести на нет ваш план аварийного восстановления.

Cassandra предоставляет два механизма резервного копирования данных: *снимки* и *инкрементные резервные копии*. Снимок содержит полную резервную копию, а инкрементные копии позволяют копировать только изменения.

Полные, инкрементные и дифференциальные резервные копии

К резервному копированию баз данных есть три стандартных подхода.

- *Полная резервная копия* содержит полное состояние всей базы данных (или отдельных таблиц), ее создание обходится дороже всего.
- *Инкрементная резервная копия* содержит изменения за некоторый период времени, обычно с момента создания предыдущей инкрементной копии. Серия инкрементных резервных копий образует дифференциальную резервную копию.
- *Дифференциальная резервная копия* содержит все изменения, произведенные с момента создания предыдущей полной копии.

В Cassandra нет встроенного механизма дифференциальных копий, но есть возможность полного и инкрементного резервного копирования.

Снимки и резервные копии Cassandra – взаимно дополняющие механизмы, в совокупности обеспечивающие надежную схему резервного копирования и восстановления.

В обоих случаях создаются физические ссылки на файлы SSTable, что позволяет не создавать дополнительных файлов, существующих непродолжительное время. Однако эти файлы могут пополняться, пока производится уплотнение, а файлы, удаленные из каталога данных,держиваются с помощью ссылок.

Задача копирования файлов в другое место и их удаления для освобождения места на диске возлагается на пользователя. Но эти задачи легко автоматизировать, что и сделано во многих инструментах, в т. ч. в программе DataStax OpsCenter, которую мы скоро рассмотрим. Из инструментов с открытым исходным кодом упомянем Tablesnap Джереми Гроссера (<https://github.com/JeremyGrosser/tablesnap>).

Создание снимка

Цель снимка – сделать копию некоторых или всех пространств ключей и таблиц, хранящихся в узле, и сохранить ее в виде отдельного файла базы данных. Резервные копии можно затем переместить в другое место или оставить там, где они находятся, на случай, если они понадобятся для восстановления. Перед созданием снимка Cassandra производитброс, а затем создает физические ссылки на каждый файл SSTable.

Создать снимок просто:

```
$ nodetool snapshot
Requested creating snapshot(s) for [all keyspaces] with snapshot name
[1452265846037]
Snapshot directory: 1452265846037
```

Эта команда создает снимок всех пространств ключей на сервере, включая системное пространство ключей system. Чтобы сделать снимок только одного пространства ключей, его имя нужно передать в качестве аргумента: nodetool snapshot hotel. Можно также задать имя конкретной таблицы с помощью флага -cf.

Команда nodetool listsnapshots выводит список всех сделанных снимков:

```
$ nodetool listsnapshots
Snapshot name  Keyspace name  Column family name  True size  Size on disk
1452265846037  hotel          pois_by_hotel        0 bytes    13 bytes
1452265846037  hotel          hotels            0 bytes    13 bytes
...
Total TrueDiskSpaceUsed: 0 bytes
```

Чтобы найти снимки в файловой системе, вспомните, что каталог данных организован в виде подкаталогов, содержащих пространства ключей и таблицы. В каталоге каждой таблицы имеется подкаталог *snapshots*, а каждый снимок хранится в его подкаталоге с именем, равным временной метке, соответствующей моменту создания. Например, снимок таблицы *hotels* находится в каталоге

```
$CASSANDRA_HOME/data/data/hotel/hotels-b9282710a78a11e5a  
0a5fb1a2fbe47/snapshots/1452265846037/
```

В каждом снимке имеется также файл *manifest.json*, в котором перечислены включенные в снимок файлы SSTable. Он позволяет проверить, все ли содержимое снимка в наличии.



Снимки нескольких узлов на момент времени

Команда *nodetool snapshot* применяется только к одному серверу. Чтобы получить снимок нескольких узлов на момент времени, нужно запустить ее одновременно на нескольких серверах с помощью какой-нибудь программы параллельного ssh, например *pssh*.

Cassandra предоставляет также функцию *автоматического снимка* (*auto snapshot*), которая создает снимок при каждом выполнении операций *DROP KEYSPACE*, *DROP TABLE* или *TRUNCATE*. Она страхует от не преднамеренной потери данных и активируется с помощью свойства *auto_snapshot* в файле *cassandra.yaml*. Есть также свойство *snapshot_before_compaction* (сделать снимок перед уплотнением), которое по умолчанию равно *false*.

Удаление снимка

Снимки можно удалять, например после перемещения на постоянное хранение на другой территории. Рекомендуется удалять старые снимки перед созданием новых.

Для стирания снимков можно удалить файлы вручную или воспользоваться командой *nodetool clearsnapshot*, которая принимает имя пространства ключей в качестве необязательного аргумента.

Включение инкрементного резервного копирования

После создания снимка можно включить режим инкрементного резервного копирования командой *nodetool enablebackup*. Эта команда применяется ко всем пространствам ключей и таблицам, хранящимся в узле.

Проверить, включено ли инкрементное резервное копирование, позволяет команда nodetool statusbackup, а выключить его – команда nodetool disablebackup.

Если инкрементное резервное копирование включено, то Cassandra создает резервные копии в процессе сброса файлов SSTable на диск. Копия состоит из физических ссылок на каждый файл данных и создается в каталоге *backups*, например:

```
$CASSANDRA_HOME/data/data/hotel/hotels-b9282710a78a11e5a  
0a5fb1a2fbe47/backups/
```

Автоматическим созданием инкрементных резервных копий управляет свойство *incremental_backups* в файле *cassandra.yaml*.

После создания снимка и перемещения его в место постоянного хранения инкрементные копии можно спокойно удалить.

Восстановление из снимка

Процесс восстановления узла из резервных копий начинается с подбора последнего созданного снимка и всех созданных после него инкрементных копий. Снимки и инкрементные копии обрабатываются одинаково.

Прежде чем начинать восстановление узла, рекомендуется усечь все таблицы и тем самым стереть изменения, произведенные после создания снимка.



Не забудьте восстановить схему!

Имейте в виду, что Cassandra не включает схему базы данных в состав снимков и инкрементных резервных копий. Поэтому позаботьтесь о том, чтобы до начала восстановления схема оказалась на месте. По счастью, это нетрудно сделать с помощью команды `cqlsh DESCRIBE TABLES`, которую легко включить в скрипт.

Если с момента создания снимка не изменились ни топология кластера, ни диапазоны маркеров в узлах, ни коэффициент репликации для включенных в снимок таблиц, то можно скопировать файлы SSTable в каталог данных на каждом узле. Если узел работает, то уведомить Cassandra о наличии новых данных позволит команда `nodetool refresh`.

Если топология, диапазоны маркеров или коэффициент репликации изменились, то для загрузки данных придется воспользоваться программой `sstableloader`. В некоторых отношениях она ведет себя как узел Cassandra: использует протокол распространения сплетен,

чтобы узнать об узлах кластера, но не регистрирует себя в качестве узла. Для передачи файлов SSTable на узлы используются потоковые библиотеки Cassandra. Программа sstableloader не копирует файлы SSTable напрямую на каждый узел, а вставляет данные из каждого файла SSTable в кластер, давая разделителю и стратегии репликации выполнить свою часть работы.

Программа sstableloader полезна также для перемещения данных между кластерами.

УТИЛИТЫ ДЛЯ РАБОТЫ С ФАЙЛАМИ SSTABLE

В каталогах *bin* и *tools/bin* есть несколько утилит, работающих непосредственно с файлами данных SSTable в узле Cassandra. Эти файлы имеют расширение *.db*, например:

```
$CASSANDRA_HOME/data/hotels-b9282710a78a11e5a0a5fb1a2fb  
efd47/ma-1-big-Data.db
```

Помимо программ sstablemetadata, sstableloader и sstableupgrade, с которыми мы уже знакомы, предлагаются следующие утилиты:

- sstableutil выводит список файлов SSTable для таблицы с указанным именем;
- sstablekeys выводит список ключей раздела, хранящихся в данном файле SSTable;
- sstableverify проверяет файлы SSTable для пространства ключей и таблицы с указанными именами, выявляя файлы с ошибками или поврежденными данными. Это автономный вариант команды nodetool verify;
- sstablescrub – автономный вариант команды nodetool scrub. Поскольку для ее работы не нужна сеть, ее можно использовать для удаления поврежденных данных из файлов SSTable. Если обнаружатся поврежденные строки, то нужно будет выполнить исправление;
- sstablerepairedset помечает конкретные файлы SSTable как исправленные или неисправленные, чтобы упростить переход на режим инкрементного исправления. Поскольку этот режим подразумевается по умолчанию, начиная с версии 2.2, эта утилита не нужна для кластеров, созданных в версии 2.2 или позже.

Некоторые утилиты помогают управлять уплотнением, о котором мы еще поговорим в главе 12.

- `sstableexpiredblockers` находит блокирующие файлы SSTable, которые мешают удалению некоторого файла SSTable. Она выводит все файлы SSTable, блокирующие удаление других файлов, и таким образом позволяет определить, почему некоторый файл все еще находится на диске.
- `sstablelevelreset` сбрасывает в 0 уровень для заданного набора файлов SSTable, что приводит к принудительному уплотнению файлов при следующей операции уплотнения.
- `sstableofflinerelevel` переназначает уровни заданным файлам SSTable, используя стратегию LeveledCompactionStrategy. Это полезно, если в течение короткого времени был вставлен большой объем данных, например в результате массового импорта.
- `sstablesplit` разбивает файлы SSTable на меньшие – заданного максимального размера. Это полезно, если в результате полного уплотнения образовались большие таблицы, которые в противном случае пришлось бы уплотнять долго.

В обычных условиях нужда в этих программах возникает не часто, но они весьма полезны для отладки и лучшего понимания работы механизмов хранения данных в Cassandra. Утилиты, модифицирующие файлы SSTable, – `sstablelevelreset`, `sstablerepairedset`, `sstablesplit`, `sstableofflinerelevel` – следует запускать, когда Cassandra не работает на локальном хосте.

Средства обслуживания

Без сомнения, для обслуживания кластера Cassandra хватило бы и одной утилиты `nodetool`, но во многих организациях, особенно эксплуатирующих большие кластеры, предпочитают более удобные инструменты, предоставляющие средства автоматизации обслуживания и улучшенную визуализацию.

Дадим краткий обзор двух таких инструментов: DataStax OpsCenter и Netflix Priam.

DataStax OpsCenter

DataStax OpsCenter – это веб-приложение для управления и мониторинга кластера Cassandra, которое автоматизирует многие задачи обслуживания, в т. ч. и рассмотренные выше в этой главе. Существуют два издания OpsCenter: бесплатное (Community Edition), которое управляет кластерами, построенными на базе дистрибутива Apache

Cassandra, и платное (Enterprise) для управления кластерами типа DataStax Enterprise.

Основой OpsCenter является информационная панель, на которой наглядно отображается состояние кластера (рис. 11.1).

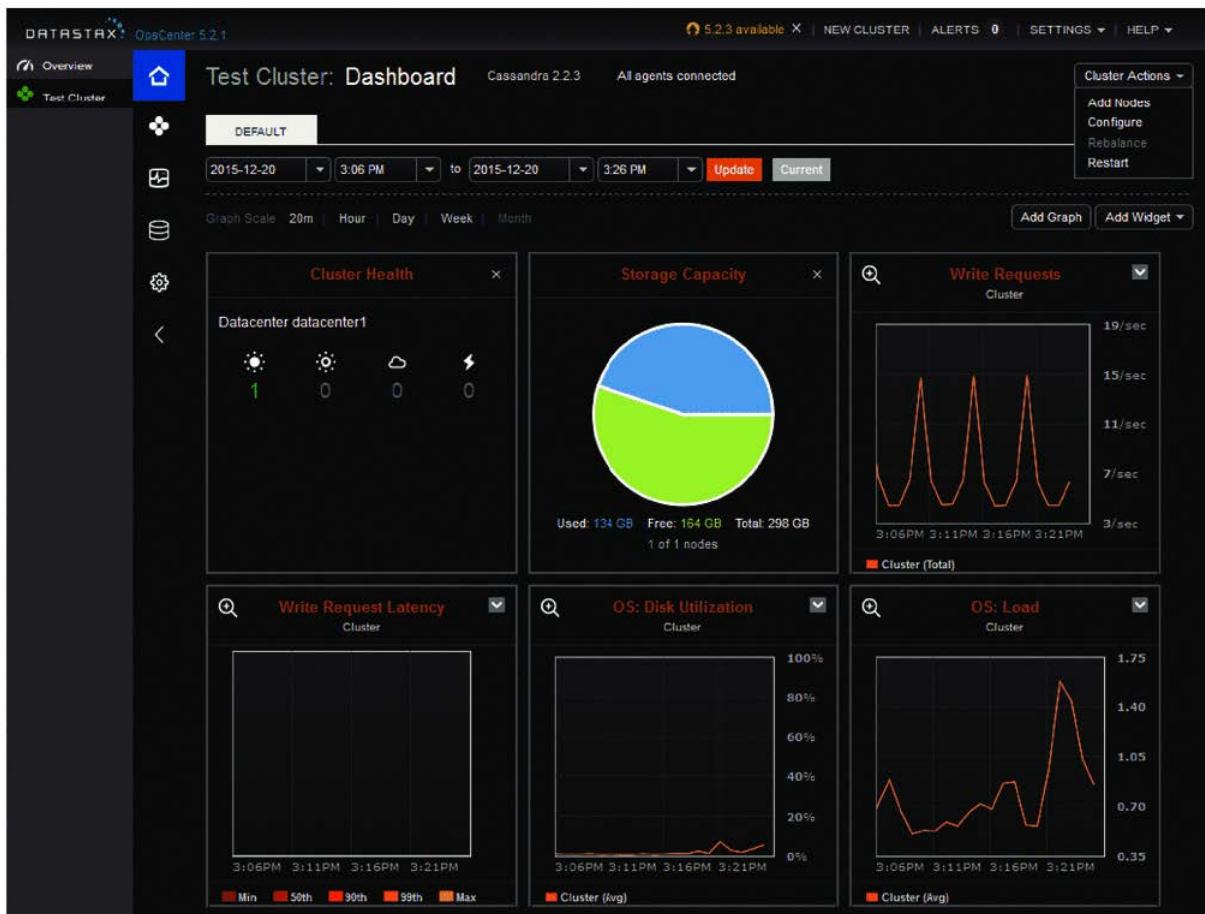


Рис. 11.1 ♦ Информационная панель DataStax OpsCenter для тестового кластера

Помимо состояния кластера, на информационной панели отображаются метрики, измеряющие емкость и задержку записи. Панель можно настроить, добавив графики для многих других метрик, которые Cassandra предоставляет с помощью JMX. Издание Enterprise Edition позволяет, кроме того, настраивать пороги метрик, при достижении которых генерируются оповещения и уведомления по электронной почте.

В правом верхнем углу представлены доступные операции над кластером: добавление узлов, настройка параметров в файлах *cassandra.yaml* на любом узле и перезапуск кластера. Имеется также команда Rebalance, позволяющая автоматически перераспределить диапазоны маркеров в кластерах, где не используются виртуальные узлы.

В издание DataStax Enterprise входит также служба исправления Repair Service, которая автоматизирует исправления на всех узлах кластера. Эта служба работает непрерывно, применяя метод инкрементного исправления к поддиапазонам. Служба Repair Service следит за ходом исправления и при необходимости замедляет его темп, чтобы уменьшить негативное влияние на кластер. После исправления всех поддиапазонов служба начинает новый раунд.

Представление OpsCenter Nodes дает полезную графическую картину узлов кластера в виде кольца (рис. 11.2).

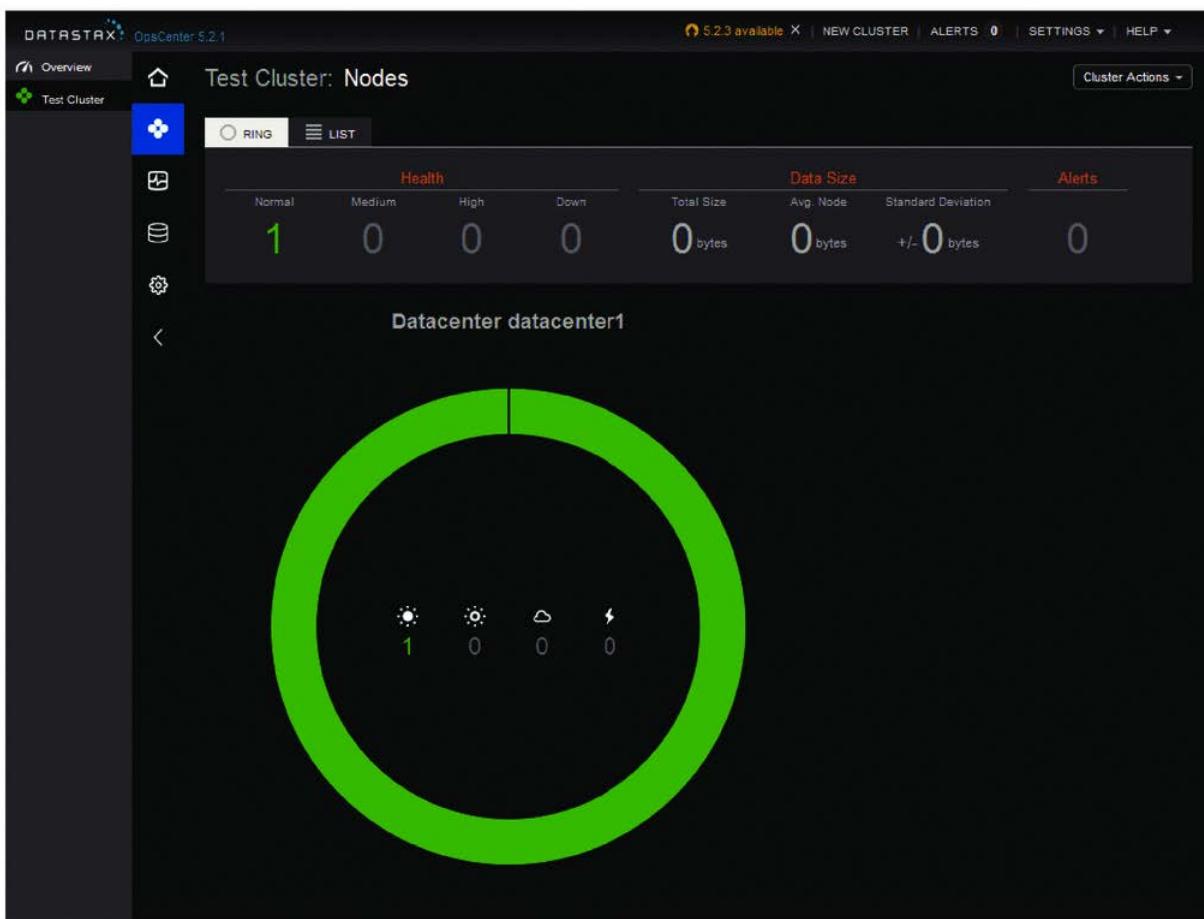


Рис. 11.2 ❖ Представление DataStax OpsCenter Nodes

В представлении Nodes мы можем выбрать узел по его положению в кольце и посмотреть сведения об использовании памяти, емкости хранилища, нагрузке, пулах потоков и другие метрики. Можно также останавливать и запускать узлы и выполнять различные служебные действия: очистку, уплотнение, сброс, исправление, вывод из эксплуатации и дренаж (команда drain).



Использование отдельного кластера для хранения данных OpsCenter

Приложение OpsCenter хранит метаданные и метрики управляемых кластеров в базе данных Cassandra. Можно настроить OpsCenter так, чтобы все таблицы находились в каком-то существующем кластере, но рекомендуется создать для этой цели отдельный кластер. В таком случае OpsCenter не будет мешать работе производственных кластеров.

Netflix Priam

Программа Priam (<https://github.com/Netflix/Priam>) разработана компанией Netflix для управления своими кластерами Cassandra. В греческой мифологии Приам был царем Трои и отцом Кассандры. Priam автоматизирует развертывание, запуск и останов узлов, а также операции резервного копирования и восстановления.

Priam хорошо интегрирован с облачной средой Amazon Web Services (AWS), хотя ее наличие и не требуется. Например, Priam поддерживает развертывание Cassandra в двух автоматически масштабируемых группах (ASG), автоматизированное резервное копирование снимков в службу Simple Storage Service (S3) и настройку сетевых групп и групп безопасности для кластеров Cassandra, размещенных в нескольких регионах.

У Priam нет пользовательского интерфейса, но есть REST-совместимый API, на основе которого можно создать собственный интерфейс или напрямую работать с программой curl. API предоставляет возможность запускать и останавливать узлы, выполнять команды nodetool (их результаты представляются в формате JSON) и производить резервное копирование и восстановление.

Резюме

В этой главе мы обсудили некоторые способы взаимодействия с Cassandra для выполнения стандартных задач обслуживания: добавления, исключения и замены узлов, резервного копирования и восстановления данных с помощью снимков и др. Мы также рассмотрели некоторые инструменты, помогающие автоматизировать решение этих задач с целью ускорения обслуживания и уменьшения числа ошибок.

Глава 12

Настройка производительности

В этой главе мы рассмотрим, как повысить производительность Cassandra. В конфигурационном файле и на уровне отдельных таблиц есть немало параметров для этих целей. В большинстве случаев вполне достаточно параметров по умолчанию, но иногда их нужно изменить. Мы разберемся, как и почему следует производить такие изменения.

Мы также научимся работать с программой нагружочного тестирования `cassandra-stress`, входящей в состав дистрибутива Cassandra. Благодаря ей мы можем быть уверены, что после изменения параметров Cassandra будет хорошо вести себя в производственной среде.

Управление производительностью

Для достижения и поддержания высокого уровня производительности кластера полезно рассматривать управление производительностью как процесс, который начинается с определения архитектуры приложения и продолжается на всех этапах разработки и эксплуатации.

Постановка целей

Прежде чем приступать к настройке производительности, важно четко определить цели, при этом не важно, что вы делаете: только начинаете развертывать приложение в кластере Cassandra или обслуживаете существующее приложение.

Планируя кластер, нужно понимать, как он будет использоваться: количество клиентов, предполагаемые способы обращения, ожидаемые периоды пиковой нагрузки и т. д. Это очень полезно для планирования как начальной вычислительной мощности кластера, так и его роста. Мы вернемся к этой теме в главе 14.

Важная часть такого планирования – четкое определение целевых показателей производительности в терминах пропускной способности (количество обслуженных запросов в единицу времени) и задержки (времени завершения одного запроса).

Предположим, к примеру, что мы разрабатываем сайт электронной коммерции, в котором используется спроектированная в главе 5 модель данных. Можно было бы так сформулировать целевые показатели производительности кластера:

Кластер должен поддерживать в секунду 30 000 операций чтения из таблиц `available_rooms_by_hotel_date` с задержкой 3 мс в 95%-ном процентиле.

В этой формулировке участвуют и пропускная способность, и задержка. Ниже мы узнаем, как их измерить с помощью команды `nodetool tablestats`.

Каковы бы не были целевые показатели, важно помнить, что настройка производительности – всегда результат компромиссов. Если целевые показатели четко определены, то будет проще решить, какие компромиссы приемлемы в приложении. Например:

- разрешить сжатие файлов SSTable, чтобы сэкономить место на диске, но ценой дополнительного потребления ресурсов процессора;
- искусственно ограничить сетевой трафик и использование потоков, что позволит держать под контролем потребление ресурсов сети и процессора ценой снижения пропускной способности и увеличения задержки;
- увеличить или уменьшить число потоков, выделенных под определенные задачи, например операции чтения, записи или уплотнения, чтобы изменить их приоритет относительно других задач или поддержать дополнительных клиентов;
- увеличить размер кучи, чтобы уменьшить время обработки запроса.

И это лишь малая толика компромиссов, на которые приходится идти при настройке производительности. Остальные мы рассмотрим на протяжении этой главы.

Мониторинг производительности

По мере роста размера кластера, увеличения числа клиентов и добавления все новых пространств ключей и таблиц требования к кластеру становятся разнонаправленными. Все большую важность приобретают частые замеры текущих показателей и их сравнение с целевыми.

В главе 10 мы узнали о различных метриках, раскрываемых через JMX, включая и относящиеся к производительности метрики класса Cassandra StorageProxy и отдельных таблиц. Там же были рассмотрены команды утилиты nodetool для вывода различной статистики, в т. ч. nodetool tpstats и nodetool tablestats; мы обсудили также, как эти сведения помогают выявлять проблемы, связанные с нагрузкой и задержкой.

Теперь познакомимся с двумя дополнительными командами nodetool, которые выводят статистику в виде гистограмм: proxyhistograms и tablehistograms. Для начала рассмотрим вывод команды nodetool proxyhistograms:

```
$ nodetool proxyhistograms
```

```
proxy histograms
```

Percentile	Read Latency (micros)	Write Latency (micros)	Range Latency (micros)
50%	654.95	0.00	1629.72
75%	943.13	0.00	5839.59
95%	4055.27	0.00	62479.63
98%	62479.63	0.00	107964.79
99%	107964.79	0.00	129557.75
Min	263.21	0.00	545.79
Max	107964.79	0.00	155469.30

Здесь показана задержка операций чтения, записи и запросов по диапазону, для которых узел, отправивший запрос, выступал в роли координатора. Данные представлены в форме процентиелей, а также минимума и максимума в микросекундах. Выполнив эту команду на нескольких узлах, мы сможем выявить медленные узлы кластера. Большая задержка для запросов по диапазону (сотни миллисекунд и более) может свидетельствовать о клиентах, отправляющих неэффективные запросы, например включающие фразу ALLOW FILTERING или требующие просмотра индекса.

