

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего
образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

ОСНОВЫ NOSQL СУБД

УЧЕБНОЕ ПОСОБИЕ

Направление подготовки:

09.04.02 «Информационные системы и технологии»

Квалификация выпускника: магистр

Ставрополь, 2016

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ.....	4
РАЗДЕЛ 1. ОСНОВЫ BIG DATA	6
1.1 Развитие вычислительных технологий и стратегии их реализации	7
1.2 Разработка стратегий управления данными	24
1.3 NoSQL и Big Data	36
1.4 Современное состояние Big Data	43
Заключение	50
Контрольные вопросы.....	51
РАЗДЕЛ 2. МОДЕЛИ И КОНЦЕПЦИИ BIG DATA	53
2.1 Технологии Big Data	53
2.2 Модель программирования Big Data.....	58
2.3 Типы данных NoSQL	65
Заключение	72
Контрольные вопросы.....	72
РАЗДЕЛ 3. ОСНОВЫ NOSQL	74
3.1 Архитектура хранилища данных	74
3.2 Архитектура распределенного хранилища HBase	83
3.3 Документо-ориентированное хранилище	87
3.4 Частично согласованные нереляционные базы данных	98
Заключение	102
Контрольные вопросы.....	103
РАЗДЕЛ 4. ФРЕЙМВОРК РАСПРЕДЕЛЕННОГО ПРОГРАММИРОВАНИЯ	104
4.1 Основы Hadoop	104
4.2 Элементы кластера Hadoop.....	109
4.3 Дистрибутивы и экосистема Hadoop	115
4.4 Применение и ограничения Hadoop	120
Заключение	123

Контрольные вопросы.....	124
РАЗДЕЛ 5. YARN	125
5.1 Основы YARN	125
5.2 Компоненты YARN	129
5.3 YARN и MapReduce.....	138
5.4 Применения YARN.....	142
Заключение	153
Контрольные вопросы.....	154
6. ПАТТЕРНЫ MAPREDUCE И BIG DATA	155
6.1 Объединение	155
6.2 Сортировка	165
6.3 Примеры	171
Заключение	172
Контрольные вопросы.....	173
7. MONGODB	174
7.1 Основы MongoDB	174
7.2 Оболочка MongoDB Shell	190
Заключение	199
Контрольные вопросы.....	200
ЗАКЛЮЧЕНИЕ	201
СПИСОК ЛИТЕРАТУРЫ	202
Основная литература	202
Дополнительная литература	202

ВВЕДЕНИЕ

Современные технологии обработки и хранения данных не ограничиваются только реляционными данными. Непрерывное увеличение объемов обрабатываемых данных приводит к поиску новых программных и аппаратных решений для обеспечения высоких характеристик хранилищ информации. Одним из таких подходов является применение специализированных систем управления базами данных (СУБД), существенно отличающихся от ранее используемых – это нереляционные базы данных. СУБД, разрабатываемые в рамках данного подхода, получили название NoSQL СУБД. Несмотря на принципиальные отличия от ранее применяемых, представители технологии NoSQL позволяют, при грамотном применении, обеспечивать существенные преимущества по сравнению с реляционными СУБД. Подход NoSQL затронул не только программную часть информационных систем; существенно пересмотрены принципы построения центров обработки данных для использования в рамках NoSQL.

В рамках развития высокопроизводительных систем обработки данных появились сложные технологии, которые не относятся к СУБД, например технология Hadoop. Данный фреймворк представляет собой целый стек технологий, обеспечивающих слой виртуализации и сервисов для высокопроизводительных программных комплексов. Hadoop рассчитан на применение в распределенной гетерогенной среде, данная технология не предполагает применения какого-либо одного языка программирования или СУБД – это именно уровень в многослойной информационной системе.

Потребность в высокопроизводительных, распределенных и масштабируемых СУБД, вызванная объективными причинами техногенного характера, приводит к неуклонному росту востребованности специалистов по работе с данными, аналитиков в области Big Data на рынках труда практически всех стран. Большинство современных университетов предоставляют возможность подготовки специалистов по направлениям Big Data и NoSQL.

Таким образом, при проектировании высокопроизводительных информационных систем на основе баз данных необходимо учитывать тенденции и современное состояние программных и аппаратных средств обработки данных.

В рамках данного пособия рассматривается комплекс вопросов, направленных на всестороннее изучение процессов, протекающих в области Data Science. Изложение материала пособия построено по принципу «от теории к практике». Первые разделы пособия отражают исторические аспекты NoSQL-подхода; теоретические подходы, лежащие в основе науки о данных. Последние разделы посвящены изложению методов работы с высокопроизводительными системами обработки данных уровня предприятия.

Пособие позволяет учащемуся получить исчерпывающие теоретические представления о таких технологиях как Hadoop, NoSQL, а также обеспечивает магистра достаточной информацией для самостоятельных исследований в области Big Data.

РАЗДЕЛ 1. ОСНОВЫ BIG DATA

На современном этапе развития корпоративных информационных систем (ИС) продолжаются поиски гибких методологий и процессов разработки и проектирования. Подобные технологии сами непрерывно эволюционируют и имеют огромное значение для бизнеса, государственных учреждений. Важнейшим этапом проектирования ИС является создание своеобразной дорожной карты, которая определяет приоритетные этапы реализации проекта, имеющие критическое значение для проекта в целом. Организации, успешно разрабатывающие и проектирующие решения нового поколения в области ИТ, следуют передовым методологиям проектирования, основываясь на собственном опыте внедрения и развертывания ИС. Отдельные методологии проектирования ИС будут рассмотрены в данном разделе. С другой стороны, предприятия, использующие передовые технологии из области Big Data или Internet of Things (IoT) в рамках инициативных проектов, экспериментов или тестовых заданий, очень часто затрудняются обнаружить реальную выгоду от подобных проектов или от непосредственного внедрения новых решений. Многие пользователи не могут выявить устойчивых связей между разрозненными требованиями к эффективности внутри предприятия. В таком случае передовые ИТ-проекты рано или поздно достигают стадии, когда необходимо принять решение о прекращении их финансирования или о неприменимости подобных решений в рамках конкретного предприятия. К сожалению, подобные ситуации встречается на практике, но и они лишены своих положительных моментов.

В данном разделе учебного пособия мы рассмотрим передовые технологии Big Data, которые, в общепринятом случае, включают технологии традиционных хранилищ, основанных на реляционных системах управления базами данных (РСУБД), кластерах Hadoop, базах данных NoSQL и прочих решениях для управления данными.

1.1 Развитие вычислительных технологий и стратегии их реализации

В начале данного раздела рассмотрим каким образом технологии больших данных (Big Data) влились в исторический процесс развития информационных технологий.

Начнем изложение со времени, когда подобные технологии даже не были реализованы в виде проектов. На рис. 1.1 демонстрируется схема, отражающая эволюцию технологий обработки данных.

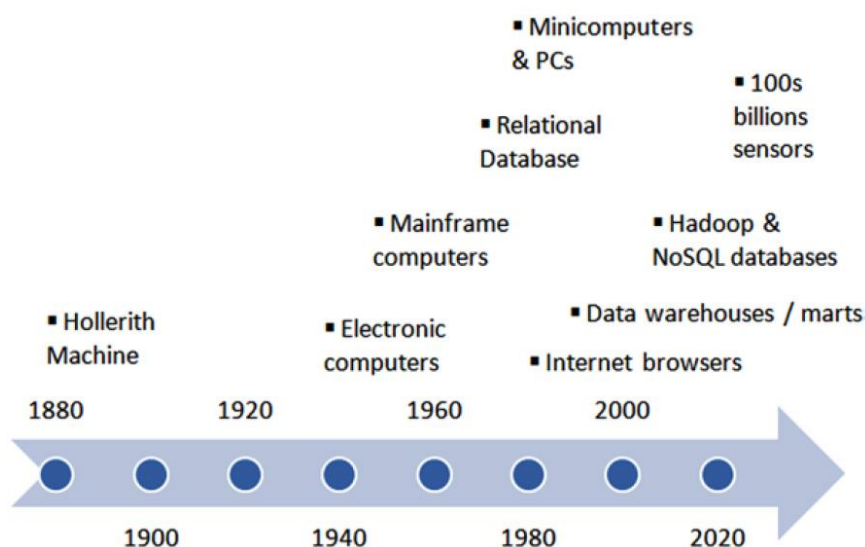


Рисунок 1.1 – Эволюция технологий обработки информации

1.1.1 Краткая история

Существует много мнений о том, каким образом описывать развитие вычислительных технологий. Данное историческое описание начинается в то время, когда вычислительной уже вышли за рамки механических вычислителей. Начнем с рассмотрения решений для обработки данных, ориентированная на предоставление информации в определенном формате. Многие считают, что основной предпосылкой к появлению табличного формата данных, стало широкое распространение оборудования на перфокартах, предложенного Германом Холлери.

Первое применение подобной технологии было реализовано при автоматизации переписи населения в США. Принципы организации данного

процесса были хорошо изучены, когда в 1880 г. Холлери предложил его решение. Многие века, государственные структуры выполняли накопление данных о различных аспектах жизни людей. В процессе такого накопления, стала проявляться структура массива данных, требуемая для хранения и обработки информации и содержащая следующие поля: имя гражданина, пол, адрес, возраст, собственность, место рождения, уровень образования и т.д. Постепенное увеличение ключевых показателей (KPI) с одновременным увеличением численности населения привели к необходимости внедрения автоматизированного подхода к хранению и обработке данных. Перфокарты Холлери обеспечивали подобную автоматизацию.

В 1930 г. данная технология стала популярной по всему миру и широко применялась для автоматизации различных процессов во всех областях.

В 1940 г., с началом Второй Мировой Войны, перед учеными встали сложные проблемы военной направленности, связанные с ускорением вычислений. К подобным проблемам относились шифрование, расчет баллистических траекторий. Необходимость автоматизации подобных решений привела к появлению электронно-вычислительных устройств, состоящих из переключателей, вакуумных трубок и проводников, заключенных в шкафы, которые заполняли комнату или здание. После войны, в продолжении исследований в военной сфере, стали появляться компьютеры для автоматизации коммерческих предприятий в области финансов и др.

Следующие десятилетия были связаны с развитием и распространением современных операционных систем, программного обеспечения, языков программирования (для упрощения и ускорения разработки приложений) и баз данных (для быстрого и простого хранения данных). Баз данных эволюционировали от иерархических к более гибким реляционным, хранящим данные в табличном виде. Таблицы связаны внешним ключом посредством введения дублирующихся столбцов. Структурированный язык запросов (SQL), вскоре стал стандартным средством доступа к реляционной базе данных.

В начале 1970-х годов разработка приложений была сфокусирована на обработке и построении отчетов по часто изменяющимся данным; подобные технологии получили название – оперативная обработка транзакций (OLTP). Разработка программного обеспечения была основана на предположении о необходимости достижения ключевых бизнес-показателей (показателей эффективности) или удовлетворении текущих экономических потребностей предприятия. Несмотря на то, что транзисторы и интегральные микросхемы существенно увеличили эффективность таких систем, стоимость вычислительных систем, мейнфреймов и программного обеспечения оставалась слишком высокой и являлась основным сдерживающим фактором развития IT.

Ситуация изменилась с появлением дешевых мини-ЭВМ, а затем персональных компьютеров, в конце 1970-х начале 1980-х годов. Электронные таблицы и реляционные базы данных сделали доступным более гибкий анализ данных, что привело к появлению систем поддержки принятия решений. С течением времени данные стали более распределенными, появлялось понимание того, что различные подходы к сбору данных привели к противоречивым результатам в области анализа данных. На современном этапе развития технологий обработки данных возникла объективная необходимость в новых подходах к архитектуре информационных систем.

1.1.2 Хранилища данных

Билла Инмона часто называют человеком, который первым предложил технологию «хранилище данных». Он описывал хранилище данных как «предметно-ориентированная, интегрированная, энергонезависимая, и изменяемая во времени коллекция данных, предназначенная для обеспечения поддержки принятия решений». В начале 1990-х, он разработал концепцию корпоративного хранилища данных (КХД). Корпоративное хранилище рассматривалось как централизованное решение для всех исторических данных для компании. КХД представлено моделью данных в третьей

нормальной форме (3НФ), где все атрибуты являются атомарными и содержат уникальные значения. Аналогичные схемы использовались в базах данных OLTP.

Рисунок 1.2 демонстрирует небольшой фрагмент модели в третьей нормальной форме для информационного хранилища компании по продаже авиабилетов. Очевидно, что такая схема позволяет анализировать независимо транзакции отдельного покупателя, статистику мест (seat), статистику сегментов, цен, скидки для постоянных клиентов.

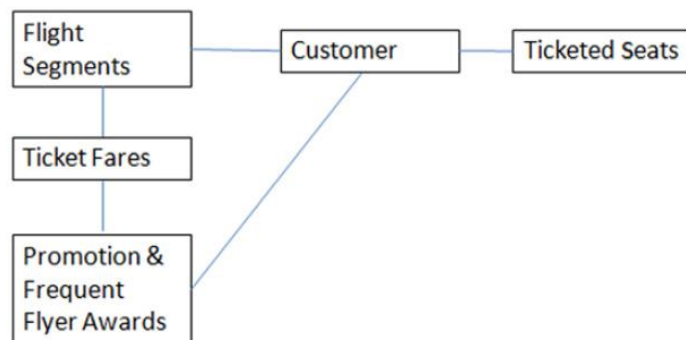


Рисунок 1.2 – Схема в третьей нормальной форме

В КХД загружаются данные из таблиц OLTP-системы через предварительные подсистемы преобразования. Преобразование используются для обеспечения согласованности данных при их извлечении из источников различного типа, а также для обеспечения поддержки бизнес-правил качества данных и стандартов. В первых хранилищах процессы извлечения, преобразования и загрузки данных между источниками и целевым хранилищем, часто выполнялась на регулярной основе (еженедельно, ежемесячно) в пакетном режиме. Но из-за потребности бизнеса видеть изменяющиеся данные в режиме реального времени, наблюдалась тенденция к все более частой загрузке данных в хранилища. Современные технологии обеспечивают загрузку данных в непрерывном режиме, без задержек (наличие задержки может быть связано со сложными алгоритмами преобразования данных). Многие организации обнаружили, что единственный способ уменьшения задержки, вызванной преобразованием данных, – это применение жестких правил к данным в первичных OLTP-системах. Этом обеспечивается

качество и согласованность данных источника, а также уменьшается сложность процедур преобразования.

Изначально большинство практических решений были ориентированы на сбор всех данных, которые могли быть размещены в хранилище, полагая, что бизнес-аналитики могут определить, что с ней делать впоследствии. Такой подход часто приводит к негативным последствиям, когда бизнес-аналитики не могли легко манипулировать необходимыми данными. Многие бизнес-аналитики просто выгружали данные из корпоративных хранилищ в электронные таблицы и уже после обрабатывали их. При этом часто эти выгрузки дополнялись данными из других источников. То есть единого стандарта на обработку данных в хранилищах не существовало. Подобная ситуация привела к тому, что технология корпоративных хранилищ данных во многих источниках характеризовалась как неудачная и требующая переосмысления.

В рамках продолжающихся дебатов об эффективности КХД, Ральф Кимбалл представил подход, который обеспечивал бизнес-аналитикам возможность выполнения специальных запросов в более естественной понятной манере. Предложенная им схема «звезда» содержала таблицу фактов, окруженную таблицами-измерениями. Эта схема широко использовалась в витринах данных.

Рисунок 1.3 демонстрирует эффективность схемы «звезда» при реализации витрины данных на примере авиакомпании. Допустим, требуется определить всех покупателей авиабилетов из США в Мексику в июле 2014 г. Как видно из схемы, транзакции с клиентами задействуют таблицу фактов. Таблицы измерений (место отправления (origination) и назначения(destination)) содержат географическую детализацию информации (континент, страну, штат или провинцию, город, и идентификатор аэропорта). Таблица измерения «Time Dimension» позволяют определить периоды времени (год, месяц, неделя, день, час дня).

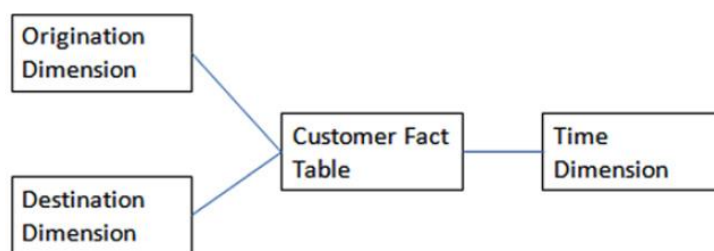


Рисунок 1.3 – Simple star schema

Не все реляционные базы данных со схемой «звезда» изначально обеспечивали оптимальную производительность. Подобные проблемы производительности привели к созданию многомерной аналитической системы обработки транзакций (MOLAP), специально предназначенных для обработки данных, организованных в виде схемы звезда. MOLAP-системы демонстрировали высокую производительность, так как были организованы в виде кубов данных. На рис. 1.4 схематично представлен трехмерный куб данных.

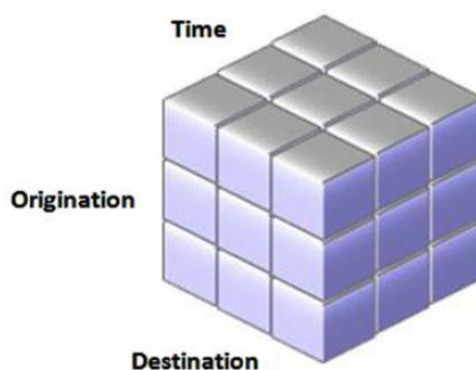


Рисунок 1.4 – Three-dimensional cube representation

Со временем, когда механизмы реляционных БД были существенно оптимизированы, были достигнуты высокие показатели производительности запросов на схемах типа «звезда». Подобные системы получили название реляционной системы аналитической обработки данных (ROLAP).

1.1.3 Зависимые и независимые витрины данных

В середине 90-х годов было множество дебатов об эффективности КХД по сравнению с витринами данных. Аналитики отметили, что в схеме «звезда» легче манипулировать данными (а также возможно разворачивать на ее основе

витрины данных); программисты стали создавать специализированные представления для КХД. Но разработка надстроек над КХД для преодоления противоречий не являлась достаточно гибким и быстрым процессом и подобные технологии не могли удовлетворить растущие потребности рынка.

Также часто возникает вторая проблема. Когда отдельные витрины данных определены и развернуты независимо друг от друга и не следуют правилам определения данных в рамках КХД, может возникнуть вопрос о месте расположения дублирующихся данных. Рисунок 1.5 иллюстрирует проблемы, которые могут возникнуть, когда различные направления бизнеса используют свои собственные независимые витрины данных и извлекают данные непосредственно из базы данных OLTP-источников. На практике, сложность реализации такой системы существенно выше, чем то, что показано на рис. 1.5, так как данные могут передаваться еще и между витринами. Также могут быть добавлены электронные таблицы, выступающие в качестве средства бизнес-аналитики. Организации, которые используют подобные системы, тратят большое количество времени на накладные сопутствующие согласования, на выявление достоверных и своевременных данных.

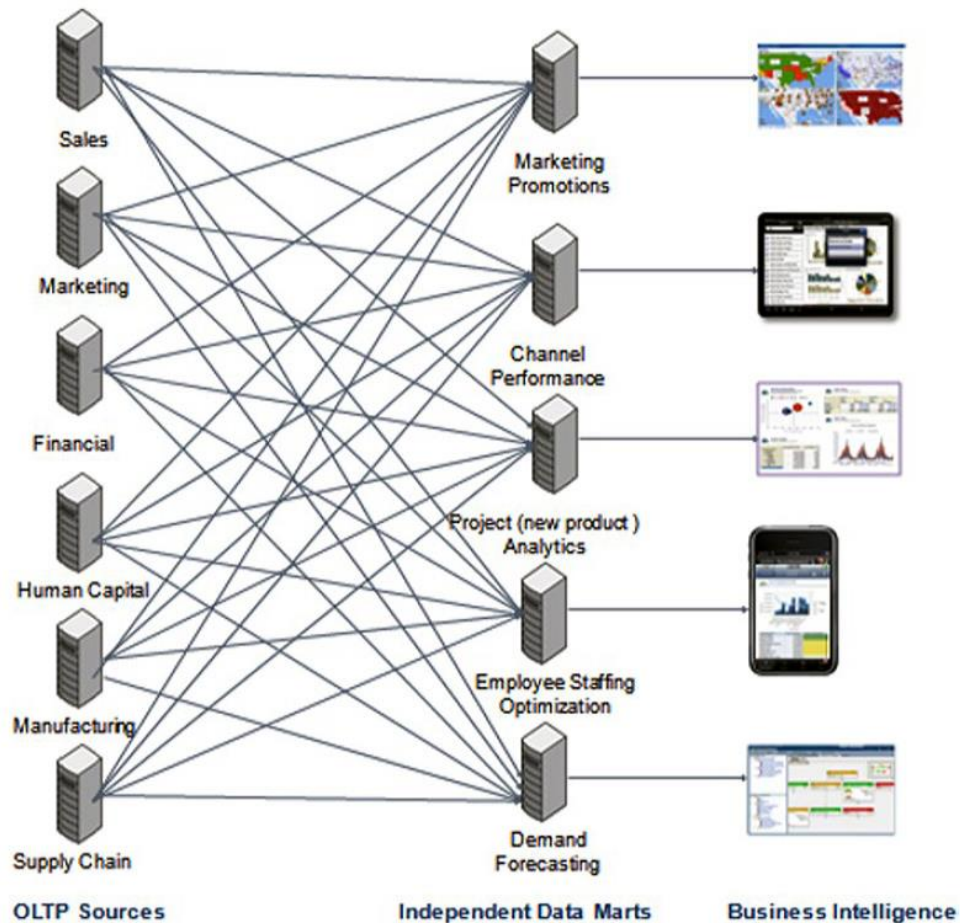


Рисунок 1.5 – Независимые витрины данных с собственными подсистемами загрузки

Существовали веские аргументы, указывающие на эффективность как витрин, так и OLAP-систем. С ростом объемов хранимых данных, аналитики сходились во мнении о необходимости применения комбинированного подхода. КХД реализовывались и расширялись постепенно по мере появления новых источников данных, также постепенно реализовывались витрины данных. Витрины данных реализовывались как системы, зависящие от КХД. КХД оставались хранилищами данных во временном измерении, тогда как данные, отображаемые в витринах извлекаются из EDW. Существовало исключение, когда КХД не использовались в качестве источника – использование источника данных третьей стороны, когда они задействованы в ограниченном сегменте деятельности компании. Эти уникальные данные проецировались только на одну витрину данных. Рисунок 1.6 иллюстрирует схему взаимодействия КХД и зависимых витрин данных.

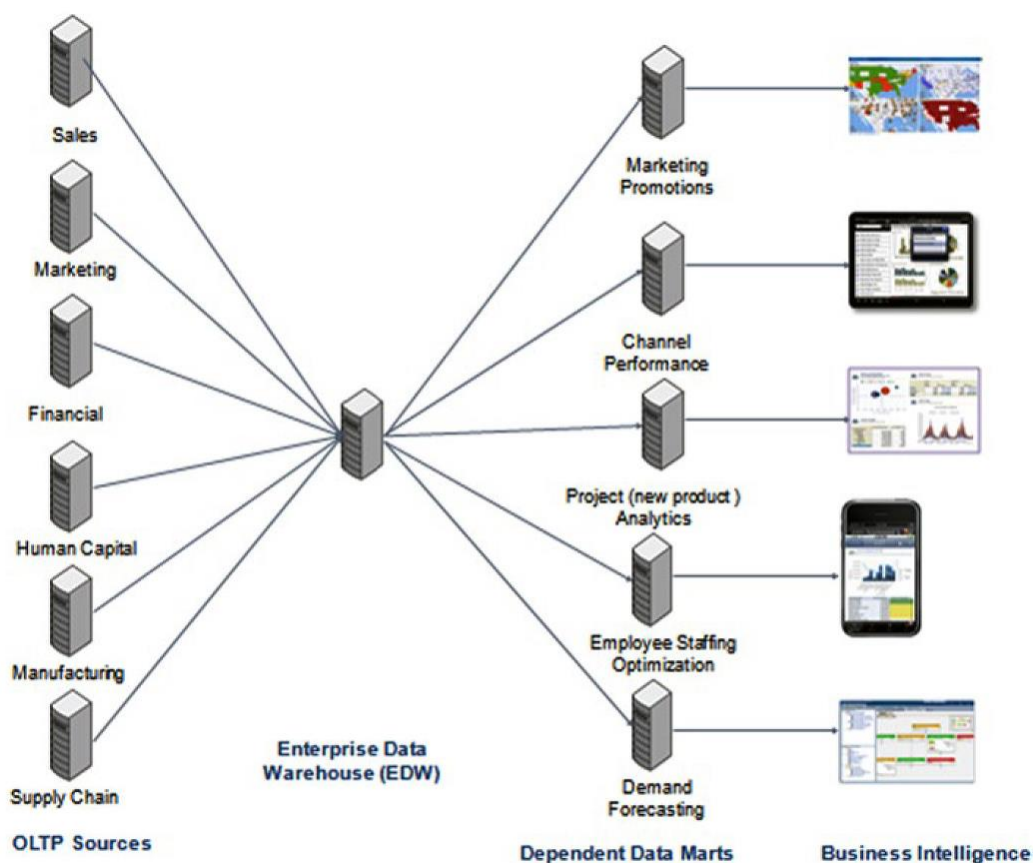


Рисунок 1.6 – Зависимые витрины данных

Такой подход часто требует определения согласованных измерений для установления согласованности между витрины данных. В таком случае согласованными измерениями, можно представить один запрос, который получает данные из нескольких витрин данных (так как измерения представляют одни и те же иерархии в различных витринах данных).

При первом использовании корпоративных хранилищ данных и витрин перед архитекторами ИС стали вопросы обеспечения доступности, отказоустойчивости и безопасности. Удовлетворение растущих требований к централизованным хранилищам приводит к росту новых возможностей и инструментов в дистрибутивах. Об этом свидетельствует растущий интерес к инструментам, направленным на повышение доступности, восстанавливаемости, безопасности в последних версиях программного обеспечения с открытым исходным кодом от различных производителей.

1.1.4 Инкрементальный подход

Первые опыты внедрения хранилищ данных укладывались в концепцию «paralysis by over analysis» (примерно «больше простоя, чем анализа»). Это связано с тем, что разработчики большое внимание уделяли проектированию системы, а не удовлетворению бизнес-требований. Проектирование КХД часто длилось более 12 месяцев и этот процесс происходил без учета требований бизнеса. При проектировании КХД применялся подход, который называется «водопад»; он предполагает фиксацию внимания разработчика на перечне работ (процедур), в то время как время разработки и затрачиваемые ресурсы являются вторичными и изменяемыми.

Схематично, концепция подхода «водопад» представлена на рис. 1.7.

Затянутые во времени планы проекта, задержки и отсутствие внимания к бизнесу часто приводило к отказу от централизованной разработки, проектированию и развертыванию независимых витрин данных, а также созданию витрин, основанных на электронных таблицах. Такое состояние было вызвано необходимостью решения текущих проблем компании.

В свете этих проблем, многие разработчики отказались от подхода «водопад» и перешли к гибкому поэтапному (инкрементальному) подходу. Инкрементальный подход учитывает тесные связи ИТ-проектировщиками и специалистами бизнеса.

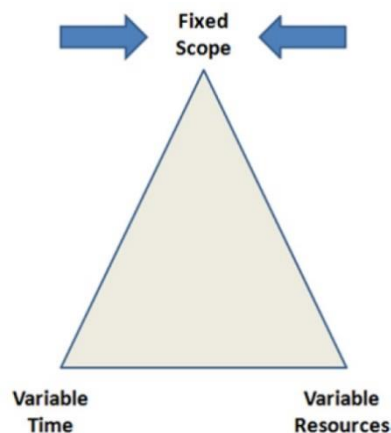


Рисунок 1.7 – Подход «водопад»

Временные рамки 120 дней или менее для реализации решения и оценки его эффективности в достижении бизнес-целей стали обычным явлением во многих организациях. На рис. 1.8 показана схема инкрементального подхода с фиксированными временными рамками и ограниченными доступными ресурсами.

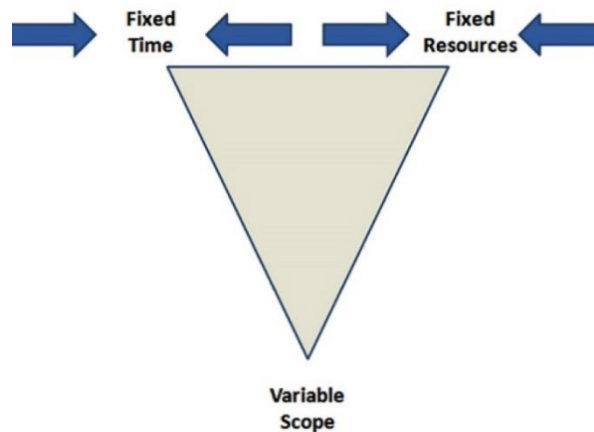


Рисунок 1.8 – Инкрементальный подход

Хотя на рис. 1.8 демонстрируется переменный объем работ, реальный инкрементальный подход предполагает наличие фиксированного показателя, коррелирующего с бизнес-целями компании на каждой итерации процесс проектирования. Таким образом, на практике, применяемая методика представляет собой сбалансированное единство пошагового подхода и подхода «водопад».

В рамках данного подхода, эффективность решения повторно оценивается на каждом шаге жизненного цикла. Определение и оценка КХД и зависимых витрин данных с все более уменьшающимся шагом означает, что ИТ-проект и разработка должны быть скорректированы, так как результаты использования смещаются от бизнес-ожиданий. Рентабельность инвестиций может быть вычислена через регулярные промежутки времени и отражать любые количественные целевые показатели.

Некоторые компании выбирают стратегию при которой бизнес-аналитики присутствуют в штате компании и непрерывно осуществляют мониторинг бизнес-требований. Другие компании обращаются к опыту

создания центров аналитической обработки, объединяющих работу бизнес-аналитиков и IT-архитекторов. Гибкость взаимодействия между этими категориями специалистов имеет более важное значение для успеха бизнеса, чем выбранный подход к проектированию КХД.

Успешные проекты Big Data, обеспечивающие новые решения в области бизнеса, часто разрабатываются с применением инкрементального подхода. В современной IT-индустрии продолжают обсуждаться передовые методик и факторов, влияющих на эффективность бизнеса.

1.1.5 Стратегии реализации

Ранее, реализация отдельного хранилища представляла собой индивидуальный проект, адаптированный к конкретной организации. Так как теоретические аспекты моделей данных не были в достаточной степени проработаны, то, фактически, модель разрабатывалась с нуля. Непредсказуемые характеристики и график нагрузки приводили к неопределенным требованиям к серверному оборудованию. Но со временем были получены результаты всестороннего изучения практического опыта использования КХД и были применены технологии, разработанные с учетом пожеланий разработчиков.

Появились технологии использования предопределенных моделей данных, разработанных на основе общих потребностей аналитиков. Эти модели были доступны для промышленного применения (производство, торговля, банковский сектор) и представлены в виде горизонтальных аналитических решений (расчет финансовых показателей, анализ продаж, маркетинговые исследования и планирование поставок). Такие модели сегодня доступны от поставщиков программного обеспечения и консалтинговых компаний, и отличаются наличием логических связей, а иногда включают в себя физическую схему. Они охватывают ключевые области бизнеса и содержат таблицы и другие элементы данных, необходимые для начала развертывания хранилища данных. Некоторые из них также

содержат с ETL-скрипты, используемые для извлечения данных из популярных ERP- и CRM-источников. Конечно, большинство организаций вынуждены настраивать модели, основываясь на их собственных уникальных бизнес-требованиях. Тем не менее, модели выступают базой для сложных проектов на основе хранилищ данных и развертываются наиболее успешно при использовании поэтапного (инкрементального) подхода.

Ранее уже было отмечено, что конфигурирование серверов и хранилища данных также как и его разработка и проектирование представляет собой самостоятельную сложную задачу.

Учитывая, что рост объемов данных происходит гораздо более быстрыми темпами, чем увеличение объемов доступного дискового пространства, множество систем обработки данных имеют существенное ограничение, если не было уделено достаточное внимание архитектуре используемого решения. В последние годы развертывание платформ, легко встраиваемых в бизнес процессы предприятия, для хранилищ данных и киосков данных стало довольно распространенным явлением. Есть несколько таких решений от производителей реляционных СУБД, которые также предоставляют серверы и системы хранения. Появление Flash-памяти в системах хранения привело к увеличению скоростных характеристик; СУБД были оптимизированы для работы с новыми типами носителей. Позднее, такие факторы как снижение стоимости памяти, появление новых процессоров, способных работать с огромными массивами оперативной памяти, и дальнейшее развитие баз данных в области совершенствования технологий хранения и извлечения данных, привело к увеличению производительности запросов по выборке данных и аналитических запросов. Все это позволило упростить механизмы оптимизации баз данных и снизить сложность задачи проектирования.

Некоторые из ограничений, присущих серверам и системам хранения были преодолены, но неизбежно возникают другие естественные ограничения, вызванные естественными физическими рамками аппаратных систем. При

этом специалисты в области анализа данных непрерывно внедряют новые технологии и, таким образом, раздвигают рамки ограничений.

В настоящее время, количество внедрений Hadoop-кластеров и решений на базе NoSQL не так велико, но непрерывно увеличивается. Также наблюдается рост серверов высокой доступности на базе этих технологий.

Так как ценность подобных решений для бизнеса неоспорима, то все большее значение приобретают такие показатели как время проектирования и реализации, простота обслуживания. Таким образом, ожидается, что востребованность систем NoSQL и Big Data будет неуклонно расти, как в свое время наблюдался рост технологий хранилищ данных.

1.1.6 Инструментарий интеллектуального анализа данных

Каким образом осуществляется доступ к данным и как они используются целиком определяется потребностями и навыками специалистов отдельных направлений бизнеса. Для тех, кому необходимо принимать решения на основе анализа данных, рынок предлагает инструменты, которые они могут использовать могут варьироваться от простых инструментов готовой отчетности до чрезвычайно сложных инструментов анализа данных. Иногда современная инфраструктура также предоставляет «двигатели» (engines) для автоматического принятия решений и выполнения действий, а также инструменты исследования информации. Рисунок 1.9 иллюстрирует зависимость размера сообщества для определенной группы инструментов и методик от сложности этих инструментов.