Данные, напечатанные командой proxyhistograms, полезны для выявления общих проблем производительности, но часто нас больше интересует производительность отдельных таблиц. Именно для этого

предназначена команда nodetool tablehistograms. Посмотрим, что она выводит для таблицы available_rooms_by_hotel_date:

```
nodetool tablehistograms hotel available_rooms_by_hotel_date
hotel/available_rooms_by_hotel_date histograms
Percentile   SSTables  Write Latency  Read Latency Partition Size Cell Count
                                         (micros)      (micros) (bytes)
50%          0.00      0.00          0.00        2759       179
75%          0.00      0.00          0.00        2759       179
95%          0.00      0.00          0.00        2759       179
98%          0.00      0.00          0.00        2759       179
99%          0.00      0.00          0.00        2759       179
Min          0.00      0.00          0.00        2300       150
Max          0.00      0.00          0.00        2759       179
```

Результаты похожи. Вместо данных о задержке запросов по диапазону выведено количество файлов SSTable на один запрос. Печатаются размер раздела и число ячеек, это еще один способ найти большие разделы.



Сброс метрик

Отметим, что в версиях Cassandra вплоть до 3.X метрики подсчитываются с момента запуска узла. Чтобы сбросить метрики в начальное значение, нужно перезагрузить узел. В запросе JIRA под номером CASSANDRA-8433 (<https://issues.apache.org/jira/browse/CASSANDRA-8433>) высказано предложение добавить возможность сброса метрик через JMX и с помощью nodetool.

Познакомившись с метриками и поняв, какую информацию о кластере они сообщают, можно перейти к следующей задаче: решить, за какими метриками наблюдать, и даже реализовать автоматизированные оповещения о том, что целевые показатели производительности не удовлетворяются. Это позволяет сделать как DataStax OpsCenter, так и любой каркас мониторинга метрик на базе JMX.

Анализ проблем с производительностью

Довольно часто бывает, что кластер, который работал хорошо, со временем начинает деградировать. Обнаружив проблему с производительностью, нужно как можно скорее проанализировать ее, чтобы не допустить дальнейшего ухудшения ситуации. Ваша задача – определить и устранить истинную причину.

В этой главе мы будем говорить о многих конфигурационных параметрах, которые можно использовать для настройки производи-

тельности кластера в целом, т. е. распространяющихся сразу на все пространства ключей и таблицы. Но важно также попытаться локализовать проблему до уровня конкретных таблиц или даже запросов.

На самом деле важнейшим фактором, влияющим на производительность кластера Cassandra, обычно является качество модели данных. Например, если таблица спроектирована так, что появляются разделы с растущим числом строк, то производительность кластера постепенно снижается, и проявляется это в ошибках исправления или в ошибках потоковой перекачки данных при добавлении новых узлов. Наоборот, если ключ раздела чрезмерно ограничительный, то будут образовываться разделы с узкими строками, а значит, для ответа даже на простой запрос придется читать много разделов.



Опасайтесь больших разделов

Обнаружить большие разделы помогает не только команда `nodetool tablehistograms`, но и поиск в журнале предупреждений вида «*Writing large partition*» (Запись в большой раздел) или «*Compacting large partition*» (Уплотнение большого раздела). Порог, после которого появляется предупреждение об уплотнении больших разделов, устанавливается с помощью свойства `compaction_large_partition_warning_threshold_mb` в файле `cassandra.yaml`.

Помните также об антипаттернах, обсуждавшихся в главе 5, в частности очередях и других решениях, которые порождают много надгробий.

Трассировка

Если вы нашли конкретную таблицу и запрос, из-за которого возникают проблемы, то дальше для получения детальной информации можно воспользоваться трассировкой. Трассировка – бесценный инструмент, позволяющий разобраться в обмене данными между клиентами и узлами, участвующими в запросе, и увидеть, сколько времени тратится на каждом шаге. Это помогает понять, как влияют на производительность решения, принятые при проектировании модели данных, а также выбор коэффициентов репликации и уровней согласованности.

Получить доступ к данным трассировки можно несколькими способами. Сначала посмотрим, как это работает в `cqlsh`. Включим режим трассировки, а затем выполним простую команду:

```
cqlsh:hotel> TRACING ON
Now Tracing is enabled
cqlsh:hotel> SELECT * from hotels where id='AZ123';
```

```

id      | address | name                  | phone        | pois
-----+-----+-----+-----+-----+-----+
AZ123 | null    | Super Hotel Suites at WestWorld | 1-888-999-9999 | null
(1 rows)

```

Tracing session: 6669f210-de99-11e5-bdb9-59bbf54c4f73

activity	timestamp	source	source_elapsed
Execute CQL3 query	2016-02-28 21:03:33.503000	127.0.0.1	0
Parsing SELECT * ...	2016-02-28 21:03:33.505000	127.0.0.1	41491
...			

Для краткости часть распечатки опущена, но, выполнив подобный запрос самостоятельно, вы увидите различные этапы: подготовку команд, исправление на этапе чтения, поиск в кэше ключей, поиск данных в таблицах в памяти и в файлах SSTable, взаимодействия между узлами, а также время, потраченное на каждом шаге, в микросекундах.

Следует обращать особое внимание на запросы, влекущие за собой большой объем межузловых взаимодействий, поскольку это может быть признаком неудачного проектирования схемы. Например, запрос, требующий обращения к вторичному индексу, скорее всего, приведет к взаимодействию с большинством или даже со всеми узлами кластера.

Достигнув цели, вы можете отключить трассировку в `cqlsh` командой `TRACING OFF`.

Информация о трассировке доступна также клиентам на основе драйверов DataStax. Модифицируем пример из главы 8, чтобы понять, как работать с трассировкой средствами драйвера DataStax для Java. Выделенный код служит для включения трассировки на уровне запроса и распечатки полученных результатов:

```

SimpleStatement hotelSelect = session.newSimpleStatement(
    "SELECT * FROM hotels WHERE id='AZ123'");
hotelSelect.enableTracing();

ResultSet hotelSelectResult = session.execute(hotelSelect);

QueryTrace queryTrace = hotelSelectResult.getExecutionInfo().getQueryTrace();

System.out.printf("Trace id: %s\n", queryTrace.getTraceId());
System.out.printf("%-42s | %-12s | %-10s \n", "activity",
    "timestamp", "source");

System.out.println("-----"
    + "-----+-----");

```

```

SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss.SSS");

for (QueryTrace.Event event : queryTrace.getEvents()) {
    System.out.printf("%42s | %12s | %10s\n",
        event.getDescription(),
        dateFormat.format((event.getTimestamp())),
        event.getSource());
}

```

Трассировка включается отдельно для каждого объекта Statement или PreparedStatement методом enableTracing(). Трассы мы получаем от результатов запроса. Вы наверняка обратили внимание, что метод Session.execute() всегда возвращает объект ResultSet, даже если запрос отличен от SELECT. Это позволяет получить метаданные запроса с помощью объекта ExecutionInfo, несмотря на отсутствие возвращенных результатов. В состав метаданных входит достигнутый уровень согласованности, адрес узла-координатора и других узлов, участвовавших в выполнении запроса, и информация о трассировке и разбиении на страницы.

Выполнение этого кода дает примерно такие же результаты, как в cqlsh:

```

Trace id: 58b22960-90cb-11e5-897a-a9fa1d00bce

activity                                | timestamp      | source
-----+-----+-----+
    Parsing SELECT * FROM hotels WHERE id=? | 20:44:34.550 | 127.0.0.1
                                         Preparing statement | 20:44:34.550 | 127.0.0.1
                                         Read-repair DC_LOCAL | 20:44:34.551 | 127.0.0.1
Executing single-partition query on hotels | 20:44:34.551 | 127.0.0.1
                                         Acquiring sstable references | 20:44:34.551 | 127.0.0.1
                                         Merging memtable contents | 20:44:34.551 | 127.0.0.1
                                         Merging data from sstable 3 | 20:44:34.552 | 127.0.0.1
Bloom filter allows skipping sstable 3 | 20:44:34.552 | 127.0.0.1
                                         Merging data from sstable 2 | 20:44:34.552 | 127.0.0.1
Bloom filter allows skipping sstable 2 | 20:44:34.552 | 127.0.0.1
                                         Read 1 live and 0 tombstone cells | 20:44:34.552 | 127.0.0.1

```

Трассировку поддерживает также DataStax DevCenter, где она включена по умолчанию. Для просмотра трассы любого запроса в DevCenter нужно перейти на вкладку «Query Trace» в нижней половине экрана, как показано на рис. 12.1 (панели «Connection», «CQL Scripts», «Schema» и «Outline» для удобства восприятия свернуты).

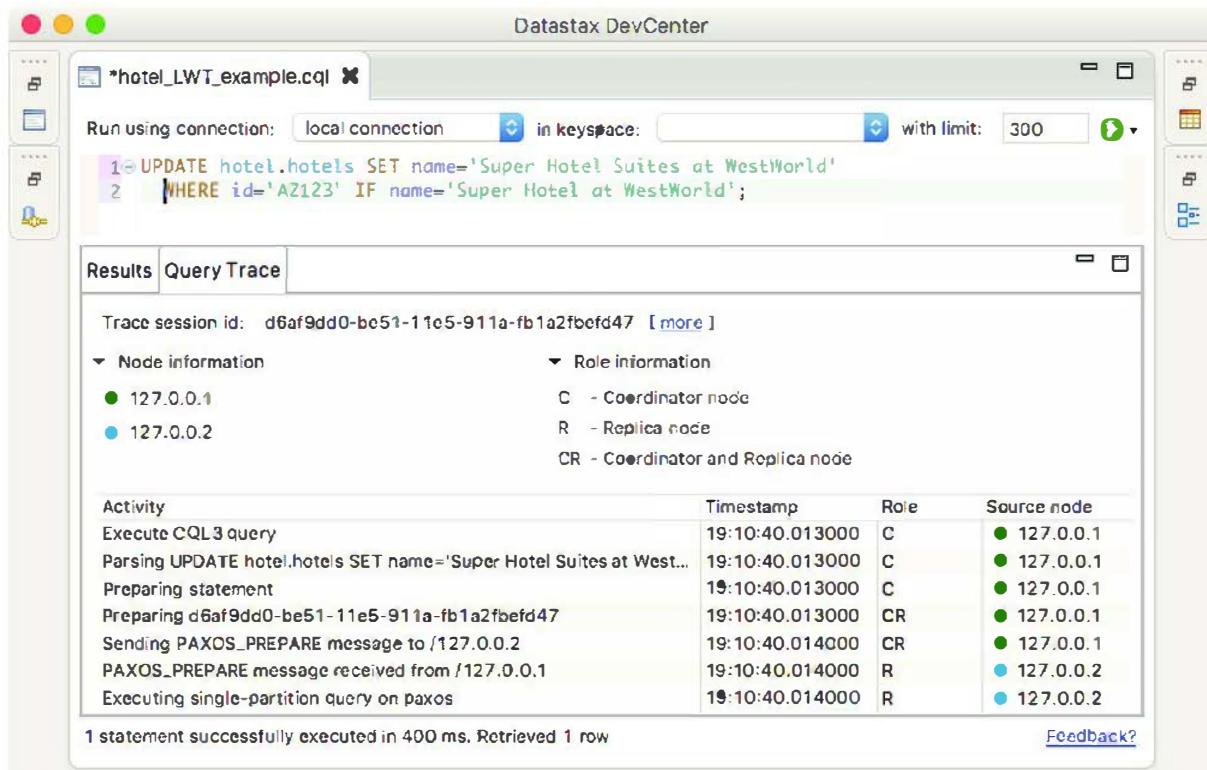


Рис. 12.1 ♦ Просмотр результатов трассировки запроса в DataStax DevCenter

Команда nodetool settraceprobability позволяет настроить отдельные узлы, так чтобы они трассировали все или некоторые запросы. Она принимает число от 0.0 (по умолчанию) до 1.0, где 0.0 означает выключение трассировки, а 1.0 – трассировку всех запросов. Это никак не связано с трассировкой отдельных запросов по требованию клиента. Следует проявлять осмотрительность при задании частоты трассировки, поскольку в типичном сеансе трассировки как минимум 10 операций записи. Поэтому задание частоты 1.0 способно создать непомерную нагрузку на занятый кластер, так что лучше ограничиться значением порядка 0.01 или 0.001.



Трассы не хранятся вечно

Cassandra хранит результаты трассировки запросов в пространстве ключей `system_traces`. Начиная с версии 2.2 трассировка используется также для хранения результатов операций исправления. Чтобы предотвратить переполнение диска, для этих таблиц установлен срок хранения (TTL). Величину TTL можно задать с помощью свойств `tracetype_query_ttl` и `tracetype_repair_ttl` в файле `cassandra.yaml`.

Методика настройки

Выявив коренную причину проблем с производительностью, можно переходить к настройке. Рекомендуется изменять по одному параметру за раз и тестировать результат.

Количество одновременных изменений конфигурации следует ограничивать, чтобы лучше понимать последствия каждого изменения. Возможно, для достижения желаемого эффекта понадобится несколько попыток.

Иногда вернуть приемлемую производительность удается простым добавлением ресурсов, например памяти или дополнительных узлов, но важно убедиться, что вы таким образом не прячете дефекты проектирования или конфигурации. Бывает и так, что достичь требуемых показателей производительности путем одной лишь настройки не получается и приходится вносить изменения в проект.

Памятую об описанной методике, рассмотрим, какие параметры влияют на производительность кластера Cassandra. Мы обсудим как параметры, задаваемые в файлах *cassandra.yaml* и *cassandra-env.sh*, так и на уровне отдельных таблиц средствами CQL.

Кэширование

Кэши применяются для ускорения операций чтения. Данные хранятся в дополнительной памяти, чтобы уменьшить число обращений к диску. Чем больше размер кэша, тем чаще происходит «попадание» – обнаружение искомых данных в памяти.

В Cassandra имеются три кэша: кэш ключей, кэш строк и кэш счетчиков. Для кэширования строк задается максимальное число хранимых в памяти строк из одного раздела. Если для некоторой таблицы используется кэш строк, то использовать для нее еще и кэш ключей не имеет смысла.

Таким образом, стратегию кэширования следует выбирать, принимая во внимание несколько факторов:

- проанализируйте запросы и используйте тот тип кэша, который максимально им соответствует;
- не позволяйте кэшу занимать большую часть кучи;
- сравните размер строк с размером ключей. Как правило, ключ должен быть гораздо короче полной строки.

Теперь поговорим о конфигурационных параметрах каждого кэша.

Кэш ключей

В кэше ключей Cassandra хранится отображение ключей разделов на записи индекса строк, что ускоряет доступ к файлам SSTable при чтении. Использование кэша ключей настраивается на уровне отдельной таблицы. Для примера настроим в cqlsh параметры кэширования таблицы hotels:

```
cqlsh:hotel> DESCRIBE TABLE hotels;

CREATE TABLE hotel.hotels (
    id text PRIMARY KEY,
    address frozen<address>,
    name text,
    phone text,
    pois set<text>
) WITH bloom_filter_fp_chance = 0.01
    AND caching = { 'keys' : 'ALL', 'rows_per_partition' : 'NONE' }

...
```

Поскольку кэш ключей значительно ускоряет чтение, потребляя немного дополнительной памяти, по умолчанию он включен, т. е. ключи кэшируются, даже если вы не задали этого режима при создании таблицы. Отключить кэширование ключей позволяет команда ALTER TABLE:

```
cqlsh:hotel> ALTER TABLE hotels
    WITH caching = { 'keys' : 'NONE', 'rows_per_partition' : 'NONE' };
```

Атрибут keys может принимать значение ALL или NONE.

Параметр key_cache_size_in_mb определяет максимальный размер памяти, отведенной под кэш; эта память разделяется между всеми таблицами. По умолчанию выделяется 5% кучи JVM или 100 МБ в зависимости от того, что меньше.

Кэш строк

В кэше строк хранятся целые строки. Он ускоряет чтение строк, к которым часто производится обращение, ценой дополнительного расхода памяти.

Выбирать размер кэша строк нужно осмотрительно, потому что зачастую этот кэш создает больше проблем, чем решает. Если набор данных невелик и целиком помещается в памяти, то кэш строк может дать впечатляющее ускорение, но оно не ощутимо, когда набор побольше, так что приходится читать данные с диска.

Если чтение из таблицы производится гораздо чаще, чем запись, то задание чрезмерно большого кэша строк без нужды потребляет значительные ресурсы сервера. Если отношение числа операций чтения к числу операций записи не очень велико, но существуют очень длинные строки (содержащие сотни столбцов), то, перед тем как задавать размер кэша, нужно будет проделать некоторые вычисления. И наконец, если нельзя сказать, что к некоторым строкам обращаются очень часто, а к остальным очень редко, то заметного выигрыша ожидать не следует.

По указанным причинам кэширование строк обычно дает меньший выигрыш, чем кэширование ключей. Попробуйте в качестве альтернативы поэкспериментировать с механизмом кэширования файлов, предоставляемым самой операционной системой.

Как и в случае кэширования ключей, порядок использования кэша строк можно задать на уровне таблицы. Параметр `rows_per_partition` определяет, сколько будет кэшироваться строк из одного раздела. По умолчанию он равен `NONE`, т. е. строки не кэшируются. Можно также задать в качестве значения положительное число или `ALL`. В следующей команде CQL задан кэш на 200 строк:

```
cqlsh:hotel> ALTER TABLE hotels
    WITH caching = { 'keys' : 'NONE', 'rows_per_partition' : '200' };
```

Реализацию кэша строк можно подменить, задав свойство `row_cache_class_name`. По умолчанию подразумевается выделение кэша не из кучи, реализованное классом `org.apache.cassandra.OHCProvider`. Ранее использовался класс `SerializingCacheProvider`.

Кэш счетчиков

Этот кэш увеличивает производительность счетчиков за счет уменьшения конкуренции за блокировку при доступе к самым часто используемым счетчикам. Возможности задавать кэширование счетчиков на уровне таблиц нет.

Параметр `counter_cache_size_in_mb` определяет максимальный размер памяти, отведенной под кэш счетчиков и разделяемой между всеми таблицами. По умолчанию выделяется 2,5% кучи JVM или 50 МБ в зависимости от того, что меньше.

Параметры, управляющие сохранением кэшей

Cassandra позволяет периодически сохранять кэши на диске, чтобы после запуска их можно было прочитать и не тратить время на прогрев. Параметры, управляющие сохранением, похожи для всех кэшей.

- Файлы, содержащие кэши, хранятся в каталоге, определяемом свойством `saved_caches`. Периодичность записи в файлы (в секундах) задается свойствами `key_cache_save_period`, `row_cache_save_period` и `counter_cache_save_period`, по умолчанию равными соответственно 14 400 (4 часа), 0 (выключено) и 7200 (2 часа).
- Кэши индексируются по значениям ключей. Количество ключей, сохраняемых в файле, задается свойствами `key_cache_keys_to_save`, `row_cache_keys_to_save` и `counter_cache_keys_to_save` соответственно.



Управление кэшами из nodetool

Cassandra предоставляет также возможность управлять кэшированием из утилиты nodetool.

- Для очистки кэшей служат команды `invalidatekeycache`, `invalidaterowcache` и `invalidatecountercache`.
- Для переопределения сконфигурированных размеров кэшей ключей и строк служит команда `setcachecapacity`.
- Для переопределения сконфигурированного числа элементов кэшей ключей и строк, сохраняемых в файле, служит команда `setcachekeystosave`.

Не забывайте, что после перезапуска узла будут восстановлены значения этих параметров, заданные в файле `cassandra.yaml`.

Таблицы в памяти

Cassandra использует таблицы в памяти для ускорения записи. Каждой хранимой таблице соответствует таблица в памяти. Таблицы в памяти сбрасываются в файлы SSTable на диске, когда достигнут порогового размера журнала фиксаций или самой таблицы в памяти.

Таблицы в памяти могут храниться в куче Java, в памяти операционной системы (вне кучи) или в обоих местах. Предельные размеры памяти, выделенной в куче и вне кучи, задаются свойствами `memtable_heap_space_in_mb` и `memtable_offheap_space_in_mb` соответственно. По умолчанию оба значения равны 1/4 общего размера кучи, заданного в файле `cassandra-env.sh`. Выделение области под таблицы в памяти уменьшает размер памяти, доступной для кэширования и других внутренних структур Cassandra, поэтому настраивайте осторожно и небольшими шагами.

Повлиять на то, как Cassandra выделяет память, позволяет свойство `memtable_allocation_type`. Оно задает сменную реализацию абстрактного класса `org.apache.cassandra.utils.memory.MemtablePool`,

используемую для управления памятью, занятой таблицами в памяти. Значение по умолчанию `heap_buffers` заставляет Cassandra выделять эту память из кучи, пользуясь API Java New I/O (NIO), а значение `offheap_buffers` – использовать тот же Java NIO для выделения памяти как из кучи, так и не из кучи. Если задано значение `offheap_objects`, то используется только память операционной системы, так что Cassandra несет всю ответственность за управление памятью и сборку мусора. Эта возможность плохо документирована, поэтому рекомендуем оставить режим по умолчанию, пока не наберетесь опыта.

К настройке таблиц в памяти имеет также отношение свойство `memtable_flush_writers`, по умолчанию равное 2. Оно определяет, сколько потоков будет занято выгрузкой таблиц в памяти на диск, когда в этом возникнет необходимость. Если каталоги с данными размещены на SSD-дисках, то рекомендуется увеличить этот параметр, сделав его равным количеству процессорных ядер, но не более 8. Если куча очень велика, то увеличение этого параметра также может повысить производительность, поскольку эти потоки блокируются во время дискового ввода-вывода.

Можно также задать сброс таблиц в памяти на диск по времени в команде CQL `CREATE TABLE` или `ALTER TABLE`. Интервал между сбросами конкретной таблицы в памяти на диск задает параметр `memtable_flush_period_in_ms`.

Установка этого свойства делает запись на диск более предсказуемой, но вместе с тем порождает больше файлов SSTable и приводит к более частым уплотнениям, что может негативно отразиться на скорости чтения. По умолчанию подразумевается значение 0, т. е. сброс по времени выключен и происходит только по достижении порогового размера журнала фиксаций или таблицы в памяти.

Журналы фиксаций

При обработке запросов на обновление Cassandra пишет в два набора файлов: журнал фиксаций и файлы SSTable. У них разное назначение, и различия нужно хорошо понимать для правильной настройки.

Напомним, что *журнал фиксаций* можно рассматривать как краткосрочное хранилище, гарантирующее сохранность данных в случае краха или остановки узла в момент, когда таблицы в памяти еще не сброшены на диск. Заявленная цель достигается благодаря накатыванию журнала фиксаций на этапе перезапуска узла. На самом деле

это единственное место, где журнал фиксаций читается; клиенты никогда и ничего из него не читают. Однако обычная операция записи в журнал фиксаций является блокирующей, поэтому с точки зрения производительности было бы плохо заставлять клиентов дожидаться ее завершения.

С помощью свойства `commitlog_segment_size_in_mb` можно указать максимальный размер журнала фиксаций, по достижении которого запись в него прекращается и создается новый журнал. По умолчанию он равен 32 МБ. Механизм аналогичен ротации журналов в таких библиотеках, как Log4J или Logback.

Общий размер всех журналов фиксаций задается свойством `commitlog_total_space_in_mb`. Чем больше это значение, тем реже Cassandra должна будет сбрасывать таблицы на диск.

Журналы фиксаций периодически удаляются – после того как все находящиеся в них данные успешно записаны в файлы SSTable. Поэтому журналы фиксаций гораздо меньше файлов SSTable, так что удваивать размер дисков необязательно; об этом следует помнить при определении характеристик оборудования.

Чтобы в журнале поместились большие данных, можно включить его сжатие с помощью свойства `commitlog_compression`. Поддерживаются алгоритмы сжатия LZ4, Snappy и Deflate. Но сжатие потребляет ресурсы процессора.

Дополнительный параметр `commitlog_sync` относится к операции синхронизации журнала фиксаций. Он может принимать одно из двух значений: `periodic` или `batch`. По умолчанию подразумевается `periodic`, и это означает, что сервер окончательно фиксирует операции записи один раз в заданный промежуток времени. Промежуток задается свойством `commitlog_sync_period_in_ms`, которое по умолчанию равно 10 000 (10 секунд).

Желая гарантировать долговечность данных в кластере Cassandra, на всякий случай проверьте этот параметр. Если сервер фиксирует обработанные операции записи периодически, то есть возможность потерять данные, которые еще не были перенесены на диск из кэша с отложенной записью.

Если установлен режим `batch`, то запись в журнал фиксаций блокирует выполнение программы, пока данные не окажутся на диске (Cassandra не подтверждает операцию записи, пока журнал фиксаций не синхронизирован с диском). После изменения этого параметра нужно померить производительность, поскольку налицо неизбежный компромисс: заставляя Cassandra производить запись безотлага-

гательно, мы ограничиваем ее свободу управления собственными ресурсами. Если вы все-таки решите установить свойство `commitlog_sync` в `batch`, то подберите также подходящее значение свойства `commit_log_sync_batch_window_in_ms`, равное количеству миллисекунд между последовательными операциями синхронизации.

Файлы SSTable

В отличие от журнала фиксаций, запись файлов SSTable на диск производится асинхронно. При использовании вращающихся дисков рекомендуется размещать файлы данных и журнал фиксаций на разных дисках для достижения максимальной производительности. В случае SSD-дисков это не имеет значения.

Cassandra, как и многие другие базы данных, весьма чувствительна к быстродействию жестких дисков и процессоров. Чтобы в полной мере задействовать мощные средства распараллеливания, встроенные в Cassandra, лучше иметь несколько процессорных ядер с умеренным быстродействием, чем одно или два очень быстрых. Постарайтесь установить максимально быстрые диски – лучше SSD, если бюджет позволяет. Если используются вращающиеся диски, то их должно быть, по меньшей мере два, – чтобы разнести журналы фиксаций и файлы данных и тем самым сократить конкуренцию за ресурсы ввода-вывода.

При чтении файлов SSTable с диска Cassandra использует *буферный кэш* (иногда его называют *буферным пулом*), чтобы уменьшить число операций ввода-вывода. Этот кэш размещается вне кучи, но его размер по умолчанию равен 512 МБ или 1/4 размера кучи Java в зависимости от того, что меньше. Задать размер буферного кэша позволяет свойство `file_cache_size_in_mb` в файле `cassandra.yaml`. Можно также разрешить Cassandra размещать буфера в куче Java, если выделенная область вне кучи переполнилась; для этого нужно присвоить свойству `buffer_pool_use_heap_if_exhausted` значение `true`.

В главе 9 было сказано, что Cassandra хранит сводные индексы файлов SSTable для ускорения доступа к ним. По умолчанию под эти индексы отводится 5% кучи Java, но это значение можно переопределить, задав свойство `index_summary_capacity_in_mb` в файле `cassandra.yaml`. Чтобы не выходить за установленные пределы, Cassandra сворачивает индексы, ассоциированные с редко читаемыми таблицами. Поскольку темп обращений к таблице со временем может изменяться, Cassandra производит переиндексирование файлов на диске с час-

тотой, определяемой свойством `summary_resize_interval_in_minutes`, которое по умолчанию равно 60.

В Cassandra также имеется возможность изменить относительный размер памяти, выделяемой для индексов над разными таблицами. Этим управляют свойства `min_index_interval` и `max_index_interval`, задаваемые в командах CQL `CREATE TABLE` и `ALTER TABLE`. Они задают минимальное и максимальное число элементов индекса для данного файла SSTable.

Вручение напоминаний

Вручение напоминаний – один из механизмов, с помощью которых Cassandra поддерживает синхронизацию данных в кластере. Как мы уже знаем, узел-координатор может хранить копию данных, предназначенных для узла, который в течение некоторого времени не подает признаков жизни. Этот механизм допускает настройку в двух отношениях: сколько места на диске отвести под напоминания и как быстро доставлять напоминания после того, как узел вновь поднимется.

Для контроля над сетевым трафиком в момент доставки напоминаний служит свойство `hinted_handoff_throttle_in_kb`, а во время выполнения его можно установить командой `nodetool sethintedhandoff-throttlekb`.

Этот параметр дросселирования по умолчанию равен 1024, или 1 МБ/с, он задает верхний предел потребления полосы пропускания, доступной узлу-приемнику напоминаний. Например, в кластере с тремя узлами каждый из двух узлов, доставляющих напоминания третьему, должен будет ограничить темп доставки половиной этого значения, т. е. 512 КБ/с.

Отметим, что задание режима дросселирования несколько отличается от настройки других механизмов Cassandra, поскольку поведение, наблюдаемое узлом-приемником напоминаний, определяется значениями свойств, заданными на других узлах. Следите за тем, чтобы этот параметр был задан одинаково на всех узлах, если не хотите, чтобы кластер пришел в замешательство.

В версиях, предшествующих 3.0, напоминания хранятся в таблице, которая не реплицируется на другие узлы, а начиная с версии 3.0 – в каталоге, который определяется свойством `hints_directory`, по умолчанию равным `data/hints`. Чтобы ограничить место на диске, отведенное под напоминания, воспользуйтесь свойством `max_hints_file_size_in_mb`.

Чтобы стереть все напоминания, ожидающие доставки на один или несколько узлов, выполните команду `nodetool truncatehints`, указав список IP-адресов или имен хостов. Недоставленные напоминания удаляются по истечении срока, заданного свойством `max_hint_window_in_ms`.

Можно также полностью запретить или разрешить вручение напоминаний (см. главу 10). Некоторые пользователи так и поступают, чтобы сэкономить место на диске и ресурсы сети, но вообще-то механизм вручения напоминаний потребляет не так много ресурсов, если принять во внимание предоставляемые им дополнительные гарантии согласованности, особенно по сравнению со стоимостью исправления узла.

Уплотнение

Cassandra позволяет настраивать механизм уплотнения, в частности задавать объем ресурсов, выделяемых для этой цели на узле, и стратегию уплотнения для каждой таблицы.

Выбор подходящей стратегии уплотнения, безусловно, может повлиять на производительность. Рассмотрим имеющиеся стратегии и обсудим, когда использовать ту или другую.

SizeTieredCompactionStrategy

`SizeTieredCompactionStrategy` (**STCS**) – стратегия уплотнения по умолчанию, в большинстве случаев ее и нужно использовать. Она собирает файлы SSTables в группы по размеру. Когда в одной группе набирается достаточно файлов SSTable (по умолчанию не меньше 4), запускается уплотнение с целью объединить эти файлы в один больший. По мере накопления данных групп становится больше. Стратегия STCS особенно хороша для таблиц с интенсивной записью, но в меньшей степени подходит для таблиц с интенсивным чтением, поскольку данные из одной логической строки могут быть разбросаны по нескольким файлам SSTable – в среднем порядка 10.

LeveledCompactionStrategy

При использовании стратегии `LeveledCompactionStrategy` (**LCS**) создаются файлы SSTable фиксированного размера (по умолчанию 5 МБ), они собираются в уровни, каждый из которых содержит в 10 раз больше файлов, чем предыдущий. LCS гарантирует, что любая строка присутствует не более чем в одном файле SSTable

каждого уровня. LCS тратит дополнительные ресурсы ввода-вывода на минимизацию числа файлов, содержащих данную строку; в среднем число таких файлов равно 1,11. Эту стратегию следует использовать, когда операций чтения гораздо больше, чем операций записи, или когда требуется предсказуемая задержка. LCS не оптимальна, если кластер уже перегружен операциями ввода-вывода. Если доминируют операции записи, то Cassandra будет трудно справиться с нагрузкой.

DateTieredCompactionStrategy

Стратегия DateTieredCompactionStrategy (DTCS) появилась в версиях 2.0.11 и 2.1.1. Она предназначена для улучшения скорости чтения временных рядов, особенно когда преимущественно идут обращения к последним записанным данным. Принцип работы основан на группировке файлов SSTable в окна по времени записи данных. Уплотнение выполняется только в пределах окна. Поскольку стратегия DTCS сравнительно новая и ориентирована на очень специальный случай, применяйте ее только после тщательного исследования.

Каждая стратегия имеет свои настраиваемые параметры. Подробностисмотрите в документации.

Тестирование стратегий уплотнения в режиме анализа записи

Чтобы протестировать другую стратегию уплотнения, не обязательно изменять ее во всем кластере. Просто создайте тестовый узел, работающий в режиме анализа записи (write survey mode).

Для этого добавьте в файл `cassandra-env.sh` на тестовом узле такие строки:

```
JVM_OPTS="$JVM_OPTS -Dcassandra.write_survey=true"
JVM_OPTS="$JVM_OPTS -Djoin_ring=false"
```

Когда узел поднимется, вы сможете обратиться к MBean-объекту `org.apache.cassandra.db.ColumnFamilyStoreMBean` для интересующей вас таблицы в разделе `org.apache.cassandra.db > Tables` и настроить стратегию уплотнения. Запишите в атрибут `CompactionStrategyClass` полное имя класса стратегии.

После этого изменения конфигурации добавьте узел в кольцо командой `nodetool join`, чтобы он начал принимать запросы на запись. Запись на тестовом узле лишь немного увеличивает нагрузку на кластер и не

учитывается при подсчете уровней согласованности. Теперь можно начать мониторинг производительности записи на данном узле с помощью команд `nodetool tablestats` и `tablehistograms`.

Чтобы протестировать влияние новой стратегии уплотнения на чтение, нужно остановить узел, запустить его как автономную машину, а затем провести тестирование производительности чтения.

Режим анализа записи полезен и для тестирования других изменений конфигурации и даже новой версии Cassandra.

На уровне таблицы задается также *порог уплотнения*. Речь идет о количестве файлов SSTable, стоящих в очереди в ожидании начала частичного уплотнения. По умолчанию минимальное число равно 4, максимальное – 32. Задавать слишком малое значение не стоит, потому что тогда Cassandra будет тратить много времени, сражаясь с клиентами за ресурсы, необходимые для частого выполнения ненужных уплотнений. Но и слишком большое значение вредно, так как тогда Cassandra будет тратить уйму ресурсов на выполнение сразу большой партии уплотнений, а для клиентов ресурсов останется меньше.

Порог уплотнения задается в команде CQL `CREATE TABLE` или `ALTER TABLE`. Но его можно также просмотреть или переопределить на конкретном узле командами `nodetool getcompactionthreshold` или `setcompactionthreshold` соответственно:

```
$ nodetool getcompactionthreshold hotel hotels
Current compaction thresholds for hotel/hotels:
  min = 4, max = 32
$ nodetool setcompactionthreshold hotel hotels 8 32
```

Уплотнение требует значительных ресурсов ввода-вывода и процессора, поэтому Cassandra предоставляет средства для мониторинга процесса уплотнения и определения условий его выполнения.

Для наблюдения за состоянием уплотнения на каком-то узле служит команда `nodetool compactionstats`, которая выводит число обработанных байтов и полное число байтов для каждого активного уплотнения (столбец ID мы для краткости опустили):

```
$ nodetool compactionstats
pending tasks: 1
id  compaction type   keyspace   table   completed   total   unit   progress
...  Compaction      hotel      hotels  57957241  127536780 bytes 45.44%
Active compaction remaining time : 0h00m12s
```

Если вы замечаете, что число ожидающих уплотнений начало увеличиваться, воспользуйтесь командами `nodetool getcompactionthroughput` и `setcompactionthroughput` для получения и установки ограничения, которое Cassandra применяет ко всем операциям уплотнения в кластере. Эта величина задается свойством `compaction_throughput_mb_per_sec` в файле `cassandra.yaml`. Если сделать его равным 0, то дросселирование полностью отключается, но значение по умолчанию 16 МБ/с подходит для большинства случаев, когда запись не слишком интенсивна.

Если ситуация не улучшилась, попробуйте увеличить число потоков, выделенных для уплотнения, изменив свойство `concurrent_compactors` в файле `cassandra.yaml`, или прямо во время работы с помощью объекта `CompactionManagerMBean`. По умолчанию его значение равно минимуму из числа дисков и числа процессорных ядер, причем абсолютный минимум равен 2, а абсолютный максимум – 8.

Хотя это встречается довольно редко, массивное уплотнение может отрицательно сказаться на производительности кластера. Для остановки всех активных операций уплотнения в узле можно воспользоваться командой `nodetool stop`. Можно также выбрать конкретное уплотнение по его идентификатору, причем список идентификаторов дает команда `compactionstats`. Cassandra запланирует все остановленные операции уплотнения на более позднее время.

Команда `nodetool compact` запускает полное уплотнение. Но имейте в виду, что это весьма накладная операция. Поведение `nodetool compact` зависит от используемой стратегии уплотнения. Влияние каждой стратегии на использование диска мы рассмотрим в главе 14.

Команда `nodetool compactionhistory` выводит статистические сведения о завершенных уплотнениях, в т. ч. размер данных до и после уплотнения и число объединенных строк. Распечатка довольно длинная, поэтому мы ее опустили.

Параллелизм и многопоточность

От многих хранилищ данных Cassandra отличается тем, что запись производится гораздо быстрее чтения. Два параметра управляют числом потоков, занятых операциями чтения и записи: `concurrent_reads` и `concurrent_writes`. Значений по умолчанию, как правило, вполне достаточно.

Параметр `concurrent_reads` определяет, сколько запросов на чтение узел может обслужить одновременно. По умолчанию значение равно

32, но должно быть равно числу дисков, используемых для хранения данных, умноженному на 16. Связано это с тем, что когда набор данных не помещается в имеющуюся память, операция чтения оказывается ограничена скоростью ввода-вывода.

Параметр `concurrent_writes` ведет себя несколько иначе. Его следует сопоставлять с числом клиентов, которые одновременно пишут на сервер. Если Cassandra обеспечивает работу сервера приложений, то значение по умолчанию этого параметра (32) следует заменить, привятив к числу потоков, используемых сервером приложений для подключения к Cassandra. В написанных на Java серверах приложений пул соединений с базой данных обычно насчитывает не больше 20–30 элементов, но если используется кластер таких серверов, то это число нужно умножить на соответствующий коэффициент.

Еще два параметра – `concurrent_counter_writes` и `concurrent_materialized_view_writes` – предназначены для управления специальными видами операций записи. Поскольку запись в счетчик и в материализованное представление подразумевает предварительное чтение, то лучше всего присвоить этим параметрам значение, равное минимуму из `concurrent_reads` и `concurrent_writes`.

В файле `cassandra.yaml` есть еще несколько свойств, управляющих числом потоков в пулах, отвечающих за реализацию различных ступеней архитектуры SEDA. С некоторыми из них мы уже встречались, а сейчас приведем полную сводку.

`max_hints_delivery_threads`

Максимальное число потоков, зарезервированных для доставки напоминаний.

`memtable_flush_writers`

Число потоков, зарезервированных для сброса таблиц в памяти на диск.

`concurrent_compactors`

Число потоков, зарезервированных для уплотнения.

`native_transport_max_threads`

Максимальное число потоков, зарезервированных для обработки входящих CQL-запросов (существуют также свойства `rpc_min_threads` и `rpc_max_threads`, относящиеся к более не используемому интерфейсу Thrift).

Отметим, что некоторые из этих свойств позволяют Cassandra динамически выделять и освобождать потоки, не превышая максималь-

но допустимого значения, тогда как другие задают статическое число потоков. Уменьшение и увеличение этих свойств влияет на то, как Cassandra использует ресурсы процессора и какая часть ресурсов ввода-вывода выделяется для различных действий.

Сеть и тайм-ауты

Поскольку Cassandra – распределенная система, она включает механизмы поведения в ненадежной сети, в т. ч. повторы, тайм-ауты и регулирование интенсивности трафика – дросселирование. Мы уже обсуждали два способа реализации логики повторов в Cassandra – политика `RetryPolicy` в клиентских драйверах DataStax и упреждающее чтение в драйверах и самих узлах.

Теперь рассмотрим механизмы, с помощью которых Cassandra избегает бесконечного ожидания ответов от других узлов. В табл. 12.1 перечислены относящиеся к тайм-аутам свойства в файле *cassandra.yaml*.

Таблица 12.1. Тайм-ауты узлов в Cassandra

Имя свойства	Значение по умолчанию	Описание
<code>read_request_timeout_in_ms</code>	5000 (5 секунд)	Сколько времени координатор будет ждать завершения чтения
<code>range_request_timeout_in_ms</code>	10 000 (10 секунд)	Сколько времени координатор будет ждать завершения запроса по диапазону
<code>write_request_timeout_in_ms</code>	2000 (2 секунды)	Сколько времени координатор будет ждать завершения записи
<code>counter_write_request_timeout_in_ms</code>	5000 (5 секунд)	Сколько времени координатор будет ждать записи счетчика
<code>cas_contention_timeout_in_ms</code>	1000 (1 секунда)	Сколько времени координатор будет пытаться повторять облегченную транзакцию
<code>truncate_request_timeout_in_ms</code>	60 000 (1 минута)	Сколько времени координатор будет ждать завершения усечения (включая создание снимка)
<code>streaming_socket_timeout_in_ms</code>	3 600 000 (1 час)	Сколько времени узел будет ждать завершения потоковой перекачки
<code>request_timeout_in_ms</code>	10 000 (10 секунд)	Тайм-аут по умолчанию для разных других операций

Вообще говоря, эти значения тайм-аутов вполне приемлемы, но, возможно, вы захотите подкорректировать их для своей сети.

С тайм-аутами связано также свойство `cross_node_timeout`, по умолчанию равное `false`. Если в среде присутствует NTP-сервер, то лучше включить этот режим, чтобы узлы могли точнее оценить, когда координатор прекратит ждать завершения долгого запроса по тайм-ауту, и быстрее освободить ресурсы.

Cassandra предоставляет также несколько свойств, которые позволяют ограничить долю полосы пропускания сети, доступную различным операциям. Их точная настройка предотвращает затопление сети запросами Cassandra ценой увеличения времени их выполнения. Например, свойства `stream_throughput_outbound_megabits_per_sec` и `inter_dc_stream_throughput_outbound_megabits_per_sec` определяют действующее на уровне потока ограничение полосы пропускания, доступной операциям передачи файлов другим узлам в локальном и удаленном ЦОДе соответственно.

Ограничения для вручения напоминаний и воспроизведения журнала пакетов работают несколько иначе. Значения свойств `hinted_handoff_throttle_in_kb` и `batchlog_replay_throttle_in_kb` интерпретируются как максимальные величины пропускной способности для кластера в целом и потому пропорционально распределяются между узлами согласно следующей формуле:

$$T_x = \frac{T_t}{N_n - 1}.$$

Иными словами, пропускная способность узла x (T_x) равна полной пропускной способности (T_t), поделенной на число узлов в кластере (N_n) минус 1.

Наконец, несколько свойств позволяет ограничить трафик, адресованный порту CQL на каждом узле. Они могут быть полезны в случае, когда вы не можете контролировать клиентские приложения, обращающиеся к кластеру. По умолчанию свойство `native_transport_max_frame_size_in_mb`, определяющее максимальный размер кадра, равно 256. Более длинные запросы узел будет отвергать.

Узел может также ограничить максимальное число одновременных клиентских подключений с помощью свойства `native_transport_max_concurrent_connections`, которое по умолчанию равно `-1` (не ограничено). Изменяя это свойство, позаботьтесь о его согласованности со свойствами `concurrent_readers` и `concurrent_writers`.

Чтобы ограничить число одновременных подключений одного IP-адреса, задайте свойство `native_transport_max_concurrent_connections_per_ip`, которое по умолчанию тоже равно `-1` (не ограничено).

Параметры JVM

Cassandra позволяет настроить разнообразные параметры, управляющие запуском JVM на сервере, выделение памяти под кучу Java и т. п. В этом разделе мы обсудим, как настраивать процедуру запуска.

В Windows скрипт запуска называется *cassandra.bat*, в Linux – *cassandra.sh*. Для запуска сервера нужно просто выполнить этот скрипт, в котором присутствует несколько параметров по умолчанию. Но в каталоге *conf* есть еще один скрипт, в котором заданы различные параметры, относящиеся к запуску Cassandra. Этот файл называется *cassandra-env.sh* (*cassandra-env.ps1* в Windows) и предназначен для того, чтобы вынести некоторые параметры, в т. ч. настройки JVM, в отдельное место, где ими будет проще управлять.

Для повышения производительности попробуйте настроить параметры, передаваемые JVM при запуске. Основные параметры JVM включены в файл *cassandra-env.sh*, а рекомендации по их настройке приведены ниже. Для настройки нужно открыть файл *cassandra-env.sh*, изменить значения параметров и перезапустить сервер.



Файл jvm.options

В версии Cassandra 3.0 в каталоге *conf* появился еще один настроечный файл – *jvm.options*. Он предназначен для того, чтобы вынести параметры JVM, относящиеся к размеру кучи и сборке мусора, из файла *cassandra.in.sh* в отдельный файл, поскольку именно эти параметры чаще всего модифицируются. Файлы *jvm.options* и *cassandra.in.sh* подгружаются скриптом *cassandra-env.sh*.

Память

По умолчанию в Cassandra используется следующий алгоритм задания размера кучи JVM: если объем оперативной памяти на машине меньше 1 ГБ, то под кучу отводится 50% оперативной памяти. Если объем оперативной памяти превышает 4 ГБ, то под кучу отводится 25%, но не более 8 ГБ. Для настройки минимального и максимального размеров кучи используются флаги *-Xms* и *-Xmx*. Они должны быть одинаковы, чтобы вся куча была зафиксирована в памяти и не выгружалась операционной системой. Не рекомендуется задавать размер кучи больше 8 ГБ, если используется сборщик мусора Concurrent Mark Sweep (CMS), поскольку при таком большом размере паузы, обусловленные сборкой мусора, становятся слишком долгими.

При настройке производительности разумно для начала установить минимальный и максимальный размеры кучи и ничего больше.

И лишь после опробования в реальной среде, выполнения тестов производительности с помощью инструментов анализа кучи и наблюдения за поведением конкретных приложений стоит переходить к другим параметрам JVM. Увидев прогресс после настройки параметров JVM, не начинайте прыгать от радости. Нужна проверка в реальных условиях.

В общем случае имеет смысл задать режим, при котором содержимое кучи выгружается при нехватке памяти. В файле *cassandra-env.sh* по умолчанию так и сделано с помощью параметра `-XX:+HeapDumpOnOutOfMemory`. Это стоит делать, если начинают сыпаться ошибки из-за отсутствия памяти.

Сборка мусора

Куча в Java разбита на две области: старые и молодые объекты. Пространство молодых объектов разбито далее на область «Eden space» (Эдем), из которой выделяется память под новые объекты, и область «Survivor space» (выжившие), в которую из Эдема перемещаются объекты, пережившие хотя бы одну сборку мусора.

Для молодого поколения Cassandra использует параллельный копирующий сборщик мусора, этот режим задан параметром `-XX:+UseParNewGC`. Коэффициентом выживания называется отношение размера Эдема к размеру области выживших в молодой части кучи. Увеличивать этот коэффициент имеет смысл, если приложение создает много новых объектов, которые живут недолго, а уменьшать, если значительная часть созданных объектов живет долго. В Cassandra этот коэффициент равен 8 (параметр `-XX:SurvivorRatio`), т. е. область выживших в 8 раз меньше Эдема. Такое высокое значение объясняется тем, что объекты долго находятся в таблицах в памяти.

В заголовке каждого объекта в Java есть поле возраста, показывающее, сколько раз он был скопирован внутри области молодого поколения. Объекты, пережившие сборку мусора в молодом поколении, копируются в новую область, и такое копирование обходится не даром. Поскольку долгоживущие объекты могут копироваться много раз, настройка этого значения иногда повышает производительность. Соответствующий параметр `-XX:MaxTenuringThreshold` в Cassandra по умолчанию равен 1. Сделайте его равным 0, если хотите, чтобы выжившие молодые объекты сразу перемещались в старое поколение. Этот и предыдущий параметры имеют смысл настраивать одновременно.

По умолчанию в современных версиях Cassandra для старого поколения используется сборщик мусора Concurrent Mark Sweep (CMS),

это задается параметром `XX:+UseConcMarkSweepGC`. В этом режиме потребляется больше памяти и процессорного времени, чтобы часто запускать сборку мусора по ходу работы приложения и тем самым уменьшить неизбежную на это время паузу. При такой стратегии важно, чтобы минимальный и максимальный размеры кучи совпадали, иначе JVM будет тратить много времени на первоначальное увеличение кучи.

Переход на сборщик мусора Garbage First

Сборщик мусора Garbage First (он же G1GC) появился в Java 7 с целью улучшить и в конечном итоге заменить сборщик мусора CMS, особенно на многопроцессорных машинах с большим объемом памяти.

G1GC делит кучу на несколько областей равного размера и динамически отдает их Эдему, выжившим и старому поколению, так что каждое поколение является логическим объединением необязательно смежных областей памяти. При таком подходе сборка мусора может производиться непрерывно и требует меньше останавливающих всю работу пауз, характерных для традиционных сборщиков мусора.

G1GC почти не нуждается в настройке; при проектировании предполагалось, что нужно будет задавать только минимальный и максимальный размеры кучи и длительность паузы. Чем короче пауза, тем чаще происходит сборка мусора.

Разработчики уже собирались сделать G1GC сборщиком мусора по умолчанию в версии Cassandra 3.0 release, но это решение было отменено из-за проблемы CASSANDRA-10403 (<https://issues.apache.org/jira/browse/CASSANDRA-10403>), в описании которой указано, что этот сборщик мусора работает хуже CMS, если размер кучи меньше 8 ГБ.

Для тех, кто хочет поэкспериментировать с G1GC и большой кучей в версии Cassandra 2.2 или более поздней, необходимые параметры имеются в файле `cassandra-env.sh` (или `jvm.options`).

Надо полагать, что G1GC все-таки станет сборщиком мусора по умолчанию в будущей версии Cassandra, скорее всего, это будет с увязано с поддержкой Java 9, в которой G1GC будет использоваться по умолчанию на уровне Java.

Есть несколько параметров, позволяющих лучше понять, как ведет себя сборка мусора в узлах кластера Cassandra. Параметр `gc_warn_threshold_in_ms` в файле `cassandra.yaml` задает длительность паузы, по истечении которой Cassandra выводит в журнал предупреждение. По умолчанию он равен 1000 мс (1 секунде). Можно также поручить

JVM печать подробных сведений о сборке мусора, задав параметры в файле *cassandra-env.sh* или *jvm.options*.