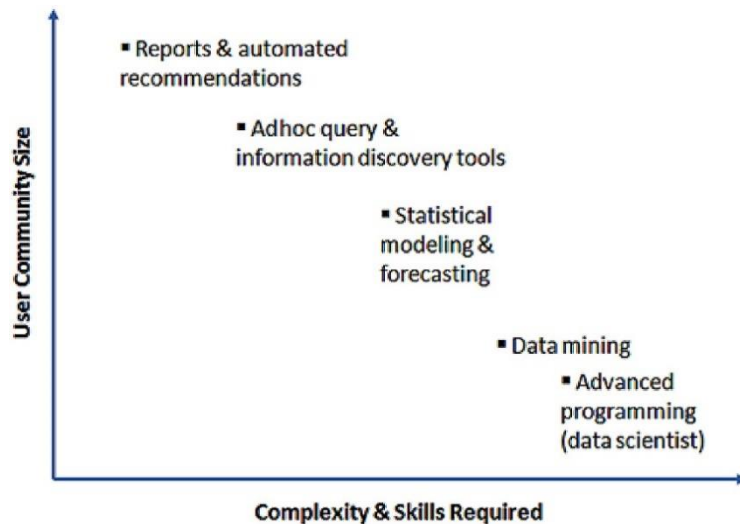


Рисунок 1.9 – Размер сообщества разработчиков и пользователей инструментария анализа данных и сложность инструментария

Самый простой способ донести информацию до бизнес-аналитика – использование предварительно определенных форм отчетов, в которых отображаются ключевые показатели эффективности. Отчеты имеют ограниченную гибкость в информации, которую можно просматривать, но они обеспечивают возможность стать потребителями информации широкому сообществу пользователей. Инструментарий отчетности, используемый разработчиками, генерирует SQL-запросы для запроса данных. Разработчики часто судят о качестве инструментальных средств построения отчетности по ясности представления показателей эффективности, а также по легкости и гибкости процедур генерации, публикации и распространения отчетов. Например, большинство шаблонов отчетов поддерживает публикацию в форматах PDF, DOC, XML и XLS.

Специальные запросы и инструменты анализа обеспечивают большую степень гибкости, так как бизнес-аналитики могут создавать свои собственные «что, если» вопросы путем самостоятельного обращения к таблицам. Разработчики могут создавать метаданные, чтобы перевести имена таблиц в значимые бизнес-ориентированные описания данных. Легкость, с которой пользователи ИС могут обращаться к данным также зависит от лежащей в

основе базы данных схемы. Как было рассмотрено ранее, схема «звезда» с измерениями и иерархическая схема являются удобными для навигации.

Типичные специализированные запросы, инструменты анализа и отчетности можно увидеть при использовании таких решений как Oracle Business Intelligence Foundation Suite, SAP Business Objects, IBM Cognos, MicroStrategy, Tableau, QlikView, Pentaho и Tibco JasperSoft. Безусловно, многие могут сказать, что Microsoft Excel является наиболее популярным инструментом для выполнения такого рода работ, но подавляющее большинство организаций применяет широкий спектр инструментов анализа данных от различных производителей.

Некоторые специалисты бизнес-анализа имеют дело с огромными объемами данных, при этом их задача – выявить скрытые закономерности и/или спрогнозировать будущие значения целевых показателей по этим данным. Число таких специалистов неуклонно растет. Методы анализа варьируются от простых статистических расчетов, изучаемых в университете (среднее значение, стандартное отклонение, и т.д.) до сложных математических моделей, основанных алгоритмах интеллектуального анализа данных.

Статистические функции, с которыми работают бизнес-аналитики, можно разделить на следующие виды:

- простые статистические функции, такие как суммирование, сортировка, упорядочивание, построение гистограмм;
- определение плотности вероятности, дискретной вероятности;
- специальные функции (такие как гамма-функция);
- тестовые функции (такие как хи-квадрат, простая и взвешенная каппа-функция, функция корреляции).

Алгоритмы интеллектуального анализа данных используются в случае, когда необходимо понять, какие переменные имеют решающее значение для точного прогнозирования результатов и в определении прогнозных моделей, которые впоследствии будут использованы для прогнозирования результатов.

Такие модели часто применяются там, где присутствуют сотни переменных, но только десяток или меньше, которые действительно влияют на результат. Алгоритмы интеллектуального анализа данных можно классифицировать следующим образом:

- алгоритмы кластеризации: применяются для того, чтобы определить какому именно выходному подмножеству принадлежат исходные элементы;
- логические модели: применяются для определения причинно-следственных связей (и часто связаны в выведении целых деревьев решений);
- искусственные нейронные сети: обучаемые абстрактные модели, способные выполнять широкий спектр операций анализа после процедуры обучения;
- алгоритмы обнаружения аномалий: используются для выявления исключений из правил, аномальных событий.

Поставщики, осуществляющие поставки ПО для статистического анализа и интеллектуального анализа данных: SAS Institute, IBM SPSS, R (открытый проект для статистических вычислений), and Oracle Advanced Analytics.

В начале этого века, было установлено, что присутствует растущая потребность в изучении больших наборов данных, которые могут включать структурированные, полуструктурированные и потоковые данные. Рассмотренные средства анализа информации позволяют проводить анализ данных, которые не имеют predetermined схемы. Такие инструменты обращаются, как правило, к своим хранилищам информации, например Oracle Endeca Information Discovery, они также могут быть основаны на Hadoop.

Анализируемые данные, как правило, загружаются из OLTP-источников, корпоративных хранилищ данных, NoSQL или Hadoop. Такие решения как Tibco Spotfire, Oracle Big Data Discovery и некоторые другие инструменты могут непосредственно обращаться к Hadoop и использовать данную технологию для информационного поиска. Существует ограниченный круг задач, которые требуют использования систем реального времени.

Например, могут возникнуть задачи информирования посетителя сайта о интересующих его товарах в момент посещения им сайта, или информирование о необходимости замены оборудования из-за наличия прогноза о его скором выходе из строя.

Данные о функционировании веб-сайта обычно анализируются с использованием моделей прогнозирования. Результаты предсказания обычно формируются в виде обновлений системы управления сайтом в реальном времени. Система, затем, предоставляет определенную страницу или сообщение. Чем больше функционирует сайт, тем более качественные предсказания выдает модель. Часто совместно с системой автоматического управления сайтом применяется подсистема отчетов для мониторинга состояния системы. Другие задачи, такие как предсказание отказов оборудования, могут потребовать немедленного реагирования без глубокого анализа.

Системы управления, основанные на правилах, или системы, управляемые событиями, могут быть запрограммированы на автоматическое принятие мер в случае возникновения условного события. Они часто развертываются в IoT-решениях, чтобы выполнить действие, основываясь на показаниях датчиков.

1.2 Разработка стратегий управления данными

Изменения требований бизнеса к рабочей нагрузке провоцируют появление новых технических требований, которые влекут развитие реляционных баз данных и внедрение новых возможностей, нацеленных на удовлетворение этих требований. Несмотря на постоянное совершенствование, были выявлены недостатки реляционной архитектуры, вытекающие из модели данных, характеризуемые высокими накладными расходами при извлечении данных из таблиц. Это в значительной степени способствовало появлению баз данных в рамках подхода NoSQL и Hadoop.

Так совпало, что подобные технологии появились в то время, когда набирают популярность решения с открытым исходным кодом, что, в свою очередь, способствует развитию нереляционных подходов. В модели с открытым исходным кодом, производители и отдельные лица имеют доступ к исходному коду и эти разработчики предоставляют обновления и утилиты, доступные всем. Исходный код для баз данных NoSQL может быть лицензирован Apache Software Foundation и GNU. Лицензии Hadoop могут быть получены от Apache. Поскольку новые функции встроены в новых версиях открытым исходным кодом, производители программного обеспечения могут выбирать, что включать в их собственные дистрибутивы. Хотя дистрибутивы можно скачать бесплатно, производители считают, что они могут в конечном счете стать прибыльным и успешным компаниями, получая доходы за счет подписки (включая поддержку), а также путем предоставления услуг за отдельную плату

1.2.1 Базы данных NoSQL

Появление термины «NoSQL» относится к концу 1990-х годов, он предназначался для описания широкого класса нереляционных СУБД, предназначенных для обеспечения быстрого обновления данных и манипулирования огромными объемами данных, обеспечивая при этом горизонтальную масштабируемость. Отсутствие такого поведения в реляционных СУБД стало проблемой для некоторых интернет-приложений (таких как торговые платформы и социальные сети), где высокая производительность обновления являлась критической.

На ранних стадиях развития базы данных NoSQL не поддерживали язык SQL, оправдывая название этого класса. Но позднее поддержка SQL была в определенной степени реализована во многих NoSQL-базах данных. На ранних стадиях NoSQL-базах отсутствовали свойства, характерные для реляционных СУБД: атомарность, согласованность, изолированность и

долговечность (свойства ACID). Но поддержка подобного поведения не преследуется в NoSQL-решениях, так как влечет высокие накладные расходы и сводит на нет все преимущества. На сегодняшний день во многих NoSQL-базах данных заявлена поддержка некоторых из ACID-свойств. Тем не менее, очевидно, что новые системы не предназначены для использования в качестве замены для реляционных баз данных, OLTP или систем, предполагающих соединения многих типов данных по нескольким измерениям.

Существуют различные типы баз данных NoSQL. К ним относятся следующие:

1. Пары ключ/значение: базы данных, которые оперируют парами «ключ-значение» или «ключ-набор значений», а также характеризующиеся большим количеством легковесных операций, изменяемым числом значений, привязанных к одному ключу.

2. Ориентированные на использование столбцов (семейства столбцов): Базы данных, основанные на парах «ключ-значение» часто представляются в виде двумерного массива, поведение которого близко к классической таблице.

3. Документо-ориентированные: технология близкая к столбцовой архитектуре, но направлена на хранение данных в формате отдельных документов, при этом допускающих произвольную вложенность, произвольную структурную сложность.

4. Графовые: базы данных использующие терминологию узлов и связей.

Свойство горизонтальной масштабируемости баз данных NoSQL обеспечивается за счет механизма шардинга (sharding). Шардинг – это механизм распределения данных по нескольким независимым серверам (или узлам) в кластере. Производительность такого решения зависит от вычислительной мощности отдельного узла, но также определяется и тем, насколько эффективно произведено распределение данных между узлами. Например, если все часто используемые данные помещены на один узел, и вся активность кластера связана с обращением к часто используемым данным, то такое решение является плохо масштабированным. Многие производители баз

данных NoSQL уделяют большое внимание механизмам автоматического шардинга для обеспечения лучшей балансировки нагрузки и упрощения добавления/удаления узлов кластера.

Несмотря на то, что NoSQL-базы не так надежны в части поддержки высокой доступности, они предполагают наличие эффективных механизмов репликации, обеспечивающих функционирование БД в случае отказа узлов кластера. Копии данных, обычно, распределяются по узлам кластера, отличным от узла, на котором содержится первичный блок данных.

Существуют десятки различных баз данных NoSQL различных типов. Наиболее известные: Apache Cassandra, MongoDB, Amazon DynamoDB, Oracle NoSQL Database, IBM Cloudant, Couchbase, Inc., и MarkLogic. По мере того как список возможностей для этих баз данных может быстро измениться, понимание возможности, предоставляемые версией рассматривается для развертывания очень важно. В качестве примера, некоторые добавляют в памяти возможности только в их более поздних версиях.

Так как список особенностей подобных СУБД очень быстро меняется, понимание возможностей, предоставляемых конкретной версией, является важным моментом при развертывании системы. Например, некоторые СУБД предоставляют возможность работы с данным непосредственно в оперативной памяти (in-memory), но только в последних версиях.

1.2.2 Hadoop. История развития, особенности и инструменты

Различные потоковые данные с веб-сайтов все чаще вызывает проблемы для компаний и организаций, стремящихся проанализировать эффективность своих поисковых систем. Такие потоки данных включают в себя встроенные идентификаторы и данные значения, другие вспомогательные символы. Очевидно, что этот тип данных не вписывается в реляционную архитектуру.

Над этой проблемой работал Дугласс Каттинг (Douglass Cutting); его работы привели к разработке новой системы еще в 2002 г, которую он назвал Nutch. В 2003 и 2004 годах, компания Google опубликовала две важные статьи, описывающие файловую систему Google (GFS) и MapReduce. Понятие распределенной файловой системы не было новым в то время, но в документах Google изложено видение того, как решить проблему поиска.

Каттинг понимал важность исследований Google и реализовал изменения своей системы в соответствии со статьями компании. Технология MapReduce позволяла выполнять разделение потоков данных и уменьшать объемы обрабатываемых данных в каждом отдельном потоке. Файловая система GFS обеспечивала масштабирование, и это свойство являлось очень важным, учитывая взрывной рост количества сайтов в сети Интернет.

В 2006 г. к проекту присоединилась компания Yahoo!, технология была переименована, также был выбран символ технологии (слон). Так появился Hadoop. С этого года проект Hadoop стал проектом Apache Software Foundation.

Распределенная файловая система обеспечивает распараллеливание рабочей нагрузки для систем обработки огромных объемов данных. MapReduce обеспечивает необходимую масштабируемость. Когда комбинация этих технологий была представлена, ее описывали как решение проблемы больших объемов данных (Big Data), то есть проблемы, возникающей при обработке данных, характеризуемых большим объемом, высокой скоростью обработки и разнообразными форматами данных. Со временем, понятие Big Data приобрело более широкий смысл – так производители обозначают широкий класс средств для решения различных похожих задач.

На сегодняшний день, кластеры Hadoop считаются идеальным решением для различных типов высоконагруженных систем. Некоторые из этих кластеров в настоящее время используются в высокоскоростных ETL-

системах обработки для обеспечения параллельного преобразования между исходными системами и хранилищами данных. Другие кластеры Hadoop используются в аналитических системах прогнозирования, где используются такие технологии как R, SAS или собственная подсистема машинного обучения на базе Hadoop и библиотека интеллектуального анализа данных Mahout. Специалисты в области анализа данных иногда пишут код (используя Java, Python, Ruby on Rails и другие языки) и встраивают поддержку MapReduce или Mahout в этот код, чтобы выявить закономерности в данных. Многие разработчики организуют доступ к данным в кластере посредством различных интерфейсов SQL, таких как Hive, Impala, или других похожих инструментов.

Apache Software Foundation обеспечивает площадку для развития и создания инструментов Hadoop, которые реализуются в виде отдельных проектов. Как только появляется новый релиз, результаты нового проекта могут появиться в дистрибутивах других производителей, таких как Cloudera, Hortonworks, IBM, MapR и Pivotal. Состояние проекта Apache Hadoop отображается на сайте apache.org.

Некоторые базовые технологии из области управления данными:

1. HDFS – the Hadoop Distributed File System. Распределенная файловая система Hadoop.
2. Parquet – сжатый формат хранения данных, основанный на столбца, для системы Hadoop.
3. Sentry – система управления ролями, правами доступа, авторизацией, доступом к данным и метаданным, хранящимся в кластере Hadoop.
4. Spark – система in-memory обработки данных.
5. YARN – фреймворк для планирования и управления вычислительными задачами и ресурсами кластера Hadoop.
6. Zookeeper – сервис для управления взаимодействием распределенных приложений.

Важные инструменты для передачи и хранения данных в Hadoop:

1. Flume – сервис для сбора и агрегирования потоковых данных, включая логи и данные о событиях HDFS.
2. Kafka – платформа для обеспечения механизма публикации/подписки (Pub/sub) для обработки событий передачи данных в режиме реального времени.
3. Sqoop – инструмент для передачи данных между Hadoop и базами данных.

Hadoop включает следующие программные инструменты, API и утилиты:

1. Hive – инструмент для предоставления возможности выборки данных в SQL-стиле из хранилища Hadoop.
2. MapReduce – это парадигма программирования, применяемая в ранних версиях Hadoop. Предполагает выполнение этапа «map» (фильтрация и сортировка) и, затем, «reduce» (агрегирующие операции) над данными, распределенными между узлами кластера.
3. Oozie – планировщик заданий, используемый при управлении задачами Hadoop.
4. Pig – фреймворк для параллельной обработки данных и язык управления потоками данных.
5. Spark GraphX – API для обеспечения просмотра данных в виде графов и коллекций, обеспечивающий преобразование и слияние графов.
6. Spark MLlib – библиотека машинного обучения, реализованная для Spark.
7. Spark SQL – API для обеспечения выполнения запросов в Spark в контексте Hive.
8. Spark Streaming – специализированный API для создания процессов в Spark.
9. Solr – платформа полнотекстового индексирования и поиска.

Разработчики приложений Hadoop иногда используют возможности, предоставляемые базами данных NoSQL, которые являются частью проектируемого ими решения. HBase представляет собой базу данных NoSQL на основе столбцов, разворачиваемую на HDFS и предоставляющую доступ на чтение и запись. Подобная БД полезна при обработке больших объемов разряженных данных. В дополнение к поддержке специализированных запросов (ad hoc), HBase часто используется для получения обобщающих сводок данных.

При распределении кластера Hadoop на серверах и системах хранения требуется назначить имена узлов: узлов данных и узлов, которые будут предоставлять различные сервисы. Корректное развертывание служб кластера позволяет избежать многих проблем, например, отказа кластера Hadoop. Данные, как правило, защищены тройной репликацией, чтобы гарантировать, их доступность в случае сбоя узла.

До сих пор ведутся дебаты о целесообразности применения кластера Hadoop в качестве компонента информационной системы вместо хранилища данных на основе реляционной СУБД. В таблице 1.1 обобщена информация о достоинствах и недостатках каждого вида систем.

Таблица 1.1 –Различия между Hadoop и хранилищами данных (реляционными базами данных)

Характеристика	Hadoop	Хранилище данных
Значимость данных	Данные, как правило, смешанного качества и значимости – наиболее важный объем	Данные только высокого качества – этот параметр имеет решающее значение
Схема	Чаще всего используется распределённая файловая система	Чаще всего 3НФ и гибридная схема «звезда»
Типичная загрузка	Анализ информации, прогнозирование, ETL-обработка	Отчетность, специализированные запросы, OLAP

Источник данных	Различные типы данных от структурированных до потоковых	Структурированные типы данных
Доступность	Репликация данных по узлам кластера	Гарантированная целостность
Защищенность	Авторизация, шифрование, списки контроля доступа	Также как в Hadoop, но с более детализированным контролем доступа
Масштабируемость	Распределение по сотням узлов кластера, сотни Петабайт данных	Распределение по нескольким узлам, обычно сотни Терабайт данных или несколько Петабайт

1.2.3 Интернет вещей

В последнее десятилетие растет востребованность систем отчетности и систем анализа для данных, собранных с датчиков и управляющих устройств различного назначения. Дебаты о пользе таких решений и, одновременно, тестирование подобных систем, начались с 1980-х годов в рамках экспериментальных проектов по разработке автоматизированных устройств, подключенных к Интернет. В начале XXI века появилось множество теорий об особом виде межмашинного взаимодействия (machine-to-machine, M2M), а также о новых возможностях, ассоциированных с новым видом технологий. Появилось множество шуток типа «заговора кухонной техники». Тем не менее, вскоре стало очевидно, что последствия развития новой технологии гораздо более значимы, чем просто появление множества новых устройств; появился новый термин для описания сложившегося положения – «Интернет вещей» (Internet of Things, IoT).

Следует отметить, что еще до появления общепринятой терминологии, многие производители компьютерных устройств закладывали возможность получения данных об окружающей среде в своих продуктах, предусмотрев в их конструкции специализированные датчики и контроллеры.

Производителями также предлагались программные системы для сбора данных с датчиков. Полезность подобных устройств в бизнесе, в прошлом десятилетии, не была оценена по достоинству, а программное обеспечение для анализа данных IoT не было достаточно эффективным. Таким образом, даже в случае массового получения данных через специализированные датчики, они не анализировались, и очень часто просто игнорировались.

Но так как проектирование IT-продукции требует больших временных затрат, а сами высокотехнологичные товары обладают коротким сроком жизни («моральное старение»), то инженеры закладывали в устройства датчики и дополнительные возможности на перспективу. То есть разработчики предполагали, что рано или поздно IoT-технологии будут оценены бизнесом. Было очевидно, что данные могут помочь при анализе качества продуктов и услуг, предсказании отказов, автоматическом подключении услуг, при анализе параметров окружающей среды и автоматическом определении оптимальных параметров энергосбережения, и еще сотни применений.

Очень часто спорят о необходимости перехода на автоматические системы управления большими данными: просят привести аргументы необходимости перехода к Hadoop и технологиям Big Data. Области, которые предполагают использование датчиков и интеллектуальных контроллеров в реальных сферах бизнеса и производства, не могут обойтись без подобных систем. Наличие большого потока данных от датчиков приводит к необходимости высокопроизводительного анализа данных, получаемых с IoT-устройств.

Растущий спрос на датчики привел к интенсификации исследований в данной области и, в результате, к миниатюризации и снижению стоимости устройств. Миниатюризация связана со снижением энергопотребления. В результате этого процесса, миллиарды датчиков функционируют сегодня по всему миру, и это число, как ожидается, в ближайшее время вырастет до сотни

миллиардов устройств к 2020 г. Такая популярность подстегивает исследования в области IoT.

Параллельно с технологическим развитием, наблюдается процесс развития программного обеспечения для обработки данных в сфере IoT. Базы данных NoSQL и системы Hadoop уже повсеместно использовались в качестве ядра веб-сайтов, социальных сетей и пр., но оказалось, что эти технологии максимально эффективно удовлетворяют требованиям обработки данных от датчиков и контроллеров.

Сфера применения IoT: анализ работы двигателей, мобильные устройства, мониторинг здоровья, отслеживание почтовых перевозок и сотни других. На рис. 1.10 схематично показаны ключевые компоненты IoT.

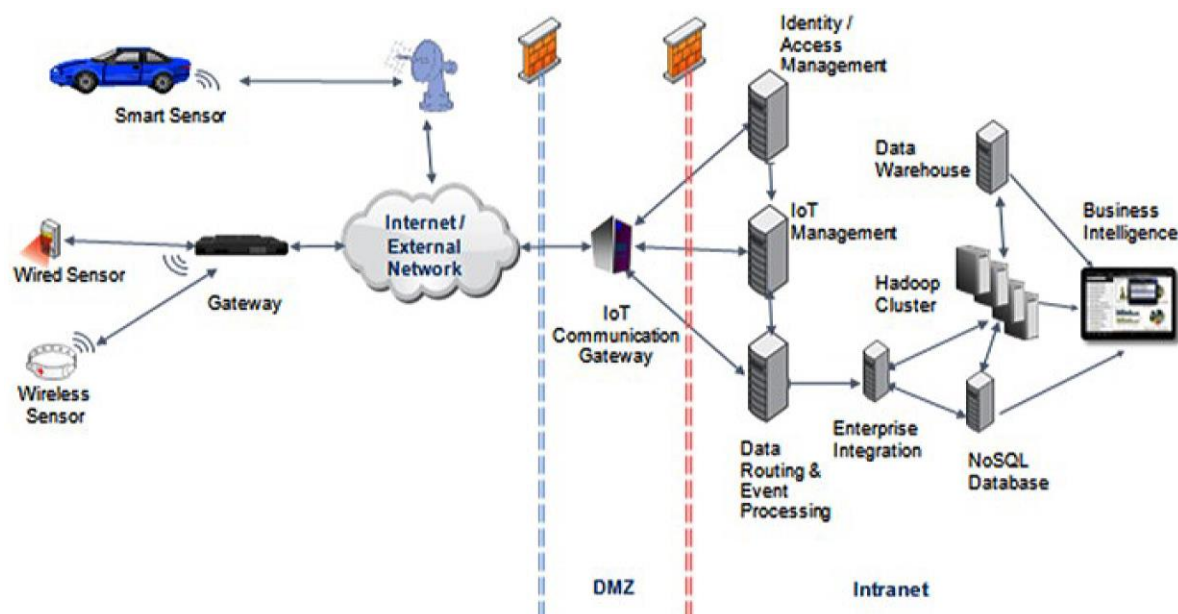


Рисунок 1.10 – Компоненты IoT

До настоящего времени в рамках пособия рассматривалось использование систем управления данными в качестве компонента для интеллектуального анализа для бизнеса. Интернет вещей вводит дополнительные требования для обеспечения маршрутизации данных, обработки событий, управления ресурсами и управления программным обеспечением для датчиков. Требуются новые технологии и средства идентификации, а также специализированные коммуникационные каналы для связи датчиков и устройств корпоративной и глобальной сетей. Несмотря на обилие проблем, самой сложной задачей является отсутствие единых стандартов в области IoT и компаний их разрабатывающих и поддерживающих. Среди существующих стандартов, применяемых в сфере IoT, можно выделить: Open Internet Consortium (архитектура взаимодействия end-to-end для IoT), IETF (описывает взаимодействие и форматы передачи), AllSeen Alliance (для маршрутизации передачи данных между устройствами), IPSO Alliance (для представления данных), Open Mobile Alliance (для управления устройствами и подключения датчиков), и Thread Group (для сетей и систем «умный дом»). На сегодняшний день Industrial Internet Consortium предпринимает попытки стандартизировать решения IoT для промышленного применения.

Компании, занимающиеся анализом данных в области IoT, предлагают свои собственные решения. Некоторые из них, проектируют и разрабатывают комплексные решения. Другие – проектируют и публикуют программное обеспечение для интеллектуального взаимодействия с сенсорными устройствами (обычно используется Java), выполняют поддержку ПО, а в итоге фокусируют свои усилия на сборе и анализе данных. Но существуют участники рынка, которые работают только в сфере анализа данных, получая их от компаний, которые занимаются передачей и непосредственным получением данных с датчиков. Материал пособия охватывает все три описанных сценария использования баз данных NoSQL и систем Hadoop для Интернета вещей. Но в пособии не рассматриваются технологии,

свойственные только для сферы IoT, затрагивающие такие аспекты как: передача сообщений, безопасность, взаимодействие и питание датчиков.

1.3 NoSQL и Big Data

Термин «NoSQL» представляет собой комбинацию слов («No» и «SQL»), что указывает на противопоставление данного подхода технологии SQL. Но на самом деле более точным термином был бы NoRDMS или NoRelational, но из-за простого произношения был выбран именно такой вариант.

Со временем, возникали предложения об использовании NonRel в качестве альтернативы NoSQL, или версии о расшифровке аббревиатуры NoSQL как «не только SQL». Каким бы ни был буквальный смысл, NoSQL используется сегодня в качестве обобщающего термина для всех баз данных и хранилищ данных, которые не реализуют популярные и хорошо разработанные принципы построения РСУБД; они часто связаны с большими наборами данных, а также с веб-технологиями. Все вышесказанное означает, что NoSQL – это не единственный продукт и не единственная технология. Это целый класс продуктов и, одновременно, семейство разнообразных, а иногда и связанных между собой, концепций хранения и манипулирования данными.

Big Data (дословно «Большие данные») – это термин, используемый для описания нашей способности анализировать постоянно растущие объемы данных в мире. Как называть данную сферу деятельности – большие данные, анализ данных, бизнес-аналитика, – не имеет существенного значения. Что действительно имеет значение, так это, что в настоящее время существуют технологии сбора и анализа данных такого уровня, который был не доступен еще несколько лет назад. Big Data привносит инновационные концепции в бизнес, промышленность, научные исследования и многие другие сферы жизни.

1.3.1 История возникновения нереляционных подходов

Нереляционные базы данных не являются новой технологией. Подобные системы хранения появились практически одновременно с первыми вычислительными машинами.

Нереляционные базы данных получили широкое распространение во времена мэйнфреймов и существовали в рамках специализированных проектов для узких практических областей, например в системах хранения исторических данных о авторизации и аутентификации. Тем не менее, нереляционные хранилища, появившиеся в рамках NoSQL-подхода, представляют собой реинкарнацию, которая учитывает изменившиеся условия IT-индустрии и широкое распространение масштабируемых Интернет-приложений. Эти хранилища, большей частью, ориентируются на распределенные и параллельные вычисления.

Начиная с Inktomi, которая может рассматриваться как первая настоящая поисковая система, и заканчивая Google, становится очевидным, что широко распространенные РСУБД имеют существенные недостатки при обработке больших объемов данных. Эти проблемы касаются производительности, эффективности распараллеливания, масштабирования и стоимости вычислений.

Сложности применения РСУБД в масштабируемых веб-системах обработки данных не являются уникальными для данного класса программ. РСУБД предполагает наличие четкой структуры данных. Предполагается, что данные плотные и однотипные по своей природе.

РСУБД разрабатываются из расчета, что свойства данных известны заранее и взаимосвязи между элементами данных заранее известны и устойчивы. Также предполагается, что индексы могут быть последовательно определены на наборах данных, а также могут быть эффективно использованы для быстрой обработки запросов. К сожалению, эффективность применения РСУБД резко снижается, как только эти предположения оказываются не верными. При попытке адаптировать РСУБД и устранить указанные

противоречия, фактически, получаются СУБД, построенные по принципам NoSQL-подхода.

Гибкость NoSQL-решений имеет свою цену. NoSQL частично устраняет проблемы, присущие РСУБД, делает возможной работу с большими объемами разряженных данных, но, одновременно, исключает возможность использование транзакций, механизмов индексирования. Большинство производителей NoSQL-решений не были реализованы средства поддержки языка SQL, и попытки реализовать такую поддержку предпринимаются до сих пор.

Компания Google, в последние несколько лет, существенно продвинулась в разработке поискового сервиса и прочих приложений, включая Google Maps, Google Earth, Gmail, Google Finance и Google Apps. Подход Google заключается в раздельном решении проблем на различных уровня стека семейства приложений. Google нацелена на построение масштабируемой архитектуры для параллельной обработки огромного объема данных. При этом компания разработала целый набор технологий, включая распределенную файловую систему, хранилище на основе семейства столбцов, распределенную систему координации и среду параллельного выполнения программ на основе парадигмы MapReduce. Компанией Google были опубликованы работы, разъясняющие принципы функционирования ключевых технологий. Наиболее важными из них являются [6-9].

Данные публикации вызвали большой интерес в сообществе open-source-разработчиков. Так разработчики первой свободно распространяемой поисковой системы Lucene, развили некоторые идеи Google в своей версии сервиса. Впоследствии команда Lucene работала в Yahoo!, где с помощью других разработчиков, они создали «параллельную вселенную», которая копировала все элементы стека распределенных вычислений от Google. Эта свободно распространяемая технология – Hadoop, вместе со своими сопутствующими продуктами, сервисами, фреймворками и библиотеками [10].

Не будем подробно рассматривать историю развития Hadoop, но следует отметить, что первая версия продукта реализовывала принципы NoSQL. Не важно кто придумал NoSQL и когда, но важно, что появление Hadoop стало причиной взрывного развития данного направления. Одновременно, успешные проекты Google способствовали повсеместному внедрению нового поколения систем распределенных вычислений, Hadoop-систем и NoSQL-решений.

Через год после публикаций Google, явившихся катализатором развития в области параллельной масштабируемой обработки данных, компания Amazon представила результат свои успешные наработки. В 2007 году компанией Amazon была представлена идея построения распределенной системы высокой доступности Dynamo.

После успешной апробации NoSQL-технологии гигантами IT-индустрии – Google и Amazon, – стали появляться новые продукты в данном сегменте. Многие разработчики стали использовать новые подходы в различных областях, на различных предприятиях: от стартапов до крупных корпораций. Технологии NoSQL стали востребованными. Менее чем за 5 лет, NoSQL и связанные с ними концепции управления большими объемами данных стали популярными и успешные варианты использования были продемонстрированы многими известными компаниями, включая Facebook, Netflix, Yahoo, eBay, Hulu, IBM и др. Многие из этих компаний также способствуют распространению их собственных расширений и свободно распространяемых продуктов в мире.

1.3.2 Базовые принципы Big Data

Рассмотрим важный вопрос: когда можно говорить, что данные являются «большими данными»? Ответ на этот вопрос зависит от того, кого вы спрашиваете и когда вы задаете этот вопрос. В настоящее время, любой набор данных размером несколько терабайт классифицируется как «большие данные». Обычно это размер, при котором набор данных достаточно большой,

чтобы возникла необходимость распределить его по нескольким узлам. Это также размер, при котором традиционные методы РСУБД демонстрируют снижение эффективности обработки.

Байт – это 8 бит. В системе СИ увеличение порядка на 3 (в 1000 раз) добавляет к единице измерения определенную приставку:

Килобайт (КБ), Kilobyte (kB) – 10^3 байт;

Мегабайт (МБ), Megabyte (MB) – 10^6 байт;

Гигабайт (ГБ), Gigabyte (GB) – 10^9 байт;

Терабайт (ТБ), Terabyte (TB) – 10^{12} байт;

Петабайт (ПБ), Petabyte (PB) – 10^{15} байт;

Эксабайт (ЭБ), Exabyte (EB) – 10^{18} байт;

Зеттабайт (ЗБ), Zettabyte (ZB) – 10^{21} байт;

Йотабайт (ИБ), Yottabyte (YB) – 10^{24} байт.

В традиционной двоичной интерпретации увеличение разрядности происходит в число раз, кратное 2, то есть умножение осуществляется на 2^{10} (или 1 024), а не на 10^3 (или 1 000). Для предотвращения двусмысленности, используется альтернативная система единиц измерения:

Кибибайт (КиБ), Kibibyte (KiB) – 2^{10} байт;

Мебибайт (МиБ), Mebibyte (MiB) – 2^{20} байт;

Гибибайт (ГиБ), Gibibyte (GiB) – 2^{30} байт;

Тебибайт (ТиБ), Tebibyte (TiB) – 2^{40} байт;

Пебибайт (ПиБ), Pebibyte (PiB) – 2^{50} байт;

Эксбибайт (ЭиБ), Exbibyte (EiB) – 2^{60} байт;

Зебибайт (ЗиБ), Zebibyte (ZiB) – 2^{70} байт;

Йобибайт (ЙиБ), Yobibyte (YiB) – 2^{80} байт.

Несколько лет назад, Терабайт данных для персонального компьютера являлся достаточно большим объемом. Сегодня, подобный объем жесткого диска для хранения данных и данных для восстановления общедоступен. Ожидается, что в ближайшие несколько лет объем жесткого диска персонального компьютера увеличится до нескольких терабайт. Сейчас мы

живем в эпоху взрывного роста объемов обрабатываемых данных. Цифровые камеры, блоги, социальные сети, твиты, электронные документы, музыка и видео – объем различных типов данных растет быстрыми темпами. Каждый человек потребляет и способствует генерации данных одновременно.

Трудно оценить истинные размеры накопленных или размер данных в сети Интернет, но некоторые исследования, приблизительные оценки показывают, что это очень большой объем, который превышает зетабайт.

Так как объем данных увеличивается, а также появляются все новые разнообразные источники данных, то вскоре человечеству предстоит столкнуться с более серьезными проблемами в данной сфере:

1. Сложная реализация эффективного хранения и организации доступа к большим объемам данных. Дополнительные требования отказоустойчивости и резервного копирования еще больше усложняет ситуацию.

2. Манипулирование большими наборами данных требует выполнения множества параллельных процессов. Реализация восстановления в случае возникновения сбоев в процессе такого параллельного выполнения в достаточно короткий период времени, является сложной задачей.

3. Одной из сложнейших проблем в области управления данных является управление данными в условиях постоянно меняющейся схемы и метаданных, работа со слабоструктурированными и неструктурированными данными, генерируемыми различными источниками.

Therefore, the ways and means of storing and retrieving large amounts of data need newer approaches beyond our current methods. NoSQL and related big-data solutions are a first step forward in that direction.

Одновременно с ростом объемов данных, происходит развитие аппаратных технологий информационных систем в сфере хранения данных. Обычные характеристики современного жесткого диска: объем 1 ТБ, скорость доступа к данным 300 Мбит/с, скорость вращения 7200 оборотов в минуту. С такими параметрами для считывания всего содержимого диска (1 ТБ)

занимает около 55 минут (очевидно, что при увеличении объема общее время считывания только вырастет). Кроме того, следует учитывать, что скорость 300 Мбит/с является пиковой, текущая скорость считывания может уменьшаться в реальных устройствах и составлять до 65% от пиковой – это связано с конструктивными особенностями HDD. Твердотельные накопители (SSD) являются альтернативой для вращающихся носителей. В SSD используются микросхемы, в отличие от электромеханических вращающихся дисков HDD. Такие диски сохраняют данные в энергозависимой оперативной памяти.

Ожидается, что SSD-накопители будут обеспечивать более высокую скорость доступа и улучшенные характеристики для операций ввода-вывода (Input/Output operations Per Second, IOPS) по сравнению с традиционными HDD. В конце 2009 и начале 2010 года, компания Toshiba анонсировала SSD-накопители, которые могут обеспечить скорость доступа более Гбит/с. Но SSD обладают и недостатками, один из них – высокая стоимость. Учитывая, что скорость доступа к диску является ограничивающим фактором, становится очевидным направление развития систем хранения: добавление в систему узлов хранения с относительно небольшим объемом, а не хранение данных на едином огромном устройстве. Таким образом, параллельно с проблемой роста объемов хранимых данных, сосуществует проблема масштабирования.

1.3.3 Масштабируемость

Масштабируемость – это способность системы увеличивать пропускную способность в случае увеличения объемов ресурсов и увеличения нагрузки. Масштабируемость может быть достигнута либо путем предоставления больших и мощных ресурсов (прежде всего аппаратных) для удовлетворения дополнительных требований, или, в альтернативном варианте, подобное поведение может быть достигнуто за счет построения кластера на базе обычных компьютеров (кластер при этом функционирует как единая ЭВМ).

Первый вариант реализации масштабирования – использование дорогих, высокопроизводительных компьютеров, – известен как вертикальное масштабирование. Использование супер-ЭВМ с многоядерными процессорами и большими дисковыми массивами, подключенными напрямую, – типичный вариант вертикального масштабирования. Такие варианты являются дорогими. Альтернатива вертикальной масштабируемости – горизонтальная масштабируемость. Горизонтальная масштабируемость предполагает объединение недорогих компьютеров в кластер, который масштабируется по мере увеличения нагрузки за счет добавления новых ЭВМ.

С появлением Big Data и необходимости параллельной обработки при манипулировании большими объемами данных привело к повсеместному внедрению горизонтально масштабируемых систем. Некоторые из этих подобных систем в компаниях Google, Amazon, Facebook, eBay и Yahoo! задействуют большое количество серверов (число которых иногда переваливает за тысячу или сотни тысяч).

Обработка данных, распределенных по узлам кластера, осуществляется параллельно. При такой реализации, модель MapReduce, возможно, обеспечивает один из наилучших подходов для обработки больших данных в горизонтальных кластерных системах.

1.4 Современное состояние Big Data

Понятие «большие данные» (Big Data) – это не просто указание на постоянно увеличивающиеся размеры наборов данных, доступные для анализа. Этот термин указывает также на высокую скорость генерации данных, их разнородность.

Величина объемов данных – это только одно измерение Big Data. На сегодняшний день человечество генерирует объем данных не доступный для прошлых веков. Например, если взять объем данных, сгенерированный с начала истории человечества и до 2000 г, то мы получим размер,

генерируемый человечеством каждый несколько минут. Фактически, более 90% данных в мире, были созданы в прошедшие несколько лет.

Скорость обработки – это другое важное измерение Big Data. Скорость, с которой человек производит новые данные и обменивается ими огромна. Каждую минуту мы посылаем более 200 миллионов электронных писем, почти 2 миллиона кликов в Facebook, отправляем около 300 000 твитов и загружаем почти 200 000 фотографий на Facebook, 100 часов видео на Youtube. Одновременно, следует учитывать более 3,5 миллионов поисковых запросов только на Google.com, все данные генерируемые сенсорами, электронные финансовые транзакции, данные мобильных сетей и т.д.

Структура данных – третья составляющая Big Data. В прошлом в основном работали с структурированными данными: определяли состав таблиц и следили за целостностью данных. Менее структурированные данные, такие как текстовые файлы, фотографии, видео-контент и многое другое, были в значительной степени игнорированы (не укладывались в рамки реляционной концепции). Сегодня есть возможность использовать и анализировать разнообразные данные, в том числе рукописный текст, речь, включая голосовые интонации, а также биометрические данные, фотографий и видео контент.

1.4.1 Правила 5V

Для описания феномена Big Data, часто используют «правило пяти V» (5V's Rule): объем (volume), скорость (velocity), разнородность (variety), достоверность (veracity) и значимость (value).

1. Объем (volume) указывает на огромные массивы данных, генерируемые каждую секунду. Просто учитываются: все электронные письма, сообщения в Twitter, фотографии, видео клипы, данные датчиков и т.д. Сейчас счет идет не на терабайты, а на эксабайты и йотобайты. Только на Facebook мы отправляем 10 млрд сообщений в день, нажимаем кнопку «Like» 4,5 миллиарда раз и загружаем 350 млн. новых фотографий каждый день.

Подобная ситуация приводит к тому, что наборы данных слишком велики, чтобы хранить и анализировать их, используя традиционные технологии баз данных. С помощью технологии Big Data, возможно хранить и использовать эти наборы данных с помощью распределенных систем (горизонтально масштабированных и оснащенных специализированным программным обеспечением).

2. Скорость (velocity) указывает на скорость создания и передачи данных. Например, сообщения в социальных сетях распространяются в считанные секунды, с огромной скоростью происходит выявление фактов мошенничества с банковскими картами, миллисекунды занимает принятие решений в торговых системах. Технологии больших данных позволяют нам анализировать данные в процессе их формирования без непосредственной записи их в базу данных.

3. Разнородность (variety) указывает на разнотипность данных, которые могут использоваться. В прошлом обработка была сфокусирована на структурированных данных, которые просто укладывались в концепцию реляционных баз данных. Это такие данные как финансовые показатели (объемы продаж по группам продукции или регионе). Но 80% данных в мире сейчас является неструктурированными, и поэтому не могут быть эффективно представлены в таблицах. Благодаря современным информационным технологиям стало возможным использование различных типов данных (структурированные и неструктурированные), в том числе сообщений, голосовых сообщений в социальных сетях, изображений, данных с датчиков, видео или голосовых записей. При этом сохраняется возможность обработки структурированных данных.

4. Достоверность (veracity) указывает на способность данных отражать реальное представление человека о этих данных, или свойство данных адекватно описывать предметную область. Для большинства типов больших данных, качество и точность являются плохо контролируемыми (например, опечатки в сообщениях, сленг, плохое произношение в голосовых

сообщениях), но Big Data и интеллектуальные технологии сейчас позволяет работать с такими некачественными данными. Огромные объемы данных часто компенсируют недостаточное качество и точность.

5. Значимость (value) – это еще один V-аспект Big Data. Значимость определяет ценность данных для принятия решений, их полезность. Так что можно смело утверждать, что значимость – это самое важное в больших данных. В сфере бизнеса непрерывно предпринимаются попытки по максимуму реализовать преимущества Big Data. При этом, как выясняется на практике, бизнес может понести убытки вследствие неправильного понимания Big Data.

По мере развития технологий Big Data, изменяются, дополняются теоретические принципы, лежащие в их основе. Например, иногда говорят «правиле семи V» (7V's rule), добавляя следующие измерения Big Data:

1. Изменчивость (variability). Указывает на тот факт, что данные постоянно изменяются. Это особенно ярко проявляется при обработке голосовых данных. Брайан Хопкинс (главный аналитик компании Forrester) приводит суперкомпьютер IBM Watson как яркий пример этого факта. При участии в интеллектуальной игре Jeopardy, суперкомпьютеру IBM Watson пришлось «глубоко проанализировать смысл ответа и ... чтобы определить смысл вопроса». Слова не имеют статического значения, их смысл может меняться в зависимости от контекста.

2. Визуализация (visualization). После того, как данные обработаны, необходимо определить способ представления результатов в таком виде, чтобы они были легко понимаемыми.

1.4.2 Применение Big Data

В настоящее время открываются все новые способы использования Big Data. Рассмотрим лишь некоторые направления в науке, информационных технологиях и анализе:

1. Понимание потребностей и стимулирование покупательского спроса. Компании используют Big Data для лучшего понимания потребностей и информирования покупателей путем анализа данных об операциях купли-продажи и других источников (социальных сетей, геолокации т.д.). на сегодняшний день эта сфера Big Data является наиболее востребованной и, одновременно, изученной. Big Data используется для анализа привычек, потребностей, предпочтений потенциальных покупателей. Компании стремятся расширить традиционно анализируемые наборы данных за счет учета данных из социальных сетей, веб-браузеров, а также текстовой информации и данных сенсоров, чтобы видеть более полную картину о рынке. Конечная цель – построение модели прогноза. В качестве примера можно привести систему американской торговой компании Target, которая с высокой точностью может предсказать рождение ребенка у какого-либо клиента компании (потенциального покупателя).

Использование Big Data позволяет телекоммуникационным компаниям прогнозировать отток клиентов. Компания Wal-Mart может предсказать, какие продукты будут продаваться; страховые компании могут определить насколько хорошими водителями являются их клиенты. Даже правительственные избирательные кампании могут быть оптимизированы с использованием Big Data. Некоторые считают, победа Обамы после президентской избирательной кампании 2012 года была обусловлена использованием Big Data.

2. Понимание и оптимизация бизнес-процессов. Компании стремятся оптимизировать бизнес-процессы путем отслеживания и анализа маршрутов доставки, цепи поставок. Используются данные для оптимизации циклов обслуживания оборудования и прогнозирования сбоев оборудования. Розничные продавцы могут оптимизировать свои акции на основе прогнозов, полученных из данных социальных сетей, интеллектуального веб-анализа, прогнозов погоды и т.д. Один из бизнес-процессов, в котором накапливаются большие объемы данных – цепочки поставок или оптимизации маршрута

доставки. В данной области датчики географического положения и радиочастотной идентификации используются для отслеживания товаров, транспортных средств, а также для оптимизации маршрутов путем анализа данных в реальном времени. HR бизнес-процессы также улучшаются за счет использования Big Data.

3. Персональный мониторинг состояния здоровья и массовое здравоохранение. Big Data используется в здравоохранении для разработки новых препаратов для лечения рака, оптимизации курсов лечения и предсказания заболеваний. Вычислительная мощность больших аналитических систем Big Data позволяет декодировать целые строки ДНК в течение нескольких минут и позволяет находить новые методы лечения, лучше понимать и предсказывать штаммы болезней. Клинические испытания будущего не будут ограничены малыми размерами выборки, но потенциально может включать в себя огромные массивы данных с персональных датчиков, принадлежащих миллионам людей. Методы Big Data уже используются для мониторинга состояния новорожденных. Посредством записи и анализа стенограммы каждого удара сердца и данных о дыхании для каждого младенца, интеллектуальной обработки данных, система способна предсказывать инфекции за 24 часа до появления каких-либо физических симптомов. Таким образом, врачи имеют возможность вмешаться заранее, предотвратив нежелательное развитие. Методы Big Data позволяют отслеживать и прогнозировать эпидемии и вспышки болезней. Интеграция данных из медицинской документации с социальными сетями позволяет следить за вспышками гриппа в режиме реального времени, просто слушая и анализируя, о чем говорят и пишут люди.

4. Персональная оценка эффективности и оптимизация производительности. Big Data используется для анализа и повышения персональной эффективности (в спорте, на работе или дома); данные с датчиков в оборудовании и носимых устройствах могут быть объединены с данными видео-аналитики, чтобы получить целостную картину, выявить

скрытые закономерности. Большие данные могут повысить эффективность не только в масштабах компаний и государственных учреждений, технологии Big Data могут улучшить жизнь каждого отдельного человека. Каждый может извлечь пользу из данных, полученных от носимых устройств, таких как смарт-часы или смарт-браслеты.

5. Безопасность. Большие данные широко используются для повышения безопасности силовыми структурами и органами правопорядка. Например, Агентство национальной безопасности (NSA) в США использует Big Data в борьбе с терроризмом (возможно, прослушивая и законопослушных граждан при этом). Часто технологии Big Data применяются для обнаружения кибератак. Правоохранительные органы используют инструменты Big Data для поиска и поимки преступников, для прогнозирования уровня преступности. Сотрудники кредитных компаний, анализируя данные, выявляют мошеннические операции на финансовых рынках.

6. Улучшение и оптимизация городов и стран. Big Data используются, чтобы улучшить многие аспекты жизни городов и стран. Например, Big Data позволяет городам оптимизировать потоки транспортного трафика на основе информации о загруженности дорог в реальном времени, данных соцсетей и метеорологических данных. Во всех странах ведутся разработки компонентов систем типа «умный дом».

7. Спорт высоких достижений. Большинство видов спорта в настоящее время используют достижения Big Data. Существует инструмент IBM SlamTracker для обслуживания теннисных турниров; используются системы видео-аналитики, которые отслеживают эффективность каждого игрока в футбол или бейсбол; разработаны датчики в спортивном оборудовании, таком как корзины, шары, мячи, ворота и т.д. — все это позволяет реализовать обратную связь (через смартфоны и облачные сервисы). Многие элитные команды также отслеживают спортсменов за пределами баз подготовки с использованием смарт-технологий: режимы питания и сна, состояние в соцсетях.

8. Научные исследования. Под влиянием Big Data трансформируются и научные исследования. Например, Большой адронный коллайдер (БАК) в лаборатории ядерных исследований CERN – самый мощный ускоритель частиц. В процессе своей работы коллайдер генерирует огромные объемы данных. Центр обработки данных (ЦОД) CERN насчитывает 65 000 процессоров для анализа 30 Петабайт данных. При этом CERN задействует компьютеры более чем 150 ЦОД по всему миру.

9. Оптимизация производительности автоматизированных систем. Аналитические методы Big Data помогают машинам и устройствам становятся умнее, автономнее. Например, инструменты Google, используемые для автономного управления автомобилем. Инструменты больших данных также используются для оптимизации энергосистем посредством анализа данных от интеллектуальных счетчиков. Big Data применяют для оптимизации производительности компьютеров и хранилищ данных. Optimizing Machine and Device Performance. Big data analytics help machines and devices become smarter and more autonomous.

10. Финансовые торговые системы. Заключительное направление использования Big Data связана с высокочастотным подходом (High-Frequency Trading, HFT) в автоматизированных системах рыночных торгов. Алгоритмы Big Data используются для принятия решений о купле/продаже. Решения принимаются в автоматическом режиме за доли секунды на основе анализа данных из различных источников: новостных лент, социальных сетей.

Список сфер применения не является законченным, он постоянно пополняется и расширяется.

Заключение

Большие данные на сегодняшний день являются одним из передовых направлений в IT. Двумя главными факторами успеха Big Data являются появление фреймворка распределенных вычислений Hadoop и развитие

облачных технологий. Эти факторы способствовали массовому внедрению Big Data в различных сферах бизнеса, притом с низкими затратами. Hadoop является краеугольным камнем всех решений для Big Data. Хотя другие модели программирования, такие как MPP и BSP, появились, чтобы справиться с самыми сложными проблемами, все они зависят от Hadoop в той или иной форме, когда масштабы обрабатываемых данных превышают сотни терабайт. Глубокое понимание механизмов повысить эффективность реализуемых ими проектов.

В этом разделе пособия рассматривается понятие NoSQL. Приводятся исторический обзор развития технологии и освещаются основные механизмы NoSQL. Кроме основ данного подхода рассмотрены дистрибутивы, области применения. Одновременно выявляется, что NoSQL – это не решение всех проблем в сфере управления данными, и, безусловно, подходу свойственны определенные недостатки. Большинство решений NoSQL хорошо масштабируются при увеличении объемов обрабатываемых данных и работают на множестве узлов кластера. Распределенная обработка больших данных является сложной задачей, и нуждается в новых специальных методах.

Современное поколение разработчиков пришло в NoSQL из области реляционных баз данных, осознавая изменения, отличия этих подходов. Разработчик должен хорошо изучить NoSQL-технологии прежде чем принимать решение об их использовании в реальных проектах.

Контрольные вопросы

1. Назовите основные факторы появления и развития Big Data.
2. Опишите области в которых возникают проблемы обработки больших объемов данных.
3. Что такое зависимые и независимые витрины данных?
4. Охарактеризуйте ключевые принципы инкрементального подхода при разработке хранилищ данных.

5. Поясните содержание терминов: «NoSQL», «Hadoop», «IoT», «Big Data».
6. Что такое концепция (правило) пяти V?
7. Перечислите направления применения технологий Big Data.

РАЗДЕЛ 2. МОДЕЛИ И КОНЦЕПЦИИ BIG DATA

В рамках данного пособия будем, для упрощения, считать данные «большими» если они не могут обрабатываться (или просто храниться) с использованием мощности только одной ЭВМ в рамках заданного соглашения об уровне услуг (service level agreements, SLA). Последняя часть этого определения имеет решающее значение. Можно обрабатывать практически любой объем данных на одной машине. Даже данные, слишком большие для хранения на одном носителе, могут быть считаны из разделяемого хранилища, таких как сетевые устройства хранения данных (NAS). Тем не менее, количество времени, которое потребуется, чтобы обработать эти данные будет слишком большим.

Рассмотрим простой пример. Средний размер обрабатываемых данных составляет 200 ГБ, предположим, что мы можем считывать около 50 МБ в секунду. С учетом предположения скорости чтения, нам понадобится 2 секунды для последовательного чтения 100 МБ данных с диска, и чтение всей выборки данных (200 ГБ) займет 1 час. Теперь предположим, что процесс обработки необходимо завершить в течении 5 минут. Если 200 ГБ данных распределить по вычислительным узлам, всего мы имеем в наличии 100 узлов и каждый узел способен обрабатывать свою порцию данных локально, то весь процесс вычислений может быть завершена за 1 минуту.

Этот простой пример демонстрирует, что Big Data – это контекстно зависимая технология, причем контекст в данном случае определяется бизнес потребностями.

2.1 Технологии Big Data

Для всех технологий Big Data можно выделить набор общих характеристик:

1. Данные распределяются по нескольким вычислительным узлам (скорость ввода-вывода сети во много раз меньше скорости ввода-вывода диска).
2. Приложения распространяются по узлам и доставляются к данным вместо распределения (физического перемещения) данных.
3. Как можно больший объем данных обрабатывается локально (в пределах одного вычислительного узла).
4. Диски с произвольным доступом заменяются носителями с последовательным доступом (скорость передачи во много раз меньше времени поиска).

Целью всех технологических парадигм мира Big Data является распараллеливание ввода/вывода (I/O) для повышения производительности.

2.1.1 Данные на нескольких узлах

По определению большой данные – это данные, которые не могут быть обработаны с использованием ресурсов одной машины. Одним из факторов популярности Big Data послужило использование массово выпускаемых вычислительных средств. Обычная массовая ЭВМ располагает диском 2-4 ТБ. Так как на практике подход Big Data связан с обработкой гораздо больших выборок, то данные будут распределены между несколькими узлами. Следует обратить внимание, что не обязательно иметь в наличии десятки терабайт данных, чтобы реализовать распределенную обработку на нескольких узлах. Большинство узлов обрабатывают небольшую порцию данных локально. Так как в обработке данных участвует большое количество узлов, то основная трудность – распределение данных по узлам. Таким образом даже набор в 500 ГБ будет распределяться по нескольким узлам, даже если обработку такого небольшого набора данных способна реализовать одна ЭВМ. Такое распределение преследует следующие цели:

1. Каждый блок данных реплицируются (распределяется) на более чем один узел (фактор репликации в Hadoop по умолчанию равен 3). Это делает

систему более гибкой по отношению к сбоям. При сбое одного узла, другие узлы содержат копии данных, размещенных на сбойном узле.

2. Для параллельной обработки в процессе участвуют несколько узлов. Таким образом, когда 50 ГБ данных совместно обрабатываются на 10 узлах, каждый из которых реализует вычисления на своем локальном наборе данных, может быть достигнуто улучшение производительности в 5-10 раз. Может возникнуть вопрос: почему все данные не в сетевой файловой системе (NFS), в котором каждый узел может читать свою часть данных? Ответ в том, что чтение с локального диска значительно быстрее, чем чтение из сети. Системы Big Data нацелены на локальные вычисления, потому что библиотеки приложений копируются на каждый узел кластера перед запуском задания (экземпляр приложения). Эти процессы будут рассмотрены в следующем подразделе.

2.1.2 Доставка приложений к данным

Еще один ключевой принцип Big Data. Для тех, кто хорошо знаком с архитектурой корпоративных приложений Java EE, очевидны принципы построения многоуровневых приложений. В трехуровневой модели программирования данные обрабатываются централизованно на уровне приложений; данные доставляются на уровень приложений с уровня данных по сети. В рамках такого подхода общепринята централизованная обработка данных, хотя сами данные могут быть распределены.

В области Big Data такая загрузка сети недопустима. Перемещение терабайт данных на уровне приложений будет перегружать сеть и вести к появлению серьёзных недостатков, вплоть до сбоя системы. В мире Big Data сами данные распределены по узлам, а приложение доставляется к данным. Важно отметить, что подобный процесс технологически реализуется очень сложно. Необходимо осуществить доставку не только приложения к данным, но и всех библиотек. Если кластер имеет сотни узлов, то легко понять, почему процесс обслуживания и развертывания становится очень затранным. Поэтому

системы Big Data проектируются для централизованного развертывания кода, а сама система обладает механизмом для перемещения программного кода на каждый узел кластера до начала выполнения задания.

2.1.3 Локальная обработка данных на узле кластера

Эта особенность систем Big Data является естественным следствием предыдущих двух. Все модели программирования в Big Data являются ориентированными на распределенную и параллельную обработку. Сетевой ввод-вывод происходит является на порядок медленнее, чем ввод-вывод с использованием локального диска. Так как данные были распространены по узлам, а библиотеки приложений доставлены к данным, то основная цель заключается в реализации локальной обработки.

Несмотря на то, что обработка данных локально на узле является предпочтительной, в системах Big Data не всегда удастся реализовать такое поведение. Правильнее говорить, что система Big Data планирует задания таким образом, чтобы код приложения оказался как можно ближе к данным. Некоторые решения предполагают выборки данных между узлами, по меньшей мере, результатов с каждого узла (MapReduce). Передача небольшого объема агрегированных данных более оптимальна чем работа с исходным массивом данных. Поэтому загрузка сети будет незначительна (но не всегда).

2.1.4 Последовательное чтение и прямой доступ

Во-первых, необходимо определиться с механизмом чтения данных с диска. Головка диска должна располагаться в месте физического расположения данных на диске. Процесс позиционирования требует времени и известен как операция поиска. Как только голова диска установила стартовую позицию, данные считываются с диска последовательно. Это называется операцией передачи. Время операции поиска составляет приблизительно 10 миллисекунд; время операции передачи данных составляет порядка 20 миллисекунд (за 1 МБ). Это означает, что при чтении 100 МБ

данных фрагментами по 1 МБ из различных секций, весь процесс займет $10 * 100 = 1$ секунда (на поиск), $20 * 100 = 2$ секунды (на передачу). Таким образом, чтобы считать 100 МБ уйдет в общей сложности 3 секунды. Однако, если 100 МБ читаются последовательно с диска, то это займет 2,01 секунд.

Следует обратить внимание, что в расчетах использовались характеристики дисков 2009 года. Очевидно, что за прошедшие года они поменялись, но временные пропорции остаются. Большинство моделей Big Data, ориентированы на использование этой особенности. Данные последовательно считываются диска и фильтруются в основной памяти. Для сравнения, обычные модели РСУБД являются ориентированными на чтение с произвольным доступом.

2.1.5 Пример

Предположим, что вам необходимо получить общее количество продаж за 2000 год в разрезе регионов, при этом данные о продажах распределяются случайным образом на нескольких узлах. Технология Big Data для достижения этой цели может быть реализована несколькими этапами:

1. Каждый узел считывает все данные по продажам локально и производит фильтрацию всех операций, оставляя только данные за 2000 год. Данные распределяются случайным образом во всех узлах и считываются с диска последовательного. Фильтрация происходит в основной памяти, чтобы избежать расходов на время дискового поиска.

2. Каждый узел строит гистограмму продаж для каждого региона, добавляя в выборку новый элемент-регион по мере его нахождения в исходных данных (приложение присутствует на каждом узле и производит обработку локальных данных).

3. Когда все узлы завершили процесс подсчета продаж для каждого региона, они посылают соответствующий сигнал зарезервированному узлу (узел сборки), который определяется предварительно перед началом вычислительного процесса.

4. Узел сборки получает данные о продажах от каждого узла и складывает соответствующие значения, полученные от каждого узла для каждого региона.

5. Узел сборки выполняет сортировку полученной выборки по региону и отдает результирующий набор.

Этот вычислительный процесс демонстрирует типичный принцип Big Data: требование максимизации пропускной способности (объем работ за единицу времени) превалирует над требованием минимизации латентности (как быстро выполняется ответ на запрос).

2.2 Модель программирования Big Data

Основные модели программирования Big Data:

1. Массивно параллельные вычислительные системы (MPP): Greenplum от EMC и Netezza от IBM.

2. Системы In-memory: Oracle Exalytics и SAP HANA.

3. Системы MapReduce: Hadoop, как наиболее распространенный подход в области Big Data.

4. Системы Bulk synchronous parallel (BSP): Apache HAMA и Apache Giraph.

2.2.1 Массивно параллельные вычислительные системы (MPP)

По своей сути MPP системы используют некоторые формы разделения данных на основе значений, содержащихся в столбце или наборе столбцов. Например, в примере из подраздела 2.1.5 можно секционировать исходные данные таким образом, чтобы отдельный узел содержал данные только по определенному региону. Этот метод секционирования позволил бы получить результат для отдельного региона на одной машине. Очевидно также ограничение подобных систем: необходимо принимать решение о

распределении на этапе проектирования. Выбор критерия сегментации данных обосновывается бизнес-требованиями. При таком подходе, система плохо подходит для ad-hoc запросов. Регламентированные запросы будут обеспечивать высокую скорость работы, так как учитывают особенности распределения данных между узлами. Для преодоления данного ограничения, в данных системах предусмотрена возможность хранения нескольких вариантов наборов данных, разделенных различными критериями. В зависимости от запроса происходит обращение к тому или иному набору данных.

MPP-системы используют модель программирования, в которой реализованы следующие этапы вычислений (на примере предметной области в подразделе 2.1.5):

1. Данные распределяются в соответствии с критерием региона по отдельным узлам.
2. Каждый узел обладает требуемым приложением и библиотеками для реализации вычислений.
3. Доступ к данным осуществляется локально. Исключение из правила – выполнение запроса не учитывающего механизма распределения данных; в этом случае каждая задача получает данные от других узлов кластера через сеть.
4. Каждая задача читает данные последовательно. Фильтрация осуществляется в памяти.

2.2.2 In-Memory базы данных

С точки зрения реализуемых процедур, базы данных in-memory аналогичны MPP-системам. Различие в реализации: каждый узел in-memory системы обладает огромным объемом оперативной памяти (ОП, ОЗУ), большая часть данных обрабатывается именно в ОП. Система SAP HANA функционирует именно по этому принципу. Другие системы, такие как Oracle Exalytics используют специализированные аппаратные решения,

гарантирующие, что множество компьютеров функционируют как одна супер-ЭВМ. По сути, базы данных в памяти как MPP-базы данных с SQL-интерфейсом.

Одним из основных недостатков коммерческих реализаций баз данных в памяти является то, что на аппаратном уровне требуется реализовывать механизм блокировок. Кроме того, учитывая, что системы используют проприетарные и специализированные аппаратные средства, – они обычно дорогие. Попытка использовать неспециализированное оборудование для баз данных в памяти увеличивает размер кластера. Рассмотрим, например, сервер имеющий 25 ГБ ОЗУ. Попытка размещения 1 ТБ в памяти потребует более чем 40 узлов (учитываются другие виды серверной активности). Объем 1 ТБ – небольшой набор данных, но кластер разрастается до 40 узлов.

Следующие положения отражают каким образом базы данных in-memory реализуют принципы Big Data:

1. Данные распределяются в соответствии с принадлежностью к региону по узлам. Каждый узел перегружает данные в память.
2. Каждый узел содержит все необходимые приложения и библиотеки.
3. Каждый узел оперирует локальными данными. Исключение из правила – выполнение запроса не учитывающего механизма распределения данных; в этом случае каждая задача получает данных от других узлов кластера через сеть.
4. Так как данные размещены в ОЗУ, то требование последовательного чтения не выполняется (только в случае начальной загрузки данных в оперативную память).

2.2.3 Системы MapReduce

MapReduce – это парадигма программирования, на которую ориентировано данное пособие. Это наиболее распространенная технология из всех моделей программирования. Наиболее важные характеристики MapReduce для реализации Hadoop:

1. Подход ориентирован на использование доступного аппаратного обеспечения. При этом категория «доступные компьютеры» не включает настольные ЭВМ и ноутбуки. Узлы кластера являются серверами масштаба предприятия, но использующие распространенные широко доступные компоненты.

2. Данные не распределяются по узлам в соответствии с каким-либо критерием.

3. Программист должен реализовать два процесса: map (Мар-шаг) и reduce (Reduce-шаг).

MapReduce – это особенная технология в области высокопроизводительных баз данных. На высоком уровне MapReduce-система предполагает определения двух процессов: Мар-стадия и Reduce-стадия. Когда Hadoop используется для реализации MapReduce, данные обычно распределяются блоками по 64-128 МБ, и каждый блок реплицируется дважды (3-кратная репликация используется по умолчанию в Hadoop). В примере вычисления объемов продаж за 2000 год, данные будут загружены в распределенную файловую систему Hadoop (HDFS) блоками (64-128 МБ). В начале запуска процесса MapReduce система передает все библиотеки приложения (включая, определяемые программистом, процессы map и reduce) на каждый узел.

Каждый узел выполняет map-процесс, который считывает данные по продажам локально. Данные считываются по блокам и фильтруются (требуется только 2000 год). Затем каждый map-процесс предоставляет готовый выходной набор в формате ключ/значение. В данном примере ключом является регион, значение – объем продаж, записанный в данной записи.

В заключении, заранее определенное количество reduce-процессов получает на вход результаты от map-процессов. При этом каждый ключ ставится в соответствие только одному reduce-процессу, что гарантирует обработку данных региона только одним процессом. Выходные данные reduce-процесса записываются в HDFS.

Клиенты, затем, выполняют сортировку по полю «Регион», читая данные из HDFS. Данный шаг может быть выполнен на reduce-стадии, так как процесс свертки получает регионы в сортированном виде. При реализации данного примера для обеспечения такого поведения нужно определить единственный reduce-процесс. Это вызвано тем, что передача данным между map- и reduce-процессами задействует сеть, что является критическим аспектом (bottleneck) данной технологии.

Технология MapReduce реализует принципы Big Data следующим образом:

1. Данные разделяются на блоки в системе HDFS. Так как HDFS – распределенная файловая система, блоки данных распределяются по узлам.
2. Библиотеки приложения, включая map- и reduce-код приложения, распространяются на все узлы кластера.
3. Каждый узел осуществляет локальное чтение. Map-процессы выполняются на каждом узле и обрабатывают свою порцию блоков данных (в большинстве случаев, сопоставление задач и блоков данных – задача планировщика задач, который может выделить задаче удаленные блоки для поддержания равномерной загрузки всех узлов).
4. Данные читаются последовательно в пределах каждой задачи.

Важное ограничение модели MapReduce: парадигмы не подходит для итерационных алгоритмов. Подавляющее большинство алгоритмов в науке носит итеративный характер – каждая итерация приближает решение. При таком подходе, MapReduce парадигма требует каждую итерацию алгоритма выполнять как отдельную работу MapReduce. В таких приложениях обычно каждая итерация использует данные, полученные на предыдущей итерации. Так как MapReduce читает данные из постоянного хранилища, каждой итерации необходимо сохранять свои результаты в хранилище для работы следующей итерации над ними.

2.2.4 BSP-системы

Системы класса BSP работают аналогично системам, созданным в рамках MapReduce-парадигмы. Но вместо завершения работы процессов MapReduce, система BSP состоит из списка процессов (аналогичных тар-процессам), которые используют барьерную синхронизацию, пересылают данные центральному узлу и обмениваются с ним информацией. Когда итерация завершена, центральный узел сигнализирует о начале новой итерации. Барьерная синхронизация – это распространенная методика в параллельном программировании. Подобная техника применяется, когда вычисления осуществляются множеством узлов, но в определенные моменты необходимо согласовать потоки. Этот паттерн используется, когда все потоки должны завершить свою работу к определенному моменту, до момента, когда будет принято решение о продолжении или отмене обработки. На практике барьерная синхронизация часто используется в параллельных вычислительных системах.

2.2.5 Системы Big Data и системы, основанные на транзакциях

Важно понимать, как развивалась концепция транзакций в контексте технологий Big Data. Этот процесс аналогичен развитию концепции NoSQL. Hadoop поддерживает HBase в качестве NoSQL-хранилища. Но можно использовать также Cassandra или NoSQL-системы облачного типа такие как Amazon Dynamo

Пользователи РСУБД предполагают, что подобные решения полностью поддерживают ACID-свойства, хотя за такую поддержку приходится платить производительностью. Когда база данных вынуждена обрабатывать миллионы транзакций в секунду, такой режим работы представляет собой сложный технологический вызов, даже при смягчении свойств ACID.

В такой ситуации необходимо идти на компромиссы и подобное поведение выражается в т.н. CAP-теореме (теорема Брюера). CAP означает аббревиатуру:

1. Consistency (согласованность): все узлы имеют доступ к одним и тем же данным в течении всего времени.

2. Availability (доступность): гарантируется, что все узлы получают ответ на запрос в определенный наперед заданный временной интервал.

3. Partition tolerance (гибкость разделения): вся система продолжает функционировать в случае выхода из строя одного узла.

CAP-теорема утверждает, что невозможно достичь всех трех свойств базы данных одновременно, но достижимы только любые два из них. В соответствии с данной теоремой можно выделить следующие типы систем:

1. Consistent and Available (согласованные и доступные). Единая база данных с поддержкой свойств ACID является примером такой системы. Подобная система не обладает гибкостью к разделению, т.е. если база данных отказала, все пользователи будут отключены от системы.

2. Consistent and partition-tolerant (согласованные и гибкие к разделению). Примером таких систем служат кластерный РСУБД. Подсистемы распределенных транзакций гарантируют, что все пользователи будут видеть одни и те же данные (согласованность), а распределенный характер хранилища позволит обеспечить работоспособность при отказах узлов кластера. Тем не менее, из-за распределённых транзакций, система будет недоступна некоторый промежуток времени из-за выполнения алгоритма двухфазной фиксации изменений. Это ограничивает количество одновременно выполняемых транзакций.

3. Available and partition-tolerant (доступные и гибкие к разделению). В данном случае речь идет о т.н. временно доступных системах. Рассмотрим, например, очень популярную платформу электронной торговли Amazon.com. Предположим, что просматривая каталог товаров, обнаружили, что две товарные позиции доступны для продажи. Но покупатель понимает, что работа системы устроена таким образом, что между событиями обнаружения товара и посылкой запроса на покупку, кто-то еще может купить этот товар. Таким образом, нет необходимости заботиться о постоянной актуальности

данных сайта, так как подобный ресурс обслуживает слишком большое количество пользователей. Обеспечение актуального состояния товарной номенклатуры, выполнение постоянных обновлений для всех пользователей, существенно снизит доступность ресурса, что, в свою очередь, приведет к падению продаж. Значит в таких системах жертвуют согласованностью для обеспечения высокой доступности и гибкости к разделению. В таких системах существуют временные окна, в которые пользователи могут видеть различные версии одних и тех же данных.