Утилита *cassandra-stress*

В комплект поставки Cassandra входит утилита *cassandra-stress*, которую можно использовать для нагружочного тестирования кластера. Для этого перейдите в каталог *<cassandra-home>/tools/bin* и выполните команду:

```
$ cassandra-stress write n=1000000
Connected to cluster: test-cluster
Datacenter: datacenter1; Host: localhost/127.0.0.1; Rack: rack1
Datacenter: datacenter1; Host: /127.0.0.2; Rack: rack1
Datacenter: datacenter1; Host: /127.0.0.3; Rack: rack1
Created keyspaces. Sleeping ls for propagation.
Sleeping 2s...
Warming up WRITE with 50000 iterations...
Running WRITE with 200 threads for 1000000 iteration
...
```

Программа выводит список узлов, к которым подключилась (в данном случае это кластер, созданный с помощью *cst*), и создает тестовое пространство ключей и в нем таблицу, куда будет записывать данные. Для прогрева инструмент производит 50 000 операций записи, а затем продолжает запись и вывод метрик, но для краткости мы все это опустили. Программа создает пул потоков (в моей системе с 200 потоками), которые работают, пока число записанных строк не достигнет одного миллиона. В конце печатается сводка результатов:

```
Results:
op rate : 7620 [WRITE:7620]
partition rate : 7620 [WRITE:7620]
row rate : 7620 [WRITE:7620]
latency mean : 26.2 [WRITE:26.2]
latency median : 2.6 [WRITE:2.6]
latency 95th percentile : 138.4 [WRITE:138.4]
latency 99th percentile : 278.8 [WRITE:278.8]
latency 99.9th percentile : 393.3 [WRITE:393.3]
latency max : 820.9 [WRITE:820.9]
Total partitions : 1000000 [WRITE:1000000]
Total errors : 0 [WRITE:0]
total gc count : 0
total gc mb : 0
```

```
total gc time (s) : 0
avg gc time(ms) : NaN
stdev gc time(ms) : 0
Total operation time : 00:02:11
```

Разберемся в увиденном. Мы сгенерировали и вставили миллион значений в абсолютно не настроенный кластер с тремя узлами. На это ушло чуть больше двух минут, т. е. мы выполняли 7620 операций записи в секунду. Медианная задержка в расчете на одну операцию составила 2,6 мс, хотя несколько операций заняло больше времени.

Поместив все данные в базу, займемся тестированием чтения:

```
$ cassandra-stress read n=200000
...
Running with 4 threadCount
Running READ with 4 threads for 200000 iteration
```

Внимательно изучив распечатку, мы увидим, что в начале работы количество потоков было невелико, а затем возрастило. В тестовом прогоне в пике число потоков превышало 600, как видно из сводки результатов:

```
Results:
op rate : 13828 [READ:13828]
partition rate : 13828 [READ:13828]
row rate : 13828 [READ:13828]
latency mean : 67.1 [READ:67.1]
latency median : 9.9 [READ:9.9]
latency 95th percentile : 333.2 [READ:333.2]
latency 99th percentile : 471.1 [READ:471.1]
latency 99.9th percentile : 627.9 [READ:627.9]
latency max : 1060.5 [READ:1060.5]
Total partitions : 200000 [READ:200000]
Total errors : 0 [READ:0]
total gc count : 0
total gc mb : 0
total gc time (s) : 0
avg gc time(ms) : NaN
stdev gc time(ms) : 0
Total operation time : 00:00:14
Improvement over 609 threadCount: 7%
```

Программа периодически выводит статистику о нескольких последних операциях. Показанный выше результат относится к последнему блоку статистики. Как видим, Cassandra читает совсем не так

быстро, как пишет; медианная задержка чтения составила примерно 10 мс. Напомним, однако, что мы ничего не настраивали, сервер работал в один поток на обычной рабочей станции вместе с другими программами, а размер базы данных составляет примерно 2 ГБ. Так или иначе, этот инструмент поможет вам оптимизировать производительность своего кластера и получить количественные показатели, на основе которых вы сможете понять, чего от него ждать.

Мы можем прогнать `cassandra-stress` и для своих таблиц, создав спецификации в виде `yaml`-файла. Например, создать файл `cqlstress-hotel.yaml`, в котором описаны запросы, читающие данные из таблицы в пространстве ключей `hotel`. Ниже показаны запросы для нагружочного тестирования таблицы `available_rooms_by_hotel_date`:

```
keyspace: hotel
table: available_rooms_by_hotel_date

columnspec:
  - name: date
    cluster: uniform(20..40)

insert:
  partitions: uniform(1..50)
  batchtype: LOGGED
  select: uniform(1..10)/10

queries:
  simple1:
    cql: select * from available_rooms_by_hotel_date
      where hotel_id = ? and date = ?
    fields: samerow
  range1:
    cql: select * from available_rooms_by_hotel_date
      where hotel_id = ? and date >= ? and date <= ?
    fields: multirow
```

Затем эти запросы выполняются утилитой `cassandra-stress`. Например, можно создать смешанную нагрузку из запросов на запись, чтение одной строки и чтение диапазона:

```
$ cassandra-stress user profile=~/cqlstress-hotel.yaml
ops\simple1=2,rangef1=1,insert=1\) no-warmup
```

Числа после запросов определяют требуемую пропорцию. В данном случае производятся три операции чтения на каждую операцию записи.



Дополнительные сведения о программе `cassandra-stress`

Программа поддерживает много других команд и параметров. Команда `cassandra-stress help` выводит их полный список, а `cassandra-stress help <command>` – подробную информацию о конкретной команде.

Будет также полезно ознакомиться с продуктом `cstar_perf` – платформой тестирования производительности с открытым исходным кодом, которую предлагает компания DataStax. Этот инструмент поддерживает нагрузочное тестирование, в т. ч. раскрутку кластера и выполнение тестов в разных версиях Cassandra или в разных кластерах с различными конфигурационными параметрами – для последующего сравнения. Имеется также веб-интерфейс для создания и выполнения тестовых скриптов и просмотра результатов. Скачать `cstar_perf` и почитать документацию можно на странице http://datastax.github.io/cstar_perf/setup_cstar_perf_tool.html.

Резюме

В этой главе мы рассмотрели управление производительностью Cassandra и различные параметры для настройки производительности, в т. ч. кэширования и памяти. Не остались без внимания и аппаратные аспекты. Мы также воспользовались программой нагрузочного тестирования `cassandra-stress` для записи и чтения миллиона строк.

Если вы только начинаете работать с Linux, но хотели бы развернуть Cassandra на этой платформе (как и рекомендуется), то почитайте статью в блоге Эла Тоби (Al Tobey) о настройке Cassandra (<https://tobert.github.io/pages/als-cassandra-21-tuning-guide.html>). В ней рассматривается несколько инструментов мониторинга в Linux, которые помогут получить представление о производительности самой базовой платформы, чтобы не искать ошибки там, где их нет. Хотя статья относится к версии Cassandra 2.1, содержащиеся в ней рекомендации применимы и в более широком контексте.

Глава 13

Безопасность

Сделать данные доступными – один из основных принципов движения «Большие данные». Это позволит совершить прорыв в методах анализа данных и принесет ощутимые выгоды бизнесу, научным кругам и обществу в целом. В то же время доступность данных входит в противоречие с усиливающимися требованиями к безопасности и конфиденциальности. Системы масштаба Интернета подвергаются постоянно изменяющимся атакам, целью которых чаще всего служат хранящиеся в них данные. Все мы знаем о многочисленных взломах, получивших широкое освещение в СМИ и приведших к потере данных, в т. ч. персональных данных, платежной информации, военных и промышленных секретов. И это только те взломы, о которых стало известно журналистам.

Одним из результатов возросшей угрозы стало ужесточение законодательных норм и требований во многих отраслях промышленности.

- Закон об охране и ответственности за информацию, полученную в результате медицинского страхования (HIPAA), принятый в США в 1996 году, предусматривает средства контроля над защитой и обработкой сведений о здоровье пациентов.
- Немецкий Федеральный закон о защите данных (BDSG) был пересмотрен в 2009 году с целью усилить регулирование в области сбора и передачи персональных данных, в т. ч. ввести ограничения на перемещение таких данных за территорию Германии и ЕС.
- Стандарт безопасности данных индустрии платежных карт (PCI DSS), выпущенный в 2006 году, – это набор отраслевых стандартов безопасной обработки данных платежных карт.
- Закон Сарбанеса-Оксли, принятый в США в 2002 году, регулирует порядок аудита и отчетности предприятий, в т. ч. сроки хранения данных, их защиту и аудит.

Это лишь несколько примеров регламентирующих и нормативных стандартов. Даже если ничто из вышеперечисленного не относится напрямую к вашему приложению, все равно велика вероятность, что какие-то нормативные акты распространяются и на вашу систему.

Общественный резонанс и нормотворческий пыл стали причиной повышенного внимания к безопасности корпоративных приложений вообще и, что для нас более важно, к безопасности баз данных NoSQL. Хотя база данных, по определению, является только частью приложения, нет сомнений, что она – важнейшая мишень атаки, поскольку именно здесь хранятся данные приложения.



Является ли безопасность слабым местом NoSQL?

В отчете журнала *Information Week* за 2012 год (<http://reports.informationweek.com/abstract/2/8758/Business-Continuity/strategy-why-nosql-equals-nosecurity.html>) сообщество NoSQL было подвергнуто критике за излишнее благодушие и недостаточное внимание к средствам безопасности в базах данных NoSQL. С тех пор многие технологии NoSQL, включая Cassandra, были значительно улучшены, но эта статья служит отрезвляющим напоминанием о нашей ответственности и о необходимости постоянно быть начеку.

По счастью, сообщество Cassandra за относительно краткий срок своего существования продемонстрировало стремление к непрерывному совершенствованию в области безопасности, как было показано в историческом обзоре в главе 2.

К числу средств обеспечения безопасности в Cassandra относятся аутентификация, ролевая авторизация и шифрование (рис. 13.1).

В этой главе мы рассмотрим все эти механизмы и способы доступа к ним из `cqlsh` и других клиентов и попутно поделимся мыслями о месте Cassandra в более широком контексте безопасности приложения.

Аутентификация и авторизация

Познакомимся со средствами аутентификации и авторизации в Cassandra.

Аутентификация по паролю

По умолчанию Cassandra позволяет любому клиенту из вашей сети подключиться к кластеру. Это не означает, что без дополнительной настройки нет никакой безопасности. Просто сконфигурирован механизм аутентификации, пускающий всех клиентов без предъявле-

ния учетных данных. Но механизм обеспечения безопасности сменный, т. е. метод аутентификации легко подменить или написать свой собственный.

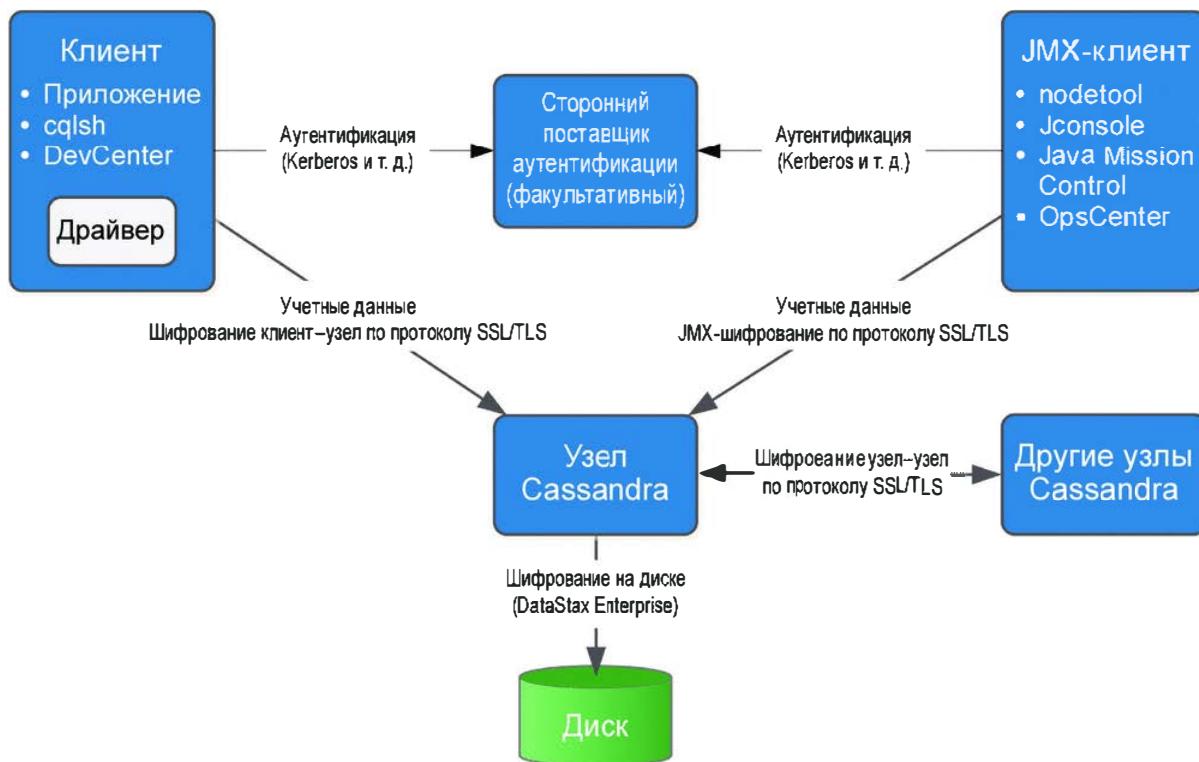


Рис. 13.1 ♦ Средства обеспечения безопасности в Cassandra

По умолчанию для аутентификации используется класс `org.apache.cassandra.auth.AllowAllAuthenticator`. Если вы хотите, чтобы клиенты предъявили свои учетные данные, воспользуйтесь вместо этого классом `org.apache.cassandra.auth.PasswordAuthenticator`. В этом разделе мы покажем, как пользоваться вторым классом.

Настройка аутентификатора

Для начала остановите кластер, чтобы можно было изменить настройки безопасности. Откройте файл `cassandra.yaml` и поищите строку «`authenticator`»:

```
authenticator: AllowAllAuthenticator
```

Измените эту строку следующим образом:

```
authenticator: PasswordAuthenticator
```

В версии Cassandra 2.2 или более поздней в файле `cassandra.yaml` имеется замечание о необходимости использовать класс `Cassandra-`

RoleManager, если применяется PasswordAuthenticator. Класс CassandraRoleManager – часть механизма авторизации в Cassandra, мы обсудим его чуть ниже.

Дополнительные поставщики аутентификации

Можно «подсунуть» Cassandra свой собственный метод аутентификации, например основанный на мандатах Kerberos, или с хранением паролей в другом месте, например в каталоге LDAP. Для создания собственной схемы аутентификации нужно лишь реализовать интерфейс IAuthenticator. В издании DataStax Enterprise Edition имеются средства интеграции с дополнительными механизмами аутентификации.

Cassandra поддерживает также сменный механизм взаимной аутентификации узлов в виде реализации интерфейса IIInternodeAuthenticator. По умолчанию используется класс AllowAllInternodeAuthenticator, не выполняющий аутентификацию вообще, но вы вправе реализовать собственный аутентификатор для защиты узла от подключения к ненадежным узлам.

Добавление пользователей

Теперь сохраните файл *cassandra.yaml* и перезапустите узел или весь кластер, после чего попытайтесь войти через cqlsh. И сразу же возникнет проблема:

```
$ bin/cqlsh
Connection error: ('Unable to connect to any servers',
 {'127.0.0.1': AuthenticationFailed('Remote end requires
 authentication.',)})
```

В предыдущих версиях Cassandra иногда разрешался вход, но запрещался любой доступ к данным. Начиная с Cassandra 2.2 пароль требуется даже для входа. По умолчанию в Cassandra существует пользователь *cassandra* с паролем «Cassandra». Попробуем войти, предъявив такие учетные данные:

```
$ bin/cqlsh -u cassandra -p cassandra
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.0.0-rcl | CQL spec 3.3.1 |
 Native protocol v4]
Use HELP for help.
cassandra@cqlsh>
```

Успешно войдя, мы увидим приглашение, из которого ясно, что мы работаем от имени пользователя *cassandra*. Первое, что нужно сде-

лать для защиты системы, – изменить пароль этого очень важного пользователя. Мы воспользовались для этой цели генератором случайных паролей:

```
cassandra@cqlsh> ALTER USER cassandra WITH PASSWORD 'Kx10*nGpB6'
```

Пароль пользователя *cassandra* должен храниться в безопасном месте.

Теперь создадим новую учетную запись. Зададим имя пользователя и пароль. Пароль не обязателен, но, конечно, рекомендуется.

```
cassandra@cqlsh> CREATE USER jeff WITH PASSWORD 'i6Xjsj!k#9';
```

Команда CREATE USER поддерживает также фразу IF NOT EXISTS, чтобы предотвратить ошибки при повторном создании уже существующего пользователя. Проверим, что пользователь успешно создан, выполнив команду LIST USERS:

```
cassandra@cqlsh> LIST USERS;
```

name	super
cassandra	True
jeff	False

(2 rows)

Обратите внимание, что пользователь *cassandra* является суперпользователем, т. е. ему разрешено выполнять любое действие. Только суперпользователь может создавать других пользователей. Мы уже изменили пароль встроенного пользователя *cassandra*. Но для пущей безопасности можно создать другого суперпользователя, а учетную запись *cassandra* удалить.



Настройка автоматического входа

Чтобы не вводить имя пользователя и пароль при каждом входе в cqlsh, создайте в своем домашнем каталоге файл *.cqlshrc* и запишите в него свои учетные данные в следующем формате:

```
; Пример файла ~/.cqlshrc
[authentication]
username = jeff
password = i6Xjsj!k#9
```

Конечно, надо обеспечить безопасность этого файла, чтобы доступ к паролю имели только авторизованные пользователи (например, вы сами).

Для работы с учетными записями пользователей предназначена также команда ALTER USER, которая изменяет пароль или статус суперпользователя, и команда DROP USER для удаления пользователя. Обычный пользователь может изменить только свой пароль, все остальные операции требуют статуса суперпользователя.

Для переключения на другую учетную запись без повторного входа в cqlsh служит команда LOGIN:

```
cassandra@cqlsh> login jeff 'i6Xjsj!k#9'
jeff@cqlsh>
```

Можно не указывать пароль в командной строке, тогда cqlsh предложит его ввести. Лучше вводить пароль по приглашению, а не в командной строке, поскольку cqlsh сохраняет все введенные команды в файле *.cassandra/cqlsh_history* в домашнем каталоге. Это относится и к паролям, введенным в команде LOGIN.

Аутентификация средствами драйвера DataStax для Java

Разумеется, ваши приложения не используют cqlsh для доступа к Cassandra, поэтому полезно научиться, как аутентифицировать клиента с помощью драйвера DataStax. Продолжая простой пример из главы 8, воспользуемся методом Cluster.Builder.withCredentials(), чтобы задать имя пользователя и пароль при конструировании экземпляра Cluster:

```
Cluster cluster = Cluster.builder().addContactPoint("127.0.0.1").
    withCredentials("jeff", "i6Xjsj!k#9").
    build();
```

В этом простом примере учетные данные защиты в код, но несложно было бы прочитать их из защищенного конфигурационного файла или запросить у пользователя. В других драйверах DataStax синтаксис похожий.

Если в узлах сконфигурирован аутентификатор, отличный от подразумеваемого по умолчанию, то и клиенты должны использовать совместимый аутентификатор. Поставщики клиентской аутентификации реализуют интерфейс com.datastax.driver.core.AuthProvider. По умолчанию используется класс PlainTextAuthProvider, экземпляр которого регистрируется при вызове метода Cluster.Builder.withCredentials().

Другие реализации, поставляемые вместе с драйвером, находятся в пакете com.datastax.driver.auth. Это класс DseAuthProvider для подключения к кластерам вида DataStax Enterprise и KerberosAuthenticator.

Поставщик выбирается при конструировании объекта Cluster путем вызова метода Cluster.Builder.withAuthProvider().

Использование класса CassandraAuthorizer

Можно, конечно, пользоваться только аутентификацией, но в большинстве случаев нужны также предоставляемые Cassandra средства авторизации. Без дополнительной настройки Cassandra предоставляет всем клиентам доступ ко всем пространствам ключей и таблицам в кластере. Но, как и в случае аутентификации, механизм авторизации сменный.

По умолчанию используется авторизатор org.apache.cassandra.auth.AllowAllAuthorizer. Для ролевого управления доступом следует задать авторизатор org.apache.cassandra.auth.CassandraAuthorizer.

Снова остановите кластер, прежде чем менять авторизатор. Найдите в файле *cassandra.yaml* строку «authorizer»:

```
authorizer: AllowAllAuthorizer
```

и замените ее такой:

```
authorizer: CassandraAuthorizer
```

После перезапуска кластера мы можем снова войти в cqlsh от имени обычного пользователя и посмотреть, к каким из введенных в предыдущих главах данным у нас есть доступ:

```
$ cqlsh -u jeff -p 'i6Xjsj!k#9'  
...  
jeff@cqlsh> DESCRIBE KEYSPACES;  
    hotel system_schema system_auth system system_distributed  
    system_traces  
  
jeff@cqlsh> USE hotel;  
jeff@cqlsh:hotel> DESCRIBE TABLES;  
  
hotels  
  
jeff@cqlsh:hotel> select * from hotels;  
Unauthorized: code=2100 [Unauthorized] message="User jeff has no  
SELECT permission on <table hotel.hotels> or any of its parents"
```

Таким образом, мы можем увидеть имена различных пространств имен и таблиц, но при попытке обратиться к данным получаем отказ.

Чтобы исправить положение, переключимся на учетную запись суперпользователя и дадим пользователю `jeff` какие-нибудь права. Например, позволим ему читать из таблицы `hotels`:

```
cassandra@cqlsh> GRANT SELECT ON hotel.hotels TO jeff;
```

Если теперь вернуться к учетной записи `jeff` и снова выполнить команду `SELECT`, то мы увидим данные, хранящиеся в таблице `hotels`.



Получение справки по командам авторизации

Для получения дополнительных сведений по настройке прав доступа пользуйтесь командами `cqlsh HELP GRANT` и `HELP PERMISSIONS`.

Ролевое управление доступом

В большом кластере Cassandra может быть много различных пространств ключей и таблиц и много пользователей. Следить за назначением прав множеству пользователей утомительно. Можно, конечно, разложить это бремя на нескольких сотрудников, но есть способ лучше.

Начиная с версии 2.2 Cassandra предоставляет ролевую модель управления доступом (RBAC), которая позволяет создавать роли и ассоциировать с ними права. Пользователю можно назначить любую комбинацию ролей. Роль может включать в себя другие роли.

Чтобы понять, как это работает, создадим роль управляющего отелем и дадим ей все права на все таблицы в пространстве ключей `hotel`:

```
cassandra@cqlsh> CREATE ROLE hotel_management;
cassandra@cqlsh> GRANT ALL ON KEYSpace hotel TO hotel_management;
```

Это простая роль, не позволяющая входить в систему непосредственно. Но можно создавать роли с привилегиями суперпользователя и роли, которые поддерживают вход и принимают пароль.

Теперь назначим эту роль нашему пользователю:

```
cassandra@cqlsh> GRANT hotel_management TO jeff;
```

Роли в Cassandra аддитивны, т. е. если хотя бы одной роли, назначенной пользователю, разрешено что-то делать, то это разрешено делать и пользователю.

Cassandra хранит информацию о пользователях и ролях в пространстве ключей `system_auth`. Если мы настроили для нашего кластера авторизацию, то к этому пространству ключей имеют доступ только администраторы, поэтому просмотрим его содержимое, войдя в `cqlsh` от имени администратора:

```
cassandra@cqlsh> DESCRIBE KEYSPACE system_auth  
CREATE KEYSPACE system_auth WITH replication = {'class': 'SimpleStrategy',  
    'replication_factor': '1'} AND durable_writes = true;  
...
```

Мы убрали распечатку результатов, но, выполнив команду самостоятельно, вы увидите таблицы, в которых хранятся роли, ассоциированные с ними права и назначения. На самом деле на уровне базы данных нет отдельного понятия пользователя – Cassandra использует понятие роли для представления как ролей, так и пользователей.



Изменение коэффициента репликации пространства ключей system_auth

Важно понимать, что изначально пространство ключей `system_auth` настроено на использование стратегии `SimpleStrategy` с коэффициентом репликации 1.

Это означает, что по умолчанию созданные нами пользователи, роли и права не будут распространяться по кластеру, пока мы не изменим стратегию репликации `system_auth` в соответствии с топологией кластера.

Шифрование

Защита частной жизни пользователя – важный аспект многих систем, особенно когда речь идет об информации, касающейся здоровья, финансового положения и прочих персональных данных. Обычно для обеспечения конфиденциальности применяется шифрование данных, так что противник, завладевший данными, не сможет ими воспользоваться, не имея ключа. Данные можно шифровать на пути через открытый Интернет и наши внутренние системы, т. е. *в движении*, или в системах, где они хранятся, т. е. *в состоянии покоя*.

Начиная с версии 3.0 Cassandra защищает данные в движении путем шифрования трафика между клиентами и серверами (узлами) и между отдельными узлами. В версии 3.0 шифрование файлов данных (данные в состоянии покоя) поддерживается только в корпоративных изданиях DataStax Cassandra.



Планы развития в области шифрования данных

В Cassandra JIRA зарегистрировано несколько запросов по расширению функций шифрования в версиях серии 3.X. Так, в версию 3.4 было добавлено:

- CASSANDRA-11040 – шифрование напоминаний (<https://issues.apache.org/jira/browse/CASSANDRA-11040>);
- CASSANDRA-6018 – шифрование журналов фиксаций (<https://issues.apache.org/jira/browse/CASSANDRA-6018>).

См. также запрос CASSANDRA-9633 (<https://issues.apache.org/jira/browse/CASSANDRA-9633>) о шифровании файлов SSTable и CASSANDRA-7922 (<https://issues.apache.org/jira/browse/CASSANDRA-7922>) – сводка запросов касательно шифрования на уровне файлов.

Прежде чем приступать к настройке шифрования в узлах, нужно проделать подготовительную работу – создать сертификаты, без которых ничего работать не будет.

SSL, TLS и сертификаты

В Cassandra для шифрования данных при передаче используется протокол Transport Layer Security (TLS). TLS часто называют по имени его предшественника – протокола Secure Sockets Layer (SSL). Это криптографический протокол защиты коммуникаций между компьютерами, предотвращающий подслушивание и манипулирование. Точнее, TLS основан на криптографии с открытым ключом (или асимметричной криптографии), когда для шифрования и дешифрирования сообщений между двумя конечными точками – клиентом и сервером – используется пара ключей.