Подобный подход очень важен в системах Big Data. MapReduce – это лишь один из компонентов экосистемы Big Data. Очень часто парадигма MapReduce рассматривается в контексте других решений, таких как HBase, которые ориентированы на использование в качестве решений электронных торгов.

2.3 Типы данных NoSQL

2.3.1 Хранилище на основе упорядоченных столбцов

Решение Bigtable компании Google предлагает модель хранения данных на основе столбцов. Это решение противоположно решению, применяемому в реляционных решениях, на основе строк. Хранилище столбцов более эффективно хранит данные. Подобный формат сокращает расходы при хранении пустых значений (NULL-значение или типизированных пустых значений).

Каждый элемент данных хранится как набор пар ключ/значение, причем каждый элемент однозначно идентифицируется ключевым атрибутом, также называемым первичным ключом. В Bigtable и его клонах такой идентификатор получил название строковый ключ. Дополнительно, как видно из названия формата, элементы данных хранятся в упорядоченном виде; элементы отсортированы и упорядочены по значению строкового ключа. Для лучшего понимания данного формата рассмотрим пример. Пусть дана простая таблица

данных, содержащая информацию о людях. Подобная таблица содержит следующие столбцы: `first_name` (имя), `last_name` (фамилия), `occupation` (адрес), `zip_code` (индекс), and `gender` (пол). Информация об отдельном человеке в такой таблице имеет вид:

`first_name: John`

`last_name: Doe`

`zip_code: 10001`

`gender: male`

Другой элемент данных может иметь иной вид:

`first_name: Jane`

`zip_code: 94303`

Строковый ключ первого элемента может быть равен 1, а второго 2. Тогда эти данные будут храниться таким образом, что элемент с ключом 1 будет размещен перед элементом с ключом 2. Также оба элемента будут выравнены относительно друг друга.

Далее, только пары ключ/значения с корректными значениями будут храниться в системе. Таким образом, возможное семейство столбцов для данного примера может быть названо как `name` и содержать столбцы `first_name` и `last_name`, которые принадлежат данному семейству. Другое семейство может быть `location` со столбцом `zip_code`. Третье семейство столбцов может быть названо как `profile`. Семейства столбцов обычно конфигурируются на этапе запуска. Сами столбцы не нуждаются в предварительном объявлении. Столбцы могут хранить данные любого типа, включая массивы байт.

Таким образом, логическое представление полученного хранилища для данного примера состоит из семейств столбцов: `name`, `location` и `profile`. Внутри каждого семейства хранятся только корректные значения. Семейство столбцов `name` хранит следующие значения:

For row-key: 1

`first_name: John`

last_name: Doe

For row-key: 2

first_name: Jane

Семейство столбцов location будет хранить:

For row-key: 1

zip_code: 10001

For row-key: 2

zip_code: 94303

Семейство столбцов profile будет хранить значения только для элемента с ключом 1:

For row-key: 1

gender: male

В практических реализациях хранилищ, семейства столбцов не изолированы в рамках одной строки. Все данные, относящиеся к одному строковому ключу хранятся совместно. Семейство столбцов выступает как ключ для столбцов, которые заключены в данном семействе, а строковый ключ выступает в качестве ключевого поля для всего набора данных.

Данный в Bigtable и его клонах хранятся в последовательной форме. По мере того как растет объем данных на отдельном узле, может возникнуть необходимости разделения узла на два. Данные сортируются и упорядочиваются не только в пределах одного узла, но и в рамках группы узлов, обеспечивающих функционирование укрупненного набора последовательных данных. Данные хранятся в отказоустойчивой форме, при котором поддерживаются три копии каждого набора.

Большинство клонов Bigtable также используют распределенной файловое хранилище, что позволяет распределять данные по узлам кластера.

Сортированная упорядоченная структура позволяет эффективно реализовать поиск данных.

HBase – это распространенное, свободно распространяемое решение на базе семейства столбцов, которое спроектировано на основе идей, лежащих в

основе Bigtable от Google. Данные в HBase обрабатываются в рамках парадигмы MapReduce. MapReduce-инструменты Hadoop могут использовать HBase как источник и/или приемника данных.

Вот некоторые open-source клоны Bigtable:

- HBase;
- Hypertable;
- Cloudata.

Хранилища на основе семейства столбцов в настоящее время широко распространены и формируют популярное NoSQL-направление. Тем не менее, NoSQL-подход включает в себе различные варианты хранилищ типа ключ/значение и документо-ориентированных хранилищ.

2.3.2 Хранилища ключ/значение

Структура данных HashMap или ассоциированный массив способна хранить набор элементов, каждый из которых представляет собой пару ключ/значение. Данная структура данных очень популярна благодаря эффективной реализации (в среднем $O(1)$) процедуры доступа к данным. Ключи в парах ключ/значение являются уникальными в пределах всего набора данных, благодаря чему просто обеспечивается поиск данных. Пары ключ/значения могут быть различного типа: некоторые предполагают хранение данных в ОЗУ, другие обеспечивают хранение на диске. Пары ключ/значение могут быть распределены по узлам кластера.

Простое и производительное хранилище типа ключ/значение – Berkeley DB от Oracle. Berkeley DB – это только движок хранилища данных, в котором и ключи и значения – байтовые массивы. Движок Berkeley DB не предполагает, что у данных могут быть смысловые значения. Berkeley DB оперирует массивами байтов и возвращает их по запросу клиентов. Данные кэшируются в ОЗУ и «сбрасываются» на диск по мере роста их объема. В Berkeley DB существует также механизм индексирования, обеспечивающий

ускорение операций поиска. Данная база данных существует с середины 1990 года. Изначально, Berkeley DB создавалась чтобы заменить NDBM от AT&T.

Другой распространенный способ организации хранилищ типа ключ/значение – кэширование. Кэширование обеспечивает снимок в памяти для тех данных, которые наиболее часто используются приложением. Основная цель подхода – минимизировать количество обращений к диску. Кэширование – это стратегия, которая реализуется на различных уровнях программного стека для увеличения производительности. Кэширование используется в операционных системах, базах данных, различных компонентах и приложениях.

Существуют надежные open-source распределенные системы кэширования наподобие EHCache (<http://ehcache.org/>), которая широко распространена при разработке Java-приложений. EHCache также может рассматриваться как решение NoSQL. Другая известная система кэширования – Memcached (<http://memcached.org/>), которая представляет собой open-source высокопроизводительную систему кэширования объектов. Memcached была создана Брэдом Фитцпатриком для LiveJournal в 2003 году.

Memcached позволяет эффективно управлять памятью ЭВМ, создавая виртуальный пул и выполняя при необходимости распределение общего адресного пространства между узлами. Это предотвращает появление ситуаций, когда один узел имеет избыток памяти, а другой испытывает недостаток в ней.

Так как NoSQL-технологии продолжают развиваться, то появляются различные хранилища типа ключ/значение. Некоторые из них строятся на основе Memcached API, другие используют Berkeley DB как низлежащее хранилище, третьи – реализуют альтернативные подходы на новых принципах.

Многие из систем, основанных на парах ключ/значение, обладают собственным API, позволяющим более просто развернуть распределенную систему. Несколько из них, такие как Redis (<http://redis.io/>), обеспечивают

более производительные API и более высокий уровень абстракции. Redis – это сервер структур данных, так как он предоставляет различные структуры данных в виде строк (последовательности символов), списков, множеств, а также карт отображения. Redis также предоставляет программисту богатый набор операций над этими структурами данных.

Некоторые хранилища на основе пар ключ/значение:

- Membase;
- Kyoto Cabinet;
- Redis.

Все три вышеуказанные системы являются эффективными хранилищами типа ключ/значение, обеспечивающие хранение данных для систем реального времени, часто используемых данных или просто для долговременного хранения данных.

Системы на основе пар ключ/значение рассматриваемые до сих пор обеспечивают надежную модель согласованности данных, которые они хранит. Однако, некоторые другие системы типа ключ/значение делают акцент на доступности вместо согласованности. Многие из них разработаны на основе Dynamo от Amazon, системы которая также основана на парах ключ/значение. Dynamo от Amazon обеспечивает высокую доступность и масштабируемость, и составляет основу отказоустойчивых систем Amazon. Амазонки вина и высокой доступности системы. Apache Cassandra, Basho Riak, и Voldemort являются open-source-проектами, в основе которых положены идеи Amazon Dynamo.

Amazon Dynamo выносит на первый план идеи высокой доступности. Наиболее важная из концепций – временная согласованность. Временная (условная, частичная) согласованность подразумевает, что там могут присутствовать небольшие интервалы несоответствия реплицированных узлов, в то время как происходит обновление данных между узлами peer-to-peer.

Временная согласованность не означает, что данные не согласованы. Это лишь слабая форма согласованности, чем та, которая определяется в принципах ACID для РСУБД.

Существуют клоны Dynamo:

- Cassandra;
- Voldemort;
- Riak.

2.3.3 Документо-ориентированные базы данных

Документо-ориентированные базы данных не являются системами автоматизации документооборота. Многие начинающие разработчики путают систему NoSQL на основе документо-ориентированной базы данных и систему управления контентом на основе документов. Слово документ в сфере баз данных подразумевает слабо структурированные наборы пар ключ/значение для описания документа в формате JSON (нотация объектов JavaScript); это не документы или электронные таблицы (хотя они также могут быть сохранены в подобных системах).

Документо-ориентированные базы данных рассматривают документ в целом, не разбивая его на составные пары имя/значение. Это позволяет оперировать коллекциями документов. Документо-ориентированные базы данных поддерживают индексирование не только на основе первичного ключа, но и с использованием различных свойств документа. На сегодняшний день доступны несколько СУБД данного типа, но наиболее популярными являются MongoDB и CouchDB.

2.3.4 Графовые базы данных

В предыдущих подразделах были рассмотрены основные направления развития продуктов NoSQL. Некоторые решения, например, основанные на графовых базах данных и XML, также могут быть отнесены к базам данных NoSQL. В данном пособии они рассматриваться не будут. Сегодня

наибольшую популярность приобрели графовые базы данных: Neo4j и FlockDB. Neo4J полностью удовлетворяет требованиям ACID. Такие СУБД облегчают обработку графов.

Заключение

В данном разделе раскрывается содержание термина NoSQL. Были рассмотрены некоторые теоретические аспекты построения систем на основе семейств столбцов, пар ключ/значение, частично согласованные базы данных и документо-ориентированные базы данных. Кроме этого, был представлен список продуктов с их характеристиками.

Очевидно, что подход NoSQL не является единственным решением проблем в области обработки данных, он также имеет свои недостатки. Тем не менее, большинство продуктов данного направления обладают свойством масштабируемости и позволяют распределить данные по узлам кластера в случае увеличения обрабатываемых данных. Обработка больших объемов данных является важной задачей, требующей поиска новых специализированных методов. Были рассмотрены такие шаблоны обработки данных как MapReduce, были раскрыты модели программирования в рамках данного подхода.

Контрольные вопросы

1. Каковы общие особенности всех методов в Big Data? Опишите каждую особенность.
2. Поясните принцип «данные распределены по нескольким узлам».
3. Поясните принцип «приложения перемещаются к данным».
4. Поясните принцип «данные обрабатываются локально».
5. Поясните принцип «последовательное чтение предпочтительнее прямого доступа».

6. Какая модель программирования наиболее распространена в области Big Data?
7. Опишите основные концепции, лежащие в основе MPP-систем баз данных.
8. Опишите основные концепции, лежащие в основе In-Memory баз данных.
9. Опишите основные концепции, лежащие в основе BSP-систем.
10. Какие типы баз данных NoSQL вы знаете? Опишите эти типы.

РАЗДЕЛ 3. ОСНОВЫ NOSQL

Базы данных на основе семейств столбцов – наиболее популярное решение среди нереляционных баз данных. Такие базы данных стали популярны благодаря беспрецедентным усилиям команды разработчиков Google и росту популярности социальных сетей Facebook, LinkedIn и Twitter. Эти системы стали своего рода символом NoSQL-революции.

Множество публикаций обеспечили внимание разработчиков со всего мира к технологиям поиска Google и открыли технологии крупно масштабных систем и больших данных: Google Earth, Google Analytics и Google Maps. Было установлено, что кластер недорогих машин способен управлять огромными объёмами данных гораздо эффективнее чем одна дорогая супер-ЭВМ. Были выдвинуты три ключевые концепции:

1. Необходимо хранить данные в распределенной сетевой среде, которая должна объединять несколько компьютеров. Файлы могут быть достаточно большими и должны храниться на нескольких вычислительных узлах.

2. Данные необходимо хранить в структуре, которая обеспечивает большую гибкость по сравнению с традиционными нормализованными реляционными базами данных. Схема хранения должна обеспечивать эффективное хранение огромных объемов разряженных данных. Необходимо изменять схему базы данных без удаления лежащих в ее основе таблиц.

3. Процесс вычислений должен строиться таким образом, чтобы данные могли обрабатываться отдельными изолированными подмножествами. Такое требование реализуется эффективно если программный код размещается физически на той же машине, где и обрабатываемые данные. Данный подход позволяет избежать необходимости передавать огромные объемы данных через сеть, что также увеличивает производительность.

3.1 Архитектура хранилища данных

3.1.1 Столбцы в реляционных базах данных

Bigtable от Google и Apache HBase (часть фреймворка Hadoop) являются базами данных, основанными на семействе столбцов, как и Hypertable и Cloudata. Архитектура этих систем варьируется, но основные принципы построения у них общие.

Поколение разработчиков нашего времени живет в эпоху реляционных баз данных. Обучение в университете, использование на работе, публикации – все направлено на изучение, внедрение и развитие систем управления реляционными базами данных. В основе концепции РСУБД лежат отношения и связи между ними.

Современное поколение разработчиков широко применяет реляционные базы данных. Фундаментальные концепции реляционных баз данных: сущности и связи между ними, изучаются в университетах, внедряются на предприятиях и обсуждаются и исследуются. Поэтому, изложение принципов построения столбцовых баз данных начнем с рассмотрения примера на основе РСУБД. Такой подход позволит более эффективно освоить материал.

В РСУБД атрибуты отношений представляются столбцами таблиц. Столбцы определяются заранее и значения хранятся для каждой строки каждого столбца. На рис. 3.1 представлена схема реляционной таблицы, которая понятно любому разработчику.

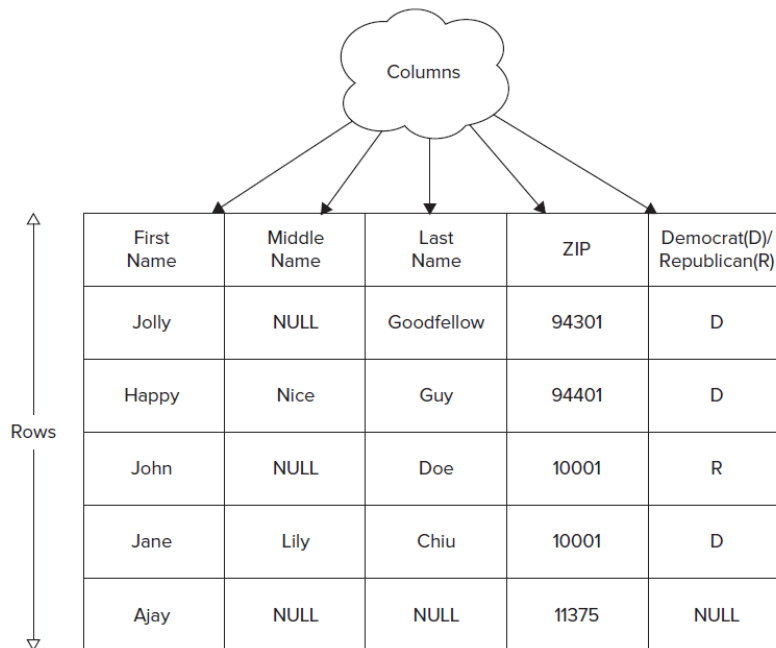


Рисунок 3.1 – Схема реляционной таблицы

В данном элементарном примере таблица содержит 5 столбцов. В РСУБД для каждого столбца определяется соответствующий тип данных. Например, для столбца «First Name» указывается тип `varchar`, для «ZIP» – `int`. В ячейках таблицы могут отсутствовать неопределенные значения (NULL-значения). Например, в примере на рис. 3.1 у Jolly Goodfellow отсутствует отчество (поле «Middle Name»).

Обычно, в таблице РСУБД определяется несколько столбцов, иногда – десятки. Внутри таблицы хранятся тысячи записей. В особых случаях таблицы могут хранить миллионы строк, но такое размещение может существенно повлиять на производительность и требуется применение специализированных механизмов оптимизации, таких как нормализация и индексирование.

В процессе использования таблицы может возникнуть ситуация, когда требуется добавить несколько столбцов, например «Street Address» и «Food Preferences». Как только добавлены новые столбцы, они не содержат значений. В ячейках будет множество NULL-значений. Такая ситуация часто встречается на практике. В этом случае, таблица будет иметь вид, представленный на рис. 3.2.

First Name	Middle Name	Last Name	Street Address	ZIP	D/R	Veg/ Non-Veg

Рисунок 3.2 – Разраженный набор данных

Предположим, что данные в таблице изменяются во времени и нам необходимо хранить различные версии данных. В таком случае можно использовать структуру напоминающую трехмерную таблиц Excel, где одним из измерений выступает время. Рисунок 3.3 демонстрирует 3-D-набор данных.

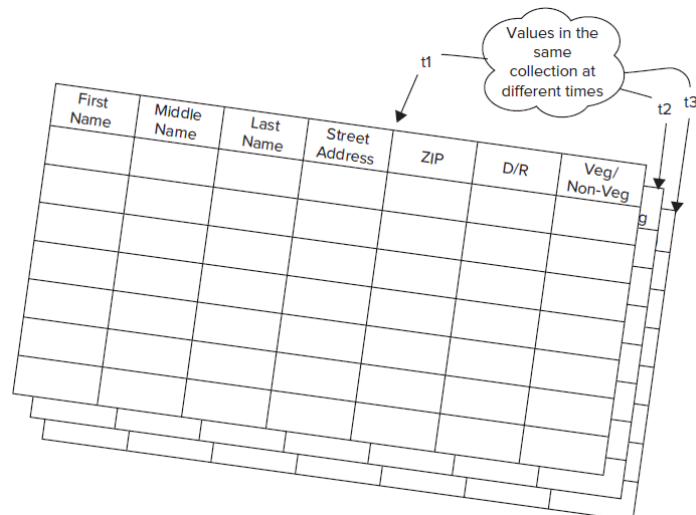


Рисунок 3.3 – 3-D-структура данных

Несмотря на то, что пример достаточно простой, вы может догадаться насколько сильно увеличиться количество пустых ячеек если в определенный момент времени все данные из таблицы будут удалены. Хранение большого количества пустых ячеек может быть сопряжено с серьезными проблемами. Точнее, проблемы связаны с особенностями РСУБД.

3.1.2 Столбцовые наборы данных и РСУБД

На основе предыдущего примера с таблицей рассмотрим столбцовые базы данных. Рассмотрение будет акцентировано на особенностях построения баз данных на основе столбцов.

Прежде всего, столбцовые базы данных отличаются отсутствием строгих требований к схеме базы данных; можно легко добавлять новые столбцы при появлении дополнительных данных. В обычном столбцовом хранилище заранее определяются семейства столбцов, а не столбцы. Семейство столбцов – это набор столбцов, сгруппированных в пакет. Столца в одном пакете логически связаны, хотя это не является обязательным требованием. Колонки семейства хранятся физически совместно и пользователь, как правило, объединяет в семейство несколько столбцов с одинаковыми характеристиками. Могут существовать различные ограничения на количество столбцов в семействе, но определение минимального количества колонок в пакете позволяет сделать схему более гибкой. В нашем примере достаточно определить три семейства: `name`, `location` и `preferences`.

В столбцовых базах данных семейство столбцов – это аналог столбца в РСУБД. Оба элемента определяются заранее, перед добавлением данных. Столбцы в РСУБД предполагают задание типа хранимых значений, что не является обязательным в столбцовых базах: они могут содержать любое количество столбцов, содержащих любые типы значений.

Каждая строка в столбцовых базах хранится только для тех столбцов, в которых для нее присутствуют корректные значения. `NULL`-значения не хранятся в базе. На рис. 3.4 показано как изменится схема данных при использовании семейств столбцов.

	name	location	preferences
	first name=>"...", last name=>"..."	zip=>"..."	d/r=>"...", veg/non-veg=>"..."

Рисунок 3.4 – Таблица в столбцовой базе данных

Кроме хорошего механизма хранения разреженных данных, столбцовые базы данных предоставляют возможность работы с данными в историческом разрезе. Поэтому постоянно развивающиеся данные будут храниться с использованием структуры, представленной на рис. 3.5.

time	name	location	preferences
t9	first name=>"...", last name=>"..."	zip=>"..."	d/r=>"...", veg/non-veg=>"..."
t8			
t7			
t5			

Рисунок 3.5 – Хранение исторических данных

На физическом уровне данные не хранятся как единая таблица, но расположены как набор семейств столбцов. Набор столбцов спроектирован для масштабируемой архитектуры и легко может оперировать миллионами столбцов и миллиардами строк. Поэтому, одна таблица часто разделяется на множество машин. Строковый ключ идентифицирует запись в столбцовой базе. Строки упорядочены и объединены в пакеты, содержащие непрерывные значения, отражающие изменение данных во времени.

Рисунок 3.6 демонстрирует физическую схему хранения таблицы.

row-key	time	name	row-key	time	location
	t9			t9	
	t8			t8	
	t7				
	t5				

row-key	time	preferences
	t9	
	t7	

Рисунок 3.6 – Разделение таблицы на физическом уровне

Обычное решение при использовании столбцовой базы данных – применение кластера. Запуск подобного решения на единственной машине осуществляется для целей тестирования. Каждая база данных на основе семейства столбцов рассчитана на определенную типовую топологию кластера и обладает своим механизмом развертывания, но все подобные решения предполагают некоторый типовой сценарий.

3.1.3 Наборы столбцов, встроенные отображения, пары ключ/значение

Несмотря на то, что представление столбцовых баз данных в виде таблиц со специальными свойствами легко понять, это создает путаницу. Часто такие термины, как столбцы и таблицы сразу же вызывают в воображении реляционные базы данных и приводит к необходимости планирования схемы данных. Это может быть вредным и часто приводит к отказу разработчиков от использования столбцовых баз данных. Это, безусловно, одна из ловушек проектирования, которую каждый разработчик должен избегать. Всегда необходимо помнить, что корректная методика работы с инструментарием важнее самого инструментария. Если РСУБД является именно тем, что нужно в конкретном проекте, то следует использовать именно эту модель. Однако, если вы используете столбцовую базу данных, в целях масштабирования огромного хранилища данных, тогда используйте NoSQL-подход без каких-либо РСУБД.

Часто, легче представлять себе столбцовые базы как набор отображений или карт. Карты или хэш-карты, которые также являются ассоциативными массивами, представляют пары ключей и соответствующих значений.

Ключи должны быть уникальными, чтобы избежать коллизий, а значения часто являются любым массивом байтов. Некоторые карты могут содержать только строковые ключи и значения, но большинство баз данных на основе столбцов не имеют такого строгого ограничения.

Не удивительно, что Bigtable от Google, настоящий прообраз для всех столбцовых баз данных текущего поколения, официально определяется как разреженная, распределенная, устойчивая, многомерная и отсортированная карта отображения.

Рассматривая пример как многомерную вложенную карту, можно создать первые два уровня ключей в JSON-подобном представлении, например, так:

```

1  "row_key_1" : {      15  "row_key_2" : {
2    "name" : {        16    "name" : {
3      ...             17      ...
4    },               18    },
5    "location" : {    19    "location" : {
6      ...             20      ...
7    },               21    },
8    "preferences" : { 22    "preferences" : {
9      ...             23      ...
10   }                24    }
11 },                25 },
                        27  "row_key_3" : {
                        28    ...
                        29  }
```

Рисунок 3.7 – JSON-подобное представление данных

Первый уровень ключа является строковым ключом, идентифицирующим запись в базе данных. Ключ второго уровня – это идентификатор уровня семейства столбцов. Три семейства столбцов были определены ранее: name, location и preferences. Следуя такому шаблону, можно определить ключ третьего уровня как идентификатор столбца. Каждая строка может охватывать различное количество столбцов внутри семейства, соответственно ключ третьего уровня может варьироваться между двумя точками многомерной карты. Добавление третьего уровня можно продемонстрировать так:

```

1  "row_key_1" : {
2      "name" : {
3          "first_name" : "Evgeny",
4          "middle_name" : "Ivanovich",
5          "last_name" : "Nikolaev"
6      },
7      "location" : {
8          "zip": "355029"
9      },
10     "preferences" : {
11         "color" : "Red"
12     }
13 },
17 "row_key_2" : {
18     "name" : {
19         "first_name" : "Ivan",
20         "middle_name" : "Sergeevich",
21         "last_name" : "Novak"
22     },
23     "location" : {
24         "zip": "355000"
25     },
26     "preferences" : {
27         "color" : "Green"
28     }
29 },

```

Рисунок 3.8 – Многомерная JSON-карта

В заключении, необходимо добавить версионирование элементов данных. Ключ третьего уровня может быть расширен путем добавления метки времени. В нашем примере для демонстрации такого поведения применяется целочисленная метка времени; показывается, что любимый цвет Николаева Евгения в момент времени 1 – красный, а в момент времени 2– зеленый.

Карта строки определяется следующим образом:

```

1  "row_key_1" : {
2      "name" : {
3          "first_name" : {
4              1: "Evgeny"
5          }
6          "last_name" : {
7              1: "Nikolaev"
8          }
9      },
10     "location" : {
11         "zip": {
12             1: "355029"
13         }
14     },
15     "preferences" : {
16         "color" : {
17             1: "Red",
18             2: "Green"
19         }
20     }
21 }

```

Рисунок 3.9 – Многомерная JSON-карта с временной меткой

3.1.4 Webtable

Никакое обсуждение баз данных на основе столбцов не обходится без определяющего примера на основе Webtable – решения, которое хранит копии просмотренных веб-страниц. Такое табличное хранилище хранит содержимое

веб-страницы, а также атрибуты, которые относятся к странице. Такие атрибуты могут быть якорем, который ссылается на страницу или MIME-типы, которые имеют отношение к содержимому страницы. Google впервые ввел это решение в своей научно-исследовательской работе по Bigtable. Webtable использует инвертированные URL веб-страниц в качестве строки-ключа. Таким образом, URL `www.example.com` подразумевает строку ключ `com.example.www`. Строковые ключи ведут к упорядочиванию строк данных в столбцовой базе. Таким образом, те строки, которые относятся к двум поддоменам одного домена `example.com`, такие как `www.example.com` и `news.example.com`, сохраняются близко друг к другу, когда инвертированный адрес используется в качестве строки-ключа. Это позволяет проще реализовать запрос на все содержимое, относящейся к одному домену. Обычно, веб-содержимое, якоря и MIME-данные обрабатываются как семейства столбцов, что приводит к модели данных на основе таблиц и семейств столбцов (рис. 3.10).

row-key	time	contents	anchor	mime
com.cnn.www	t9		cnnsi.com	
	t8		my.look.ca	
	t6	"<html>..."	my.look.ca	"text/html"
	t5	"<html>..."		
	t3	"<html>..."		

Рисунок 3.10 – Концепция Webtable

Многие open-source реализации Bigtable включают документацию по Webtable.

3.2 Архитектура распределенного хранилища HBase

Надежная архитектура HBase включает в себе несколько компонент. По крайней мере в основе распределенной системы лежит централизованный сервис для конфигурирования и синхронизации. Обзор архитектуры представлен на рис. 3.11.

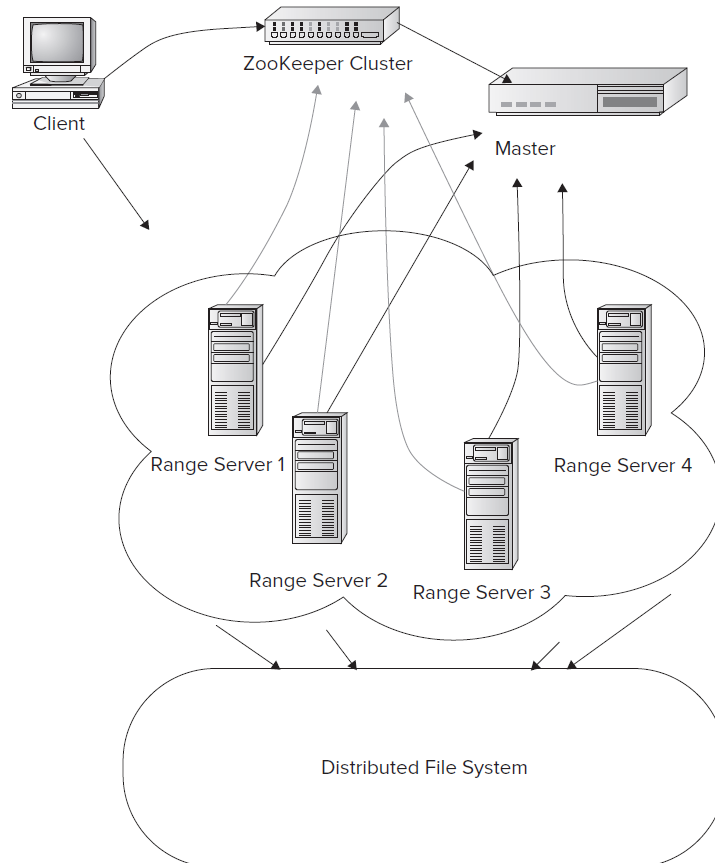


Рисунок 3.11 – Конфигурация HBase

Развертывание HBase придерживается шаблона мастер-рабочий. Поэтому, как правило, имеется мастер-процесс и набор процессов-рабочих, широко известный как сервера диапазонов. Когда HBase начинает выполняться, мастер-процесс размещает набор диапазонов на серверах диапазонов. Каждый диапазон хранит упорядоченное множество строк, где каждая строка идентифицируется уникальным строковым ключом. По мере того как число строк, хранимых в диапазоне увеличивается и превышает некоторое пороговое значение, диапазон разделяется на две части и строки делятся между двумя новыми диапазонами.

Как и большинство столбцовых баз данных, HBase хранит столбцы в семействах столбцов. Поэтому, каждая область использует отдельное

хранилище для каждого семейства столбцов таблицы. Каждое хранилище, в свою очередь, отображается на физический файл, который хранится в распределенной файловой системе. Для каждого хранилища, абстрактный уровень HBase предоставляет доступ к файловой системе с помощью тонкой прослойки сервисов, которая выступает в качестве посредника между хранилищем и физическим файлом.

Каждая область обладает своим хранилищем типа in-memory, или кэшем, и системой записи (write-ahead-log, WAL). WAL – это целое семейство технологий, обеспечивающих атомарность и долговечность (два из свойств ACID) в СУБД. WAL используется с различными СУБД, включая популярные РСУБД, такие как PostgreSQL и MySQL. В HBase разработчик может включить или отключить механизм WAL. Отключение увеличит производительность решения, но снизит надежность и возможности восстановления. Когда данные пишут в область, они сначала записываются в WAL, в случае соответствующей конфигурации. Затем, данные переписываются в хранилище in-memory для данной области. Если область памяти переполнена, то данные сбрасываются на диск и сохраняются в долговременное дисковое хранилище.

Рисунок 3.12 демонстрирует ключевые концепции использования серверов диапазонов.

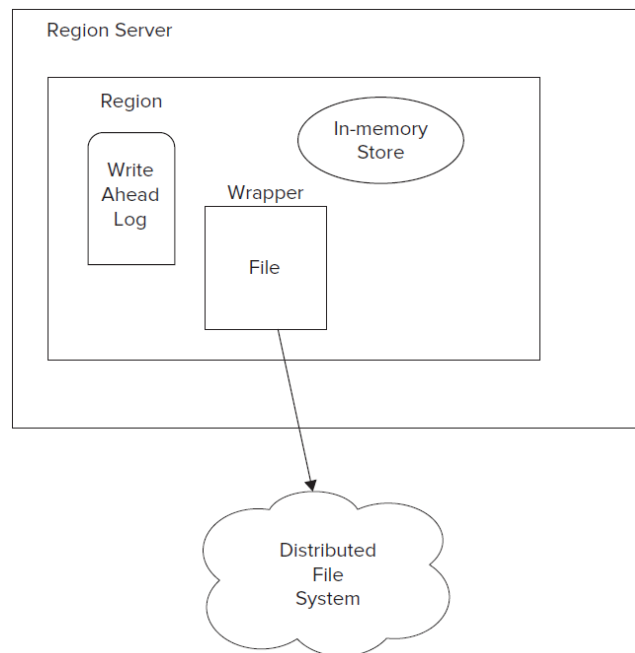


Рисунок 3.12 – Сервер диапазонов и диапазон

Если используется распределенная файловая система, как Hadoop HDFS, то шаблон мастер-работчий распространяется на базовую схему хранения. В HDFS NameNode и набор DataNodes образуют структуру, аналогичную конфигурации на основе мастер-процесса и серверов диапазонов в HBase. В такой конфигурации каждый физический файл хранилища для хранилища HBase заключается DataNode HDFS. HBase использует API файловой системы, чтобы избежать сильной связи с HDFS и таким образом API действует в качестве посредника для взаимодействия между хранилищем HBase и HDFS-файлом. API также позволяет HBase беспрепятственно работать с другими типами файловых систем. Например, HBase может использоваться с CloudStore (ранее известным как Kosmos FileSystem, KFS), вместо HDFS.

В дополнение к распределенной файловой системе для хранения, кластер HBase также использует утилиты конфигурации. В основополагающем документе по Bigtable, Google назвал эту программу конфигурации Chubby. Hadoop, будучи клоном инфраструктуры Google, использует его точный аналог Zookeeper. Zookeeper – это front-end-подсистема кластера HBase для клиентов и управления конфигурацией.

Чтобы получить доступ к HBase в первый раз, клиент получает доступ через два каталога Zookeeper. Эти каталоги называются `-ROOT-` и `.meta.` Каталоги поддерживают информацию о состоянии и местоположении всех регионов. Каталог `-ROOT-` хранит информацию о всех `.meta.`-таблицах и `.meta.`-файлах, хранит записи для пользовательского пространства таблицы, то есть для таблицы, которая содержит данные. Когда клиент хочет получить доступ к конкретной строке, он сначала опрашивает Zookeeper для доступа к каталогу `-ROOT-`. Каталог `-ROOT-` находит каталог `.meta.` для соответствующей строки, которая, в свою очередь определяет все детали для доступа к области конкретной строки. Трехступенчатый процесс доступа к строке не повторяется в следующий раз, когда клиент запрашивает данные строки повторно. Базы данных на основе столбцов в значительной степени полагаются на кэширование всей необходимой информации. Это означает, что клиенты могут обратиться непосредственно к серверу диапазонов в следующий раз, когда понадобятся данные строки. Длинный цикл поиска повторяется только если информация о области или в кэше является устаревшей или регион отключен и недоступен. Каждый диапазон часто идентифицируется наименьшим значением строкового ключа, таким образом, просматривая строку, можно определить, что строковый ключ больше или равен идентификатору диапазона.

К настоящему моменту были рассмотрены существенные концептуальные и физические модели хранения базы данных на основе столбцов. Также были рассмотрены механизмы чтения и записи данных в подобные хранилища. Расширенные функции и подробные нюансы функционирования баз данных на основе столбцов будут освещены в последующих главах.