Прежде чем устанавливать соединение, обе конечные точки должны располагать сертификатом, в котором записаны открытый и закрытый ключи. Партнеры обмениваются своими открытыми ключами, но закрытые ключи никогда и никому не передаются.

Чтобы установить соединение, клиент отправляет запрос серверу, сообщая, какие наборы шифров он поддерживает. Сервер выбирает из списка тот набор шифров, который его устраивает, и возвращает в ответ сертификат, содержащий его открытый ключ. Клиент, если захочет, может проверить открытый ключ сервера, но это не обязательно. Сервер также может потребовать, чтобы клиент представил свой открытый ключ – для выполнения двусторонней аутентификации. Клиент использует открытый ключ сервера для шифрования отправленного ему сообщения, в котором согласуется сеансовый ключ – генерированный симметричный ключ для выбранного набора шифров, который будет применяться во время последующей коммуникации.

Во многих приложениях криптографии с открытым ключом сертификаты выдает удостоверяющий центр, но, поскольку мы обычно

контролируем как узлы Cassandra, так и клиентов, такой уровень контроля нам ни к чему. Мы можем и сами сгенерировать сертификаты с помощью простой утилиты keytool, входящей в комплект поставки Java.

Ниже приведен пример использования флага `-genkey` утилиты keytool для генерации пары ключей:

```
$ keytool -genkey -keyalg RSA -alias node1 -keystore node1.keystore  
-storepass cassandra -keypass cassandra  
-dname "CN=Jeff Carpenter, OU=None, O=None, L=Scottsdale, C=USA"
```

Эта команда генерирует пару ключей для одного узла Cassandra, который мы назвали «node1», и сохраняет ее в файле, который называется *складом ключей* (keystore). Мы назвали склад ключей *node1.keystore*. Мы задали пароли для склада ключей и пары ключей, а также уникальное имя в формате протокола LDAP.

В этом примере задан минимальный набор обязательных атрибутов для генерации ключей. Можно было указать в командной строке меньше атрибутов, и тогда keytool попросила бы ввести недостающие интерактивно – с точки зрения ввода паролей, это более безопасно.

Затем мы экспортим открытый ключ из каждого сертификата в отдельный файл, который можно передать кому угодно:

```
$ keytool -export -alias node1 -file node0.cer -keystore node1.keystore  
Enter keystore password:  
Certificate stored in file <node0.cer>
```

Мы указываем, какой ключ экспортим из склада ключей, идентифицируя его тем же именем, что и раньше, и задаем имя выходного файла. keytool предлагает ввести пароль склада ключей, после чего генерирует файл сертификата.

Этот процесс следует повторить для генерации ключей каждого узла и клиента.

Шифрование трафика между узлами

Имея ключи для каждого узла Cassandra, мы можем настроить шифрование коммуникаций между узлами, задав свойство `server_encryption_options` в файле *cassandra.yaml*:

```
server_encryption_options:  
  internode_encryption: none  
  keystore: conf/.keystore  
  keystore_password: cassandra  
  truststore: conf/.truststore
```

```

truststore_password: cassandra
# Дополнительные параметры:
# protocol: TLS
# algorithm: SunX509
# store_type: JKS
# cipher_suites: [TLS_RSA_WITH_AES_128_CBC_SHA,...]
# require_client_auth: false

```

Прежде всего задается параметр `internode_encryption`. Можно указать `all`, чтобы шифровать весь трафик между узлами, `dc` – для шифрования трафика между ЦОДами или `rack` – для шифрования трафика между стойками. Задаются пароль склада ключей и путь к нему, но можно вместо этого поместить созданный ранее файл со складом ключей в каталог `conf`, подразумеваемый по умолчанию.

Затем задаются параметры для файла, аналогичного складу ключей, который называется *хранилищем доверенных сертификатов* (`truststore`). Такое хранилище генерируется для каждого узла и содержит открытые ключи всех остальных узлов кластера. Команда выглядит следующим образом:

```

$ keytool -import -v -trustcacerts -alias node1 -file node1.cer
      -keystore node1.truststore
Enter keystore password:
Re-enter new password:
Owner: CN=Jeff Carpenter, OU=None, O=None, L=Scottsdale, C=USA
Issuer: CN=Jeff Carpenter, OU=None, O=None, L=Scottsdale, C=USA
Serial number: 52cf9209
Valid from: Thu Dec 17 17:01:03 MST 2015 until: Wed Mar 16 17:01:03 MST 2016
Certificate fingerprints:
      MD5: E2:B6:07:C0:AA:BB:71:E8:47:8A:2A:81:FE:48:2F:AB
      SHA1: 42:3E:9F:85:0D:87:02:50:A7:CD:C5:EF:DD:D1:6B:C2:78:2F:B0:E7
      SHA256: C1:F0:51:5B:B6:C7:B5:8A:57:7F:D0:F2:F7:89:C7:34:30:79:30:
                  98:0B:65:75:CE:03:AB:AA:A6:E5:F5:6E:C0
      Signature algorithm name: SHA256withRSA
      Version: 3

Extensions:
#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: C2 32 58 D0 55 27 5C D2 FB 1E 50 C9 76 21 30 5C .2X.U'\...P.v!0\
0010: E6 1A 7D CF ....
]
]
```

```
Trust this certificate? [no]: y
Certificate was added to keystore
[Storing truststore.node1]
```

keytool предлагает ввести пароль нового хранилища, а затем выводит сводную информацию об импортированном ключе.

В файле *cassandra.yaml* можно задать также ряд дополнительных параметров для настройки криптографических протоколов, выбирая подходящие элементы из перечня алгоритмов и библиотек, поддерживаемых в Java. Для Java 8 описания этих элементов можно найти на странице <http://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html>.

В большинстве случаев хватает умолчаний, но полезно знать, какие существуют параметры. Как эти параметры используются в Cassandra, можно узнать из кода класса `org.apache.cassandra.security.SSLFactory`, где создаются безопасные сокеты.

Параметр `protocol` задает набор протоколов, используемый при создании экземпляра класса `javax.net.ssl.SSLContext`. В Java 8 поддерживаются протоколы SSLv2, SSLv3, TLSv1, TLSv1.1 и TLSv1.2. Сокращения SSL и TLS означают последнюю поддерживаемую версию протокола.

Параметр `algorithm` определяет алгоритм, используемый для получения экземпляра класса `javax.net.ssl.TrustManagerFactory`. По умолчанию это сертификат в формате X.509.

Параметр `store_type` определяет тип склада ключей, нужный для получения экземпляра класса `java.security.KeyStore`. Значение по умолчанию `jks` соответствует складу ключей, созданному утилитой `keytool`, а это как раз то, что нам нужно.

Параметр `cipher_suites` – это список алгоритмов шифрования в порядке предпочтения. Набор шифров используется на этапе согласования параметров соединения между клиентом и сервером, предпочтение отдается алгоритмам, расположенным ближе к началу списка. Точно так же браузер соглашает алгоритм с веб-сервером, когда вы заходите на страницу с URL-адресом типа `https`: По умолчанию более стойкие шифры расположены в начале списка. Если вы не можете полностью контролировать клиентов, то имеет смысл вообще удалить слабые шифры, чтобы исключить возможность атаки с принуждением использовать недостаточно стойкие алгоритмы (*downgrade attack*).

Наконец, мы можем включить двустороннюю аутентификацию сертификатов, когда не только сервер аутентифицирует клиента, но и клиент аутентифицирует сервер. Для этого нужно присвоить значение `true` параметру `require_client_auth`.

Шифрование трафика между клиентами и узлами

Шифрование трафика между клиентами и узлами защищает данные в процессе передачи с клиентской машины узлу кластера. Свойство `client_encryption_options` в файле `cassandra.yaml` аналогично рассмотренному выше:

```
# enable or disable client/server encryption.
client_encryption_options:
    enabled: false
    optional: false
    keystore: conf/.keystore
    keystore_password: cassandra
    # require_client_auth: false
    # Set truststore and truststore_password if require_client_auth is true
    # truststore: conf/.truststore
    # truststore_password: cassandra
    # Дополнительные параметры:
    # protocol: TLS
    # algorithm: SunX509
    # store_type: JKS
    # cipher_suites: [TLS_RSA_WITH_AES_128_CBC_SHA,...]
```

Основные отличия от `server_encryption_options` – параметр `enabled`, который включает и выключает шифрование трафика между клиентами и узлами, и параметр `optional`, который говорит, вправе ли клиент выбирать между шифрованным и нешифрованным соединениями.

Параметры `keystore` и `truststore` обычно такие же, как в свойстве `server_encryption_options`, хотя можно завести отдельные файлы для клиента.

Отметим, что если параметр `require_client_auth` равен `true`, то в файле `truststore` на каждом узле должны храниться открытые ключи всех клиентов, которые собираются использовать шифрованное соединение.

Упрощенное управление сертификатами

Настройка хранилищ доверенных сертификатов на узлах Cassandra может стать проблемой, если в кластер регулярно добавляются новые узлы или появляются новые клиенты. Общепринятая практика – использовать один и тот же сертификат для всех узлов кластера и другой сертификат для всех клиентов.

Это существенно упрощает управление кластером, поскольку при добавлении узлов или клиентов не придется модифицировать файлы

truststore и перезапускать узлы. Но приходится расплачиваться потерей точного контроля над тем, каким узлам и клиентам вы доверяете, а каким – нет.

Какую бы схему управления сертификатами вы ни выбрали, для безопасности кластера Cassandra необходимо ограничивать доступ к компьютерам, на которых работают узлы, чтобы никто не мог несанкционированно изменить конфигурацию.

Безопасность на уровне JMX

В главе 10 мы видели, как Cassandra предоставляет средства мониторинга и управления по технологии JMX. В этом разделе мы узнаем, как защитить этот интерфейс и какие относящиеся к безопасности параметры можно настроить через JMX.

Обеспечение безопасности доступа через JMX

По умолчанию Cassandra разрешает доступ через JMX только с локального хоста. Это приемлемо, если имеется прямой доступ к компьютерам, но при наличии большого кластера невозможно заходить на каждую машину, где развернуты узлы, чтобы поработать на ней с инструментами типа nodetool или OpsCenter.

Поэтому Cassandra предоставляет возможность открыть интерфейс JMX для удаленного доступа. Разумеется, абсолютно бессмысленно потратить уйму сил, чтобы обеспечить безопасность доступа по внутреннему транспортному протоколу и при этом оставить уязвимой для атаки такую крупную мишень, как JMX. Поэтому посмотрим, как обезопасить удаленный доступ через JMX.

Прежде всего остановите узел или кластер и отредактируйте файл *conf/cassandra-env.sh* (*cassandra-env.ps1* в Windows). Найдите переменную LOCAL_JMX и измените ее:

```
LOCAL_JMX=no
```

Если этой переменной присвоено любое значение, кроме «yes», то нужно изменить еще несколько свойств, чтобы разрешить удаленный доступ к порту JMX:

```
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.port=$JMX_PORT"  
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.rmi.port=$JMX_PORT"
```

Далее, существует свойство, которое определяет, используется ли SSL для шифрования JMX-соединений (подробнее об этом чуть ниже):

```
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.ssl=false"
```

И наконец, свойства для настройки удаленной аутентификации для JMX:

```
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.authenticate=true"
JVM_OPTS="$JVM_OPTS -Dcom.sun.management.jmxremote.password.file=
/etc/cassandra/jmxremote.password"
```

Где находится файл *jmxremote.password*, решать вам. Но помните, что это должно быть место, доступное только тем, кому вы готовы предоставить доступ. Содержимым файла *jmxremote.password* мы сейчас займемся, но сначала сохраните файл *cassandra-env.sh*.

В комплект поставки JRE входит шаблон файла *jmxremote.password*, он находится в каталоге *jre/lib/management*. Обычно JRE устанавливается в каталог *C:\Program Files\Java* в Windows, */Library/Java/JavaVirtualMachines* в Mac OS и в */usr/lib/java* в Linux. Скопируйте файл *jmxremote.password* в то место, которое ранее прописали в скрипте *cassandra-env.sh*, и отредактируйте его, добавив строку с именем и паролем администратора, как показано полужирным шрифтом ниже:

```
...
# monitorRole QED
# controlRole R&D
cassandra cassandra
```

Еще отредактируйте файл *jmxremote.access* в каталоге *jre/lib/management*, добавив администратору разрешение читать и записывать MBean-объекты:

```
monitorRole readonly
controlRole readwrite \
create javax.management.monitor.* , javax.management.timer.* \
unregister
cassandra readwrite
```

Задайте права доступа к файлам *jmxremote.password* и *jmxremote.access*. В идеале учетной записи, от имени которой работает Cassandra, должно быть разрешено только чтение этого файла, а пользователям, не являющимся администраторами, должен быть запрещен всякий доступ.

Наконец, перезапустите Cassandra и проверьте, правильно ли настроен защищенный доступ, выполнив команду

```
$ nodetool status -u cassandra -pw cassandra
```

Можно также настроить SSL для JMX-соединений. Для этого нужно добавить еще несколько параметров JVM в файл *cassandra-env*:

```
JVM_OPTS="${JVM_OPTS} -Dcom.sun.management.jmxremote.ssl=true"
JVM_OPTS="${JVM_OPTS} -Djavax.net.ssl.keyStore=conf/nodel.keystore"
JVM_OPTS="${JVM_OPTS} -Djavax.net.ssl.keyStorePassword=cassandra"
JVM_OPTS="${JVM_OPTS} -Djavax.net.ssl.trustStore=conf/nodel.truststore"
JVM_OPTS="${JVM_OPTS} -Djavax.net.ssl.trustStorePassword=cassandra"
JVM_OPTS="${JVM_OPTS} -Dcom.sun.management.jmxremote.ssl.need.client.
auth=true"
```

MBean-объекты, относящиеся к безопасности

О различных MBean-объектах, раскрываемых Cassandra, мы узнали в главе 10. По понятным причинам, существует не так уж много относящихся к безопасности конфигурационных параметров, которые были бы удаленно доступны через JMX, но кое-что все же имеется в домене *org.apache.cassandra.auth*.

Класс *PermissionsCacheMBean*

По умолчанию Cassandra для повышения производительности кэширует информацию о ролях и правах. Время хранения прав в кэше задается в файле *cassandra.yaml* с помощью свойства *permissions_validity_in_ms* и по умолчанию составляет 2 секунды. Класс *PermissionsCacheMBean* позволяет уменьшить или увеличить это время, а также предоставляет метод для удаления всех прав из кэша. Это полезно, если требуется изменить права в кластере, так чтобы новые значения вступили в силу немедленно.

Резюме

Cassandra – лишь одна из частей корпоративного приложения, но она играет немаловажную роль. В этой главе мы научились настраивать сменные механизмы аутентификации и авторизации в Cassandra, а также предоставлять права пользователям и ролям. Мы включили шифрование трафика между клиентами и узлами и узнали, как защитить интерфейс JMX при удаленном доступе.

Глава 14

Развертывание и интеграция

В этой, последней, главе мы поделимся еще несколькими советами о развертывании Cassandra в производственной среде. Мы поговорим о том, какие варианты следует рассмотреть при планировании развертывания, и обсудим развертывание Cassandra в различных облачных средах. И в заключение расскажем о некоторых технологиях, дополняющих Cassandra.

Планирование развертывания кластера

Залог успешного развертывания Cassandra – качественное планирование. Необходимо учесть предполагаемый объем данных в кластере, сетевое окружение и вычислительные ресурсы (физические или виртуальные), доступные узлам.

Оценка размера кластера

Важный первый шаг планирования кластера – оценка объема хранимых данных. Разумеется, в будущем вы сможете добавлять и исключать узлы, регулируя вместимость, но расчет начального размера и его изменения со временем поможет предвидеть затраты и принимать обоснованные решения при планировании конфигурации кластера.

Чтобы вычислить требуемый размер кластера, нужно сначала определить размер каждой таблицы по формулам из главы 5. При расчете учитываются типы столбцов таблицы и предположительное число строк в ней, а результатом является размер одной копии данных на диске.

Чтобы оценить истинный физический объем требуемого места на диске для данной таблицы во всем кластере, нужно еще учесть коэффициент репликации пространства ключей, в котором находится таблица, и стратегию уплотнения. Окончательная формула полного размера T_t выглядит так:

$$T_t = S_t * RF_k * CSF_t$$

где S_t – размер таблицы, вычисленный по вышеупомянутой формуле, RF_k – коэффициент репликации, CSF_t – коэффициент уплотнения таблицы, равный:

- 2 для стратегии уплотнения `SizeTieredCompactionStrategy`. В худшем для этой стратегии случае полное уплотнение потребует создания второй копии всех данных;
- 1,25 для прочих стратегий уплотнения; согласно существующим оценкам, накладные расходы при полном уплотнении составляют 25%.

Зная, сколько места на диске понадобится для хранения каждой таблицы, мы можем просуммировать эти значения по всем пространствам ключей и таблицам и получить оценку полного размера кластера.

Разделив эту величину на форматированную емкость диска, получим требуемое число дисков. Форматированная емкость составляет примерно 90% от номинальной.

Отметим, что в оценке учтены накладные расходы на полное уплотнение всех пространств ключей и таблиц. Эти расходы можно уменьшить, если гарантируется, что эксплуатационники никогда не будут выполнять такое полное уплотнение, однако это предположение выглядит рискованным.



Оценка размера системных пространств ключей Cassandra

У вдумчивого читателя может возникнуть вопрос о том, как быть с местом на диске под внутренние данные Cassandra в различных пространствах ключей. По сравнению с размером диска, это обычно несущественно. Мы создали кластер с тремя узлами, и оказалось, что без дополнительных пространств ключей на каждом узле занято примерно 18 МБ.

Конечно, эта величина может оказаться существенно больше, если вы часто пользуетесь трассировкой, но в таблицах из пространства ключей `system_traces` задан срок хранения данных, что не позволяет им со временем заполнить весь диск.

Оценив требуемый размер и количество узлов, вы сможете с большим основанием подойти к оценке начального размера кластера. Заложенная вместимость кластера зависит от ожидаемой скорости роста, которую нужно соотнести со стоимостью дополнительного оборудования, физического или виртуального.

Выбор экземпляров

Необходимо правильно выбирать вычислительные ресурсы для узлов Cassandra, не важно, работают они на физическом оборудовании или в виртуальной облачной среде. Рекомендации для современных версий (2.0 и выше) разнятся в зависимости от типа среды – разработки или промышленной эксплуатации.

Среда разработки

В среде разработки узлы Cassandra должны быть оснащены процессором как минимум с двумя ядрами и памятью объемом не менее 8 ГБ. Cassandra с успехом работает и на менее мощном оборудовании, например на платформе Raspberry Pi с 512 МБ памяти, но тогда придется потратить много сил на настройку производительности.

Производственная среда

В производственной среде узлы Cassandra должны быть оснащены процессором как минимум с восемью ядрами (на виртуальных машинах достаточно и четырех) и памятью объемом от 16 до 64 ГБ.

Развертывание с помощью Docker и других контейнеров

Сейчас быстро набирает популярность, особенно среди пользователей Linux, вариант развертывания с помощью контейнеров, например Docker.

Достоинства контейнеров часто формулируют одной фразой «строишь один раз, развертываешь всюду». Движок позволяет создавать облегченные переносимые контейнеры для программного обеспечения, которые можно легко перемещать и развертывать на разных аппаратных платформах.

В Docker контейнер представляет собой облегченную виртуальную машину, которая обеспечивает изоляцию процессов, отдельную файловую систему и сетевой адаптер. Движок Docker расположен между приложением и ОС.

Основные проблемы развертывания Cassandra в Docker связаны с сетевой инфраструктурой и каталогами для данных. По умолчанию

в Docker используется программно-определенная маршрутизация. На момент написания книги производительность этого слоя снижает производительность Cassandra примерно на 50%, поэтому рекомендуется использовать сетевой стек хост-компьютера напрямую. Это ограничивает возможности развертывания одним узлом на контейнер.

Исполнение нескольких экземпляров Cassandra в контейнере можно рекомендовать только в среде разработки. При этом следует тщательно настраивать параметры памяти, о которых шла речь в главе 12. С точки зрения управления данными, каталоги данных следует вынести за пределы контейнера. Это позволит сохранить данные в случае модернизации или остановки контейнеров.

Те же соображения применимы и к другим контейнерным технологиям, хотя детали могут различаться. Это быстро развивающаяся область, в которой изменения будут продолжаться на протяжении еще многих месяцев или лет.

Хранилище

При выборе и настройке хранилища следует иметь в виду несколько факторов, в т. ч. типы и количество дисков.

Жесткие или SSD-диски?

Cassandra поддерживает как жесткие диски (их еще называют вращающимися), так и твердотельные (SSD-диски). Как мы знаем, Cassandra в основном дописывает в конец файла, и в этом смысле жесткие диски благоприятствуют ее работе, но SSD-диски показывают более высокую общую производительность благодаря низкой задержке чтения с произвольной выборкой.

Исторически жесткие диски были дешевле, но стоимость SSD продолжает снижаться, особенно с учетом того, что поставщики облачных платформ все чаще начинают предлагать их как один из вариантов хранилища.

Конфигурация дисков

Если используются вращающиеся диски, то рекомендуется записывать данные и журналы фиксаций на разные диски. В случае SSD это не так существенно.

JBOD или RAID?

Традиционно сообщество Cassandra рекомендовало использовать для файлов данных несколько дисков, собранных в RAID-массив. Поскольку для достижения избыточности Cassandra хранит дан-

ные на нескольких узлах, число которых определяется коэффициентом репликации, то уровень RAID 0 (тот с чередованием) считается достаточным.

В последнее время пользователи Cassandra стали применять конфигурацию типа JBOD (Just a Bunch of Disks – просто пачка дисков). Этот подход обеспечивает более высокую производительность и является предпочтительным, если есть возможность заменять отдельные диски.

Избегайте разделяемых хранилищ

Выбирая хранилище, избегайте технологий Storage Area Network (SAN) и Network Attached Storage (NAS). Обе они плохо масштабируются – потребляют дополнительные сетевые ресурсы для доступа к физическому хранилищу по сети и характеризуются дополнительным временем ожидания завершения ввода-вывода при чтении и записи.

Сеть

Поскольку в основе Cassandra лежит распределенная архитектура с несколькими узлами, объединенными сетью, необходимо учитывать ряд дополнительных факторов.

Пропускная способность

Прежде всего убедитесь, что сеть достаточно надежна и способна выдержать трафик, генерируемый распределением данных между некоторыми узлами. Рекомендуемая пропускная способность – 1 ГБ и выше.

Конфигурация сети

Проверьте правильность настройки правил брандмауэра и IP-адресов узлов сети и сетевых устройств – они должны пропускать трафик в порты, используемые для внутреннего транспортного протокола CQL, межузловых коммуникаций (параметр `listen_address`), JMX и т. д. Сюда относится и передача данных между ЦОДами (топологию кластера мы обсудим чуть ниже).

Часы на всех узлах должны быть синхронизированы с помощью протокола Network Time Protocol (NTP) или иными методами. Напомним, что Cassandra перезаписывает столбец, только если временная метка нового значения позже, чем у существующего. Без синхронизации могут быть потеряны операции записи от узлов с отстающими часами.

Избегайте балансировщиков нагрузки

Балансировщики нагрузки используются во многих вычислительных средах. Они полезны для распределения входящего трафика между несколькими экземплярами службы или приложения, но использовать их вместе с Cassandra не рекомендуется. У Cassandra имеются собственные механизмы балансировки сетевого трафика между узлами, а драйверы DataStax распределяют запросы клиентов между репликами, поэтому балансировщик нагрузки не даст дополнительного выигрыша. Кроме того, размещение балансировщика нагрузки перед узлами Cassandra создает потенциальную точку общего отказа, что ставит под угрозу доступность кластера.

Тайм-ауты

Строя кластер, охватывающий несколько ЦОДов, имеет смысл измерить задержку при передаче данных между центрами и соответственно настроить тайм-ауты в файле *cassandra.yaml*.

Правильная конфигурация сети – ключ к успешному развертыванию Cassandra, не важно, идет ли речь о частном центре обработки данных, о публичном облаке, охватывающем несколько ЦОДов, или даже о гибридной облачной среде.

Развортывание в облаке

Познакомившись с основами планирования развертывания кластера, перейдем к вариантам развертывания Cassandra в трех наиболее популярных облачных средах.

Прибегая к услугам коммерческих поставщиков облачных решений, вы получаете несколько важных преимуществ. Во-первых, для обеспечения высокой доступности можно выбрать несколько ЦОДов. Расширив кластер на несколько ЦОДов в конфигурации активный–активный и внедрив надежную стратегию резервного копирования, вы сможете отказаться от создания специальной системы аварийного восстановления.

Во-вторых, коммерческие поставщики позволяют разместить данные в центрах, близких к пользователям вашего приложения, и тем самым уменьшить время отклика. Если приложение имеет сезонный характер, то кластеры в каждом ЦОДе можно уменьшать или увеличивать в соответствии с текущим спросом.

Вы можете сэкономить время, воспользовавшись готовым образом, содержащим Cassandra. Существуют также компании, предлагающие

Cassandra в виде управляемой услуги по принципу SaaS (Software-as-a-Service).



Не забывайте о стоимости облачных ресурсов

Планируя развертывание в публичном облаке, не забудьте оценить затраты на эксплуатацию кластера. Следует учитывать все ресурсы: вычислительные, сетевые и хранение данных в узлах и в резервных копиях.

Amazon Web Services

Amazon Web Services (AWS) давно уже является популярным вариантом развертывания Cassandra, о чем свидетельствуют такие ориентированные на AWS расширения, как Ec2Snitch, Ec2MultiRegionSnitch и EC2MultiRegion Address Translator в драйвере DataStax для Java.

Топология кластера

В основе AWS лежат понятия региона и зоны доступности, обычно они отображаются на ЦОДы и стойки Cassandra. На рис. 14.1 показан пример топологии кластера AWS, охватывающего регионы us-east-1 (Вирджиния) и eu-west-1 (Ирландия). Имена узлов выбраны для удобства, никакого обязательного соглашения нет.

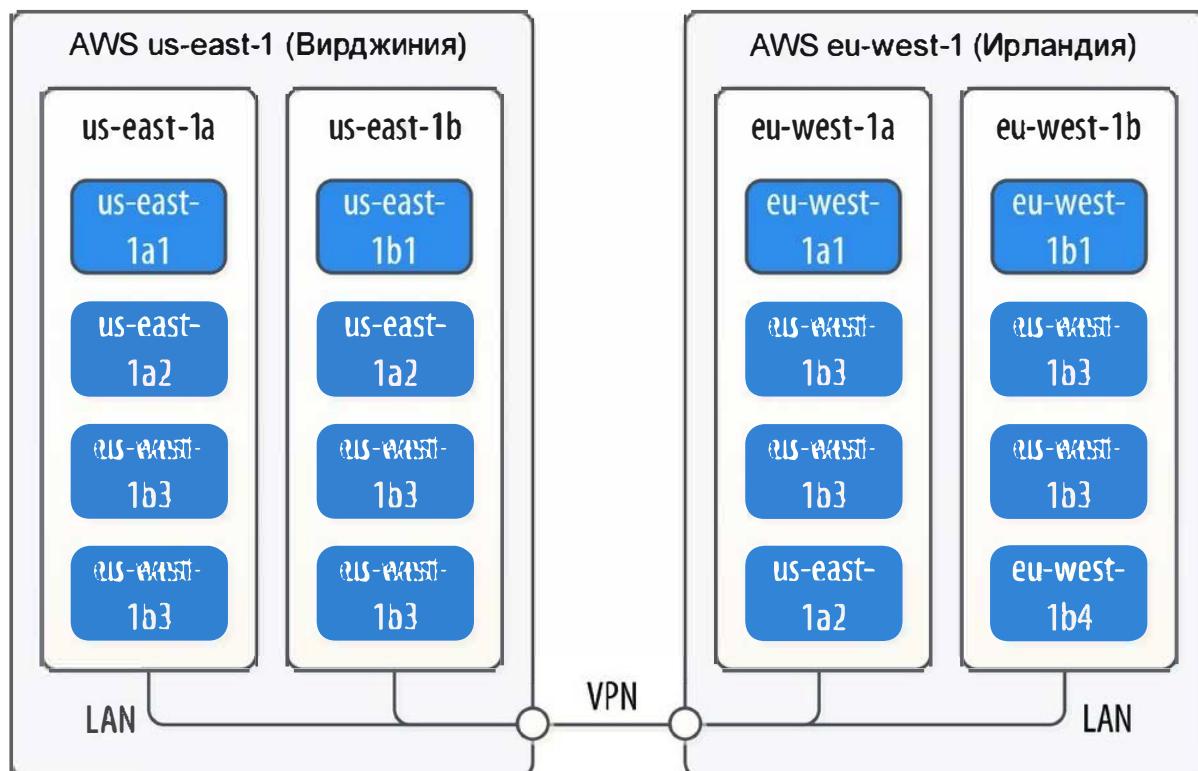


Рис. 14.1 ♦ Топология кластера в двух регионах AWS

Инстансы EC2

Облако Amazon Elastic Compute Cloud (EC2) предлагает целый ряд различных виртуальных аппаратных инстансов¹, сгруппированных в несколько классов. Для развертывания Cassandra рекомендуются классы M и I.

В M-класс входят инстансы общего назначения со сбалансированными ресурсами процессора, памяти и сети, пригодные для разработки и промышленной эксплуатации с относительно низкой нагрузкой. Старшие инстансы M-класса, например M3 и M4, оснащены SSD-дисками.

Все инстансы I-класса оснащены SSD-дисками и рассчитаны на интенсивный ввод-вывод. Стоят они дороже, но это обычно окупается для высоконагруженных кластеров.

Дополнительные сведения о различных типах инстансов приведены на странице <https://aws.amazon.com/ec2/instance-types>.

В издание DataStax Enterprise включен готовый образ машины Amazon (AMI), позволяющий упростить развертывание (начиная с версии Cassandra 2.1 поставка AMI для издания DataStax Community прекращена).

Хранилище данных

Исторически для AWS рекомендовалось использовать эфемерное хранилище, подключенное к каждому инстансу. Недостаток этого решения заключается в том, что если экземпляр, на котором работает узел, по какой-то причине остановится (а такое иногда случается в AWS), то данные будут потеряны.

В конце 2015 года было доказано, что сетевое хранилище Elastic Block Store (EBS) – надежное место для хранения данных, которое не пропадает с исчезновением инстансов EC2, но характеризуется дополнительной задержкой.

Служба AWS S3 – хороший вариант для краткосрочного и среднесрочного хранения резервных копий, а Glacier – для долгосрочного хранения.

Сеть

Если вы работаете в конфигурации с несколькими регионами, то убедитесь в наличии между ними достаточно быстрой и надежной сети. Многие пользователи считают, что использование элементов виртуального частного облака AWS (Virtual Private Cloud –

¹ Так называется экземпляр на русскоязычном сайте AWS. – *Прим. перев.*

VPC) – эффективный способ обеспечения надежного соединения между регионами с высокой пропускной способностью. Служба AWS Direct Connect предоставляет выделенную частную сеть, существуют также варианты организации виртуальной частной сети (VPN). Разумеется, эти службы не бесплатны.

Если система развернута в одном регионе или в нескольких регионах, соединенных виртуальной частной сетью, то используйте осведомитель Ec2Snitch. При развертывании в нескольких регионах, соединенных публичным Интернетом, используйте Ec2MultiRegionSnitch. И в том, и в другом случаях рекомендуется увеличить значение `phi_convict_threshold` в файле `cassandra.yaml` до 12.

Microsoft Azure

DataStax и Microsoft объединили усилия, стремясь улучшить развертывание Cassandra в облаке Microsoft Azure на платформах Windows и Ubuntu Linux.

Топология кластера

Центры обработки данных Azure имеются в разных местах, которые, как и в AWS, называются «регионами». Для управления наборами виртуальных машин (ВМ) применяется понятие *группы доступности*. В Azure ВМ в одной группе доступности могут быть назначены разным *доменам обновления*, концептуально эквивалентным стойкам Cassandra.

Экземпляры виртуальных машин

Как и в AWS, в Azure определено несколько классов ВМ. Для развертывания Cassandra подходят машины серии D, серии D v2 и серии G, оснащенные SSD-дисками. В машинах серии G имеется дополнительная память, что необходимо для описанной ниже интеграции. Дополнительные сведения о типах ВМ в Azure приведены на странице <https://azure.microsoft.com/en-us/pricing/details/virtualmachines>.

Хранилище данных

Azure предлагает стандартные SSD-диски для всех вышеупомянутых экземпляров. Кроме того, можно перейти на хранилище Premium Storage, предлагающее сетевые SSD-диски емкостью до 1 ТБ.

Сеть

Для развертывания в одном и нескольких регионах рекомендуется использовать публичные IP-адреса. Чтобы узлы могли

определить топологию кластера, используйте осведомитель `GossipingPropertyFileSnitch`. Для организации сети между регионами предлагаются технологии `VPN Gateways` и `Express Route`, обеспечивающие пропускную способность до 2 ГБ/с.

Google Cloud Platform

Облачная платформа Google (Google Cloud Platform) предоставляет облачные вычисления, хостинг приложений, сетевую инфраструктуру, систему хранения данных и различные приложения, предлагаемые в форме `SaaS` (`Software-as-a-Service` – программа как услуга), например API `Google Translate` и `Prediction`.

Топология кластера

Вычислительная среда `Google Compute Environment (GCE)` состоит из регионов и зон, соответствующих ЦОДам и стойкам `Cassandra`.

Экземпляры виртуальных машин

Для развертывания `Cassandra` рекомендуется использовать типы машин `GCE n1-standard` и `n1-highmemory`. Для быстрого запуска `Cassandra` на платформе `Google Cloud` используется `Cloud Launcher`. На странице <https://cloud.google.com/launcher/?q=cassandra> вы найдете варианты создания кластера несколькими щелчками мыши.

Хранилище данных

`GCE` предлагает как для эфемерных (`local-ssd`), так и для сетевых (`pd-ssd`) накопителей возможность использовать вращающиеся (`pd-hdd`) или SSD- (`local-ssd`) диски. Для хранения резервных копий `Cassandra` можно использовать три сервиса, отличающихся стоимостью и доступностью: `Standard`, `Durable Reduced Availability (DRA)` и `Nearline`.

Сеть

`GoogleCloudSnitch` – специальный класс осведомителя, предназначенный только для `GCE`. Между регионами можно организовать `VPN`.

Интеграция

В главе 2 отмечалось, что `Cassandra` – отличное решение для многих приложений, но не претендует на включение всего и вся. В этом разделе мы обсудим несколько технологий, которые в сочетании

с Cassandra позволяют более качественно решать такие задачи, как поиск и анализ данных.

Apache Lucene, SOLR и Elasticsearch

Одна из функций, востребованная в приложениях на основе Cassandra, – полнотекстовый поиск. Ее можно добавить к Cassandra с помощью Apache Lucene (<http://lucene.apache.org/>) – движка распределенного индексирования и поиска и его подпроекта Apache Solr (<http://lucene.apache.org/solr/>), дополняющего движок Lucene API для работы с REST и JSON.

Elasticsearch (<https://github.com/elastic/elasticsearch>) – еще один популярный поисковый каркас, построенный поверх Apache Lucene. Он поддерживает многоарендную архитектуру и предоставляет доступ к Java и JSON по протоколу HTTP.

Компания Stratio написала подключаемый модуль (<https://github.com/Stratio/cassandra-lucene-index>), который позволяет заменить стандартную реализацию вторичных индексов в Cassandra индексом Lucene. При использовании такой интеграции на каждом узле Cassandra появляется индекс Lucene, обеспечивающий высокоскоростной поиск.

Apache Hadoop

Каркас Apache Hadoop предоставляет средства распределенного хранения и обработки больших наборов данных на стандартном дешевом оборудовании. Работа над этой темой началась в Google в начале 2000-х годов. Обнаружилось, что несколько групп внутри Google занималось реализацией похожей функциональности с целью обработки данных и что во всех созданных инструментах было две фазы: распределения и редукции. Функция распределения получает на вход исходные данные и порождает промежуточные значения, а функция редукции обрабатывает промежуточные значения и порождает конечный результат. В 2006 году Дуг Каттинг (Doug Cutting) написал открытую реализацию файловой системы Google File System (<http://research.google.com/archive/gfs.html>) и технологии MapReduce (<http://research.google.com/archive/mapreduce.html>), положив тем самым начало Hadoop.

Как и Cassandra, Hadoop основан на распределенной архитектуре узлов, объединенных в кластеры. Интеграция обычно подразумевает установку Hadoop на каждый узел Cassandra, который играет роль поставщика данных. Чтобы использовать Cassandra в качестве источника данных,

нужно запустить на каждом узле Cassandra два компонента Hadoop: Task Tracker и Data Node. Тогда после инициализации задачи MapReduce (обычно на узле, внешнем по отношению к кластеру Cassandra) компонент Job Tracker сможет запрашивать у Cassandra данные в процессе сопровождения заданий распределения и редукции (рис. 14.2).

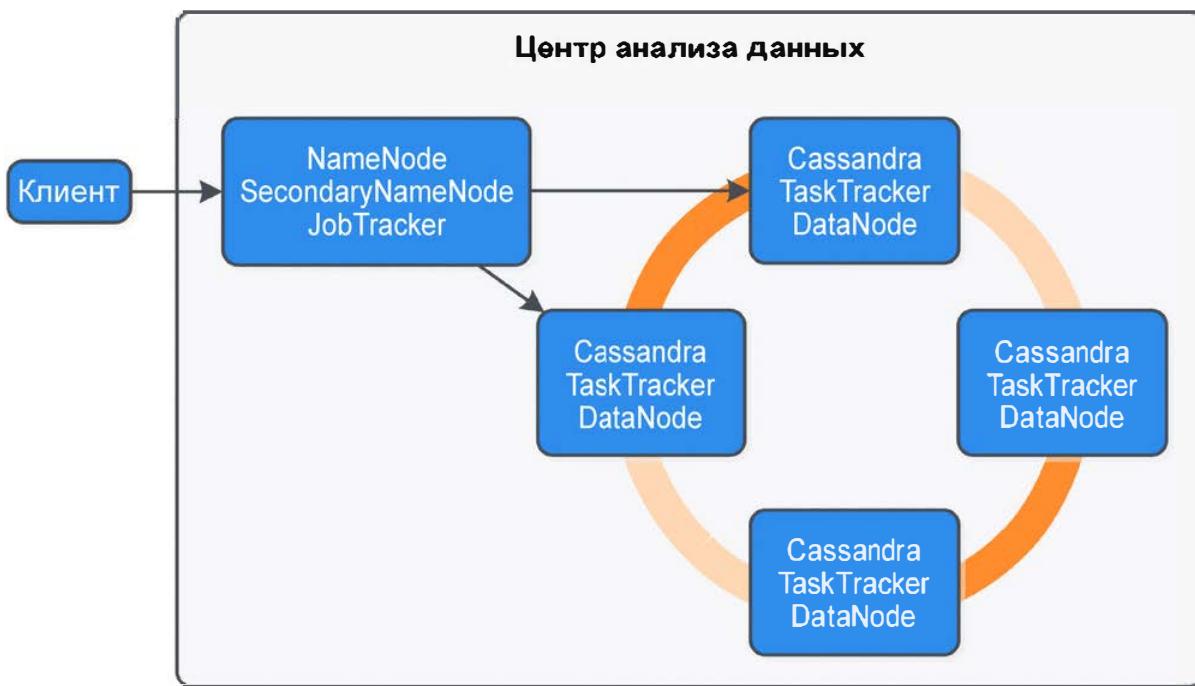


Рис. 14.2 ❖ Топология совмещенного кластера Hadoop-Cassandra

С конца 2000-х годов Hadoop оказался в центре внимания сообщества больших данных, и интерес к нему только растет. В результате вокруг Hadoop сформировалась целая экосистема расширений, например Apache Pig – каркас и язык для анализа данных и Apache Hive – средство создания складов данных. Сообщество Hadoop постепенно уходит от технологии MapReduce из-за ее сравнительно низкой производительности (все данные записываются на диск, даже если они нужны только на этапах промежуточной обработки), высокого потребления памяти, интуитивно не очевидного API и принципиально пакетного режима обработки. Все это стало стимулом для появления проекта Apache Spark, ставшего значительным достижением.

Apache Spark

в соавторстве с Патриком Макфейдином

Apache Spark (<http://spark.apache.org/>) – сравнительно новый каркас для анализа данных, предоставляющий архитектуру массово-параллельной обработки больших объемов данных с простым API.

Первоначально Spark был разработан в Калифорнийском университете в Беркли в 2009 году, в 2010 г. был опубликован исходный код, а в 2014 г. проект перешел под крыло Apache.

В отличие от Hadoop, где промежуточные результаты записываются на диск, движок Spark ориентирован на максимальное использование оперативной памяти при минимальном доступе к диску и сети. В Spark применяется потоковая, а не пакетная обработка, что позволяет многократно – до 100 раз – увеличить скорость по сравнению с Hadoop. К тому же API Spark гораздо проще, чем у Hadoop.

Основной единицей представления данных в Spark является *устойчивый распределенный набор данных* (Resilient Distributed Dataset – RDD). Это описание подлежащих обработке данных, например файла или коллекции. После создания RDD содержащиеся в нем данные можно преобразовывать с помощью вызовов API, как если бы все они находились на одной машине. Однако в действительности разделы RDD могут располагаться на нескольких узлах. Все разделы можно обрабатывать параллельно для получения окончательного результата. RDD поддерживает знакомые операции map и reduce, а также дополнительные операции, например: count, filter, union и distinct. Полный список преобразованийсмотрите в документации по Spark (<http://spark.apache.org/docs/latest/programming-guide.html#transformations>).

Помимо базового движка, в состав Spark входят библиотеки, ориентированные на различные виды обработки. Все это подпроекты, управляемые независимо от ядра Spark, но примерно с таким же графиком выпуска версий.

- SparkSQL предоставляет знакомый синтаксис SQL и реляционных запросов к структурированным данным.
- MLlib – библиотека алгоритмов машинного обучения для Spark.
- SparkR – поддержка статистического языка R в задачах Spark.
- GraphX – библиотека для анализа графов и коллекций.
- Spark Streaming – обработка постоянных потоков данных в режиме, близком к реальному времени.

Примеры использования Spark в сочетании с Cassandra

Как уже было сказано, Apache Cassandra – отличный выбор для транзакционных систем большого масштаба, нуждающихся в максимальной доступности. А Apache Spark прекрасно зарекомендовал

себя в области анализа больших данных с возможностью масштабирования. Сочетание того и другого открывает много интересных возможностей.

Рассмотрим, к примеру, временные ряды большого объема. Система сбора данных о погоде с тысяч датчиков – отличный кандидат для применения Cassandra. Но проанализировать собранные данные в Cassandra трудно, так как аналитические возможности CQL ограничены. Добавление же в систему Spark открывает новые способы использования собранных данных. Например, можно построить предопределенные агрегаты исходных данных и сохранить их в таблицах Cassandra, сделав доступными клиентским приложениям. Это приближает средства аналитики к пользователям, не заставляя их выполнять сложные запросы к хранилищам данных и дожидаться ответа.

Или взять гостиничное приложение, с которым мы работали на протяжении всей книги. Spark можно было бы использовать для выполнения различных видов анализа данных о бронировании и гостях, например для составления отчетов о динамике доходов или анализа демографического состава обезличенных гостей.

Но чего следует избегать, так это использования связки Spark-Cassandra как альтернативы рабочим нагрузкам, ориентированным на Hadoop. Cassandra хорошо подходит для транзакционной обработки больших объемов данных, но не должна рассматриваться как хранилище данных. Встретившись с ситуацией, где могут понадобиться обе технологии, сначала примените Cassandra для решения той задачи, для которой она подходит, как, например, в главе 2. А затем подумайте о добавлении Spark как средства анализа и обогащения хранящихся в Cassandra данных без включения дорогих и сложных систем извлечения, преобразования и загрузки данных (ETL).

Разворачивание Spark вместе с Cassandra

Cassandra распределяет данные между узлами, ориентируясь на назначение маркеров. Существующим распределением можно с успехом воспользоваться для распараллеливания задач Spark. Поскольку каждый узел содержит подмножество данных кластера, то для интеграции Spark с Cassandra рекомендуется размещать компонент Spark Worker на каждом узле Cassandra (рис. 14.3).

Когда компонент Spark Master получает задачу, компоненты Spark Worker на каждом узле запускают исполнителей Spark Executor для выполнения работы. С помощью соединителя spark-cassandra-connector, играющего роль проводника, данные, необходимые задаче,

читаются по возможности из локального узла. О соединителях мы поговорим чуть ниже.

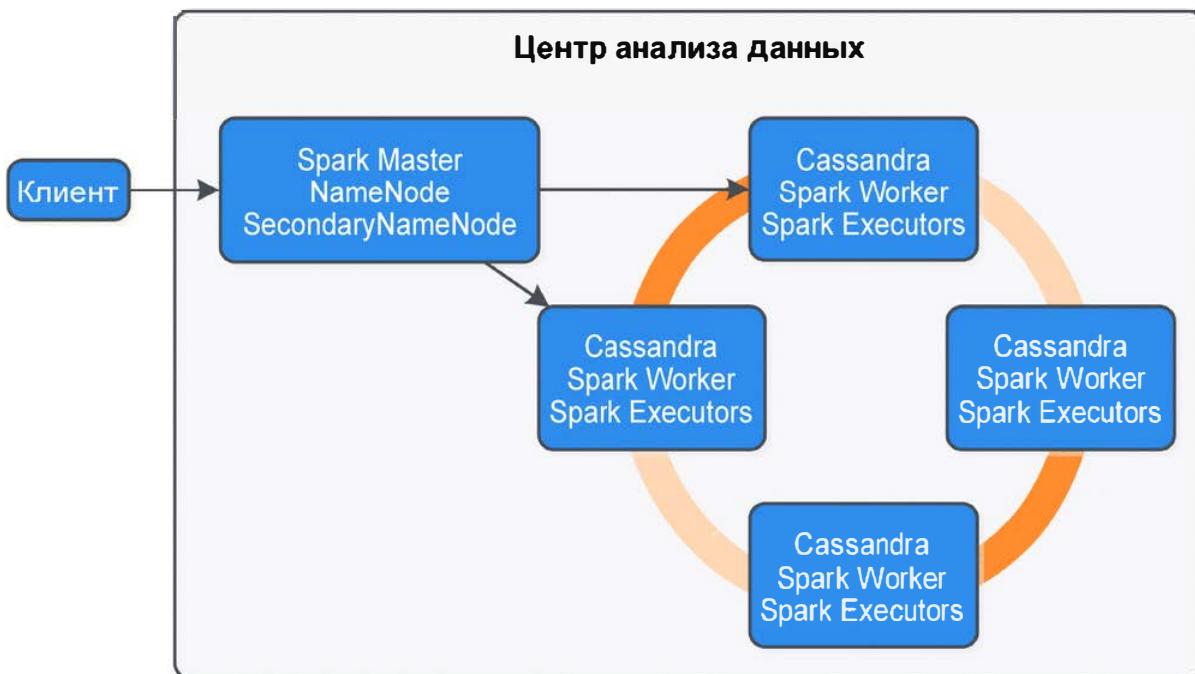


Рис. 14.3 ♦ Топология совмещенного кластера Spark-Cassandra

Поскольку каждый узел содержит часть всех данных кластера, каждый Spark Worker должен будет обработать только это локальное подмножество данных. В качестве примера возьмем подсчет числа записей в таблице. В каждом узле хранится некоторый диапазон данных таблицы. Записи подсчитываются локально, а затем полученные счетчики суммируются.

При таком подходе локальность данных задействуется по максимуму, что позволяет повысить производительность и снизить потребление ресурсов аналитическими задачами. Исполнители Spark Executor обращаются к сети, только когда возникает необходимость объединить свои данные с данными из других узлов. По мере роста кластера достигаемый таким образом выигрыш только увеличивается.

Использование отдельного ЦОДа для анализа данных

При совместном развертывании Cassandra и средств анализа данных, например Spark, часто создают отдельный ЦОД для задач аналитики. Достоинство такого решения в том, что ресурсы, необходимые для анализа, не отвлекаются от других задач, решаемых кластером. Аналитический ЦОД можно сделать «виртуальным», т. е. построить его на том оборудовании, которое уже имеется в помещении,

где организован основной кластер. Воспользовавшись стратегией NetworkTopologyStrategy, мы можем задать для аналитического ЦОДа меньший коэффициент репликации, поскольку необходимая для него доступность, как правило, ниже.

Соединитель *spark-cassandra-connector*

spark-cassandra-connector (<https://github.com/datastax/spark-cassandra-connector>) – это проект с открытым исходным кодом на GitHub, спонсируемый компанией DataStax. Клиенты могут использовать соединитель как проводник, позволяющий читать и записывать данные в таблицы Cassandra через Spark. Соединитель предоставляет, в частности, средства для выполнения SQL-запросов и фильтрации на стороне сервера. Соединитель написан на Scala, но имеется и Java API. С помощью вызовов API spark-cassandra-connector обеспечивает прямой доступ к данным, хранящимся в Cassandra. При доступе к данным из Spark соединитель использует Cassandra как источник данных, производя необходимые преобразования.

Чтобы воспользоваться spark-cassandra-connector, нужно будет скачать как соединитель, так и сам Spark. Подробное руководство по установке мы приводить не будем, но основные моменты опишем. Желающим более детально ознакомиться с предметом рекомендуем вышедшую в издательстве O'Reilly книгу «Learning Spark»¹. Вы можете скачать уже скомпилированную версию Spark или собрать программу из исходного кода самостоятельно.