3.3 Документо-ориентированное хранилище

MongoDB является хранилищем на основе документов, в котором документы сгруппированы в коллекции. Коллекции могут быть приближенно рассмотрены как реляционные таблицы. Тем не менее, коллекции не накладывают строгие ограничения схемы, которые приняты в отношении реляционных таблиц. Произвольные документы могут быть сгруппированы вместе в одной коллекции. Документы в коллекции должны быть структурно похожи для эффективной индексации. Коллекции могут быть разделены с помощью пространств имен, но внутренне представление не является иерархическим.

Каждый документ хранится в формате BSON. BSON является двоичным представлением формата документа JSON-типа, структура которого близка к набору пар ключ/значение. BSON является надстройкой JSON и поддерживает дополнительные типы, такие как регулярные выражения, двоичные данные и даты. Каждый документ имеет уникальный идентификатор, который MongoDB может генерировать, если он не указывается явно. Генерация происходит когда данные добавляются в коллекцию. Структура автоматически сгенерированного идентификатора показана на рис. 3.13.

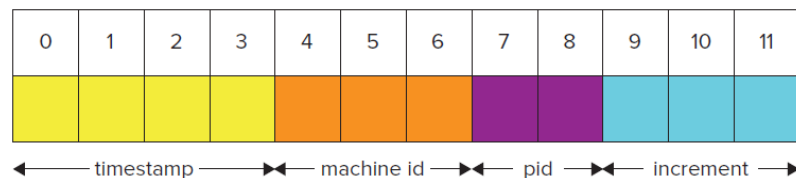


Рисунок 3.13 – Структура идентификатора

Драйвера и клиенты MongoDB выполняют сериализацию и десериализацию BSON. Сервер MongoDB, с другой стороны, понимает формат BSON и не требует дополнительных накладных расходов на сериализацию. Двоичные представления читаются в том же формате, в каком они передаются по сети. Это обеспечивает увеличение производительности.

Высокая производительность является важной концепцией в философии MongoDB, которая построена на подобных концепциях. Одной из таких

концепций является использование mapped-memory файлов для реализации хранилища.

3.3.1 Memory-Mapped файлы

Memory-Mapped файл представляет собой сегмент виртуальной памяти, который отображен байт-в-байт на файл или файл-подобный ресурс, на который можно ссылаться через дескриптор файла. Из этого следует, что приложения могут взаимодействовать с такими файлами, как если бы они были частью первичной памяти. Это, очевидно, повышает производительность ввода/вывода по сравнению с обычным дисковым чтением и записью. Доступ и манипулирование данными в памяти гораздо быстрее, чем реализация аналогичных системных вызовов. Кроме того, во многих операционных системах, таких как Linux, отображаемые на файлы регионы памяти являются частью буфера страниц дисковой памяти в оперативной памяти. Этот буфер обычно называется кэшем страниц. Он реализован в ядре операционной системы.

Стратегия MongoDB использования памяти, отображаемой на файлы, для хранения является достаточно интеллектуальной, но не лишенной недостатков. Во-первых, memory-mapped файлы подразумевают, что не существует никакого разделения между кэшем операционной системы и кэшем базы данных. Это означает, что также отсутствует избыточность. Во-вторых, кэширование управляется операционной системой, а отображение виртуальной памяти работает не одинаково на различных операционных системах. Таким образом, политики кэширования, которые определяют, что хранится в кэше, а что нет, также варьируются от одной операционной системы к другой. В-третьих, MongoDB может расширить свой кэш базы данных, чтобы использовать всю доступную память без каких-либо дополнительных настроек. Это означает, что производительность MongoDB может быть увеличена за счет приращения объема RAM и выделения большей виртуальной памяти.

Отображение памяти также предполагает некоторые ограничения. Например, реализация MongoDB ограничивает размер данных до 2 ГБ на 32-разрядных системах. Это ограничение не актуально для 64-битных систем.

Размер базы данных ограничен не только по размеру. Дополнительные ограничения лимитируют размер каждого документа и количество коллекций сервера MongoDB. Документ не может быть больше, чем 8 МБ. Это, очевидно, означает, что использование MongoDB для хранения больших элементов не подходит. При хранении больших документов необходимо использовать GridFS. Кроме того, существует ограничение на количество пространств имен, которые могут быть определены в экземпляре базы данных. По умолчанию, количество пространств имен – 24000. Каждая коллекция и каждый индекс использует пространства имен. Соответственно, два индекса на коллекцию позволит организовать максимум 8000 коллекций на одну базу данных. Как правило, такое количество достаточно. Тем не менее, если нужно, то можно увеличить количество пространств имен до числа, превышающего 24000.

Увеличение размера пространства имен влечет определенные последствия. Каждое пространство имен коллекций использует несколько килобайт. В MongoDB индекс реализован в виде В-дерева. Каждая страница В-дерева имеет размер 8 КБ. Таким образом, добавление дополнительных пространств имен, для коллекций или индексов, предполагает добавление несколько килобайт для каждого дополнительного экземпляра. Пространства имен для имени MyDB базы данных MongoDB сохраняются в файле с именем mydb.ns. Файл mydb.ns может вырасти до максимального размера 2 ГБ. Ограничение размера mydb.ns ограничивает и рост базы данных – понимание этого намного важнее чем знание моделей поведения коллекций и индексов.

3.3.2 Коллекции и индексы MongoDB

Нет формулы, которая определяла бы оптимальное количество коллекций в базе данных, но желательно избегать создания коллекций, объединяющих разрозненные данные в единую коллекцию. Такое хранение

создает сложности для индексирования. Одним из правил является следующее: дайте ответ на вопрос «Часто ли вам требуется запрашивать из коллекции разрозненные данные?». Если ответ «Да», то данные целесообразно сохранить вместе в одной коллекции.

Иногда коллекция может расти очень быстро и создается угроза превышения лимита в 2 ГБ. Тогда возникают предпосылки для использования ограниченных коллекций (capped collections). Ограниченная коллекция в MongoDB как стек, который имеет предопределенный размер. Когда сборную коллекцию превышает свой лимит, старые записи коллекции будут удалены. Старые записи определяются в соответствии с алгоритмом LRU (Least Recently Used). Документ, выбираемый из ограниченной коллекции, следует алгоритму LIFO (Last-In-First-Out).

Поле `_id` индексирует каждую коллекцию MongoDB. Дополнительно, программист может создавать индексы по любым другим атрибутам документа. При запросе документы выбираются в соответствии с их порядком, определяемым по `_id`. Только ограниченные коллекции используют порядок доступа LIFO.

Курсоры возвращают данные в виде пакетов, каждый ограничен размером 8 МиБ. Обновление записей происходит локально. MongoDB предоставляет повышенную производительность, но это достигается за счет снижения надежности.

3.3.3 Надежность и долговечность

Прежде всего MongoDB не всегда реализует принцип атомарности, а также не определяет транзакционный механизм или уровни изоляции в процессе параллельных операций. Только определенный класс операций, операции модификации, предполагают атомарную согласованность.

MongoDB определяет несколько операций модификаций для атомарного обновления (таблица 3.1).

Table 3.1 – Операции модификации MongoDB

Name	Description
\$inc	Инкрементация значения определенного поля
\$set	Установка значения поля
\$unset	Удаление поля
\$push	Добавление значения к текущему значению поля
\$pushAll	Добавление значения к массиву значений полей
\$addToSet	Добавление значения в массив (успешно только при отсутствии такого значения в массиве)
\$pop	Удаление последнего элемента в массиве
\$pull	Удаление всех вхождений значения в массиве
\$pullAll	Удаление всех значений, указанных в массиве, из значения поля
\$rename	Переименование поля

Отсутствие уровней изоляции также иногда приводит к проблеме чтения фантомных данных. Курсоры не обновляются автоматически в случае изменения данных, на которых они были получены. По умолчанию, MongoDB сбрасывает на диск данные раз в минуту. Данные записывают на диск, когда происходит вставка или обновление. Любой сбой между двумя синхронизациями может привести к несогласованности. Можно настроить MongoDB таким образом, чтобы повысить частоту синхронизации, но это приведет к снижению производительности.

Для обеспечения отказоустойчивости применяется механизм репликации. Два экземпляра MongoDB могут быть развернуты в конфигурации master-slave (ведущий-ведомый) и осуществлять репликацию и поддерживать данные синхронизированными. Репликация – это асинхронный процесс, поэтому изменения не применяются мгновенно.

Тем не менее, лучше использовать механизм репликации, чем не реализовать ни одного доступного механизма поддержки стабильности. В текущей версии MongoDB пара реплик master и slave была заменена набором реплик, содержащим три реплики. Одна реплика из набора играет роль master, две другие – slave. Набор реплик позволяет реализовать автоматическое восстановление и автоматическое переключение на исправные узлы.

Если репликация рассматривается в качестве механизма восстановления после сбоев, то шардинг (sharding) связан с горизонтальным масштабированием систем MongoDB.

3.3.4 Горизонтальное масштабирование

Одна из главных причин использования MongoDB – это отсутствие необходимости задавать схему, высокая производительность и масштабируемость. В более поздних версиях MongoDB поддерживает механизм автоматического шардинга для простого горизонтального масштабирования.

Идея шардинга схожа с концепцией мастер-рабочий в столбцовых базах данных, когда данных распределяются на множестве серверов диапазонов. В MongoDB упорядоченные коллекции можно сохранять на множестве машин кластера. Каждая машина, на которой хранится часть коллекции, называется шардом. Шарды, таким образом, реплицированы для обеспечения отказоустойчивости. Таким образом, большая коллекция может быть разделена по 4 шардам и каждый шард реплицируется 3 раза. Такое размещение создает 12 узлов на сервере MongoDB. Две дополнительные копии каждого шарда играют роль узлов восстановления.

Шарды – это понятие уровня коллекций, а не уровня базы данных. Поэтому одна коллекция в базе данных может размещаться на единственном узле, тогда как другая коллекция может быть разнесена по нескольким узлам.

Каждый шард хранит последовательный набор упорядоченных документов. Такой набор называется чанк (chunk). Чанк – это новый термин в области баз данных, но в MongoDB это устоявшийся жаргонный термин. Каждый чанк идентифицируется тремя атрибутами: ключ первого документа (min key), ключ последнего документа (max key) и коллекция.

Коллекция может быть шардирована на основе любого паттерна с применением ключей шардинга. Любое поле документа или комбинация полей могут быть использованы для создания ключа шардинга. Ключи

шардинга содержат дополнительное свойство, указывающее направление упорядочивания. Направление упорядочивания может быть равно 1, что значит упорядочивание по возрастанию, или -1 – упорядочивание по убыванию. Выбор ключа шардинга – важный этап конфигурирования, ключ должен позволять эффективно разделять наборы больших данных и балансировать нагрузку.

Все данные о шардах и чанках содержатся в метаданных на сервере конфигурации. Как и сами шарды, конфигурационные сервера также поддерживают механизм репликации для повышения отказоустойчивости.

Процессы клиентов получают доступ к кластеру MongoDB через процесс mongos. Процесс mongos не имеет постоянного состояния – конфигурация определяется серверами конфигурации. В кластере может существовать несколько процессов mongos. Эти процессы отвечают за маршрутизацию пользовательских запросов, а также за объединение результатов запросов, если это требуется. Запрос к кластеру MongoDB может быть целевым или глобальным. Все запросы, которые используют ключ шардинга – целевые, в противном случае – глобальные. Целевые запросы более эффективны. Глобальные запросы можно рассматривать как команды, приводящие к полному перебору коллекции.

Рисунок 3.14 демонстрирует архитектуру шардинга для MongoDB.

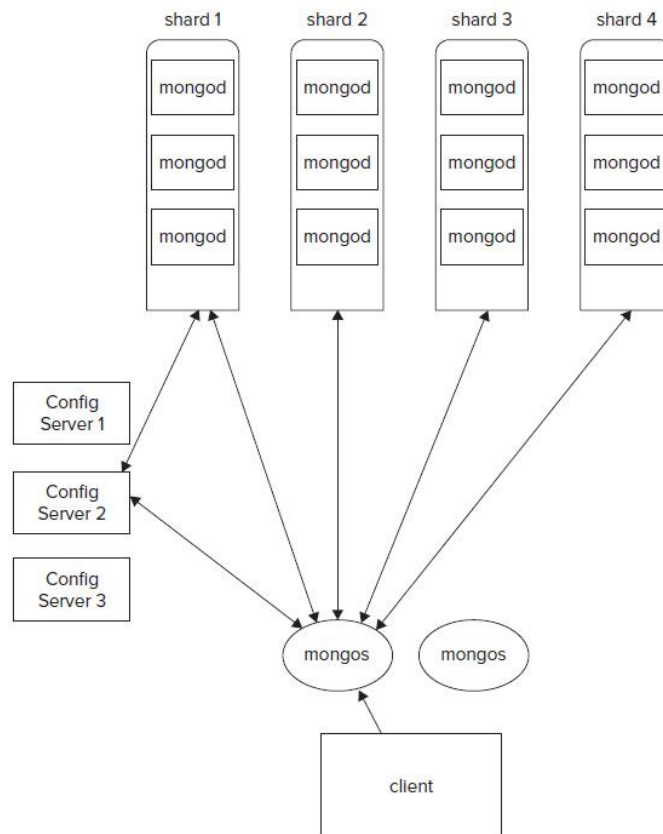


Рисунок 3.14 – Топология связей для реализации шардинга MongoDB

3.3.5 Хранилище типа ключ/значение в Memcached и Redis

Несмотря на то, что все хранилища типа ключ/значение отличаются, у них существуют общие базовые принципы построения. Например, они хранят данные в виде карт. Рассмотрим внутренние механизмы Memcached и Redis, чтобы продемонстрировать структуру надежных хранилищ типа ключ/значение.

Memcached – это распределенная высокопроизводительная система кэширования объектов. Она чрезвычайно популярна и используется в высокопроизводительных системах с высоким трафиком. Memcached используется на таких ресурсах как Facebook, Twitter, Википедия и YouTube. Memcached является простой системой с небольшим набором функций. Например, нет поддержки резервного копирования, отказоустойчивости, или восстановления. Memcached имеет простой API и может использоваться совместно с любым языком веб-программирования. Основная цель использования Memcached в стеке приложений – уменьшить нагрузку на базу

данных. На рис. 3.15 представлена возможная конфигурация для Memcached для веб-приложения.

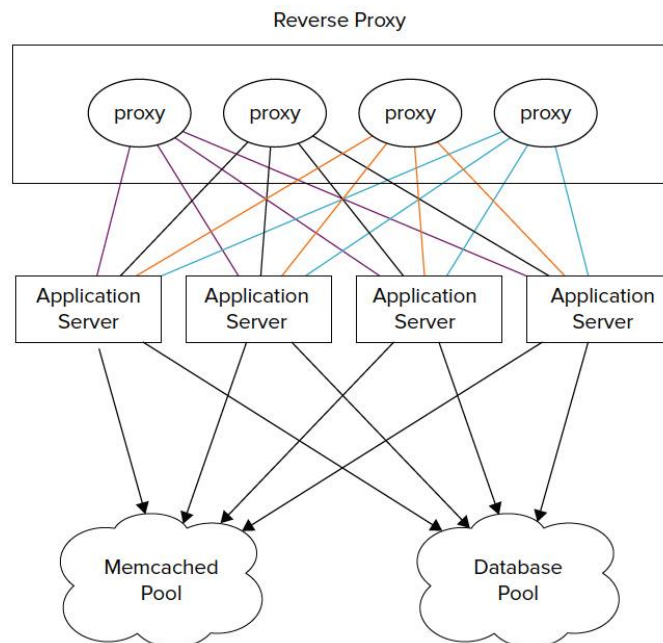


Рисунок 3.15 – Конфигурация Memcached

Центральным ядром Memcached является распределитель slab. Memcached сохраняет данные в slab-памяти. Slab состоит из страниц, которые распределяются по чанкам или корзинам. Наименьший размер slab равен 1 КБ, размер может расти кратно множителю 1,25. Memcached может хранить данные размером до 1 МБ. Доступ к данным осуществляется по ключу. Размер ключа – до 250 байт. Каждый кэшированный объект хранится в чанке ограниченного размера. Это означает, что объект размером 1,4 КБ будет храниться в чанке размером 1,5625 КБ ($1,25^2$). Безусловно, такое поведение приводит к затратам памяти, особенно в случаях, когда объекты ненамного превосходят размер чанка. По умолчанию, Memcached использует всю доступную память, размер ограничивается только архитектурой базового хранилища. На рис. 3.16 схематично представлен принцип размещения данных в Memcached.

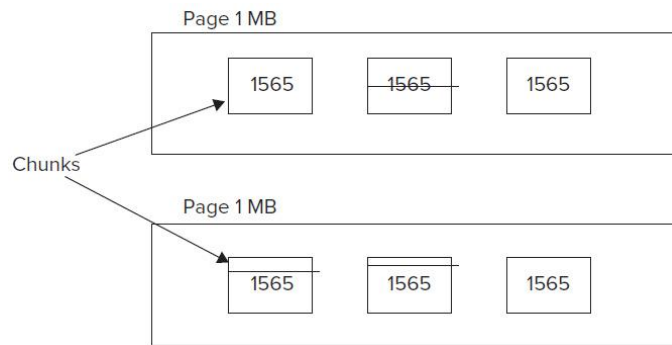


Рисунок 3.16 – Конфигурация Memcached

Алгоритмы LRU отвечает за удаление устаревших данных из кэша. Алгоритмы LRU основывается на работе фрагментированной slab-памяти. Фрагментация появляется при хранении и очистке памяти от объектов. Перераспределение памяти решает часть этой проблемы.

Memcached – это система кэширования объектов, которая не выполняет организацию элементов данных в коллекциях, таких как списки, наборы, отсортированные наборы, или карты. С другой стороны, Redis обеспечивает поддержку всех этих структур данных. Redis – это система подобная Memcached, но более надежная.

Все, что есть в Redis представляется в виде строки. Даже коллекции типа списков, множеств, сортированных множеств и карт, представлены в виде строк. В Redis определена специальная структура данных, называемая SDS (simple dynamic string). Эта структура состоит из трех частей:

- 1) buff – символьный массив для хранения строк;
- 2) len – численное значение, указывающее на длину массива buff;
- 3) free – количество дополнительных байт, доступных для использования.

Данные Redis располагаются в первичной памяти, перемещение их на диск происходит в определенные моменты. В отличие от MongoDB, эта система не использует memory-mapped файлы. Вместо этого, в Redis реализован собственная подсистема виртуальной памяти.

Дополнительно, в Redis реализован менеджер виртуальной памяти, который координирует неблокирующие сокетные операции. Архитектура Redis показана на рис. 3.17.

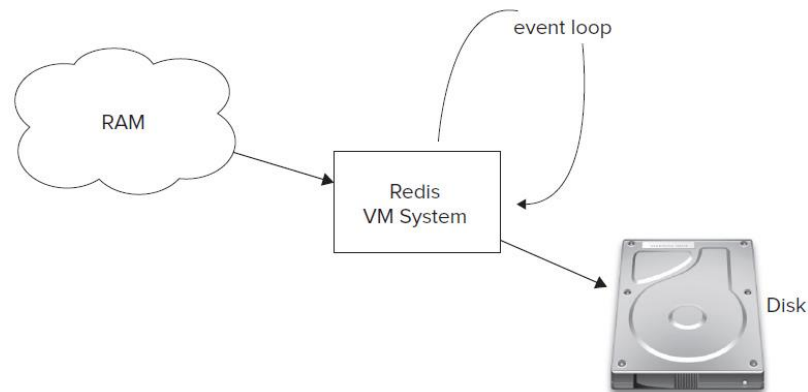


Рисунок 3.17 – Схема архитектуры Redis

Система Redis не связана с swar-системой операционной системы, потому что:

1. Объекты Redis не реализуют маппинг один-к-одному с swar-страницами. Swar-страницы имеют размер 4,096 байт, а объекты Redis могут занимать более одной страницы или в одну swar-страницу могут быть помещены более одного объекта. В системе Redis, даже в случае низкого процента востребованных объектов в памяти, остаются доступными большое количество swar-страниц. Система swar операционной системы следит за использованием объектов (востребованностью данных). То есть, данные удаляются из swapping-системы даже если востребован один байт в странице.

2. В отличие от MongoDB, система Redis работает с различными форматами данных в памяти и на диске. Данные на диске хранятся в более компактном сжатом формате.

3.4 Частично согласованные нереляционные базы данных

В то время как Bigtable от Google рассматривается как реализация столбцовой базы данных, Dynamo от Amazon представляется собой частично согласованное хранилище. Основные идеи реализации подобной системы

были представлены в 2007 на симпозиуме по принципам функционирования операционных систем. Теоретические идеи Dynamo были реализованы на практике в таких системах как Apache Cassandra, Voldemort, Riak и Dynomite.

Amazon Dynamo является ядром для различных сервисов Amazon, которые реализуют единую систему e-commerce. Эта система предъявляет особые требования: высокая доступность и устойчивость к сбоям.

Наборы данных в Dynamo являются структурированными; выполнение запросов, основанных на первичных ключах покрывают большинство потребностей. Реляционные связи в таких системах не используются. Система Dynamo построена на идеи согласованного хэширования, версификации объектов.

Система Dynamo поддерживает простой интерфейс для хранилища данных (get-and-put). Запрос Put включает в себе данные, связанные с версией объекта, которые сохраняются в контексте. Dynamo построена таким образом, чтобы обеспечивать инкрементальное масштабирование по мере роста объемов данных. Дополнительно, система использует согласованное хэширование для более эффективного распределения.

3.4.1 Согласованное хэширование

Согласованное хэширование – это важный принцип при работе с hash-таблицами. В данном подходе добавление или удаление слотов не оказывает существенного влияния на отображение ключей и слотов. Рассмотрим элементарную схему хэширования.

Размер базы данных ограничен не только по размеру. Дополнительные ограничения лимитируют размер каждого документа и количество коллекций сервера MongoDB. Документ не может быть больше, чем 8 МБ. Это, очевидно, означает, что использование MongoDB для хранения больших элементов не подходит. При хранении больших документов необходимо использовать GridFS. Кроме того, существует ограничение на количество пространств имен, которые могут быть определены в экземпляре базы данных. По умолчанию,

количество пространств имен – 24000. Каждая коллекция и каждый индекс использует пространства имен. Соответственно, два индекса на коллекцию позволит организовать максимум 8000 коллекций на одну базу данных. Как правило, такое количество достаточно. Тем не менее, если нужно, то можно увеличить количество пространств имен до числа, превышающего 24000.

Увеличение размера пространства имен влечет определенные последствия. Каждое пространство имен коллекций использует несколько килобайт. В MongoDB индекс реализован в виде В-дерева. Каждая страница В-дерева имеет размер 8 КБ. Таким образом, добавление дополнительных пространств имен, для коллекций или индексов, предполагает добавление несколько килобайт для каждого дополнительного экземпляра. Пространства имен для имени MyDB базы данных MongoDB сохраняются в файле с именем mydb.ns. Файл mydb.ns может вырасти до максимального размера 2 ГБ. Ограничение размера mydb.ns ограничивает и рост базы данных – понимание этого намного важнее чем знание моделей поведения коллекций и индексов.

Очень часто спорят о необходимости перехода на автоматические системы управления большими данными: просят привести аргументы необходимости перехода к Hadoop и технологиям Big Data. Области, которые предполагают использование датчиков и интеллектуальных контроллеров в реальных сферах бизнеса и производства, не могут обойтись без подобных систем. Наличие большого потока данных от датчиков приводит к необходимости высокопроизводительного анализа данных, получаемых с IoT-устройств.

3.4.2 Версионирование объектов

В большой распределенной и масштабируемой системы, Acid-транзакции накладывают огромные накладные расходы. Так что "Динамо" предлагает управление версиями объектов и векторных часов для поддержания согласованности. Давайте попробуем разобраться, как вектор, Часы работы с помощью примера.

Будем считать, что четыре хакеров, Джо, Хиллари, Эрик, и Аджай, решили встретиться и поговорить о векторе часы. Джо предполагает, что они все встретились в пало-Альто. Потом Хиллари и Эрик встречаются на работе и решают, что горы могут быть лучшим местом для встречи. В тот же день, Эрик и сообщение Аджай друг друга и сделать вывод, что встречи в Лос-Альтос может быть лучшей идеей. Когда день встречи приезжает, Джо электронные письма всем встретиться-напоминание и адрес отель в пало-Альто. Хиллари отвечает, что сменилось место проведения, чтобы посмотреть на горы и Эйджей говорит, что это Лос-Альтос. Оба утверждают, что Эрик знает решение. Теперь Эрик связаться, чтобы решить вопрос. На данном этапе вы можете создать векторные часы для разрешения конфликта.

Вектор часы могут быть созданы для каждого из трех значений за тем, пало-Альто, Маунтин-вью, Лос-Альтос, следующим образом: Venue: Palo Alto:

Vector Clock: Joe (ver 1).

Venue: Mountain View:

Vector Clock: Joe (ver 1), Hillary (ver 1), Eric (ver 1).

Venue: Los Altos:

Vector Clock: Joe (ver 1), Ajay (ver 1), Eric (ver 1).

Векторные часы для Маунтин-вью и Лос-Альтос включают первоначально выбор Джо, потому что всем было известно. Векторные часы для вид на горы базируется на ответ Хиллари и векторных часов для Лос-Альтос базируется на ответ Аджая. Вид на горы и Лос-Альтос векторные часы не синхронизированы, потому что они не спускают друг от друга. Вектор часы должен быть версии больше или равно всех значений в другой вектор часы, чтобы сходиться с нее.

Наконец, Джо завладеет Эрик по телефону и просит его разрешить недоумение. Эрик понимает проблемы и быстро решает, что встреча в Маунтин-вью, вероятно, лучшая идея. Теперь Джо вытягивает обновленные векторные часы следующим образом: Venue: Palo Alto:

Vector Clock: Joe (ver 1).

Venue: Mountain View:

Vector Clock: Joe (ver 1), Hillary (ver 1), Ajay (ver 0), Eric (ver 2).

Venue: Los Altos:

Vector Clock: Joe (ver 1), Hillary (ver 0), Ajay (ver 1), Eric (ver 1).

Версия 0 создана для Хиллари и Аджай в векторе часов на площадках они не предложили, но теперь в курсе. Теперь, векторные часы спускаются друг от друга и видом на горы является местом проведения встречи. Из примера, вы бы заметили, что вектор часы не только помогают определить порядок событий, но и помочь в решении каких-либо несоответствий путем выявления коренных причин этих проблем.

Независимо от объекта управления версиями, Динамо использует полученные на основе членства для узлов и использует намекнул передачу последовательности.

Протокол сплетни-это стиль коммуникационного протокола, выполненная в виде сплетен или слухов в социальных сетях и офисах. Протокол связи включает в себя периодические сплетни, пара мудрых, межпроцессных взаимодействий. Надежность невелика и коллегиального выбора часто является случайной.

Заключение

В данном разделе рассмотрены базовые принципы построения систем NoSQL. Рассмотрены простые модели данных, схемы хранения и конфигурации, применяемые в распространенных NoSQL-хранилищах. Представлены типичные компоненты столбцовых баз данных, для демонстрации принципов функционирования использовалась база данных HBase.

Были рассмотрены внутренние механизмы функционирования баз данных различного типа: документо-ориентированные и ключ/значение.

Были изложены теоретические основы построения частично согласованных баз данных.

Контрольные вопросы

1. Опишите ключевые принципы построения реляционных баз данных.
2. Каковы главные отличия столбцовых баз данных и РСУБД?
3. Поясните содержание терминов: «встроенные карты», «пара ключ/значение».
4. В каких приложениях традиционно применяется Webtable?
5. Опишите обычную конфигурацию HBase.
6. Что такое сервер диапазонов в HBase?
7. Что такое регион в HBase?
8. Опишите структуру уникального идентификатора в MongoDB.
9. Что такое Memory-Mapped файл?
10. Опишите ключевые принципы согласованного хэширования.

РАЗДЕЛ 4. ФРЕЙМВОРК РАСПРЕДЕЛЕННОГО ПРОГРАММИРОВАНИЯ

Hadoop является свободно распространяемым фреймворком для разработки и выполнения распределенных приложений, обрабатывающих большие объемы данных. Распределенные вычисления – это достаточно широкое понятие (и постоянно изменяющееся), но применительно к Hadoop можно сформулировать следующие базовые концепции:

1. Доступность. Hadoop выполняется на кластерах общедоступных ЭВМ или в рамках облачных вычислительных сервисов таких как Elastic Compute Cloud (EC2) от Amazon.

2. Надёжность. Так как используются доступные ЭВМ, Hadoop разрабатывался с учетом возможных отказов оборудования. Система способна обрабатывать подобные отказы достаточно эффективно.

3. Масштабирование. Масштабирование Hadoop является линейным по отношению к росту объемов обрабатываемых данных. Масштабирование осуществляется путем добавления новых узлов кластера.

4. Простота. Hadoop позволяет пользователям быстро создавать эффективный параллельный программный код.

Доступность и простота Hadoop позволяют создавать и запускать масштабные распределенные решения. Даже студент уровня магистра может достаточно быстро, с минимальными затратами реализовать кластер Hadoop. Но с другой стороны, надежность и масштабируемость делают данную технологию приемлемой для таких высоконагруженных систем как Yahoo и Facebook. Такие особенности Hadoop делают его популярной технологией как в академической среде, так и в промышленном использовании.

4.1 Основы Hadoop

На рис. 4.1 демонстрирует схема кластера Hadoop. Очевидно, что Hadoop-кластер – это множество ЭВМ массовой доступности, объединенных сетью, расположенных, чаще всего, в едином центре обработки данных. Хранилище данных и вычисления сосредоточены внутри своеобразного облака на базе машин кластера. Различные пользователи могут посылать работы кластеру, располагаясь при этом удаленно относительно кластера.

Закон Мура все также продолжает действовать, но увеличение размеров серверов (и стоимости) в настоящее время не является единственным решением для решения проблем в области больших данных. Другим популярным направлением развития является использование множества дешевых, серийных ЭВМ, функционирующих как единое целое в рамках распределенной системы.

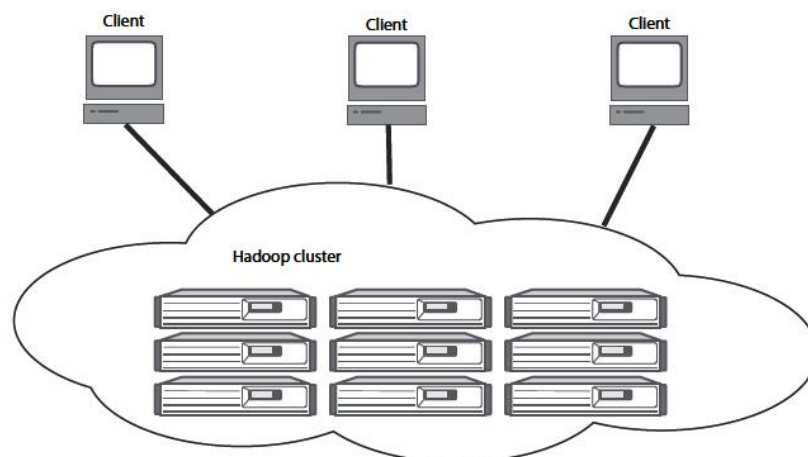


Рисунок 4.1 – Схема кластера Hadoop

Для лучшего понимания причин популярности распределенных систем (по сравнению с огромными высокопроизводительными серверами) рассмотрим стоимость текущих технологий ввода-вывода. Дорогие сервера с четырехканальной системой ввода-вывода демонстрируют пропускную способность 100 МБ/с. Эта скорость передачи данных позволяет считать 4 ТБ данных за 1 час. В системе Hadoop аналогичный набор данных разделяется на отдельные блоки (обычно по 64 МБ), распределяемые по машинам кластера с использованием HDFS (Hadoop Distributed File System).

При наличии репликации, кластер ЭВМ может читать данные параллельно и обеспечивает более высокую пропускную способность. Подобная конфигурация серийных компьютеров оказывается дешевле, чем применение одного высокопроизводительного сервера.

Рассмотренные аспекты демонстрирует эффективность применения Hadoop, особенно в сравнении с традиционными высокопроизводительными серверами. Можно сравнить Hadoop с другими распределенными системами обработки информации. Система SETI@home использует программу экранной заставки для объединения усилий множества компьютеров по всему миру в поиске внеземной жизни. В системе SETI@home центральный сервер хранит оцифрованные радиосигналы из космоса и передают их через Интернет на клиентские ПК для поиска аномальных признаков. Этот подход перемещает данные на рабочее место клиента (к программе-заставке). После расчета результаты возвращаются на сервер.

Hadoop и отличается от подобных систем, таких как SETI@Home, прежде всего, своей концепцией работы с данными. В рамках SETI@Home данные передаются между сервером и многочисленными клиентами. Этот принцип хорошо работает в системах с высокой вычислительной интенсивностью, но для систем с высокой интенсивностью изменения данных объем пересылаемых данных становится слишком большим. Hadoop перемещает код, обрабатывающий данные, к набору данных, а не наоборот. На рис. 4.1 видно, что в кластере Hadoop присутствуют как сами данные, так и код для их обработки. Клиенты отправляют на сервер только MapReduce-программы, которые, как правило, небольшие (часто порядка килобайт). Еще более важным принципом Hadoop является следование философии «код доставляется к данным» внутри самого кластера Hadoop. Данные разбиваются на блоки и распределены по кластеру, и, по возможности, вычисления реализуются на том же узле кластера, на котором находятся требуемые исходные данные.

Философия «код доставляется к данным» имеет большое значение для типа вычислений, реализуемого в Hadoop – интенсивного использования данных. Исполняемых программ (кодов) на порядок меньше чем данных, а также они занимают существенно меньший объем, что существенно облегчает их пересылку. Пересылка данных занимает гораздо большее время.

Hadoop заполняет рыночную нишу IT-рынка, связанную с системами эффективного хранения и вычислениями на основе больших объемов данных. Распределённая система Hadoop включает распределенную файловую систему, позволяющую программам выполняться параллельно на кластере машин (рис. 4.2).

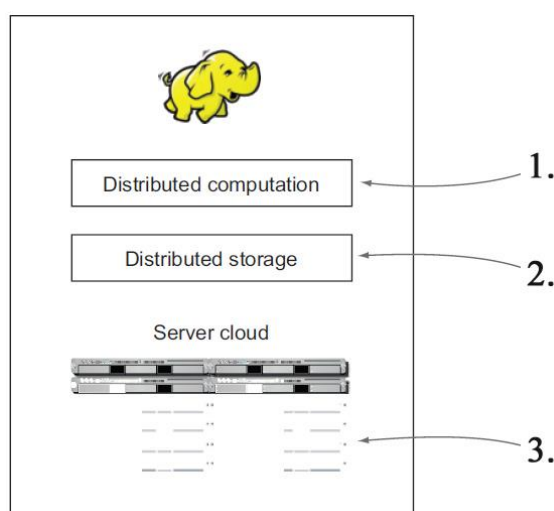


Рисунок 4.2 – Окружение Hadoop, развернутое на кластере доступных ЭВМ

На рис. 4.2:

1. Вычислительный уровень представлен планировщиком заданий и фреймворком распределенного программирования MapReduce.
2. Хранилище данных на основе HDFS.
3. Аппаратная часть на основе доступных компьютеров.

В системе Hadoop выполняются процессы в рамках парадигмы MapReduce.

Hadoop является платформой для распределенного хранения данных и вычислений. Это решение было предложено для устранения проблем масштабируемости, существующих в системе Nutch (проект с открытым

исходным кодом). Одновременно, компанией Google были опубликованы документы, описывающие распределенную файловую систему GFS (Google File System) и MapReduce (технология параллельного программирования). Успешная реализация концепций Google в проекте Nutch привела к появлению двух новых проектов: Hadoop и Apache.

В данном подразделе рассматриваются архитектурные особенности Hadoop, исследуются сферы его промышленного применения и анализируются недостатки данного решения.

Hadoop, как показано на рис. 4.3, использует архитектурный шаблон master-slave и состоит из следующих компонентов:

1. Распределенная файловая система Hadoop (Hadoop Distributed File System, HDFS).
2. Представленный в Hadoop второй версии планировщик и менеджер ресурсов YARN (Yet Another Resource Negotiator). Все YARN-приложения запускаются на кластере Hadoop.
3. MapReduce – пакетно-ориентированный вычислительный движок.

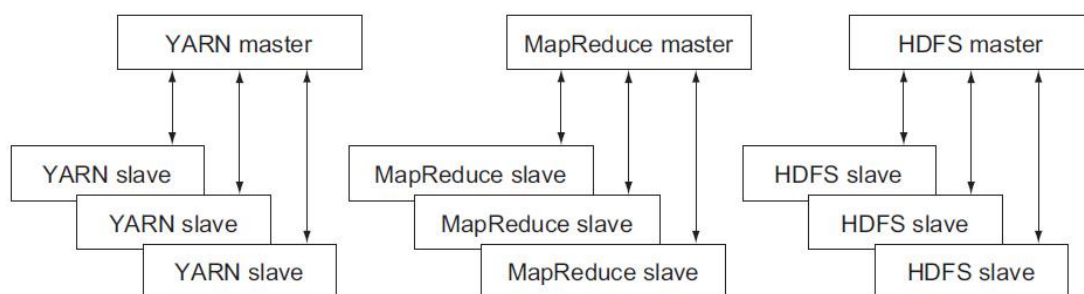


Рисунок 4.3 – Архитектура высокого уровня Hadoop 2

На рис. 4.3 показаны следующие компоненты:

1. Процесс YARN-master выполняет распределение работ для приложений YARN.
2. Элемент MapReduce-master отвечает за распределение вычислительной нагрузки по slave-узлам кластера.
3. HDFS-master отвечает за распределение данных по узлам кластера, данный компонент отслеживает местоположение данных.

Основные характеристики Hadoop: распределение данных и параллельные вычисления над большими массивами данных. Возможности хранения и вычислительные возможности Hadoop масштабируются путем добавления новых узлов в кластер Hadoop; кластер с сотней узлов способен просто обрабатывать объемы данных в несколько петабайт.

4.2 Элементы кластера Hadoop

4.2.1 HDFS

HDFS – это компонент хранения данных для Hadoop. Это распределенная файловая система, которая была спроектирована на замену GFS от Google. HDFS спроектирована для обеспечения высокой пропускной способности и демонстрирует высокую эффективность при чтении и записи больших файлов данных (гигабайт и больше). Для обеспечения высокой пропускной способности, HDFS использует необычно большой (для файловой системы) размер блока и оптимизацию размещения данных для сокращения расходов на сетевые операции ввода/вывода.

Масштабируемость и высокая доступность также являются ключевыми особенностями HDFS, которые достигаются благодаря репликации данных и механизмам защиты от сбоев. HDFS реплицирует файлы с настраиваемой кратностью, при этом обеспечивается защита от программных и аппаратных сбоев, а также автоматическое перераспределение данных при отказе узлов кластера.

На рис. 4.4 показана схема взаимосвязи логических элементов HDFS: NameNode и DataNode. Также показано приложение, которое задействует библиотеку Hadoop filesystem library для доступа к HDFS.

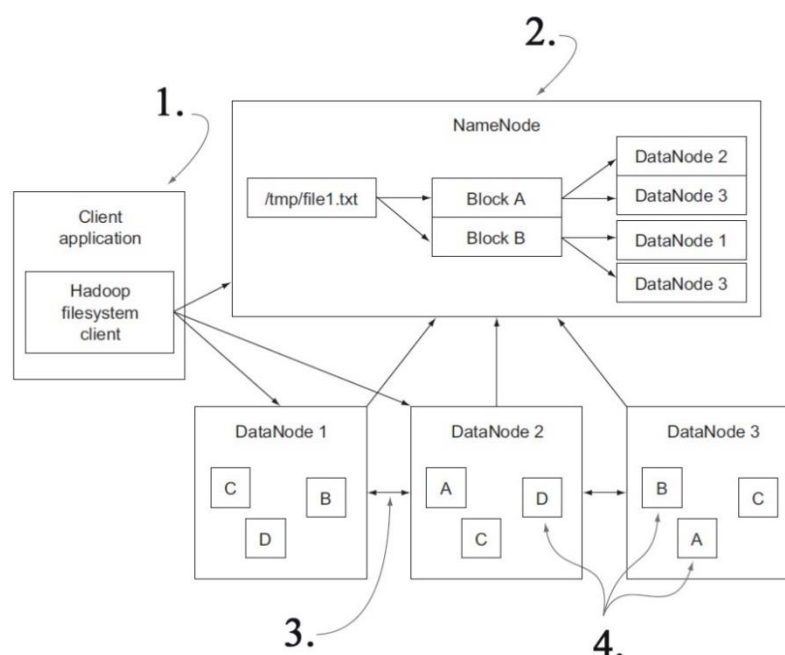


Рисунок 4.4 – Архитектура кластера HDFS в Hadoop 2

На рис. 4.4 показаны следующие элементы:

1. HDFS-клиент обращается к узлу NameNode для запроса метаданных, а также к узлам DataNodes для получения данных.
2. Узел NameNode содержит в памяти метаданные о файловой системе: каким образом узлы DataNodes распределяют блоки файлов данных.
3. Узлы DataNodes взаимодействуют друг с другом для управления файлами.
4. Файлы разбиваются на блоки, каждый файл может быть многократно реплицирован (по умолчанию каждый блок реплицируется трижды).

В Hadoop 2 были представлены две важные технологии для HDFS: HDFS Federation и High Availability (HA).

1. Federation позволяет распределять метаданные HDFS по множеству узлов NameNode. Такое поведение повышает масштабируемость HDFS и обеспечивает изолированность, позволяя различным приложениям запускать собственные NameNode без опасения нанесения вреда другим узлам NameNode этого же кластера.

2. High Availability исключает критическую уязвимость в HDFS, которая существовала в Hadoop 1, когда отказ единственного NameNode мог привести

к отказу всего кластера. HDFS НА также обеспечивает механизм автоматического восстановления кластера.

4.2.2 YARN

YARN – это автоматический планировщик ресурсов кластера. Эта технология была представлена в Hadoop версии 2. Технология была нацелена на устранение архитектурных недостатков Hadoop 1:

1. При развертывании кластера с более чем 4000 узлов возникали проблемы масштабируемости, добавление дополнительных узлов не давало эффекта роста производительности.

2. Поддерживалась только MapReduce-модель программ. Это означало, что кластер не был рассчитан на такие задачи как машинное обучение, где алгоритмы носят итерационный характер.

2. Only MapReduce workloads were supported, which meant it wasn't suited to run execution models such as machine learning algorithms that often require

В Hadoop 2 данные проблемы были разрешены путем передачи функций распределения от MapReduce и его переработки в планировщик, получивший название YARN. С такими изменениями кластер Hadoop больше не ограничивается запуском только MapReduce-программ. YARN делает возможными различные типы загрузок кластера и поддерживает альтернативные модели вычислений, такие как обработка графов, потоковая обработка.

YARN обладает простой архитектурой, так как его функции ограничены: этот компонент просто планирует и управляет ресурсами кластера Hadoop.

Рисунок 4.5 демонстрирует взаимосвязь компонентов YARN; ResourceManager и NodeManager. Также показаны специфические компоненты приложения YARN: клиент YARN-приложения, ApplicationMaster и контейнер. Hadoop 2 предоставляет другое важное нововведение: возможность создавать контейнеры в различных конфигурациях.

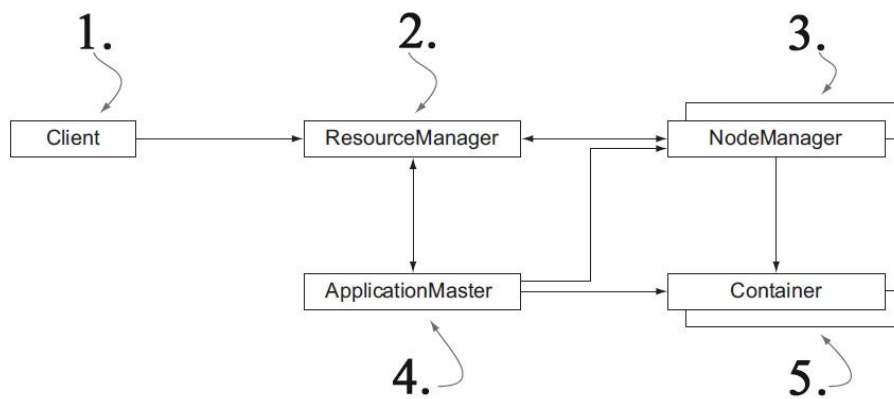


Рисунок 4.5 – Взаимодействия компонентов YARN

На рис. 4.5 показана схема характерных взаимодействий между компонентами ядра YARN и компонентами YARN-приложения:

1. Клиент YARN отвечает за создание YARN-приложения.
2. ResourceManager – это master-процесс YARN, отвечающий за управление и планирование ресурсов, называемых контейнерами.
3. NodeManager – это slave-процесс YARN, который запускается на каждом узле. Процесс отвечает за запуск и управление контейнерами.
4. ApplicationMaster создается процессом ResourceManager и отвечает за выполнение специфической работы: запрашивание контейнеров.
5. Контейнер – это специализированный процесс YARN, реализующий функционал приложения.

В Hadoop 1 существовало понятие «слот», которое отражало фиксированное количество map- и reduce-процессов, которые запускаются на одном узле. Это было расточительно с точки зрения использования кластера. Как результат появлялись неиспользуемые ресурсы и ограничения на объем используемой памяти для процессов map и reduce. С YARN каждый контейнер запрашивает у ApplicationMaster свой объем памяти и процессорное время. Это дает YARN полный контроль над ресурсами, необходимыми для выполнения работы.

4.2.3 MapReduce

MapReduce – это фреймворк распределенных вычислений, созданный на основе технологий, описание которых было опубликовано в работах Google. Технология позволяет распараллелить работу по обработке большого объема исходных данных, например, объединение логов веб-сайта и данных из OLTP-базы данных для построения модели, объясняющей каким образом пользователи взаимодействуют с веб-ресурсом. Выполнение подобной задачи может занять несколько дней при использовании традиционных подходов, но время может быть сокращено до нескольких минут с помощью MapReduce в Hadoop-кластере.

Модель параллельной обработки данных MapReduce за счет абстрагирования позволяет исключить большинство проблем, возникающих при работе с распределенными системами, такими как необходимость распараллеливания, распределение нагрузки, а также ненадежность оборудования и программного обеспечения. Используя высокий уровень абстракции в рамках парадигмы MapReduce программист может сосредоточиться на удовлетворении потребностей бизнеса, а не на нюансах построения распределенной системы. MapReduce расщепляет задачу, представленную клиентом на мелкие параллельные подзадачи типа map и reduce как показано на рис. 4.6. Конструкции Map и reduce заимствованы из функционального языка программирования Лисп; они используют модель, которая исключает любые взаимозависимости между параллельными подзадачами, которые могли бы повлечь необходимость синхронизации и обмена состояниями.

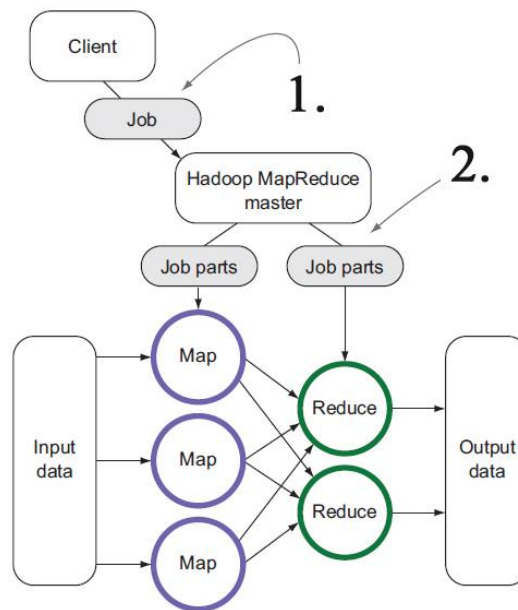


Рисунок 4.6 – Распараллеливание задачи MapReduce

На рис. 4.6 показаны различные этапы разделения задачи на уменьшенные подзадачи типа map и reduce:

1. Клиент запускает на выполнение работу MapReduce.
2. MapReduce выполняет декомпозицию работы на map- и reduce-подзадачи, а также распределяет их по slave-узлам кластера, где они и будут выполняться.

Задача программиста сводится к разработке map- и reduce-функций, причем map-функция на выходе отдает пары ключ/значение, а reduce-функция использует их для получения результата.

Мар-функция получает на вход пары ключ/значение, которые представляют логические записи из источника данных. Если источник файловый, то под записью понимается строка из файла, а если источник таблица – строка таблицы.

С помощью мар-функции получают ноль и более выходных пар ключ/значение на каждую входную пару. Например, если мар-функция является функцией фильтрации, то возврат будет в виде множества пар, удовлетворяющих условию фильтрации. Или, это может быть операция демультиплексирования, когда входная пара порождает на выходе несколько пар.

С появлением YARN в Hadoop 2, MapReduce был переработан и выполняется в качестве YARN-приложения; также используется название MapReduce 2 (или MRv2). С точки зрения разработчика MapReduce в Hadoop 2 работает в большей степени без изменений, а код, написанный для Hadoop 1, будет выполняться и в Hadoop версии 2.

4.3 Дистрибутивы и экосистема Hadoop

4.3.1 Экосистемы Hadoop

Экосистема Hadoop разнообразна и растет постоянно. Невозможно отслеживать все различные проекты, которые связаны с Hadoop в той или иной форме. В пособии акцент делается на средства, которые в настоящее время получают наибольшее распространение среди пользователей, как показано на рис. 4.7.

Термин «доступное оборудование» (commodity hardware) часто используется при описании Hadoop и требований к оборудованию. Это правда, что системы Hadoop могут работать на любых старых серверах, но все таки производительность оборудования оказывает влияние на работу Hadoop, а также старое оборудование может увеличить количество простоев из-за ремонта. Следовательно, «доступное оборудование» описывает серверное оборудование среднего ценового диапазона с двумя процессорными слотами, с технологиями исправления ошибок памяти, SATA-дисками, оптимизированными для RAID-хранилища. Применение RAID на узле кластера DataNode, на котором развернута HDFS, крайне не рекомендуется, так как HDFS уже предполагает встроенный механизм репликации и механизм исправления ошибок. На узле NameNode поддержка RAID наоборот рекомендуется для повышения безопасности. HDFS использует диски для длительного хранения метаданных о файловой системе.

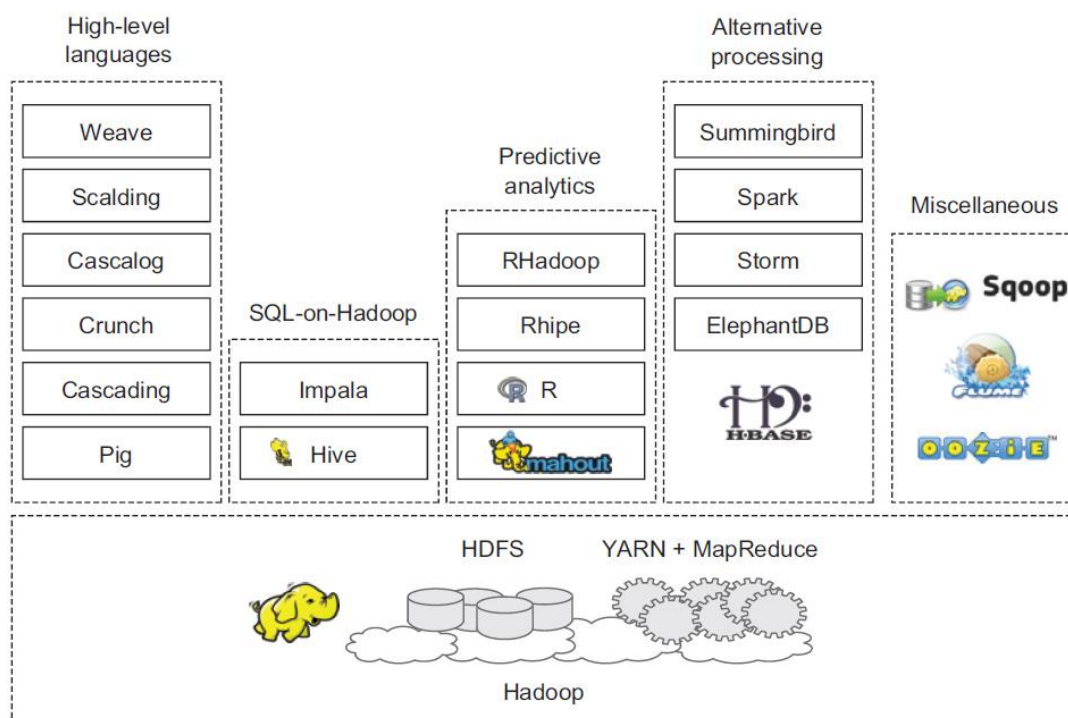


Рисунок 4.7 – Hadoop смежные технологии

С точки зрения архитектуры вычислительной сети, коммутаторов, межсетевых экранов, все узлы кластера должны иметь возможность соединения по открытым каналам. Для небольших кластеров все компьютеры должны быть оснащены сетевой картой с пропускной способностью 1 Гб и подключены к коммутатору высокого качества. Для больших кластеров используются коммутаторы на 10 Гб. Клиентские узлы должны иметь возможность обращаться ко всем master- и slave-узлам, но, если требуется, доступ к ним может быть ограничен фаерволом.

4.3.2 Дистрибутивы Hadoop

Hadoop – это проект с открытым исходным кодом от Apache, регулярные обновления которого доступны непосредственно на сайте компании (<http://hadoop.apache.org>). Пользователь имеет возможность загрузить и установить Hadoop непосредственно с веб-сайта или использовать виртуальную машину из платных дистрибутивов. Такой подход позволяет максимально быстро научиться использовать передовые технологии распределенных вычислений. После того, как пользователь определился ч

технологией и принял решение об использовании Hadoop, встает вопрос о выборе конкретного дистрибутива. Можно продолжить использование бесплатной версии Hadoop, но тогда потребуется задействовать собственных специалистов для управления кластером. Это не совсем простая задача, ее реализация успешна только в компаниях, обладающих командой инженеров с соответствующей квалификацией, которые заняты управлением кластерами.

В качестве альтернативного решения можно выбрать коммерческий дистрибутив Hadoop, который предполагает наличие дополнительного программного обеспечения для администрирования; предполагается также наличие технической поддержки. Безусловно, все дополнительные опции являются платными, но если вы запускаете высокопроизводительный сервис и не обладаете требуемой командой специалистов для поддержки инфраструктуры и сервисов, тогда коммерческая реализация Hadoop является разумным решением.

Настоятельно рекомендуется проводить консультации с основными поставщиками, чтобы получить точное понимание о возможностях, предоставляемых дистрибутивом, а также о том, насколько полно предлагаемое решение будет удовлетворять будущим требованиям, насколько эффективно решение по стоимости, наличие поддержки и т.п. Каждый поставщик будет выделять преимущества своих решений и в то же время указывать на недостатки продуктов конкурентов, так что вести переговоры с двумя или несколькими поставщиками позволит получить более реалистичное представление о дистрибутивах. Необходимо убедиться, что дистрибутив удовлетворяет всем бизнес-требованиям и обеспечивает поддержку применяемых программных и аппаратных средств; в этом можно убедиться воспользовавшись пробными версиями.

Существует целый ряд дистрибутивов на выбор, и в этом разделе будут рассмотрены наиболее популярные, а также проанализированы их преимущества.

4.3.3 Apache

Apache является организацией, которая поддерживает код ядра Hadoop и распространяет его, а так как весь код является открытым, вы имеете возможность просматривать его в предпочитаемой IDE, изучать внутренне устройство Hadoop. Основная проблема с дистрибутивами Apache – ограниченная поддержка: нет никакой гарантии, что ваш запрос будет принят и разобран. Но современное сообщество Hadoop достаточно развито и прибегнув к коллективному обсуждению найти решение проблемы можно гораздо быстрее чем ждать исправлений от службы поддержки.

Дистрибутив Apache Hadoop стал гораздо дружелюбнее, так как появился инструментарий Apache Ambari – инструмент для администрирования в графическом режиме, который помогает устанавливать и управлять кластерами. Хотя и существуют такие инструменты как Ambari, коммерческие дистрибутивы предлагают инструменты с более мощными инструментами.

4.3.4 Cloudera

Cloudera является самым штатным дистрибутивом Hadoop, и на него приходится большое количество установок. Дуг Каттинг, который вместе с Майком Каферелла создали Hadoop, является главным архитектором на Cloudera. Это означает, что исправления и пожелания имеют больше шансов быть рассмотренными в Cloudera по сравнению с Hadoop. Кроме поддержки непосредственно Hadoop, Cloudera внедряет новшества в данной области путем разработки проектов, исправляющих слабые места Hadoop. Один из ярких проектов является проект Impala, который предлагает систему SQL-on-Hadoop, аналогичный Hive, но фокусирующий основное внимание на практически мгновенном учете пользовательских пожеланий и опыта разработчиков (в противоположность Hive). Существует большое количество различных проектов, поддерживаемых в рамках Cloudera: Flume (система

распределения и хранения логов), Sqoop (перенос реляционных данных в систему Hadoop), Cloudera Search (предлагает механизм индексирования данных в реальном времени).

4.3.5 Hortonworks

Hortonworks также достаточно популярный проект. Как и Cloudera он предлагает большое количество преимуществ в области поддержки.

С точки зрения развития, Hortonworks немного отличается от Cloudera. Например, что касается Hive: если в Cloudera пошли по пути разработки полностью нового решения SQL-on-Hadoop, то в Hortonworks использовали исходный код Hive и удалили недостатки, связанные с высокой латентностью, добавив дополнительные возможности, связанные, например, с поддержкой ACID. Hortonworks является двигателем в области проектирования нового поколения платформы YARN, которая является ключевой технологией Hadoop. В Hortonworks используется Apache Ambari для администрирования. Данный проект уделяет основное внимание разработке и расширению экосистемы Apache, что несет огромные преимущества всему сообществу разработчиков, так как позволяет всем использовать инструментарий без необходимости заключения контрактов.

4.3.6 MapR

MapR не имеет такого огромного сообщества разработчиков, как ранее рассмотренные проекты. Поэтому возможности проект в области внесения исправлений и определения стратегии развития Hadoop ограничены.

С точки зрения перспектив развития, MapR использует явно альтернативный подход к поддержке Hadoop. С самого начала проекта, было решено, что HDFS не подходит для корпоративного хранилища; вместо этого была разработана собственная распределенная файловая система, которая обладает достаточно интересными возможностями, такими как POSIX-согласованность (поддержка прямого доступа для записи и атомарных

операций), высокая доступность, мониторинг NFS, зеркалирование данных, мгновенные снимки.

Некоторые из этих возможностей стали доступны в Hadoop 2, но MapR обладал этими возможностями с самого начала, что делает реализацию функций в MapR достаточно надежной. Следует отметить, что отдельные части стека технологий MapR, такие как файловая система или реализация HBase, являются закрытыми проектами и платными. Это оказывает влияние на возможности инженеров проекта по выявлению ошибок, их исправлению. В противоположность такому подходу, подходы компаний Cloudera и Hortonworks предполагают использование открытого исходного кода.

Эффективные решения от MapR доступны в облаке Amazon в качестве альтернативы Elastic MapReduce от Amazon, также проекты MapR интегрируются с Compute Cloud от Google. В данном подразделе рассмотрены преимущества и особенности MapR лишь поверхностно.

В следующем подпункте обращается внимание на области применения Hadoop.

4.4 Применение и ограничения Hadoop

Hadoop имеет высокий уровень использования в IT-компаниях, данная технология начинает все шире внедряться в различных секторах рынка, включая производство, правительственные учреждения, учреждения здравоохранения.

Компания Facebook использует Hadoop, Hive и HBase для хранения данных и обслуживания приложений в режиме реального времени. Кластеры хранилища Facebook заполнены петабайтами данных, содержат тысячи узлов, используют отдельные кластеры реального времени на основе HBase для отправки сообщений и аналитических вычислений в реальном времени.

Yahoo! использует Hadoop для машинного обучения, аналитических задач, поискового ранжирования, в системах анти спама, оптимизации бизнес

процессов, в ETL-системах и др. Система объединяет более 40 000 серверов с установленным Hadoop с 170 ПБ данных. Yahoo! также запустил первую крупномасштабную YARN-систему с 4000 узлов.

Twitter является главным новатором в области Big Data, компания внесла весомый вклад в развитие проекта Hadoop путем развития таких проектов как Scalding, Scala API для Cascading, Summingbird.

eBay, Samsung, Rackspace, J.P. Morgan, Groupon, LinkedIn, AOL, Spotify, и StumbleUpon, а также другие известные компании также заинтересованы в развитии Hadoop. Microsoft проводит совместные исследования с Hortonworks для развития технологии Hadoop на платформе Windows.

Компания Google в своей публикации, посвященной MapReduce, отметила, что использует Caffeine (собственную версию MapReduce) для создания индексов из исходных данных.

Ежедневно количество организаций, использующих Hadoop, растет и если практически все передовые компании мира используют данную технологию. Очевидно, что технология Hadoop постоянно изменяется, продолжается внедрение данной технологии для решения различных задач.

Высокая доступность и безопасность часто указываются в качестве основных проблем Hadoop. Многие из проблем были решены в Hadoop 2. Рассмотрим некоторые недостатки Hadoop.

Предприятия, использующие Hadoop, имели проблемы с доступностью и безопасностью. В Hadoop 1, все мастер-процессы являются одиночными, а это означает, что сбой мастер-процесса вызывает сбой всего кластера. В Hadoop 2, HDFS рассчитан на высокую готовность системы, а переработка архитектуры MapReduce с заменой на YARN упразднило критическое место технологии. Безопасность Hadoop представляет собой еще одну область, обладающую различными недостатками.

Высокая доступность.

Высокая доступность часто требуется на предприятиях, которые предъявляют высокие требования к времени безотказной работы, при этом

компаниях хотят быть уверены, что все системы работают постоянно, даже в случае отказа узла. До платформы Hadoop 2, мастер-процесс HDFS может выполняться только на одном узле, в результате чего возникает критическая точка отказа. В Hadoop 2 реализована технология HA (High Availability), что позволяет запускать несколько узлов NameNode для кластера Hadoop. В подобной архитектуре один из NameNode-узлов активный, а второй узел находится в ожидании. В случае планового или незапланированного отключения одного из узлов NameNode, другой предотвратит отказ системы.

Множественные центры обработки данных.

Поддержка нескольких центров обработки данных является еще одной технологией, которая все чаще востребована в секторе корпоративного программного обеспечения, так как обеспечивает надежную защиту данных, локализацию данных и их репликацию. Apache Hadoop и большая часть коммерческих дистрибутивов никогда не имели поддержки нескольких центров обработки данных, что создает проблемы для организаций, которые работают с программным обеспечением, предполагающим наличие нескольких ЦОД. WANdisco – это единственное в настоящее время доступное решение Hadoop, поддерживающее множественные ЦОД.

Безопасность.

Hadoop предлагает встроенную модель обеспечения безопасности, которая отключена по умолчанию. С отключенным механизмом безопасности, единственный механизм обеспечения безопасности, который функционирует, – это механизм HDFS по контролю за файлами и каталогами на основе прав и разрешений для пользователей. Но подобная архитектура подвержена атакам. По умолчанию все прочие сервисы Hadoop являются доступными, позволяющими всем пользователям выполнять любые типы операций, например, один пользователь может принудительно завершить задачу MapReduce другого пользователя.

Hadoop можно сконфигурировать для запуска Kerberos – сетевого протокола аутентификации, который позволит обеспечить проверку

подлинности клиентов, пользователей при доступе к компонентам Hadoop. Kerberos может быть интегрирован с Active Directory, развернутой на предприятии. При внедрении Kerberos необходимо убедиться, что любой компонент, взаимодействующий с Hadoop, поддерживает Kerberos.

Физический уровень шифрования может быть сконфигурирован в Hadoop 2, что позволяет передавать зашифрованные данные по сети. Шифрование данных в самом Hadoop не реализовано.

HDFS.

Основными недостатками HDFS являются неспособность данной файловой системы обеспечить высокий уровень доступности (в Hadoop 1.x и более ранних версиях); неэффективная обработка маленьких файлов; отсутствие сжатия. HDFS не поддерживает прямого доступа, поддержка данного типа доступа добавляется.

MapReduce.

Архитектура MapReduce не является удачным выбором в случаях требующих обработки данных в реальном времени. Данные, требующие глобальной синхронизации, или применение шардинга к изменяющимся данным, также неудачное применение MapReduce.

Несовпадение версий.

Выпуск Hadoop 2 связан с некоторыми проблемами в смысле совместимости. Для обеспечения совместимости иногда достаточно перекомпилировать код для API Hadoop 1 под Hadoop версии 2. Существуют также библиотеки, обеспечивающие совместимость различных версий.

Заключение

Hadoop – это распределенная система, спроектированная для обработки, генерации и хранения наборов данных большого размера. Реализация MapReduce обеспечивает отказоустойчивый механизм для анализа данных в гетерогенной среде на базе неструктурированных источников данных.

Технология YARN позволяет выполнять множество неравноправных приложений различными пользователями на одном кластере Hadoop.

Hadoop является инструментом, предоставляющим пользователям доступ к распределенным вычислениям.

Используя распределенное хранилище и доставку кода к данным, Hadoop позволяет избежать долговременной задержки при передаче больших объемов данных по сети. Кроме того, избыточность данных позволяет Hadoop восстановить работу кластера при сбое узла. Создание программ для кластера Hadoop представляется достаточно простым с использованием фреймворка MapReduce.

Контрольные вопросы

1. Как упрощенно объяснить назначение технологии Hadoop?
2. Каковы основные составляющие технологии Hadoop?
3. Опишите назначение HDFS.
4. Что такое YARN?
5. Как можно описать MapReduce? К чему ближе это понятие: технология, фреймворк или библиотека?
6. Какие дистрибутивы Hadoop вы знаете? С какими из них вы работали?
7. Укажите различия дистрибутивов Hadoop.

РАЗДЕЛ 5. YARN

С Hadoop 1 и более ранних версиях разработчики были ограничены только MapReduce. Обработка была достаточно эффективна, если тип работы, которая вы выполнялась на кластере, хорошо вписывается в модель обработки данных MapReduce. Но MapReduce значительно ограничивает разработчиков, которые обрабатывают графы, реализуют итеративные вычисления.

В Hadoop 2 компоненты планирования работ MapReduce были вынесены отдельно и переработаны в новый компонент, называемый YARN (Yet Another Resource Negotiator). YARN не зависит от типа работы, которая выполняется на Hadoop.

5.1 Основы YARN

MapReduce-приложения теперь называются YARN-приложениями. Сравнение нового и старого фреймворка представлено на рис. 5.1.

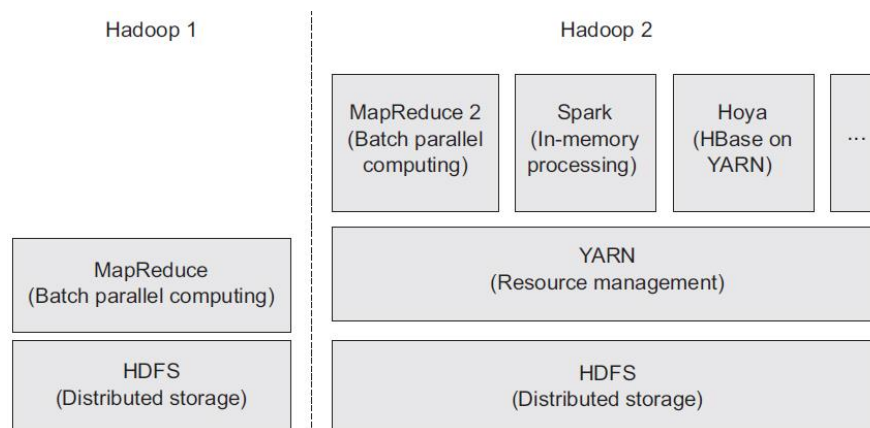


Рисунок 5.1 – Архитектура Hadoop версий 1 и 2

Таким образом, мы можем подвести итог о различиях Hadoop 1 and Hadoop 2:

1. MapReduce является только механизмом исполнения программ в версии 1.
2. HDFS остался неизменным при переходе на новую версию.

3. Hadoop 2 поддерживает любые модели выполнения, включая портирование MapReduce который в новой версии называется YARN.

4. Все механизмы исполнения в Hadoop 2 реализованы как YARN-приложения.

5. Фреймворк YARN обеспечивает распределение работ и нечувствителен к характеру выполняемых заданий.

MapReduce – мощная распределенная платформа и модель программирования, которая функционирует на основе пакетного распараллеливания задач на кластере. Несмотря на то что технология достаточно эффективна, MapReduce имеет некоторые недостатки; главным образом связанные с пакетной обработкой, в результате чего она не подходит для обработки данных в реальном времени. Фактически, YARN представляет собой распределенный планировщик задач, который отвечает за два действия:

1. Отвечает на запросы клиентов для создания контейнеров. Контейнер – это, фактически, процесс приложения.

2. Осуществляет наблюдение за контейнерами, которые запущены; останавливает их выполнение при необходимости. Выполнение контейнера может быть остановлено если планировщику YARN требуется освободить ресурсы.

В таблице 5.1 сравниваются MapReduce 1, чтобы продемонстрировать революционность новой технологии.

Таблица 5.1 – Сравнение MapReduce 1 и YARN

Характеристика	MapReduce 1	YARN
Модель выполнения	Только MapReduce-приложения поддерживаются в Hadoop 1, могут исполняться ограниченный набор типов приложений, соответствующих пакетной модели	YARN не накладывает никаких ограничений на тип работы, которая может выполняться в Hadoop; разработчик выбирает механизм исполнения (будь то обработка в реальном времени на Spark,

	обработки данных MapReduce.	обработки графов в Giraph, или пакетная обработка MapReduce).
Параллельные процессы	В MapReduce существовало понятие «слот», которое представляло конфигурацию узла, задавало максимальное количество map- и reduce-процессов.	YARN позволяет распределять ресурсы более гибко, а количество процессов лимитировано только конфигурацией и объемом доступной памяти и квотами CPU.
Ограничение памяти	Nadoop 1 также ограничивает количество слотов. Обычно кластеры Nadoop 1 конфигурируются таким образом, чтобы производство количества слотов, на максимальный размер памяти для каждого слота было меньше доступной оперативной памяти. Это часто приводит к нехватке памяти и не позволяет запускать высокоинтенсивные приложения. Nadoop 1 также ограничивает количество слотов. Обычно кластеры Nadoop 1 конфигурируются таким образом, чтобы производство количества слотов, на максимальный размер памяти для каждого слота было меньше доступной оперативной памяти. Это часто приводит к нехватке памяти и не позволяет запускать	YARN позволяет приложениям запрашивать ресурсы различных размеров. YARN позволяет задавать минимальные и максимальные ограничения на размер занимаемой памяти, а так как количество слотов больше не является фиксированным, максимальные значения могут быть значительно больше, что позволит реализовать поддержку ресурсоемких приложений. YARN обеспечивает гораздо более гибкую модель управления, которая не ограничивает количество процессов или объем оперативной памяти.