Если приложение написано на Java или Scala, а для сборки используется Maven, то добавьте в файл проекта *pom.xml* показанные ниже зависимости для доступа к файлам Spark и соединителя:

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-core_2.10</artifactId>
  <version>1.5.2</version>
</dependency>
<dependency>
  <groupId>com.datastax.spark</groupId>
  <artifactId>spark-cassandra-connector_2.10</artifactId>
  <version>1.5.0</version>
</dependency>
```

¹ Карая Х., Конвински Э., Венделл П., Захария М. Изучаем Spark: молниеносный анализ данных. М.: ДМК Пресс, 2015.

Spark поддерживает два режима развертывания: *локальный* и *кластерный*. В кластерном режиме имеются центральный узел Spark Master и много вычислительных узлов. В локальном режиме все работает на одной машине, этот режим используют в основном для разработки. В примере ниже мы работаем в локальном режиме, но для перехода в кластерный требуется всего несколько дополнительных шагов.

Рассмотрим типичные элементы API, используемые в большинстве задач Spark для доступа к данным, хранящимся в Cassandra. Примеры в этом разделе написаны на Scala из-за простоты и удобочитаемости этого языка, а также потому, что сам Spark и многие приложения для него написаны на Scala. Java API похож, но несколько многословнее; вариант того же кода на Java имеется в репозитории книги на GitHub (<https://github.com/jeffreyscarpenter/cassandra-guide/>). Чтобы подключиться из приложения Spark к Cassandra, нужно первым делом создать объект SparkContext, содержащий параметры подключения:

```
val conf = new SparkConf(true)
  .set("spark.cassandra.connection.host", "127.0.0.1")
  .setMaster("local[*]")
  .setAppName(getClass.getName)
  // необязательно
  .set("cassandra.username", "cassandra")
  .set("cassandra.password", "cassandra")

val sc = new SparkContext(conf)
```

Для установления соединения между Cassandra и Spark нужно просто указать на работающий кластер Cassandra и узел Spark Master. В этом примере показано, как соединить локальный узел Cassandra и Spark Master. При необходимости можно также указать учетные данные для подключения к Cassandra.

После того как объект SparkContext создан, можно начинать работу с данными Cassandra, для чего необходимо создать RDD, представляющий таблицу Cassandra. К примеру, создадим RDD для представления таблицы reservations_by_hotel_date из пространства ключей reservation (см. главу 5):

```
val rdd = sc.cassandraTable("reservation", "reservations_by_hotel_date")
```

Имея RDD, можно применить к нему различные преобразования и действия. Так, чтобы получить общее число бронирований, создадим действие для подсчета записей в таблице:

```
println("Число бронирований: " + rdd.count)
```

Поскольку эта операция выполняется как аналитическая задача, работающая параллельно с Cassandra, получается гораздо эффективнее, чем запрос SELECT count(*) FROM reservations из cqlsh.

Поскольку в основе RDD лежит таблица Cassandra, мы можем использовать CQL для фильтрации данных и выборки строк. В Cassandra запросы фильтрации с использованием CQL эффективны, только если в условии присутствует ключ раздела, но при выполнении запросов в виде задачи Spark это ограничение снимается.

Например, может быть поставлена задача построить отчет о бронированиях с указанной конечной датой, чтобы каждый отель знал, кто выписывается в данный день. В этом случае end_date не является ни ключом раздела, ни кластерным столбцом, однако мы можем просмотреть все данные в кластере, выбирая только бронирования с конечной датой 8 сентября 2016 г.:

```
val rdd = sc.cassandraTable("reservation",
    "reservations_by_hotel_date")
    .select("hotel_id", "confirm_number")
    .where("end_date = ?", "2016-09-08")

// Вызвать действие для запуска задачи spark
rdd.toArray.foreach(println)
```

Поиск и выборка данных – только половина дела, еще нужно сохранить данные в Cassandra. В транзакционной базе данных данные традиционно выгружаются в отдельное место для анализа. Но spark-cassandra-connector позволяет извлечь данные, преобразовать их на месте и сохранить в таблице Cassandra, избежав тем самым дорогостоящими и чреватыми ошибками процесса ETL. Сохранить данные в таблице Cassandra на удивление просто:

```
// Создать коллекцию гостей с простыми идентификаторами
val collection = sc.parallelize(Seq(("1", "Delaney", "McFadin"),
    ("2", "Quinn", "McFadin")))

// Сохранить в таблице guests
collection.saveToCassandra("reservation", "guests",
    SomeColumns("guest_id", "first_name", "last_name"))
```

Это простой пример, но идея применима к любым данным. Более интересная задача – посчитать среднюю дневную выручку отелей и записать результат в новую таблицу Cassandra. В случае датчиков

можно было бы вычислить максимальную и минимальную температуры в течение каждого дня и записать результаты в Cassandra.

Для запросов можно использовать не только Spark API. Библиотека SparkSQL позволяет формулировать сложные запросы к данным в Cassandra, применяя знакомый синтаксис SQL, причем даже такие возможности, которых нет в CQL. Нетрудно написать запросы с агрегированием, сортировкой и соединением таблиц.

Для погружения SQL-запросов в программу необходимо создать объект CassandraSQLContext:

```
// Используем SparkContext для создания CassandraSQLContext
val cc = new CassandraSQLContext(sc)

// Задаем пространство ключей
cc.setKeyspace("reservation")
val rdd = cc.cassandraSql("
SELECT hotel_id, confirm_number
FROM reservations_by_hotel_date
WHERE end_date = '2016-09-08'
ORDER BY hotel_id")