	<p>высокоинтенсивные приложения.</p> <p>Вакансии.</p> <p>Еще один недостаток MRv1 состоит в том, что его применение затруднено в высокоинтенсивных приложениях обработки данных. Программисту необходимо запускать больше слотов, чтобы увеличить количество процессов ввода/вывода, или меньше слотов, но выделять больший объем оперативной памяти.</p>	
Масштабируемость	<p>Наблюдались проблемы параллелизма, связанные с Job-Tracker, который ограничивал количество узлов в кластере Hadoop до 3000-4 000 узлов.</p>	<p>После выделения компонентов для планирования выполнения задач из MapReduce в отдельный элемент YARN, система в целом лишилась целого ряда недостатков. Масштабирование существенно упростилось. Возможно строить кластеры с размерами, превышающие размеры кластеров на основе предыдущих версий Hadoop.</p>
Выполнение	<p>Только одна версия MapReduce может поддерживаться кластером. Это было серьезной проблемой при проектировании многопользовательских систем. В подобных системах огромные ресурсы тратились на</p>	<p>MapReduce не входит в ядро Hadoop, теперь все приложения выполняются в соответствии с концепцией YARN. Это означает, что могут быть запущены несколько версий MapReduce одновременно на одном</p>

	координацию и поддержку согласованности.	кластере. Это является серьезным улучшением производительности в сфере внедрения многопользовательских систем.
--	--	--

Цель разработки YARN – увеличить число узлов кластера до 10000 узлов; дальнейшее масштабирование может быть ограничено компонентом ResourceManager, который является критической точкой данной технологии.

5.2 Компоненты YARN

YARN представляет собой совокупность фреймворка для управления ресурсами и приложений, которые выполняются на кластере.

5.2.1 YARN Framework

Параллельные вычисления, как модель программирования, могут быть определены как способ построения программного обеспечения, при котором множество программных потоков запущены параллельно, что в свою очередь, основано на предположении, что большая вычислительная задача может быть разбита на множество простых, которые решаются параллельно и независимо.

С точки зрения программиста при реализации параллельной вычислительной системы, возникает основная проблема – каким образом отобразить параллельные алгоритмические ветви на аппаратные средства компьютера.

Предположим, мы располагаем набором вычислительных ресурсов. Тогда, параллельные вычисления могут быть определены как одновременное использование всех вычислительных блоков (узлов или ЭВМ) для реализации параллельных вычислительных алгоритмов. В рамках такого подхода основная задача разбивается на множество малых, каждая из которых может решаться параллельно и независимо на отдельных вычислительных блоках.

При этом программное и аппаратное обеспечение таких вычислительных систем очень тесно связаны. Фактически, параллельные вычисления, как дисциплина, состоит из двух самостоятельных областей:

- архитектура вычислительных систем (аппаратная область);
- параллельное программирование (программная часть).

Архитектура вычислительных систем связана с изучением аппаратных средств параллельных и распределенных вычислителей, а параллельное программирование изучает методы решения задач с использованием параллельных алгоритмов, ориентированных на параллельные и распределенные вычислительные архитектуры. Для того, чтобы программист имел возможность разрабатывать параллельные программы, аппаратное обеспечение должно обеспечивать поддержку параллельного выполнения множества процессов или потоков.

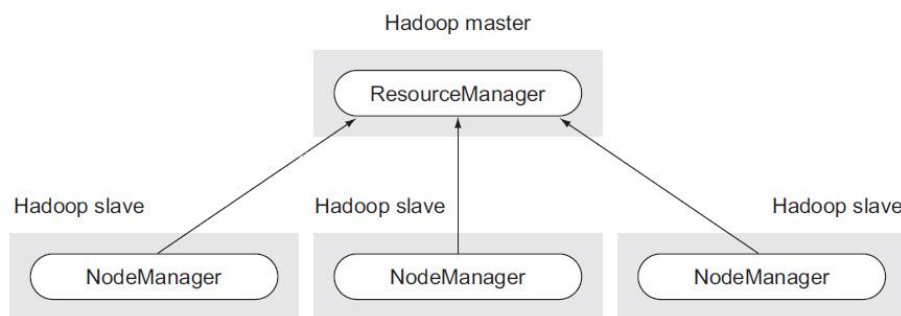


Рисунок 5.2 – Компоненты YARN

Важным элементом, который не указан на рис. 1.2 – это чипсет или «логическое ядро», т.е. интегральная схема, которая соединяет ЦП с внешним миром.

Это важный компонент материнской платы, так как каждый бит данных с устройств ввода-вывода (сетевые контроллеры, мышь, клавиатура, накопители USB, видеокарты и т.п.) проходит через чипсет. В устаревших аппаратных реализациях чипсет разделен на две отдельные микросхемы, которые называются «северный мост» (соединяет графический процессор с контроллером памяти, который управляет взаимодействием ЦП, регистрами ЦП и оперативной памятью) и «южный мост» (соединяет процессор и

периферийные устройства). Функции «северного моста» существенно изменились с появлением интерфейса PCI Express (PCIe).

Интерфейс PCIe является широкополосным. Каждая линия PCIe теоретически способна передавать данные со скоростью 500 МБ/с (полоса линии пропускания), количество таких линий в интерфейсе – 1, 4, 8 или 16. Графический процессор предполагает использование как можно большего количества линий интерфейса PCIe, видеокарты проектируются для использования всех 16 линий полосы слота PCIe. Следует понимать, что при теоретической пропускной способности 8 ГБ/с, подобный интерфейс на практике способен обеспечивать передачу данных со скоростью около 6 ГБ/с.

На рис. 1.2 также показан интегрированный контроллер памяти (memory controller) графического процессора, который является более специализированным, более упрощенным по сравнению с контроллером памяти ЦП.

5.2.2 Приложения YARN

Графический адаптер должен быть спроектирован так, чтобы учитывать взаимодействие с асинхронными устройствами-клиентами, такими как дисплеи, полоса пропускания которых фиксирована и предопределена производителем. Графический процессор проектируется также с расчетом на максимальные латентность памяти и ширину полосы пропускания интерфейса памяти. Таким образом, современные графические адаптеры могут обеспечивать ширину пропускания в 100 ГБ/с при локальном доступе графического процессора к памяти адаптера. Контроллер памяти ГП всегда интегрирован в схему процессора.

Для начала рассмотрим архитектуру Opteron от компании AMD и Nehalem от Intel. Контроллер памяти («северный мост») интегрирован непосредственно в ЦП (рис. 1.3). Такие усовершенствования призваны увеличить производительность памяти.

В случае архитектуры с несколькими ЦП (Рисунок 5.4), предполагается, что каждый ЦП имеет свой выделенный диапазон полосы пропускания. В то же время, из-за того, что многопоточные операционные системы и приложения базируются на когерентности кэша памяти (это вызвано обратной совместимостью с конфигурациями с «северным мостом»), архитектуры Opteron и Nehalem поддерживают такие технологии как HyperTransport (HT) и QuickPath Interconnect (QPI) соответственно (рис. 1.4).

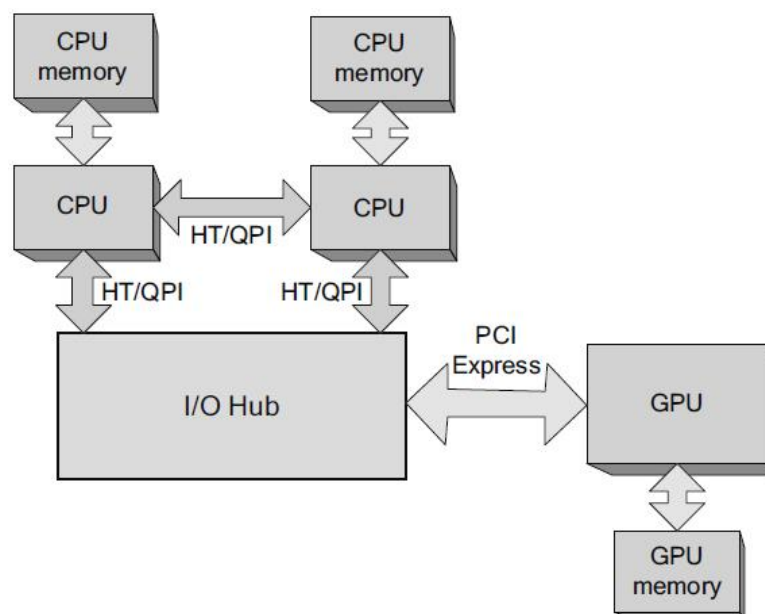


Рисунок 5.4 – Архитектура с несколькими центральными процессорами (NUMA-архитектура)

HT и QPI представляют собой интерфейсы, соединяющие центральные процессора между собой (или процессор и микросхему управления устройствами ввода-вывода, I/O hubs). В вычислительных системах с поддержкой HT/QPI каждый процессор может получать доступ к любой ячейке памяти, но наиболее высокоскоростной доступ обеспечивается только при обращении к локальной памяти, которая физически расположена в кристалле процессора.

Доступ к «нелокальной» памяти реализуется с помощью HT/QPI, при этом становятся доступными кэши других процессоров и процессор способен извлекать данные кэшированные другим процессором. Это значительно повышает производительность запросов к памяти. Упрощённо, можно сказать,

что большой объем кэшированных данных может снизить стоимость обращения к «нелокальной» памяти. При этом запрашиваемые данные могут содержаться в собственном кэше ЦП в виде специальной иерархической структуры до тех пор, пока они не будут запрошены другим процессором.

Операционные системы Windows и Linux предоставляют программистам API для обеспечения приложениям возможности размещать данные в памяти с учетом задействованного процессора, т.е. таким образом, чтобы доступ процессора к памяти происходил локально.

Безусловно, такое поведение вычислительной системы позволяет программистам писать код, представляющий угрозу производительности многопроцессорной архитектуры, но чаще возникает проблема «ложного совместного доступа», когда два потока, выполняемых на разных процессорах, вызывают множество обращений к кэшированным данным посредством HT/QPI-шины.

Таким образом, NUMA API должны использоваться с осторожностью: несмотря на увеличение производительности при использовании API, программист должен самостоятельно исключить ошибки при параллельном программировании.

Одним из подходов, позволяющих избежать трудностей «нелокального» доступа к памяти, является т.н. чередование – доступ к физической памяти квантуется по времени. В системах CUDA такой принцип реализации хорошо подходит в системе, схема которой показана на рис. 1.5: многопроцессорная система с архитектурой NUMA и доступом к графическому процессору через блок управления устройствами ввода-вывода.

Учитывая, что шина PCIe является критическим узлом («бутылочное горлышко») для производительности приложений, в большинстве систем для подключения нескольких PCIe-устройств применяется отдельный блок I/O hub (рис. 5.5).

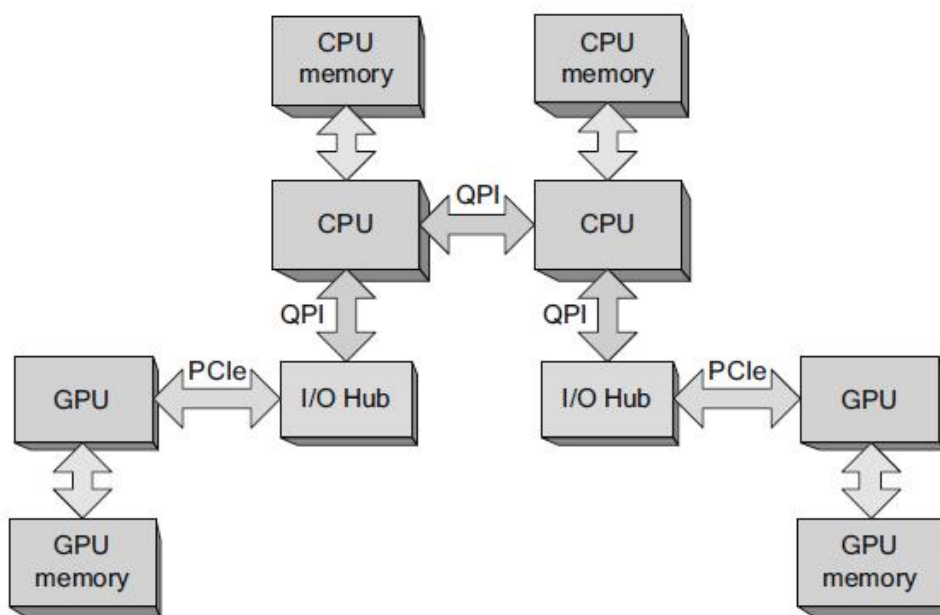


Рисунок 5.5 – Многопроцессорная система с несколькими шинами PCIe (NUMA-архитектура)

5.2.3 Конфигурации YARN

В 2004 году компания NVIDIA анонсировала интерфейс SLI (Scalable Link Interface) – технологию позволяющую нескольким ГП обеспечивать большую производительность за счет скоординированной параллельной работы. То есть к материнской плате могут быть подключены несколько видеокарт и пользователь может получить увеличение производительности в два раза при установке двух видеокарт (рис. 5.6). При этом две видеокарты функционируют как одна плата повышенной производительности.

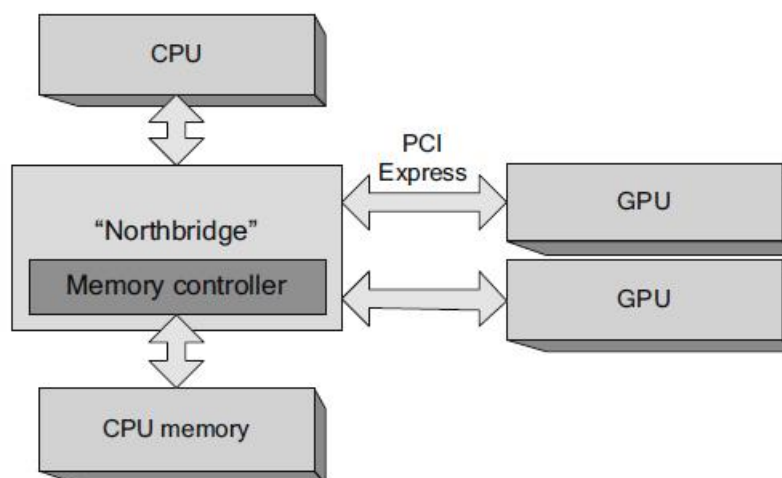


Рисунок 5.6 – Архитектура с двумя видеокартами в режиме SLI

Существуют видеокарты, которые содержат несколько графических процессоров (рис. 5.7). Такими графическими платами являются: GeForce 9800GX2 (dual-G92), the GeForce GTX 295 (dual-GT200), the GeForce GTX 590 (dual-GF110) и GeForce GTX 690 (dual-GK104). В таких видеокартах присутствует разделяемая микросхема моста, позволяющая ГП взаимодействовать с шиной PCIe.

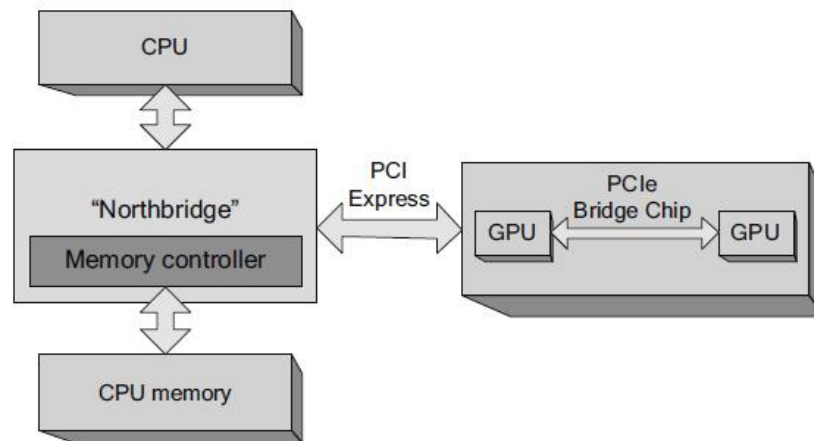


Рисунок 5.7 – Многопроцессорная видеокарта

Отдельные ГП не используют разделяемую память, вместо этого каждый ГП содержит интегрированный контроллер памяти, обеспечивающий максимальную полосу пропускания при доступе к локальной памяти, физически связанной с данным ГП. Графические процессоры могут взаимодействовать посредством вызовов типа «точка-точка» для копирования областей памяти с использованием микросхемы моста. Если графическая плата архитектуры Fermi или более поздней, то каждый ГП способен обращаться к областям памяти, принадлежащей другому ГП, но находящейся в общем глобальном адресном пространстве.

SLI – это технология компании NVIDIA, которая позволяет нескольким графическим процессорам (обычно, находящимся на одной плате) работать как единое целое, как единый графический адаптер повышенной производительности. Когда приложение загружает текстуры или другие данные, драйвер NVIDIA делает их доступными всем графическим процессорам; большинство инструкций рендеринга также доступны всем ГП.

Так как в режиме SLI графические процессоры представляют собой единое целое и приложения CUDA не предполагают непосредственного ускорения как традиционные графические приложения, разработчики CUDA должны отключать режим SLI.

Сложная высокопроизводительная система, основанная на нескольких PCIe-чипсетах показана на рис. 5.8.

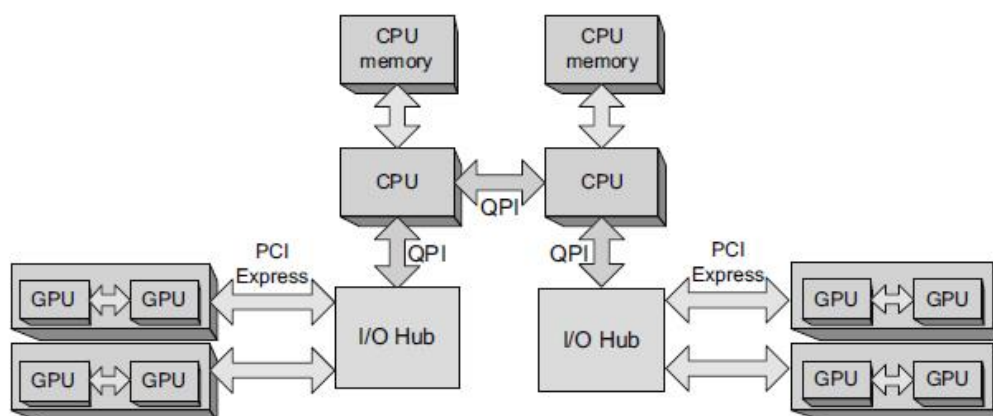


Рисунок 5.8 – Многопроцессорные видеокарты в системе с несколькими PCIe-шинами контроллерами ввода-вывода

На рис. 5.8 представлена система, в которой доступ к памяти типа «точка-точка» не может быть реализован через контроллер ввода-вывода (I/O hubs).

2.2.4 Взаимодействия YARN

Типичный современный вычислительный узел в гетерогенной системе состоит из многоядерного процессора и одного или нескольких многоядерных графических процессоров. При этом ГП не является самостоятельным устройством – в системе он выступает сопроцессором для ЦП. ГП взаимодействует с ЦП посредством PCIe (рис. 1.9). В терминологии, принятой в области графических вычислений, графический процессор называется device («устройство»), а центральный процессор – host («ведущий узел»).

Графический процессор – это специализированный вычислитель для реализации высокоинтенсивных, параллельных операций. ГП сконструирован таким образом, что большинство его транзисторов направлено на обеспечение

обработки данных, а не на кэширование или управление потоком управления как в центральном процессоре.

Графический процессор хорошо адаптирован к решению специализированных задач: одна программа, выполняемая над множеством элементов данных. Решаемые на ГП задачи отличаются высокой алгоритмической интенсивностью (под интенсивность понимают коэффициент равный отношению количества арифметических операций в программе к количеству операций обращения к памяти).

Так как одна и та же программа на графическом процессоре выполняется над несколькими элементами данных, в логике ГП упрощенно реализованы средства контроля потоком управления, а также латентность доступа к памяти нивелируется массовостью вычислений вместо большого объема кэша.

Ядра процессора реализуются для выполнения нескольких потоков, но с максимальной скоростью; для этого необходимо организовать доступность обрабатываемых данных – как только инструкция запускается на выполнение данные должны быть переданы процессору. В противном случае процессор будет простаивать в результате вынужденного ожидания данных или инструкций. Обращение к внешней памяти для процессора требует многих тактов работы, поэтому конструкция ЦП предусматривает наличие и активное использование кэшей различного уровня и объема.

Продукты компании NVIDIA доступны в рамках нескольких семейств:

1. Tegra. Графические платы данного семейства спроектированы для мобильных и встраиваемых систем (планшеты, смартфоны).
2. GeForce. Устройства для ноутбуков и настольных компьютеров.
3. Quadro. Устройства для профессиональной работы с компьютерной графикой.
4. Tesla. Устройства для высокопроизводительной обработки данных.

2.2.5 Недостатки YARN

Модель параллельной обработки данных MapReduce за счет абстрагирования позволяет исключить большинство проблем, возникающих при работе с распределенными системами, такими как необходимость распараллеливания, распределение нагрузки, а также ненадежность оборудования и программного обеспечения. Используя высокий уровень абстракции в рамках парадигмы MapReduce программист может сосредоточиться на удовлетворении потребностей бизнеса, а не на нюансах построения распределенной системы. MapReduce расщепляет задачу, представленную клиентом на мелкие параллельные подзадачи типа map и reduce как показано на рис. 4.6. Конструкции Map и reduce заимствованы из функционального языка программирования Лисп; они используют модель, которая исключает любые взаимозависимости между параллельными подзадачами, которые могли бы повлечь необходимость синхронизации и обмена состояниями.

Технология Kepler, представленная в 2012 году, предлагает гораздо более высокую вычислительную производительность по сравнению с предыдущими поколениями ГП, а также обеспечивает реализацию новых методов оптимизации и повышение объемов параллельных вычислений на ГП. Продукт Tegra K1 содержит процессор архитектуры Kepler и обеспечивает максимальную вычислительную производительность для встраиваемых систем.

5.3 YARN и MapReduce

5.3.1 Процесс взаимодействия с YARN

YARN – это автоматический планировщик ресурсов кластера. Эта технология была представлена в Hadoop версии 2. Технология была нацелена на устранение архитектурных недостатков Hadoop 1:

1. При развертывании кластера с более чем 4000 узлов возникали проблемы масштабируемости, добавление дополнительных узлов не давало эффекта роста производительности.

2. Поддерживалась только MapReduce-модель программ. Это означало, что кластер не был рассчитан на такие задачи как машинное обучение, где алгоритмы носят итерационный характер.

2. Only MapReduce workloads were supported, which meant it wasn't suited to run execution models such as machine learning algorithms that often require

В Hadoop 2 данные проблемы были разрешены путем передачи функций распределения от MapReduce и его переработки в планировщик, получивший название YARN. С такими изменениями кластер Hadoop больше не ограничивается запуском только MapReduce-программ. YARN делает возможными различные типы загрузок кластера и поддерживает альтернативные модели вычислений, такие как обработка графов, потоковая обработка.

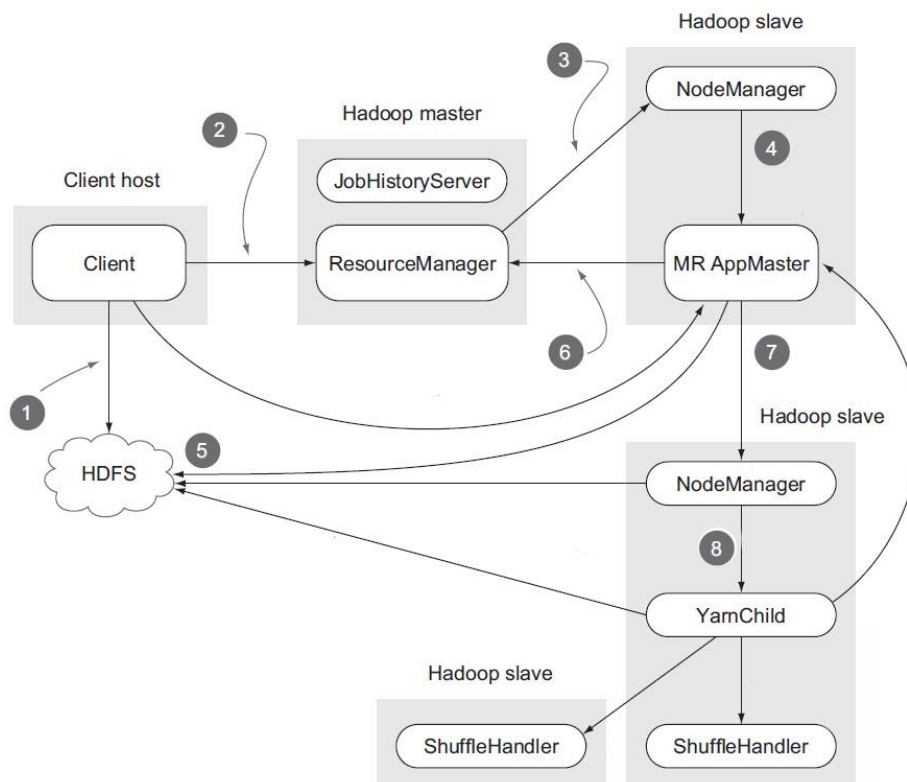


Рисунок 5.6 – Взаимодействия внутри YARN

1. Клиент запускает на выполнение работу MapReduce.

2. MapReduce выполняет декомпозицию работы на map- и reduce-подзадачи, а также распределяет их по slave-узлам кластера, где они и будут выполняться.

Задача программиста сводится к разработке map- и reduce-функций, причем map-функция на выходе отдает пары ключ/значение, а reduce-функция использует их для получения результата.

Map-функция получает на вход пары ключ/значение, которые представляют логические записи из источника данных. Если источник файловый, то под записью понимается строка из файла, а если источник таблица – строка таблицы.

С помощью map-функции получают ноль и более выходных пар ключ/значение на каждую входную пару. Например, если map-функция является функцией фильтрации, то возврат будет в виде множества пар, удовлетворяющих условию фильтрации. Или, это может быть операция демультиплексирования, когда входная пара порождает на выходе несколько пар.

С появлением YARN в Hadoop 2, MapReduce был переработан и выполняется в качестве YARN-приложения; также используется название MapReduce 2 (или MRv2). С точки зрения разработчика MapReduce в Hadoop 2 работает в большей степени без изменений, а код, написанный для Hadoop 1, будет выполняться и в Hadoop версии 2.

5.3.2 Свойства конфигурации YARN

Масштабируемость и высокая доступность также являются ключевыми особенностями HDFS, которые достигаются благодаря репликации данных и механизмам защиты от сбоев. HDFS реплицирует файлы с настраиваемой кратностью, при этом обеспечивается защита от программных и аппаратных сбоев, а также автоматическое перераспределение данных при отказе узлов кластера.

На рис. 4.4 показана схема взаимосвязи логических элементов HDFS: NameNode и DataNode. Также показано приложение, которое задействует библиотеку Hadoop filesystem library для доступа к HDFS.

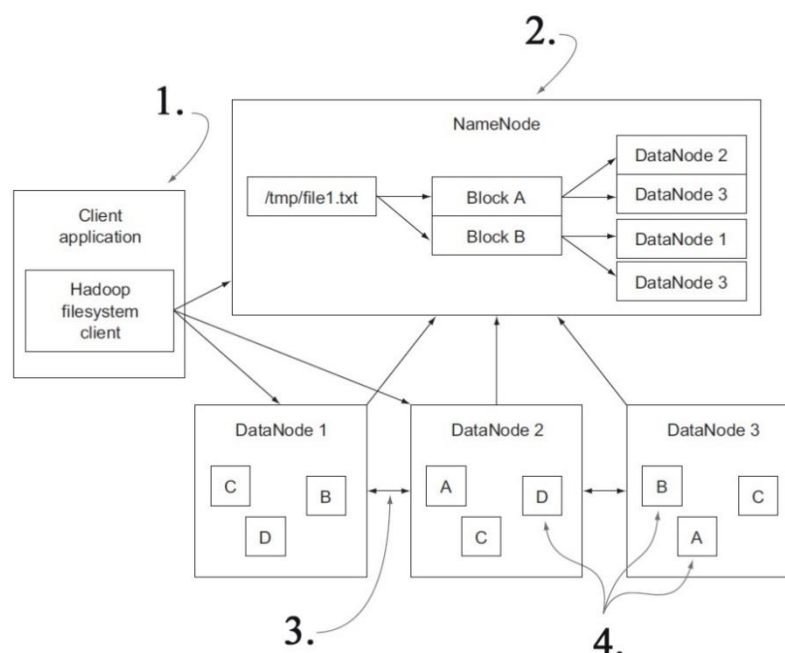


Рисунок 4.4 – Архитектура кластера HDFS в Hadoop 2

На рис. 4.4 показаны следующие элементы:

1. HDFS-клиент обращается к узлу NameNode для запроса метаданных, а также к узлам DataNodes для получения данных.
2. Узел NameNode содержит в памяти метаданные о файловой системе: каким образом узлы DataNodes распределяют блоки файлов данных.
3. Узлы DataNodes взаимодействуют друг с другом для управления файлами.
4. Файлы разбиваются на блоки, каждый файл может быть многократно реплицирован (по умолчанию каждый блок реплицируется трижды).

В Hadoop 2 были представлены две важные технологии для HDFS: HDFS Federation и High Availability (HA).

1. Federation позволяет распределять метаданные HDFS по множеству узлов NameNode. Такое поведение повышает масштабируемость HDFS и обеспечивает изолированность, позволяя различным приложениям запускать

собственные NameNode без опасения нанесения вреда другим узлам NameNode этого же кластера.

2. High Availability исключает критическую уязвимость в HDFS, которая существовала в Hadoop 1, когда отказ единственного NameNode мог привести к отказу всего кластера. HDFS HA также обеспечивает механизм автоматического восстановления кластера.

5.4 Применения YARN

Hadoop имеет высокий уровень использования в IT-компаниях, данная технология начинает все шире внедряться в различных секторах рынка, включая производство, правительственные учреждения, учреждения здравоохранения.

Компания Facebook использует Hadoop, Hive и HBase для хранения данных и обслуживания приложений в режиме реального времени. Кластеры хранилища Facebook заполнены петабайтами данных, содержат тысячи узлов, используют отдельные кластеры реального времени на основе HBase для отправки сообщений и аналитических вычислений в реальном времени.

Yahoo! использует Hadoop для машинного обучения, аналитических задач, поискового ранжирования, в системах анти спама, оптимизации бизнес процессов, в ETL-системах и др. Система объединяет более 40 000 серверов с установленным Hadoop с 170 ПБ данных. Yahoo! также запустил первую крупномасштабную YARN-систему с 4000 узлов.

Twitter является главным новатором в области Big Data, компания внесла весомый вклад в развитие проекта Hadoop путем развития таких проектов как Scalding, Scala API для Cascading, Summingbird.

eBay, Samsung, Rackspace, J.P. Morgan, Groupon, LinkedIn, AOL, Spotify, и StumbleUpon, а также другие известные компании также заинтересованы в развитии Hadoop. Microsoft проводит совместные исследования с Hortonworks для развития технологии Hadoop на платформе Windows.

Компания Google в своей публикации, посвященной MapReduce, отметила, что использует Caffeine (собственную версию MapReduce) для создания индексов из исходных данных.

Ежедневно количество организаций, использующих Hadoop, растет и если практически все передовые компании мира используют данную технологию. Очевидно, что технология Hadoop постоянно изменяется, продолжается внедрение данной технологии для решения различных задач.

Высокая доступность и безопасность часто указываются в качестве основных проблем Hadoop. Многие из проблем были решены в Hadoop 2. Рассмотрим некоторые недостатки Hadoop.

Предприятия, использующие Hadoop, имели проблемы с доступностью и безопасностью. В Hadoop 1, все мастер-процессы являются одиночными, а это означает, что сбой мастер-процесса вызывает сбой всего кластера. В Hadoop 2, HDFS рассчитан на высокую готовность системы, а переработка архитектуры MapReduce с заменой на YARN упразднило критическое место технологии. Безопасность Hadoop представляет собой еще одну область, обладающую различными недостатками.

Высокая доступность.

Высокая доступность часто требуется на предприятиях, которые предъявляют высокие требования к времени безотказной работы, при этом компании хотят быть уверены, что все системы работают постоянно, даже в случае отказа узла. До платформы Hadoop 2, мастер-процесс HDFS может выполняться только на одном узле, в результате чего возникает критическая точка отказа. В Hadoop 2 реализована технология HA (High Availability), что позволяет запускать несколько узлов NameNode для кластера Hadoop. В подобной архитектуре один из NameNode-узлов активный, а второй узел находится в ожидании. В случае планового или незапланированного отключения одного из узлов NameNode, другой предотвратит отказ системы.

Множественные центры обработки данных.

Поддержка нескольких центров обработки данных является еще одной технологией, которая все чаще востребована в секторе корпоративного программного обеспечения, так как обеспечивает надежную защиту данных, локализацию данных и их репликацию. Apache Hadoop и большая часть коммерческих дистрибутивов никогда не имели поддержки нескольких центров обработки данных, что создает проблемы для организаций, которые работают с программным обеспечением, предполагающим наличие нескольких ЦОД. WANdisco – это единственное в настоящее время доступное решение Hadoop, поддерживающее множественные ЦОД.

Безопасность.

Hadoop предлагает встроенную модель обеспечения безопасности, которая отключена по умолчанию. С отключенным механизмом безопасности, единственный механизм обеспечения безопасности, который функционирует, – это механизм HDFS по контролю за файлами и каталогами на основе прав и разрешений для пользователей. Но подобная архитектура подвержена атакам. По умолчанию все прочие сервисы Hadoop являются доступными, позволяющими всем пользователям выполнять любые типы операций, например, один пользователь может принудительно завершить задачу MapReduce другого пользователя.

Hadoop можно сконфигурировать для запуска Kerberos – сетевого протокола аутентификации, который позволит обеспечить проверку подлинности клиентов, пользователей при доступе к компонентам Hadoop. Kerberos может быть интегрирован с Active Directory, развернутой на предприятии. При внедрении Kerberos необходимо убедиться, что любой компонент, взаимодействующий с Hadoop, поддерживает Kerberos.

Физический уровень шифрования может быть сконфигурирован в Hadoop 2, что позволяет передавать зашифрованные данные по сети. Шифрование данных в самом Hadoop не реализована.

HDFS.

Основными недостатками HDFS являются неспособность данной файловой системы обеспечить высокий уровень доступности (в Hadoop 1.x и более ранних версиях); неэффективная обработка маленьких файлов; отсутствие сжатия. HDFS не поддерживает прямого доступа, поддержка данного типа доступа добавляется.