// Выполняем действие для запуска задачи SQL
rdd.collect().foreach(println)
```

Синтаксис SQL похож на рассмотренную выше задачу Spark, но более привычен пользователям, поднаторевшим в работе с традиционными базами данных. Если вы хотите исследовать данные, но не писать при этом задачи Spark, воспользуйтесь оболочкой spark-sql, которая находится в подкаталоге *bin* установочного каталога Spark.

Интеграция в DataStax Enterprise

DataStax Enterprise – коммерческая версия Cassandra, которая поддерживает многие виды интеграции, описанные в этой главе. Точнее, DSE Search обеспечивает интеграцию с Solr, а DSE Analytics – интеграцию с Apache Spark и такими элементами экосистемы Hadoop, как MapReduce, Hive и Pig.

Из дополнительных возможностей DSE отметим подключаемые модули поставщиков безопасности и конфигурацию для работы целиком в памяти, полезную приложениям, нуждающимся в сверхбыстром времени отклика.

Резюме

В этой главе мы лишь слегка затронули обширную тему развертывания и интеграции Cassandra с различными системами. Надеемся, что пробудили в вас интерес к широкому спектру направлений, в которых можно развить ваше приложение, воспользовавшись Cassandra и смежными технологиями.

Вот мы и подошли к концу путешествия. Если мы достигли поставленной цели, то вы теперь хорошо понимаете, какие задачи пригодны для использования Cassandra и как подходить к проектированию, реализации, развертыванию и обслуживанию успешных приложений.

Предметный указатель

Символы

`$CQLSH_HOST`, переменная среды, [82](#)

`$CQLSH_PORT`, переменная среды, [82](#)

А

Автоматическая начальная загрузка, [175](#)

Автоматический снимок, [307](#)

Автоматически масштабируемые группы (ASG), [313](#)

Автоматическое разбиение на страницы, [249](#)

Авторизация

использование класса

`CassandraAuthorizer`, [350](#)

ролевое управление доступом, [351](#)

Агрегаты, [245](#)

встроенные, [248](#)

написанные на Java

или JavaScript, [245](#)

пользовательские, [246](#)

Анализ данных

`Apache Spark`, [372](#)

использование отдельного ЦОДа, [375](#)

Антиуплотнение, [290](#)

Антиэнтропия, [160](#)

Архитектура Cassandra, [140](#)

антиэнтропия, исправление и деревья Меркля, [160](#)

вручение напоминаний, [154](#)

диспетчеры и службы, [164](#)

запросы

и узлы-координаторы, [150](#)

кольца и маркеры, [145](#)

многоступенчатая событийно-

ориентированная архитектура (SEDA), [162](#)

надгробья, [157](#)

осведомители, [144](#)

системные пространства ключей, [167](#)

сплетни и обнаружение отказов, [142](#)

стратегии репликации, [148](#)

таблицы в памяти, файлы SSTable и журналы фиксаций, [151](#)

уплотнение, [159](#)

уровни согласованности, [149](#)

фильтры Блума, [158](#)

центры обработки данных и стойки, [140](#)

Архитектура без разделения ресурсов, [38](#)

Асинхронное выполнение

внутренний протокол CQL, [198](#)

драйвер DataStax для Java, [201](#)

Атомарность (транзакции), [32](#)

в распределенных системах, [35](#)

Атомарные пакеты, [233](#)

Атрибуты (JMX), [266](#)

Аутентификация

двусторонняя аутентификация

сертификатов, [356](#)

поддержка в драйвере DataStax

для Java, [197](#)

по паролю, [345](#)

добавление пользователей, [347](#)

дополнительные поставщики

аутентификации, [347](#)

настройка автоматического входа, [348](#)

настройка

аутентификатора, [346](#)

средствами драйвера DataStax

для Java, [349](#)

удаленная через JMX, [359](#)

Б

Балансировщики нагрузки, вредность, 366
 Безопасность, 344
 MBean-объекты, 278, 360
 аутентификация и авторизация, 345
 законодательные нормы и требования, 344
 на уровне JMX, 358
 средства, встроенные в Cassandra, 345
 шифрование, 352
 Блоги сообщества Cassandra, 69
 Блума фильтры, 153, 158
 другие применения, 159
 информация о, 284
 настройка, 240
 существование раздела в файле SSTable, 239
 Бронирование номеров
 логическая модель данных, 124
 материализованные представления, 131
 физическая модель данных, 128
 Брюер Эрик, 52, 162
 Буферный кэш, 328

В

Версия сервера, получение в cqlsh, 84
 Вертикальное масштабирование, 28
 Виртуальные машины
 образы, 81
 экземпляры, 370
 Виртуальные узлы, 147, 181
 выключение и настройка диапазонов, 181
 создание кластера программой csm, 172
 Вложенные коллекции, 108
 Временные метки, 97
 Вручение напоминаний, 154, 226
 доступ к параметрам с помощью StorageProxyMBean, 272
 дублирование средств управления, 274
 и восстановление узла, 298

и гарантированная доставка, 155
 исключение таблицы system.hints, 169
 настройка, 329

Встроенные функции и агрегаты, 248
 Вторичные индексы, 110
 и материализованные представления, 129
 перестройка, 293
 Вход в систему
 настройка автоматического входа, 348
 от имени суперпользователя, 347
 Вывод узла из эксплуатации, 300
 Выработка консенсуса, алгоритмы, 156
 Высокая доступность, 48

Г

Гарантиированная доставка и вручение напоминаний, 155
 Гистограммы, команды формирования, 316
 Горизонтальная масштабируемость
 в базах данных NoSQL, 42
 в реляционных базах данных, 33
 Горячие узлы, 177
 Графовые базы данных, 41, 54

Д

Движок хранения, 164
 Двухфазная фиксация, 33, 156
 проблема, 34
 Денормализация, 29
 в логической модели данных о бронировании, 124
 в реляционной базе данных и в Cassandra, 116
 на стороне сервера с помощью материализованных представлений, 117
 Диапазоны маркеров, 168
 исправление поддиапазона, 291
 Динамическое осведомление, 145
 Диски
 JBOD и RAID, 364
 конфигурация, 364

Диспетчеры, 164

Дифференциальная резервная копия, 305

Документные хранилища, 41

Долговечная запись, 188, 225

Долговечность (транзакции), 33

Доступность

высокая доступность

и отказоустойчивость

в Cassandra, 47

и уровня соглашенностии, 224

теорема CAP, 52

Драйвер DataStax

для C#, 219

для C/C++, 220

для Java, 194

автентификация, 349

доступ к метаданным

кластера, 211

задание уровня соглашенностии

по умолчанию, 225

исключения, 209

кластеры и точки контакта, 195

настройка среды разработки, 195

объекты Statement, 199

отладка и мониторинг, 215

поддержка пакетов, 234

политики, 207

разбиение на страницы, 249

сеансы и пулы соединений, 197

для Node.js, 218

для PHP, 222

для Python, 217

для Ruby, 219

Ж

Журнал пакета, 235

Журналы фиксаций, 151, 183

запись в файловую систему, 227

накатывание, 152, 326

настройка, 326

З

Замораживание коллекций, 108

Запрос дайджеста, 238, 240

Защита от быстрого чтения, 252

И

Иерархические базы данных, 26

Избыточная потоковая передача, 289

Изолированность (транзакции), 33

Индексы

вторичные, 110

выделение памяти, 329

замена вторичных индексов

индексом Lucene, 371

оптимизация, 28

переиндексирование, 293

построение над объединенными файлами SSTable, 159

сводка раздела и индекс раздела, 240

сводные для файлов SSTable, 328

хранение определений, 168

Инкрементная резервная копия, 305

включение режима, 307

Инкрементное исправление, 290

переход на, 290

Исключения

в драйвере DataStax для Java, 209

и класс CompactionManager, 268

ошибки при подключении, 82

сеансы и пулы соединений, 197

Исправление, 288

nodetool repair, 288

антиэнтропийные механизмы, 226

антиэнтропия и деревья

Меркла, 160

диапазона разделителя, 291

поддиапазона, 291

полное исправление, инкрементное исправление и антиуплотнение, 290

последовательное

и параллельное, 291

рекомендации, 292

Исправление на этапе чтения, 161, 226, 241

К

Кластерные столбцы, 94, 243

определение порядка

сортировка, 118

- хранение атрибутов для запросов по диапазону, 124
- Кластеры, 95
- взаимодействие с драйвером DataStax для Java, 195
 - создание специального инициализатора, 196
 - добавление узлов, 185
 - добавление центра обработки данных, 296
 - доступ к метаданным из драйвера DataStax для Java, 211
 - информация о структуре в системных таблицах, 168
 - описание текущего кластера в cqlsh, 84
 - оценка размера, 361
 - получение информации с помощью nodetool, 279
 - руководство по переходу на новую версию, 303
 - создание, 171
 - топология кластера в AWS, 367
 - топология кластера в GCE, 370
 - топология кластера в Microsoft Azure, 369
 - топология сети, 178
 - узлы-распространители, 175
- Клиентские интерфейсы, 183
- Клиенты, 72, 193
- JMX, 263
 - определение версии в cqlsh, 84
 - поддержка операций чтения и записи, 151
 - сообщения в журнале, 78
 - трассировка, 318
 - устаревшие, Hector, Astyanax и другие, 193
- Ключи раздела, 94, 243
- добавление столбца, 135
 - явное выделение, 136
- Кодд Эдгар, 26
- двенадцать правил, 29
- Кодек пользовательский, 201
- Коллекции, 104
- list, тип данных, 106
 - map, тип данных, 107
- set, тип данных, 105
- замораживание, 108
- Кольца, 145
- динамическое присоединение, 187
 - получение информации с помощью nodetool, 280
- Команды
- в драйвере DataStax для Java, 199
 - BoundStatement, 203
 - PreparedStatement, 202
 - SchemaStatement, 214
 - SimpleStatement, 199
 - асинхронное выполнение, 201
 - включение трассировки, 320
- Компенсация, 34
- Контрольное уплотнение, 289
- Контрольные сообщения, драйвер DataStax для Java, 199
- Конференции сообщества Cassandra, 70
- Конфигурационные файлы, 171
- обновление при переходе на новую версию, 304
 - резервное копирование, 304
- Концептуальная модели данных, 113
- диаграмма сущность-связь
 - бронирования номеров в отеле, 114
- Коэффициент репликации, 51, 148, 362
- изменение, 191
 - для пространства ключей system_auth, 352
 - и уровень согласованности, 150
 - создание пространства ключей в cqlsh, 84
- Криптография
- алгоритм хэширования в классе RandomPartitioner, 176
 - алгоритмы и библиотеки, 356
- Куча JVM
- определение размера, 337
 - размер доступной памяти и использование в Cassandra, 262
 - размещение буферов, 328
 - размещение таблиц в памяти, 325

- Кэширование, 322
 виды кэширования
 в Cassandra, 154
 и запросы на чтение, 239
 информации о ролях
 и правах, 360
 и операции записи, 227
 кэш ключей, 240
 настройка и оптимизация, 323
 кэш строк, настройка
 и оптимизация, 323
 методика настройки, 322
 настройка буферного кэша, 328
 сообщения в журнале, 78
 сохранение кэшней, 324
 управление кэшами
 из nodetool, 325
- Кэш ключей, 154
 настройка и оптимизация, 323
- Кэш строк, 154
 настройка и оптимизация, 323
- Кэш счетчиков, 154, 324
- Л**
- Линеаризуемая согласованность, 156
- Локаль, установка в csqlsh, 101
- М**
- Маркеры, 145
 allocate_tokens_keyspace,
 свойство, 177
 num_tokens, свойство, 181
 вычисление для узлов, 146
 перемещение, 294
 перераспределение между
 узлами, 186
 получение списка, 173
- Массовая загрузка, 242
- Масштабирование
 в масштабе веба, 39
 в реляционных базах данных,
 проблемы, 39
 горизонтальное в базах данных
 NoSQL, 42
- Масштабируемость, 46
 проблемы в реляционных базах
 данных, 28
- Материализованные
 представления, 117, 129, 165
 дополнительные
 возможности, 131
 запись в таблицы, 228
 хранение определений
 и информации о построении, 168
- Меркла деревья, 161, 289
- Метрики
 поддерживаемые Cassandra, 276
 сбор в драйвере DataStax
 для Java, 216
 сборки мусора, 278
 сброс, 317
- Многопоточность, 333
- Моделирование данных, 113
 определение запросов
 в приложении, 119
 определение схемы базы
 данных, 135
 оценка и уточнение, 131
 вычисление размера
 раздела, 132
 оценка места, занятого
 на диске, 133
 паттерны и антипаттерны, 125
 построение концептуальной
 модели, 113
 построение логической
 модели, 120
 построение физической
 модели, 126
 материализованные
 представления, 129
 разбиение больших
 разделов, 134
 различия в проектировании
 для РСУБД и Cassandra, 115
- Модель параллелизма, 162
- Мониторинг, 255
 использование MBean-объектов
 Cassandra, 267
 производительности, 316
 протоколирование, 255
 с помощью nodetool, 278
 средствами JMX, 259
- Мягкое удаление, 157

H

Надгробья, 157, 253
 Напоминания, 154
 Настраиваемая согласованность, 48
 Настройка Cassandra, 170
 Cassandra Cluster Manager, 170
 динамическое присоединение к кольцу, 187
 добавление узлов в кластер, 185
 конфигурация узлов, 181
 осведомители, 178
 разделители, 176, 177
 создание кластера, 171
 стратегии репликации, 188
 узлы-распространители, 175
 Настройка производительности, 314
 вручение напоминаний, 329
 журналы фиксаций, 326
 кэширование, 322
 параллелизм
 и многопоточность, 333
 параметры JVM, 337
 сеть и таймауты, 335
 таблицы в памяти, 325
 уплотнение, 330
 управление
 производительностью, 314
 утилита cassandra-stress, 340
 файлы SSTable, 328
 Начальная загрузка, 175
 выключение автоматического режима, 296
 конфигурирование узлов, 295
 Неформальные встречи пользователей Cassandra, 69

O

Облачные среды
 осведомители, 145, 180
 поставщики облачных решений
 PaaS, 30
 развертывание, 366
 Amazon Web Services, 367
 Google Cloud Platform, 370
 Microsoft Azure, 369
 стоимость ресурсов, 367
 Облегченные транзакции, 156, 230

Обнаружение отказов

φ-алгоритм с накоплением, 143
 реализация в Cassandra, 144
 с помощью класса Gossiper, 142

Обнаружение отказов

с накоплением, 143
 Обновление-вставка,
 операция, 224, 231
 Оболочка CQL (cqlsh), 72
 запуск, 81
 ошибки при подключении, 82
 подключение к конкретному узлу, 82
 простые команды, 82
 HELP или ?, 82
 без параметров, 83
 история команд, 89
 нечувствительность
 к регистру, 82
 описание окружения, 84
 создание пространства ключей
 и таблицы, 84
 чтение и запись данных, 88
 трассировка, 318
 установка локали, 101

Обслуживание, 285

добавление узлов в ЦОД, 294
 исправление, 288
 обработка отказа узла, 297
 очистка данных в узле, 287
 переиндексирование, 293
 перемещение маркеров, 294
 переход на новую версию Cassandra, 303
 резервное копирование и восстановление, 305
 средства, 310

 DataStax OpsCenter, 310

 Netflix Priam, 313

 утилиты для работы с файлами SSTable, 309

Объектно-реляционное отображение (ORM), 36

Ограничение трафика (дросселирование), 336

Осведомители, 144, 171, 178

 DynamicEndpointSnitch, 180

- GossipingPropertyFileSnitch, 179
 MBean-объекты, 274
 PropertyFileSnitch, 178
 RackInferringSnitch, 179
 SimpleSnitch, 178
 использование файла свойств, 299
 конфигурирование, 296
 облачные, 180
 Оснащение инструментальными средствами, 260
 Останов Cassandra, 79
 Отказоустойчивость, 47
- П**
- Пакеты, 233
 массовая загрузка, 234
 непротоколируемые, 233
 ограничения на размер, 235
 таблица system.batchlog, 168
 Пакеты модификаций счетчиков, 233
 создание в драйверах DataStax, 235
 Память
 HeapMemoryUsage, атрибут, 266
 буферный кэш, 328
 выделение под кэш счетчиков, 324
 для таблиц в памяти, 325
 параметры JVM, 337
 Параллелизм, 333
 Пароли
 jmxremote.password, файл, 359
 в команде LOGIN, 349
 Паттерн временных рядов, 125
 Первичные ключи, 92, 242
 замороженные коллекции, 108
 неизменяемость, 103
 строк в Cassandra, 95
 уникальность, 122
 Перечисляемые типы, и CQL, 100
 Подготовленные команды
 в драйвере DataStax для Java, 202
 привязка значений, 203
 поддержка в CQL, 63
 Полная резервная копия, 305
 Полное исправление, 290
 Полное уплотнение, 160
- Пользователи
 добавление, 347
 команды ALTER USER и DROP USER, 349
 предоставление прав доступа, 351
 ролевое управление доступом, 351
 Пользовательские агрегаты, 246
 дополнительные команды, 248
 Пользовательские типы, 107
 отображение на классы Java, 205
 уменьшение дублирования столбцов, не входящих в состав первичного ключа, 128
 Пользовательские функции, 245
 безопасность, 245
 дополнительные команды, 248
 использование в пользовательских агрегатах, 246
 Поочередная модернизация, 303
 Построение логической модели данных, 120
 диаграммы Чеботко, 121
 модель данных о бронировании, 124
 модель данных отеля, 122
 Построение физической модели данных
 диаграммы Чеботко, 126
 материализованные представления, 129
 Потоки, 162
 количество активных в Cassandra, 262
 Преобразователи объектов
 в драйвере DataStax для Java, 205
 в драйвере DataStax для Python, 218
 Причинно-следственная согласованность, 49
 Проектирование от запроса, 117
 Пространства ключей, 95
 allocate_tokens_keyspace, свойство, 177
 именование, 86
 переключение на (cqlsh), 86
 получение статистики с помощью nodetool, 283

- пользовательские типы, 108
 пользовательские функции, 246
 сброс на диск, 287
 снимки, 306
 создание в cqlsh, 84
 хранение определений, 168
Протоколирование, 255
 изменение уровня, 256, 271
 изучение журналов, 258
 настройка в производственной среде, 256
 отсутствие файлов журналов, 257
 параметры протоколирования в Cassandra, 185
 просмотр и задания уровня с помощью nodetool, 281
 утилита tail, 257
Протоколируемые пакеты, 233
Протокол распространения сплетен, 142
 использование для обнаружения отказов, 142
 команда nodetool gossipinfo, 282
 сообщения в журнале, 186
Пул соединений
 внутренний протокол CQL, 193
 и сеансы, 197
Пулы потоков, 277
- P**
- Разбиение на страницы, 249
Развертывание, 361
 Spark вместе с Cassandra, 374
 в облаке, 366
 планирование развертывания кластера, 361
 с помощью Docker и других контейнеров, 363
Разделители, 147, 171, 176
 ByteOrderedPartitioner, 177
 Murmur3Partitioner, 176
 OrderPreservingPartitioner, 176
 RandomPartitioner, 176
 избегание горячих узлов, 177
 переход на новую версию Cassandra, 304
- Разделы, 94
 вычисление размера, 132
 вычисление размера ячеек, 134
 избегание больших разделов, 318
 минимизация количества просматриваемых при обработке запроса, 118
 облегченные транзакции и Paxos, 156
 оценка размера, занятого на диске, 134
 разбиение больших разделов, 134
Разделяемые хранилища, нежелательность, 365
Раскладывание по корзинам, 135
Распределенные транзакции, 33
Режим анализа записи, 331
Резервное копирование и восстановление, 305
Реляционная модель данных, 90
Реляционные базы данных, 27
 клиентские драйверы, 193
 моделирование данных, отличия от Cassandra, 115
 положение на диаграмме CAP, 54
 проблемы, 27
 сложности масштабирования, 39
 схемы, 35
 таблицы, 93
 транзакции, свойства ACID и двухфазная фиксация, 27, 32
 язык SQL, 31
Репликация, 46
 и согласованность в реляционных базах данных, 28
Реплики
 узлы координатора для операций чтения и записи, 150
 узлы-реплики на пути записи, 226
 узлы-реплики на пути чтения, 238
Ролевое управление доступом, 351
- C**
- Сборка мусора, 253
 настройка, 338
 переход на G1GC, 339
Сводка раздела, 240

- Сеансовый ключ, 353
 Сегментирование, 36
 как пример архитектуры без разделения ресурсов, 38
 стратегии, 38
 Сертификаты, 353
 двустронняя аутентификация, 356
 упрощенное управление, 357
 Сжатие
 commitlog_compression, свойство, 327
 в драйвере DataStax для Python, 218
 сообщений между клиентом и сервером в драйвере DataStax для Java, 196
 файлов SSTable, 153, 315
 файлов журналов, 257
 Синхронизация реплик, 161
 Системные пространства ключей, 84, 167
 и версии Cassandra, 168
 неизменяемость, 169
 описание атрибутов, 169
 оценка размера, 362
 список таблиц, 167
 Склад ключей, 354, 356
 Слабая согласованность, 50
 Службы
 CassandraDaemon, 164
 EmbeddedCassandraService, 164
 MBean-объекты, 278
 MessagingService, 166
 StorageProxy, класс, 165
 StorageService, класс, 165
 Снимки, 305
 восстановление из, 308
 местонахождение в файловой системе, 307
 нескольких узлов на момент времени, 307
 получение списка, 306
 создание, 306
 стирание, 307
 файлов SSTable, 273
 Согласованность, 28
 в теореме CAP, 52
 модели, 49
 настраиваемая, 48
 облегченные транзакции и Paxos, 156
 семантика ACID, 32
 Согласованность в конечном счете, 49
 информации о схеме, 214
 Сортировка, в РСУБД и в Cassandra, 118
 Составной ключ, 94
 Списки рассылки Cassandra, 68
 Срезки, 241
 Срок хранения данных (TTL), 98
 для трасс, 321
 Ссылочная целостность, 116
 Статические столбцы, 94
 Стойки, 140, 179
 в GossipingPropertyFileSnitch, 179
 в осведомителях для EC2 и Google Cloud, 180
 Столбцовые хранилища, 41
 Столбцы
 в Cassandra, 92, 94, 97
 временные метки
 для обновленных значений, 97
 имена, 86
 срок хранения данных (TTL), 98
 статические, 94
 клUSTERНЫЕ, 94
 Стратегии репликации, 148, 188
 LocalStrategy, 190
 NetworkTopologyStrategy, 179, 190, 296
 OldNetworkTopologyStrategy, 190
 SimpleStrategy, 178, 189
 для пространства ключей system, 169
 изменение коэффициента репликации, 191
 унаследованные, 149
 Строгая согласованность, 49
 Суперпользователь, 348
 Сущность-связь, диаграмма, 114

Схемы

бессхемная модель и Cassandra, 57
 восстановление, 308
 в реляционных базах данных, 35
 выявление рассогласований, 279
 доступ в кластере, 214
 конфликты при программном определении, 215
 определение схемы базы данных, 135
 применение транзакций при создании, 232

Счетчики

и запись, 228
 повышение производительности с помощью кэширования, 324

Т**Таблицы**

вычисление размера, 133
 данные о производительности, 316
 именование, 86
 оптимизация на этапе проектирования, 118
 получение описания в cqlsh, 87
 получение статистики с помощью nodetool, 283
 сброс на диск, 286
 создание в cqlsh, 86
 хранение определений, 168

Таблицы в памяти, 151

настройка, 325
 обработка запросов на чтение, 239
 принудительный сброс на диск, 286
 реализация операций в виду ступеней, 163
 статистика, 284

Таймауты, 335, 366

Текстовые типы данных, 100
 «Тик-так», 65
 Типы данных в CQL, 99
 boolean, 103
 counter, 104
 inet, 104
 времени и идентификации, 101
 коллекции, 104

пользовательские, 107

текстовые, 100
 числовые, 99

Точки общего отказа, отсутствие, 46

Трассировка, 318
 команда nodetool settraceprobability, 321
 команда TRACING ON, 318
 поддержка в DataStax DevCenter, 320
 срок хранения трасс, 321

Триггеры, 168

У

Удаление данных, 252
 мягкое, 157
 надгробья, 157, 253

Узкие строки, 93

Узлы

взаимодействия на пути записи, 226
 взаимодействия на пути чтения, 238
 виртуальные, 147
 добавление в кластер, 185
 добавление в существующий ЦОД, 294
 конфигурация, 181
 обнаружение средствами драйвера DataStax, 212
 обработка отказа, 297
 замена, 299
 исключение, 300
 остановка и запуск без прерывания работы кластера, 187
 проверка исправности, 285
 ремонт, 298
 узлы-распространители для нового узла, 175

Узлы-координаторы, 150

взаимодействия на пути записи, 226
 взаимодействия на пути чтения, 238
 в облегченных транзакциях, 156

Узлы-распространители, 171, 175, 296

- замена, 300
удаление, 303
- Универсально уникальные идентификаторы (UUID), 102
- Уничтожение узла, 302
- Уплотнение, 159, 253
антиуплотнение, 290
больших разделов, 318
контрольное, 289
настройка, 330
порог, 332
стратегии, 159, 330, 362
тестирование в режиме анализа записи, 331
утилиты для управления, 309
- Упреждающее выполнение, 252
- Уровень кэширования, 28
- Уровень подозрения, 143
- Уровни согласованности, 51, 149
для клиентов после добавления нового ЦОДа, 297
и вручение напоминаний, 154
и коэффициенты репликации, 150
исправление на этапе чтения, 241
настройка уровней согласованности чтения и записи, 237
последовательный, 232
по умолчанию, 225
при записи, 224
при удалении, 253
при чтении, 236
- Условная команда записи, 231
- Установка Cassandra, 71
- Устойчивость к разделению
в теореме CAP, 52
компромисс с согласованностью и доступностью, 48
максимизация в Cassandra, 141
- Устойчивый распределенный набор данных (RDD), 373
создание, 377
- Ф**
Файлы данных, 184
Фильтрация, 243, 316
Функции, 244
встроенные, 248
- пользовательские, 168, 245
Функция состояния, 247
- Х**
Хранилища ключей и значений, 41
Хранилище доверенных сертификатов, 355
- Ц**
Центры обработки данных, 140, 179
добавление в кластер, 296
добавление узлов, 294
использование отдельного ЦОДа для анализа данных, 375
ограничение области исправления одним ЦОДом, 289
узлы-распространители, 175
- Ч**
Чат сообщества Cassandra, 68
Чеботко диаграммы, 121
для физических моделей данных, 126
Числовые типы данных, 99
Чтение раньше записи, семантика, 157
- Ш**
Широкие строки, 93, 125
Шифрование, 352
SSL, TLS и сертификаты, 353
в драйвере DataStax для Java, 197
планы развития, 352
трафика между клиентами и узлами, 357
трафика между узлами, 354
- Э**
Эдем, область кучи, 338
Эластичная масштабируемость, 46
- Я**
Язык определения данных (DDL), 31
Ячейки (значения), количество в разделе, 132

A

AbstractCompactionStrategy, 159
 AbstractReplicationStrategy, 148
 ACID, 32
 ActiveRepairService, 289
 AddressTranslator, интерфейс, 211
 allocate_tokens_keyspace, свойство, 177
 AllowAllAuthenticator, 346
 AllowAllAuthorizer, 350
 AllowAllInternodeAuthenticator, 347
ALLOW FILTERING,
 фраза, 243, 316
 ALL, уровень согласованности, 149,
 224, 237
 и упреждающее выполнение, 252
 ALTER TABLE, команда, 96
 ALTER USER, команда, 349
 Amazon Dynamo, 45, 51
 Amazon EC2, 145, 180
 Amazon Web Services (AWS), 30
 интеграция с Priam, 313
 осведомители, 180
 развертывание Cassandra, 367
 Ant
 дополнительные цели сборки, 75
 компиляция исходного кода
 Cassandra, 74
 скачивание, 73
 ANY, уровень согласованности, 155,
 224, 227
 Apache Cassandra Wiki, 69
 Apache Cloudstack, 145, 180
 Apache Hadoop, 61, 371
 Apache Lucene, 371
 Apache Solr, 371
 APPLY BATCH, команда, 234
 ascii, тип, 101
 AS SELECT, фраза (CREATE
 MATERIALIZED VIEW), 130
 Astyanax, клиент, 194
 AuthProvider, интерфейс, 349
 auto_snapshot, свойство, 307
 avg, функция, 249

B

BEGIN BATCH, команда, 234
 Bigtable, 45

бессхемная база, 57
 положение на шкале CAP, 55
 производные базы данных, 54
 bin, каталог, 72
 BlacklistedDirectoriesMBean, 298
 blob, тип данных, 103
 boolean, тип данных, 103
 BuiltStatement, класс, 204
 ByteOrderedPartitioner, 177

C

CacheServiceMBean, 273
 CAP теорема, 51
 новый взгляд, 56
 Cassandra
 для каких проектов подходит, 67
 истоки, 58
 история версий, 60
 краткая презентация, 44
 распределенная
 и децентрализованная база
 данных, 45
 сегментирование и архитектура
 без разделения ресурсов, 38
 участие в жизни сообщества, 68
 эластичная масштабируемость, 46
 CassandraAuthorizer, 350
 Cassandra Cluster Manager. См. csm
 CassandraDaemon, 164
 cassandra-env.sh, файл, 286
 CASSANDRA_HOME, переменная
 среды, 76
 Cassandra Loader, 242
 Cassandra Query Language. См. CQL
 cassandra-rackdc.properties,
 файл, 179, 180
 CassandraRoleManager, 347
 Cassandra-Sharp, клиент
 для платформы Microsoft .NET, 194
 CassandraSQLContext, 379
 cassandra-stress, 340
 cassandra.thrift, файл, 72
 cassandra-topology.properties,
 файл, 178
 cassandra.yaml, файл, 72
 задание места расположения
 файлов данных, 77

- справочник, 181
 csm (Cassandra Cluster Manager), 170
 добавление узлов в кластер, 185
 дополнительные сведения о конфигурации кластера, 174
 запуск и проверка состояния кластера, 173
 запуск узла, 186
 остановка узла, 187
 получение списка кластеров, 172
 получение списка маркеров, принадлежащих узлу, 173
 проверка состояния кластера, 186
 проверка состояния отдельных узлов, 173
 просмотр журналов с любого узла, 186
 создание кластера, 172
 cipher_suites, параметр, 356
 clean, цель сборки, 75
 client_encryption_options, 357
 CLI (командный интерфейс), клиент, 72
 нерекомендованный, 81
 CloudstackSnitch, 180
 cluster.conf, файл (csm), 174
 cluster_name, свойство, 172
 ColumnFamilyStoreMBean, 272
 ColumnFamilyStore, класс, 164
 commit_failure_policy, свойство, 184
 commitlog_directory, свойство, 184
 CommitLogMBean, 273
 commitlog_sync, свойство, 327
 CompactionManagerMBean, 268, 273, 333
 CompactionManager, класс, 159, 268
 реализация метода stopCompaction(), 269
 concurrent_compactors, свойство, 333
 Concurrent Mark Sweep (CMS), сборщик мусора, 338
 conf, каталог, 72
 CONSISTENCY, команда, 225
 COPY, команда, 242
 counter, тип данных, 104
 COUNT, функция, 248
 CQL (Cassandra Query Language), 57, 90
 внутренний транспортный протокол, 166
 одновременная передача нескольких запросов, 198
 поддержка в драйверах DataStax, 196
 вторичные индексы, 110
 интерфейс программирования, 183
 модель данных Cassandra, 91
 типы данных, 99
 cql-rb, get-пакет, 219
 CREATE AGGREGATE, команда, 247
 CREATE CUSTOM INDEX, команда, 112
 CREATE FUNCTION, команда, 245
 фраза LANGUAGE, 245
 CREATE INDEX ON, команда, 110
 CREATE_KEYSPACE, команда, 85
 CREATE MATERIALIZED VIEW, команда, 130
 CREATE OR REPLACE, команда, 248
 CREATE TABLE, команда, 87, 243
 CREATE TYPE, команда, 108
 CREATE USER, команда, 348
 cstard_perf, платформа тестирования производительности, 343
 CSV-файлы, загрузка и выгрузка, 242
 Cygwin, эмулятор оболочки iash, 257
- D**
- data_file_directories, свойство, 184
 DataStax Enterprise, 379
 data, каталог, 76
 dateOf(), функция, 102
 DateTieredCompactionStrategy (DTCS), 160
 date, тип данных, 101
 DCAwareRoundRobinPolicy, 208
 decimal, тип данных, 100
 DefaultRetryPolicy, 209
 DELETE, команда, 88

DESCRIBE AGGREGATES, команда, 248
 DESCRIBE FUNCTIONS, команда, 248
 DESCRIBE KEYSPACE, команда, 86, 108
 DESCRIBE TABLE, команда, 87, 96
 DESCRIBE команда, 84
 DevCenter
 поддержка трассировки, 320
 редактирование схемы базы данных, 138
 diff, команда, 174, 304
 disk_failure_policy, свойство, 184, 298
 Docker, развертывание Cassandra, 363
 DowngradingConsistencyRetry-Policy, 209
 DROP AGGREGATE, команда, 248
 DROP FUNCTION, команда, 248
 DROP INDEX, команда, 111
 DROP TABLE, команда, 89
 DROP USER, команда, 349
 Dropwizard Metrics, библиотека, 277
 DseAuthProvider, 350
 DynamicEndpointSnitch, 180, 274
 DynamicEndpointSnitchMBean, 274

E

EACH_QUORUM, уровень согласованности, 224, 237, 297
 Ec2MultiRegionSnitch, 180, 369
 Ec2Snitch, 180, 369
 Elastic Block Store (EBS), 368
 Elasticsearch, 371
 EmbeddedCassandraService, 164
 EndpointSnitchInfoMBean, 274
 endpoint_snitch, свойство, 178, 296
 ExecutorService, класс, 162

F

FailureDetectorMBean, 275
 FailureDetector, класс, 144
 FallthroughRetryPolicy, 209
 file_cache_size_in_mb, свойство, 328

G

G1GC, сборщик мусора, 339
 gc_grace_seconds, свойство, 298
 gc_warn_threshold_in_ms, свойство, 339
 Google Cloud Platform, 30, 145, 370
 GoogleCloudSnitch, 180, 370
 GossiperMBean, 275
 Gossiper, класс, 142
 GossipingPropertyFileSnitch, 179, 299

H

Hadoop, 61, 371
 Hector, клиент, 193
 Helenus, клиент, 194, 218
 HELP BLOB_INPUT, команда, 103
 HintedHandOffManagerMBean, 274
 HintedHandOffManager, класс, 155
 hints_directory, свойство, 329
 HintsServiceMBean, 274
 Host.StateListener, интерфейс, 212

I

IAuthenticator, 347
 IEndpointSnitch, интерфейс, 145
 IFailureDetector, интерфейс, 144
 IF NOT EXISTS, фраза
 в команде CREATE USER, 348
 в команде INSERT, 230
 в пользовательских функциях и агрегатах, 248
 IIInternodeAuthenticator, интерфейс, 347
 IncomingTcpConnection, класс, 166
 incremental_backups, свойство, 308
 inet, тип данных, 104
 initial_token, свойство, 147, 181
 INSERT, команда, 88
 построение с помощью
 QueryBuilder, 204
 фраза IF NOT EXISTS, 230
 фраза USING TTL, 99

interface, каталог, 72
 internode_encryption, параметр, 355
 IN, оператор, 244
 IPartitioner, интерфейс, 148

IP-адреса
 RackInferringSnitch, 179
 трансляция с помощью
 AddressTranslator, 211
 узлов-распространителей, 175
 IVerbHandler, интерфейс, 163

J

jar, цель сборки, 75
 javadoc, каталог, 73
 JAVA_HOME, переменная среды, 76
 java.lang.management.ThreadMX-
 Bean, 265
 java.lang.Threading, 265
 JavaScript, язык для написания
 пользовательских функций, 245
 JBOD (just a bunch of disks), тип
 развертывания, 184, 365
 JConsole, 261
 JDBC, 193
 JMXConfigurator, 266
 JMXEnabledScheduledThreadPool-
 Executor, 277
 JMXEnabledThreadPoolExecutor, 277
 JMX (Java Management Extensions)
 Mbean-объекты, 264
 архитектура, 261
 безопасность, 358
 интеграция с библиотекой
 Metrics, 216
 использование
 для мониторинга, 255
 клиенты, отличные
 от JConsole, 263
 настройки в скрипте
 cassandra.env.sh, 185
 операции управления
 приложением, 259
 перекрытие функциональности
 с nodetool, 278
 подключение к Cassandra
 через JConsole, 261
 получение информации
 о MessagingService, 166
 разрешение удаленного
 подключения, 262
 сообщения в журнале, 78

jmxremote.access, файл, 359
 jmxremote.password, файл, 359
 Jmxterm, 264
 JRuby, 219
 JVM (Java Virtual Machine)
 версия, необходимая
 для Cassandra, 76
 задание параметров, 185
 оснащение средствами
 контроля, 260
 остановка процесса Cassandra, 80
 параметры, 337
 сборки мусора, 338
 управления памятью, 337
 параметры в файле
 cassandra-env, 360
 сообщения в журнале, 78
 управление по протоколу
 SNMP, 260
 jvm.options, файл, 337

K

KerberosAuthenticator, 350
 key_cache_size_in_mb, параметр, 323
 keys, атрибут, 323
 keytool, утилита, 354
 KillrVideo, 220

L

LeveledCompactionStrategy
 (LCS), 160, 330
 lib, каталог, 73
 LIMIT, ключевое слово, 249
 ListenableFuture, интерфейс, 202
 listen_address, свойство, 182
 listen_interface, свойство, 182
 LIST USERS, 348
 LOCAL_JMX, переменная, 358
 LOCAL_ONE, уровень
 согласованности, 224, 237
 LOCAL_QUORUM, уровень
 согласованности, 224, 237, 297
 LOCAL_SERIAL, уровень
 согласованности, 232
 LocalStrategy, 169, 190
 Log4J, 185
 logback.xml, файл, 72, 257

Logback, библиотека протоколирования, 185, 216, 255
 LoggingRetryPolicy, 210
 LOGIN, команда, 349
 logs, каталог, 77
 LZ4, алгоритм сжатия, 197

M

Management Extensions для Java (MX4J), 263
 manifest.json, файл, 307
 Mapper, класс, 206
 Maven, 74
 max_hint_window_in_ms, свойство, 298
 MBean-объекты, 164
 BlacklistedDirectoriesMBean, 298
 в Cassandra, 267
 краткий обзор, 264
 относящиеся к безопасности, 360
 MBean-объекты, относящиеся к базе данных, 270
 MD5, криптографический алгоритм, 176
 memtable_allocation_type, 325
 memtable_flush_period_in_ms, 326
 memtable_flush_writers, 326
 memtable_heap_space_in_mb, 325
 memtable_offheap_space_in_mb, 325
 MemtablePool, 325
 MessagingService, класс, 166
 Microsoft Azure, 30
 развертывание Cassandra, 369
 MIN и MAX, функции, 248
 Mission Control, 263
 MongoDB, 57
 автоматическое сегментирование, 39
 Murmur3Partitioner, 148, 176
 генерирование начальных маркеров, 182
 избегание горячих узлов, 177

N

native_transport_port, свойство, 183
 Network Time Protocol (NTP), 365

NetworkTopologyStrategy, класс, 148, 179, 190, 296
 Node.js
 драйвер DataStax, 218
 клиент Helenius, 194
 NodeProbe, класс, 278
 nodetool, 72
 выполнение команд из Priam, 313
 задание порога уплотнения, 332
 команда assassinate, 302
 команда bootstrap, 295
 команда bootstrap resume, 295
 команда clean, 191
 команда cleanup, 287
 команда clearsnapshot, 307
 команда compact, 333
 команда compactionhistory, 333
 команда compactionstats, 332
 команда decommission, 300
 команда describecluster, 279
 команда describering, 281
 команда disablebackup, 308
 команда drain, 287, 304
 команда enablebackup, 307
 команда flush, 228, 286
 команда info, 280
 команда join, 331
 команда listsnapshots, 306
 команда loadbalance, 177
 команда move, 294
 команда netstats, 301
 команда proxyhistograms, 316
 команда rebuild_index, 293
 команда refresh, 308
 команда removenode, 302
 команда repair, 191, 288, 298
 флаг -par, 291
 флаг -pr, 291
 флаг -seq, 291
 флаги -st и -et, 292
 команда ring, 173, 186, 281
 команда settraceprobability, 321
 команда snapshot, 306
 команда status, 173, 280, 285
 команда statusbackup, 308
 команда tablehistograms, 317, 332
 команда tablestats, 283, 315, 332

команда tptats, 163, 282, 285
 команда upgradestables, 304
 команды для вручения напоминаний, 329
 мониторинг, 278
 управление кэшами, 325

NoSQL

- восхождение, 40
- типы баз данных, 41

now(), функция, 102

num_tokens, свойство, 147, 181

O

OHCPProvider, класс, 324

OldNetworkTopologyStrategy, 149, 190

ONE, уровень согласованности, 149, 224, 237

OperationMode, 271

ORDER BY, фраза (команда SELECT), 244

OrderPreservingPartitioner, 176

org.apache.cassandra.concurrent, пакет, 278

org.apache.cassandra.db, пакет, 164

org.apache.cassandra.dht, пакет, 148

org.apache.cassandra.internal, 277

org.apache.cassandra.locator, пакет, 145

org.apache.cassandra.metrics, 277

org.apache.cassandra.service.paxos, 157

org.apache.cassandra.transport, пакет, 166

P

PaaS (Platform-as-a-Service), 30

PagingState, 251

PAGING, команда, 249

PasswordAuthenticator, 346

Paxos, алгоритм выработки консенсуса, 156

- таблица system.paxos, 168

Perl, клиент Perlcassa, 194

PermissionsCacheMBean, 278, 360

PHP, драйвер DataStax, 222

PlainTextAuthProvider, 349

Planet Cassandra, сайт, 60

PoolingOptions, 198

Priam, 313

PRIMARY KEY, фраза, 130

PropertyFileSnitch, 178, 299

Rycassa (драйвер для Python), 217

pylib, каталог, 73

Python, 246

- драйвер DataStax, 217
- инструмент csm, 170
- каталог pylib, 73
- клиент Rycassa, 194
- установка, 217

Q

QueryBuilder, класс, 204

- использование для удаления данных, 253

QUORUM, уровень согласованности, 149, 224, 237

R

RackInferringSnitch, 179

RAID, 364

RandomPartitioner, 148, 176

- генерирование начальных маркеров, 182

ReadRepairStage, MBean-объект, 277

Repair Service, служба, 312

ReplicationAwareTokenAllocator, 147

RetryDecision, 209

RetryPolicy, 209

- DowngradingConsistencyRetry-Policy, замечание, 209

RoundRobinPolicy, 208

row_cache_class_name, свойство, 324

rows_per_partition, свойство, 324

rpc_address, свойство, 211

rpc_keepalive, свойство, 183

rpc_port, свойство, 183

rpc_timeout_in_ms, свойство, 236

Ruby, драйвер DataStax, 219

S

SASI-индексы, 112

saved_caches, свойство, 325

Scala, 246

SchemaBuilder, 214

- SchemaChangeListener, интерфейс, 214
 SchemaStatement, 214
 SEDA (многоступенчатая событийно-ориентированная архитектура), 162, 277
 SeedProvider, интерфейс, 175
 SELECT, команда, 88, 95
 запросы по диапазону, 160
 упорядочение и фильтрация, 241
 использование функции TTL(), 98
 построение с помощью QueryBuilder, 205
 SerializingCacheProvider, 324
 server_encryption_options, свойство, 354, 357
 SHOW VERSION, команда, 84
 SimpleSeedProvider, класс, 175
 SimpleSnitch, класс, 178
 SimpleStrategy, класс, 148, 178, 189
 sizeOf(), функция, 133
 SizeTieredCompactionStrategy (STCS), 160, 330, 362
 SNAPPY, алгоритм сжатия, 197
 snapshot_before_compaction, свойство, 307
 Spark, 372
 spark-cassandra-connector, 376
 примеры использования в сочетании с Cassandra, 373
 развертывание Spark вместе с Cassandra, 374
 spark-cassandra-connector, 376
 SpeculativeExecutionPolicy, 210
 speculative_retry, свойство, 252
 SQL (Structured Query Language), 31
 SparkSQL, 379
 и CQL, 90
 SSD-диски, 328, 364
 SSLContext, объект, 356
 SSLFactory, класс, 356
 SSL (Secure Sockets Layer), 353
 шифрование JMX-соединений, 358
 ssl_storage_port, свойство, 182
 sstableloader, 309
 sstablemetadata, 290
 sstableupgrade, скрипт, 304
 SSTableWriter, интерфейс, 229
 SSTable, файлы, 151
 антиуплотнение, 160
 и удаление, 253
 компоненты, 229
 метаданные о состоянии уплотнения, 290
 настройка и оптимизация, 328
 переход на новую версию, 304
 полное уплотнение, 160
 получение информации, 272, 284
 запись на диск, 228
 утилиты для работы, 72, 309
 представление файлов данных, 184
 принудительный сброс данных, 286
 просмотр при выполнении запроса на чтение, 239
 сжатие, 153
 создание резервной копии при сбросе на диск, 308
 уплотнение, 159
 start_native_transport, свойство, 183
 stop-server, команда, 79
 storage_port, 182
 StorageProxyMBean, 272
 StorageProxy, класс, 165
 StorageServiceMBean, 270, 276
 StorageService, класс, 165
 вывод узла из эксплуатации, 300
 StreamManagerMBean, 276
 StreamManager, класс, 166
 stress-build, цель сборки, 75
 sum, функция, 249
 system_auth, пространство ключей, 351
 system.log, файл, 186, 257

T

- Tablesnap, 306
 test, цель сборки, 75
 textAsBlob(), функция, 103
 text, тип данных, 100
 ThreadMXBean, 265
 THREE, уровень согласованности, 149, 224, 237
 Thrift API (нерекомендуемый), 57, 183

cassandra.thrift, файл, 72
устаревшие клиенты, 194
timestamp, тип данных, 101
timeuuid, тип данных, 102, 107
time, тип данных, 101
TLS (Transport Layer Security), 353
TokenAwarePolicy, 208
token-generator, утилита, 181
tools, каталог, 73
TRUNCATE, команда, 89
TrustManagerFactory, класс, 356
TTL(), функция, 98
TWO, уровень согласованности, 149, 224, 237

U

unixTimestampOf(), функция, 102
UnreachableNodes, атрибут, 271
UPDATE, команда
в облегченных транзакциях, 231
фраза USING TIMESTAMP, 97

фраза USING TTL, 98
USING TIMESTAMP, фраза, 98
USING TIMESTAMP, фраза
(команда UPDATE), 97
uuid, тип данных, 102
uuid(), функция, 102

V

varchar, тип данных, 100
varint, тип данных, 100

W

WHERE, фраза
в команде CREATE
MATERIALIZED VIEW, 130
в команде SELECT, 243
добавление в SELECT с помощью
QueryBuilder, 205
writetime(), функция, 98

X

XML, базы данных, 42

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс»
наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: www.alians-kniga.ru.
Оптовые закупки: тел. (499) 782-38-89.
Электронный адрес: books@alians-kniga.ru.

Джефф Карпентер, Эбен Хьюитт

Cassandra. Полное руководство

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод *Слинкин А. А.*

Корректор *Синяева Г. И.*

Верстка *Чаннова А. А.*

Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.

Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 37,5. Тираж 200 экз.

Веб-сайт издательства: www.dmk.ru