MapReduce.

Архитектура MapReduce не является удачным выбором в случаях требующих обработки данных в реальном времени. Данные, требующие глобальной синхронизации, или применение шардинга к изменяющимся данным, также неудачное применение MapReduce.

Несовпадение версий.

Выпуск Hadoop 2 связан с некоторыми проблемами в смысле совместимости. Для обеспечения совместимости иногда достаточно перекомпилировать код для API Hadoop 1 под Hadoop версии 2. Существуют также библиотеки, обеспечивающие совместимость различных версий.

5.4.1 NoSQL

Решение Bigtable компании Google предлагает модель хранения данных на основе столбцов. Это решение противоположно решению, применяемому в реляционных решениях, на основе строк. Хранилище столбцов более эффективно хранит данные. Подобный формат сокращает расходы при хранении пустых значений (NULL-значение или типизированных пустых значений).

Каждый элемент данных хранится как набор пар ключ/значение, причем каждый элемент однозначно идентифицируется ключевым атрибутом, также называемым первичным ключом. В Bigtable и его клонах такой идентификатор получил название строковый ключ. Дополнительно, как видно из названия формата, элементы данных хранятся в упорядоченном виде; элементы отсортированы и упорядочены по значению строкового ключа. Для лучшего понимания данного формата рассмотрим пример. Пусть дана простая таблица

данных, содержащая информацию о людях. Подобная таблица содержит следующие столбцы: `first_name` (имя), `last_name` (фамилия), `occupation` (адрес), `zip_code` (индекс), and `gender` (пол). Информация об отдельном человеке в такой таблице имеет вид:

`first_name: John`

`last_name: Doe`

`zip_code: 10001`

`gender: male`

Другой элемент данных может иметь иной вид:

`first_name: Jane`

`zip_code: 94303`

Строковый ключ первого элемента может быть равен 1, а второго 2. Тогда эти данные будут храниться таким образом, что элемент с ключом 1 будет размещен перед элементом с ключом 2. Также оба элемента будут выравнены относительно друг друга.

Далее, только пары ключ/значения с корректными значениями будут храниться в системе. Таким образом, возможное семейство столбцов для данного примера может быть названо как `name` и содержать столбцы `first_name` и `last_name`, которые принадлежат данному семейству. Другое семейство может быть `location` со столбцом `zip_code`. Третье семейство столбцов может быть названо как `profile`. Семейства столбцов обычно конфигурируются на этапе запуска. Сами столбцы не нуждаются в предварительном объявлении. Столбцы могут хранить данные любого типа, включая массивы байт.

Таким образом, логическое представление полученного хранилища для данного примера состоит из семейств столбцов: `name`, `location` и `profile`. Внутри каждого семейства хранятся только корректные значения. Семейство столбцов `name` хранит следующие значения:

For row-key: 1

`first_name: John`

last_name: Doe

For row-key: 2

first_name: Jane

Семейство столбцов location будет хранить:

For row-key: 1

zip_code: 10001

For row-key: 2

zip_code: 94303

Семейство столбцов profile будет хранить значения только для элемента с ключом 1:

For row-key: 1

gender: male

В практических реализациях хранилищ, семейства столбцов не изолированы в рамках одной строки. Все данные, относящиеся к одному строковому ключу хранятся совместно. Семейство столбцов выступает как ключ для столбцов, которые заключены в данном семействе, а строковый ключ выступает в качестве ключевого поля для всего набора данных.

Данный в Bigtable и его клонах хранятся в последовательной форме. По мере того как растет объем данных на отдельном узле, может возникнуть необходимости разделения узла на два. Данные сортируются и упорядочиваются не только в пределах одного узла, но и в рамках группы узлов, обеспечивающих функционирование укрупненного набора последовательных данных. Данные хранятся в отказоустойчивой форме, при котором поддерживаются три копии каждого набора.

Большинство клонов Bigtable также используют распределенной файловое хранилище, что позволяет распределять данные по узлам кластера.

Сортированная упорядоченная структура позволяет эффективно реализовать поиск данных.

HBase – это распространенное, свободно распространяемое решение на базе семейства столбцов, которое спроектировано на основе идей, лежащих в

основе Bigtable от Google. Данные в HBase обрабатываются в рамках парадигмы MapReduce. MapReduce-инструменты Hadoop могут использовать HBase как источник и/или приемника данных.

Вот некоторые open-source клоны Bigtable:

- HBase;
- Hypertable;
- Cloudata.

Хранилища на основе семейства столбцов в настоящее время широко распространены и формируют популярное NoSQL-направление. Тем не менее, NoSQL-подход включает в себе различные варианты хранилищ типа ключ/значение и документо-ориентированных хранилищ.

5.4.2 Интерактивный SQL

Bigtable от Google и Apache HBase (часть фреймворка Hadoop) являются базами данных, основанными на семействе столбцов, как и Hypertable и Cloudata. Архитектура этих систем варьируется, но основные принципы построения у них общие.

Поколение разработчиков нашего времени живет в эпоху реляционных баз данных. Обучение в университете, использование на работе, публикации – все направлено на изучение, внедрение и развитие систем управления реляционными базами данных. В основе концепции РСУБД лежат отношения и связи между ними.

Современное поколение разработчиков широко применяет реляционные базы данных. Фундаментальные концепции реляционных баз данных: сущности и связи между ними, изучаются в университетах, внедряются на предприятиях и обсуждаются и исследуются. Поэтому, изложение принципов построения столбцовых баз данных начнем с рассмотрения примера на основе РСУБД. Такой подход позволит более эффективно освоить материал.

В РСУБД атрибуты отношений представляются столбцами таблиц. Столбцы определяются заранее и значения хранятся для каждой строки каждого

столбца. На рис. 5.1 представлена схема реляционной таблицы, которая понятно любому разработчику.

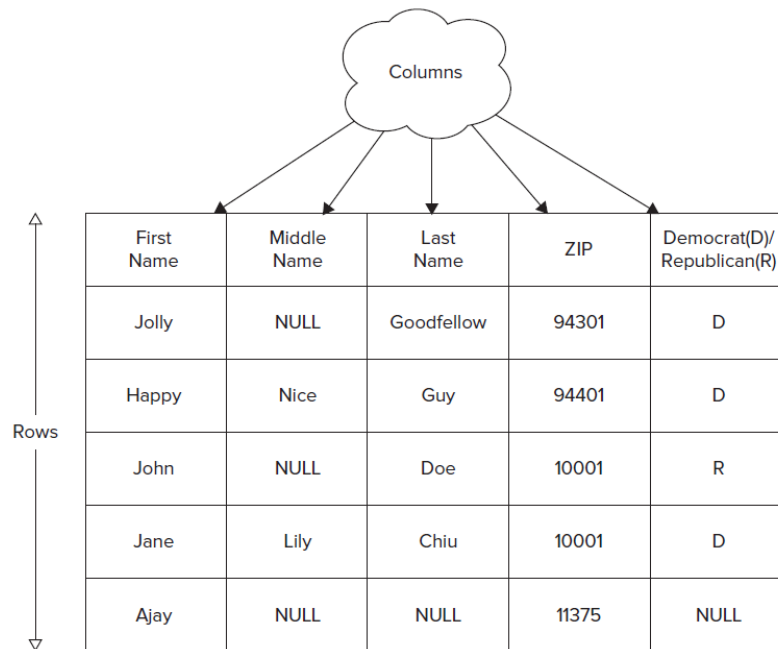


Рисунок 5.1 – Схема реляционной таблицы

В данном элементарном примере таблица содержит 5 столбцов. В РСУБД для каждого столбца определяется соответствующий тип данных. Например, для столбца «First Name» указывается тип `varchar`, для «ZIP» – `int`. В ячейках таблицы могут отсутствовать неопределенные значения (NULL-значения). Например, в примере на рис. 3.1 у Jolly Goodfellow отсутствует отчество (поле «Middle Name»).

Обычно, в таблице РСУБД определяется несколько столбцов, иногда – десятки. Внутри таблицы хранятся тысячи записей. В особых случаях таблицы могут хранить миллионы строк, но такое размещение может существенно повлиять на производительность и требуется применение специализированных механизмов оптимизации, таких как нормализация и индексирование.

В процессе использования таблицы может возникнуть ситуация, когда требуется добавить несколько столбцов, например «Street Address» и «Food Preferences». Как только добавлены новые столбцы, они не содержат значений. В ячейках будет множество NULL-значений. Такая ситуация часто

встречается на практике. В этом случае, таблица будет иметь вид, представленный на рис. 5.2.

First Name	Middle Name	Last Name	Street Address	ZIP	D/R	Veg/ Non-Veg

Рисунок 5.2 – Разраженный набор данных

Предположим, что данные в таблице изменяются во времени и нам необходимо хранить различные версии данных. В таком случае можно использовать структуру напоминающую трехмерную таблиц Excel, где одним из измерений выступает время. Рисунок 5.3 демонстрирует 3-D-набор данных.

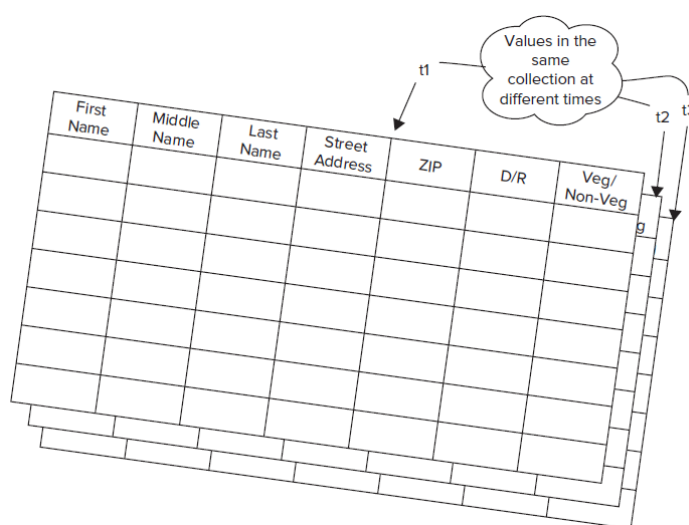


Рисунок 5.3 – 3-D-структура данных

Несмотря на то, что пример достаточно простой, вы может догадаться насколько сильно увеличиться количество пустых ячеек если в определенный момент времени все данные из таблицы будут удалены. Хранение большого количества пустых ячеек может быть сопряжено с серьезными проблемами. Точнее, проблемы связаны с особенностями РСУБД.

5.4.3 Обработка графов

В предыдущих подразделах были рассмотрены основные направления развития продуктов NoSQL. Некоторые решения, например, основанные на графовых базах данных и XML, также могут быть отнесены к базам данных NoSQL. В данном пособии они рассматриваться не будут. Сегодня наибольшую популярность приобрели графовые базы данных: Neo4j и FlockDB. Neo4J полностью удовлетворяет требованиям ACID. Такие СУБД облегчают обработку графов.

5.4.4 Обработка данных в режиме реального времени

Документо-ориентированные базы данных не являются системами автоматизации документооборота. Многие начинающие разработчики путают систему NoSQL на основе документо-ориентированной базы данных и систему управления контентом на основе документов. Слово документ в сфере баз данных подразумевает слабо структурированные наборы пар ключ/значение для описания документа в формате JSON (нотация объектов JavaScript); это не документы или электронные таблицы (хотя они также могут быть сохранены в подобных системах).

Документо-ориентированные базы данных рассматривают документ в целом, не разбивая его на составные пары имя/значение. Это позволяет оперировать коллекциями документов. Документо-ориентированные базы данных поддерживают индексирование не только на основе первичного ключа, но и с использованием различных свойств документа. На сегодняшний день доступны несколько СУБД данного типа, но наиболее популярными являются MongoDB и CouchDB.

5.4.5 Технология BSP

Модель параллельной обработки данных MapReduce за счет абстрагирования позволяет исключить большинство проблем, возникающих при работе с распределенными системами, такими как необходимость

распараллеливания, распределение нагрузки, а также ненадежность оборудования и программного обеспечения. Используя высокий уровень абстракции в рамках парадигмы MapReduce программист может сосредоточиться на удовлетворении потребностей бизнеса, а не на нюансах построения распределенной системы. MapReduce расщепляет задачу, представленную клиентом на мелкие параллельные подзадачи типа map и reduce как показано на рис. 5.6. Конструкции Map и reduce заимствованы из функционального языка программирования Лисп; они используют модель, которая исключает любые взаимозависимости между параллельными подзадачами, которые могли бы повлечь необходимость синхронизации и обмена состояниями.

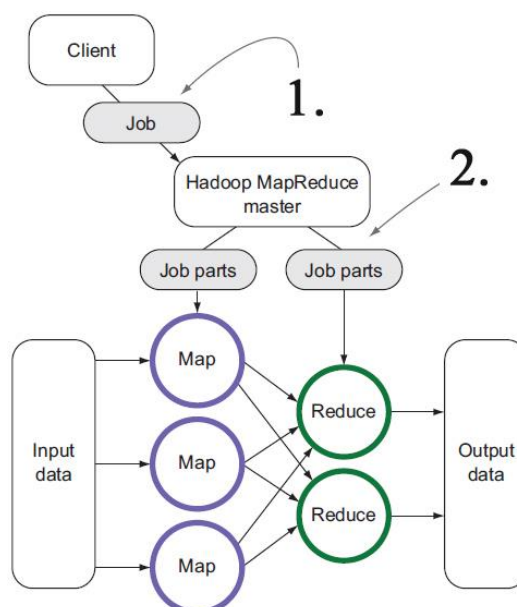


Рисунок 5.6 – Распараллеливание задачи MapReduce

На рис. 5.6 показаны различные этапы разделения задачи на уменьшенные подзадачи типа map и reduce:

1. Клиент запускает на выполнение работу MapReduce.
2. MapReduce выполняет декомпозицию работы на map- и reduce-подзадачи, а также распределяет их по slave-узлам кластера, где они и будут выполняться.

Задача программиста сводится к разработке map- и reduce-функций, причем map-функция на выходе отдает пары ключ/значение, а reduce-функция использует их для получения результата.

Мар-функция получает на вход пары ключ/значение, которые представляют логические записи из источника данных. Если источник файловый, то под записью понимается строка из файла, а если источник таблица – строка таблицы.

5.4.6 MPI, In-memory computing

Интерфейс передачи сообщений – наиболее распространенный стандарт параллельного программирования. Данная технология оперирует процессами и является часто используемой в крупных вычислительных центрах. Установка и настройка MPI-кластера требуют квалификации и определенных навыков администрирования. Параллелизм достигается за счет использования сотен, тысяч потоков одновременно на нескольких сетевых узлах; взаимодействие потоков происходит путем отправки сообщений через высокоскоростные линии связи (Ethernet, чаще SCI или InfinityBand). MPI – широко распространенное решение и достаточно сложная технология. Данное решение оправдано при наличии кластера, сконфигурированного и управляемого высококвалифицированным администратором.

Заключение

Hadoop – это распределенная система, спроектированная для обработки, генерации и хранения наборов данных большого размера. Реализация MapReduce обеспечивает отказоустойчивый механизм для анализа данных в гетерогенной среде на базе неструктурированных источников данных. Технология YARN позволяет выполнять множество неравноправных приложений различными пользователями на одном кластере Hadoop.

Hadoop является инструментом, предоставляющим пользователям доступ к распределенным вычислениям.

Используя распределенное хранилище и доставку кода к данным, Hadoop позволяет избежать долговременной задержки при передаче больших объемов данных по сети. Кроме того, избыточность данных позволяет Hadoop восстановить работу кластера при сбое узла. Создание программ для кластера Hadoop представляется достаточно простым с использованием фреймворка MapReduce.

Контрольные вопросы

1. Опишите архитектуру Hadoop 1 и Hadoop 2. Каковы основные отличия?
2. Каковы основные компоненты YARN?
3. Опишите типовые схемы взаимодействия приложений в YARN.
4. Опишите недостатки YARN.
5. Опишите области применения YARN.

6. ПАТТЕРНЫ MAPREDUCE И BIG DATA

Множество однотипных задач, возникающих при параллельном программировании, могут рассматриваться как устоявшиеся шаблоны (паттерны). Несмотря на то, что паттерны применяются во многих приложениях, необходим опыт, чтобы их выявить. Решение задач в терминах паттернов позволяет программисту абстрагироваться от самой проблемы и позволяет найти более простой и, возможно, уже апробированный метод решения задачи.

Вместе с параллелизмом в программах стала появляться неоправданная сложность и возникла необходимость в программистах, который мыслят в терминах параллелизма и конкурентного взаимодействия потоков. Программист должен постоянно помнить об особенностях параллельных моделей, когда осуществляет проектирование программных систем для GPU.

6.1 Объединение

Каждый разработчик знаком с циклами. Циклы различаются по синтаксису и своему действию в программах (for, do...while, while, foreach), а также они отличаются по характеру зависимости между итерациями цикла.

Зависимость итераций – это явление, когда одна итерация цикла зависит от одной или нескольких предыдущих. Программист должен стремиться избавиться от таких зависимостей, так как они увеличивают сложность параллельных алгоритмов. Если не удастся добиться полной независимости итераций, то алгоритм разбивается на блоки, которые выполняются параллельно. Результат из блока 0 затем передается блоку 1, затем блоку 2, и т.д.

Паттерн, основанный на итерациях, является одним из самых простых паттернов для параллельного программирования. После удаления итерационных зависимостей, программисту остается только решить каким

образом разделить все множество итераций между доступными вычислителями. Такое разделение производится с точки зрения минимизации взаимодействия между вычислителями и максимизации использования внутренних ресурсов вычислителя (регистры и разделяемая память на GPU; кэши L1, L2, L3 на CPU). Взаимодействие между вычислителями – основная трудность и часто именно это является узким местом (bottleneck) при реализации и проектировании систем.

6.1.1 Стратегии объединения

На более высоком уровне декомпозиции необходимо основываться на количестве доступных вычислителей. Для систем с использованием CPU – это число доступных аппаратных потоков (логических процессоров). Для GPU – число потоковых мультипроцессоров (SM), умноженное на максимальную нагрузку одного SM. В данном случае мы оперируем логическими, а не физическими потоками. Например, некоторые процессоры Intel способны поддерживать более одного логического потока на одном ядре CPU (т.н. hyperthreading). Графические процессора запускают множество блоков на отдельном SM, соответственно при планировании необходимо, как минимум, учитывать процессы в количестве произведения количества SM на максимальное количество блоков, поддерживаемое каждым SM.

Использование более одного потока на одном физическом вычислителе позволяет максимизировать пропускную способность таких устройств, в терминах выполнения реальных работ это означает: устройство постоянно выполняет какие-либо операции пока происходит ожидание данных из памяти или результаты операций ввода-вывода. Наблюдение за этими временными показателями полезно при анализе загрузки и позволяет существенно улучшить характеристики при реализации новых версий GPU.

Что касается CPU, то количество потоков тесно связано с производительностью, причем простое увеличение вычислителей не приводит к увеличению производительности – в большинстве случаев это приводит к

частому переключению контекста приложением или операционной системой. Такое поведение имеет критическое значение для кэшей и полосы пропускания памяти

Таким образом, многоядерные архитектуры CPU, существующие на современном этапе, сильно отличаются от GPU в области параллелизма. Практически всегда, при решении задачи на GPU, разработчик должен разбивать исходные данные на небольшие блоки. Программист также должен учитывать внутренние и внешние циклы с целью их распараллеливания.

На CPU одним из подходов является распараллеливание внешних циклов, так как количество потоков ограничено. Такое решение приемлемо, но только в случае независимости итераций.

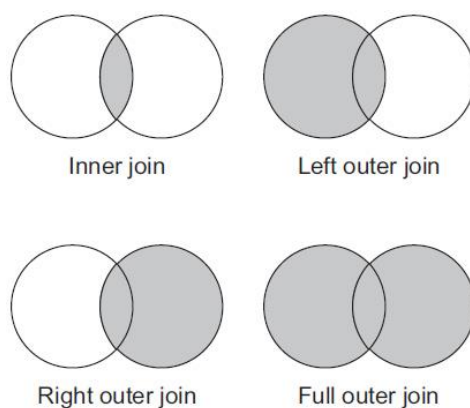


Рисунок 6.1 – Различные типы диаграмм Верна

На GPU внутренний цикл обычно реализуется в виде группы потоков в одном блоке. В таком случае сгруппированные итерации трансформируются в группы потоков, которые получают доступ к близко расположенным областям памяти. Это позволяет говорить о локальности – важной особенности в CUDA. Любой внешний цикл представляется как блок потоков.

anne	22	NY
joe	39	CO
alison	35	NY
mike	69	VA
marie	27	OR
jim	21	OR
bob	71	CA
mary	53	NY
dave	36	VA
dude	50	CA

Рисунок 6.2 – Набор данных

The logs dataset shows some user-based activity that could be extracted from application or webserver logs. The data includes the username, an action, and the source IP address. Here's the complete dataset (Рисунок 6.3):

jim	logout	93.24.237.12
mike	new_tweet	87.124.79.252
bob	new_tweet	58.133.120.100
mike	logout	55.237.104.36
jim	new_tweet	93.24.237.12
marie	view_user	122.158.130.90
jim	login	198.184.237.49
marie	login	58.133.120.100

Рисунок 6.3 – Log dataset

Рассмотрим также каким образом большинство циклов могут быть приведены в параллельную форму, а также как преобразовать внутренние и внешние циклы. Рассмотрим алгоритм обработки изображения, который предполагает выполнение итераций вдоль оси пикселей X (внутренний цикл) и вдоль оси пикселей Y (внешний цикл). В данном случае возможно преобразовать цикл, предположив, что матрица пикселей – это одномерный массив. Тогда итерации производятся над однотипными элементами-пикселями.

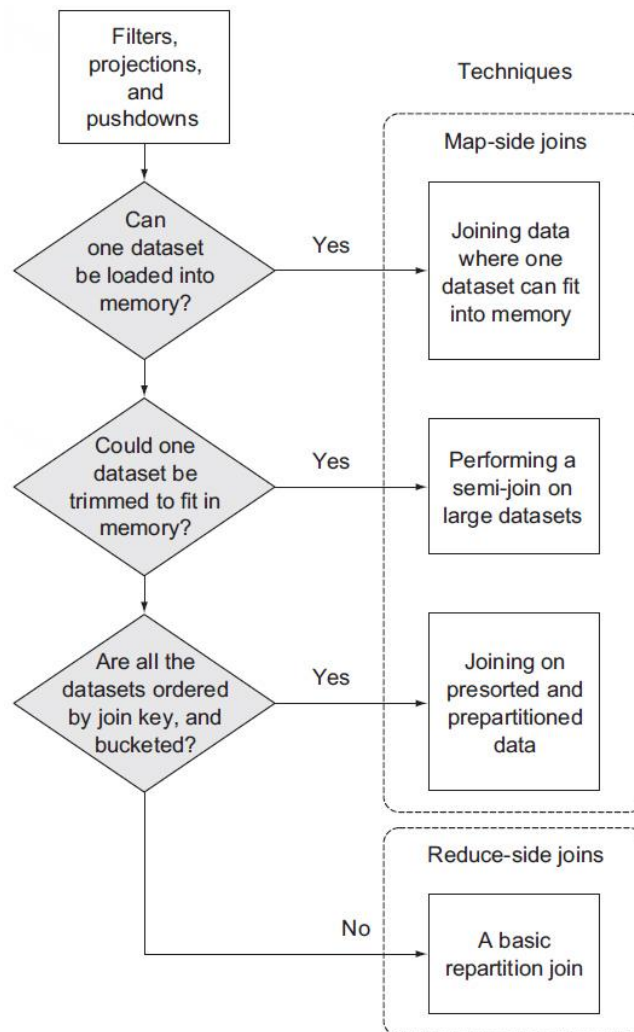


Рисунок 6.4 – Алгоритм принятия решения при выборе стратегии объединения

Паттерн fork/join – это общепринятый шаблон в промышленном программировании. Кратко, этот метод построения приложений предполагает, что программа разбивается на отдельные участки, только часть из которых является параллельными блоками, а также присутствуют области синхронизации параллельных веток программы. В определенных точках последовательного кода встречаются участки, начиная с которых вычисления могут быть распределены по нескольким процессорам. В этой точке программа «расходится» на несколько параллельных потоков (процессов). Выполнение параллельного участка происходит независимо и затем, по завершении вычислений, происходит объединений «веток» программы. Подобное поведение можно выявить при использовании OpenMP, где

параллельные участки программы маркируются специальными программными директивами.

Рисунок 6.5 демонстрирует каким образом происходит фильтрация.

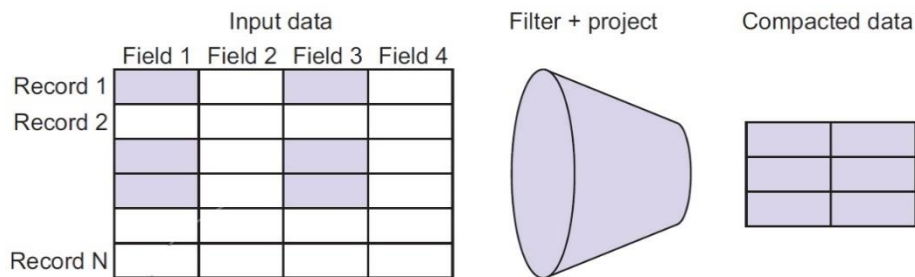


Рисунок 6.5 – Использование фильтра в reduce-процессе

В Рисунок 6.5:

Каждая очередь обрабатывается независимо, результат обработки записывается в определенное место результирующей очереди. Обычно применение данного паттерна реализуется с использованием статического разбиения данных. Это значит, что последовательный код инициирует запуск N потоков и разделит исходный массив данных на равные части между N потоками. В процессе выполнения fork/join возможны ситуации, когда все потоки равно нагружены и выполнение работы занимает одинаковое время, но чаще всего время выполнения работ на отдельных параллельных потоках отличается.

```
@Override
protected void map(LongWritable offset, Text value, Context context) {

    User user = User.fromText(value);
    if (user.getAge() >= 30) {
        context.write(new Text(user.getName()),
            new Text(user.getState()));
    }
}
```

Рисунок 6.6 – Код фильтра

Таким образом, на CPU программисты и различные многопоточные библиотеки используют ограниченное количество логических потоков, доступных в виде набора процессов. Потоки CPU обычно являются тяжеловесными, т.е. предполагают высокие накладные расходы на создание, уничтожение и управление. В таких библиотеках часто используется

механизм пула потоков, который позволяет динамически распределять нагрузку между тяжеловесными потоками, предоставляя им задачи из единой очереди.

Table 6.1 – Форматы данных

Формат	Projection pushdown supported	Predicate pushdown supported
Text (CSV, JSON, etc.)	No	No
Protocol Buffers	No	No
Thrift	No	No
Avro	No	No
Parquet	Yes	Yes

Очевидно, что преимущество обеспечивает формат Parquet.

6.1.2 Соединение типа Map-side

Memory-Mapped файл представляет собой сегмент виртуальной памяти, который отображен байт-в-байт на файл или файл-подобный ресурс, на который можно ссылаться через дескриптор файла. Из этого следует, что приложения могут взаимодействовать с такими файлами, как если бы они были частью первичной памяти. Это, очевидно, повышает производительность ввода/вывода по сравнению с обычным дисковым чтением и записью. Доступ и манипулирование данными в памяти гораздо быстрее, чем реализация аналогичных системных вызовов. Кроме того, во многих операционных системах, таких как Linux, отображаемые на файлы регионы памяти являются частью буфера страниц дисковой памяти в оперативной памяти. Этот буфер обычно называется кэшем страниц. Он реализован в ядре операционной системы.

Стратегия MongoDB использования памяти, отображаемой на файлы, для хранения является достаточно интеллектуальной, но не лишенной недостатков. Во-первых, memory-mapped файлы подразумевают, что не существует никакого разделения между кэшем операционной системы и кэшем базы данных. Это означает, что также отсутствует избыточность. Во-

вторых, кэширование управляется операционной системой, а отображение виртуальной памяти работает не одинаково на различных операционных системах. Таким образом, политики кэширования, которые определяют, что хранится в кэше, а что нет, также варьируются от одной операционной системы к другой. В-третьих, MongoDB может расширить свой кэш базы данных, чтобы использовать всю доступную память без каких-либо дополнительных настроек. Это означает, что производительность MongoDB может быть увеличена за счет приращения объема RAM и выделения большей виртуальной памяти.

Отображение памяти также предполагает некоторые ограничения. Например, реализация MongoDB ограничивает размер данных до 2 ГБ на 32-разрядных системах. Это ограничение не актуально для 64-битных систем.

Размер базы данных ограничен не только по размеру. Дополнительные ограничения лимитируют размер каждого документа и количество коллекций сервера MongoDB. Документ не может быть больше, чем 8 МБ. Это, очевидно, означает, что использование MongoDB для хранения больших элементов не подходит. При хранении больших документов необходимо использовать GridFS. Кроме того, существует ограничение на количество пространств имен, которые могут быть определены в экземпляре базы данных. По умолчанию, количество пространств имен – 24000. Каждая коллекция и каждый индекс использует пространства имен. Соответственно, два индекса на коллекцию позволит организовать максимум 8000 коллекций на одну базу данных. Как правило, такое количество достаточно. Тем не менее, если нужно, то можно увеличить количество пространств имен до числа, превышающего 24000.

Увеличение размера пространства имен влечет определенные последствия. Каждое пространство имен коллекций использует несколько килобайт. В MongoDB индекс реализован в виде B-дерева. Каждая страница B-дерева имеет размер 8 КБ. Таким образом, добавление дополнительных пространств имен, для коллекций или индексов, предполагает добавление несколько килобайт для каждого дополнительного экземпляра. Пространства

имен для имени MyDB базы данных MongoDB сохраняются в файле с именем mydb.ns. Файл mydb.ns может вырасти до максимального размера 2 ГБ. Ограничение размера mydb.ns ограничивает и рост базы данных – понимание этого намного важнее чем знание моделей поведения коллекций и индексов.

6.1.3 Соединение типа Reduce-side

В большой распределенной и масштабируемой системы, Acid-транзакции накладывают огромные накладные расходы. Так что "Динамо" предлагает управление версиями объектов и векторных часов для поддержания согласованности. Давайте попробуем разобраться, как вектор, Часы работы с помощью примера.

Будем считать, что четыре хакеров, Джо, Хиллари, Эрик, и Аджай, решили встретиться и поговорить о векторе часы. Джо предполагает, что они все встретились в пало-Альто. Потом Хиллари и Эрик встречаются на работе и решать, что горы могут быть лучшим местом для встречи. В тот же день, Эрик и сообщение Аджай друг друга и сделать вывод, что встречи в Лос-Альтос может быть лучшей идеей. Когда день встречи приезжает, Джо электронные письма всем встретиться-напоминание и адрес отель в пало-Альто. Хиллари отвечает, что сменилось место проведения, чтобы смотреть на горы и Эйджей говорит, что это Лос-Альтос. Оба утверждают, что Эрик знает решение. Теперь Эрик связаться, чтобы решить вопрос. На данном этапе вы можете создать векторные часы для разрешения конфликта.

Вектор часы могут быть созданы для каждого из трех значений за тем, пало-Альто, Маунтин-вью, Лос-Альтос, следующим образом: Venue: Palo Alto:

Vector Clock: Joe (ver 1).

Venue: Mountain View:

Vector Clock: Joe (ver 1), Hillary (ver 1), Eric (ver 1).

Venue: Los Altos:

Vector Clock: Joe (ver 1), Ajay (ver 1), Eric (ver 1).

Векторные часы для Маунтин-вью и Лос-Альтос включают первоначально выбор Джо, потому что всем было известно. Векторные часы для вид на горы базируется на ответ Хиллари и векторных часов для Лос-Альтос базируется на ответ Аджая. Вид на горы и Лос-Альтос векторные часы не синхронизированы, потому что они не спускают друг от друга. Вектор часы должен быть версии больше или равно всех значений в другой вектор часы, чтобы сходиться с нее.

Наконец, Джо завладеет Эрик по телефону и просит его разрешить недоумение. Эрик понимает проблемы и быстро решает, что встреча в Маунтин-вью, вероятно, лучшая идея. Теперь Джо вытягивает обновленные векторные часы следующим образом: Venue: Palo Alto:

Vector Clock: Joe (ver 1).

Venue: Mountain View:

Vector Clock: Joe (ver 1), Hillary (ver 1), Ajay (ver 0), Eric (ver 2).

Venue: Los Altos:

Vector Clock: Joe (ver 1), Hillary (ver 0), Ajay (ver 1), Eric (ver 1).

Версия 0 создана для Хиллари и Аджай в векторе часов на площадках они не предложили, но теперь в курсе. Теперь, векторные часы спускаются друг от друга и видом на горы является местом проведения встречи. Из примера, вы бы заметили, что вектор часы не только помогают определить порядок событий, но и помочь в решении каких-либо несоответствий путем выявления коренных причин этих проблем.

Независимо от объекта управления версиями, Динамо использует полученные на основе членства для узлов и использует намекнул передачу последовательности.

Протокол сплетни-это стиль коммуникационного протокола, выполненная в виде сплетен или слухов в социальных сетях и офисах. Протокол связи включает в себя периодические сплетни, пара мудрых, межпроцессных взаимодействий. Надежность невелика и коллегиального выбора часто является случайной.

6.1.4 Срез данных в reduce-side-соединении

С точки зрения программиста при реализации параллельной вычислительной системы, возникает основная проблема – каким образом отобразить параллельные алгоритмические ветви на аппаратные средства компьютера.

Предположим, мы располагаем набором вычислительных ресурсов. Тогда, параллельные вычисления могут быть определены как одновременное использование всех вычислительных блоков (узлов или ЭВМ) для реализации параллельных вычислительных алгоритмов. В рамках такого подхода основная задача разбивается на множество малых, каждая из которых может решаться параллельно и независимо на отдельных вычислительных блоках. При этом программное и аппаратное обеспечение таких вычислительных систем очень тесно связаны. Фактически, параллельные вычисления, как дисциплина, состоит из двух самостоятельных областей:

- архитектура вычислительных систем (аппаратная область);
- параллельное программирование (программная часть).

Архитектура вычислительных систем связана с изучением аппаратных средств параллельных и распределенных вычислителей, а параллельное программирование изучает методы решения задач с использованием параллельных алгоритмов, ориентированных на параллельные и распределенные вычислительные архитектуры. Для того, чтобы программист имел возможность разрабатывать параллельные программы, аппаратное обеспечение должно обеспечивать поддержку параллельного выполнения множества процессов или потоков.

6.2 Сортировка

Модель параллельной обработки данных MapReduce за счет абстрагирования позволяет исключить большинство проблем, возникающих

при работе с распределенными системами, такими как необходимость распараллеливания, распределение нагрузки, а также ненадежность оборудования и программного обеспечения. Используя высокий уровень абстракции в рамках парадигмы MapReduce программист может сосредоточиться на удовлетворении потребностей бизнеса, а не на нюансах построения распределенной системы. MapReduce расщепляет задачу, представленную клиентом на мелкие параллельные подзадачи типа map и reduce как показано на рис. 6.6. Конструкции Map и reduce заимствованы из функционального языка программирования Лисп; они используют модель, которая исключает любые взаимозависимости между параллельными подзадачами, которые могли бы повлечь необходимость синхронизации и обмена состояниями.

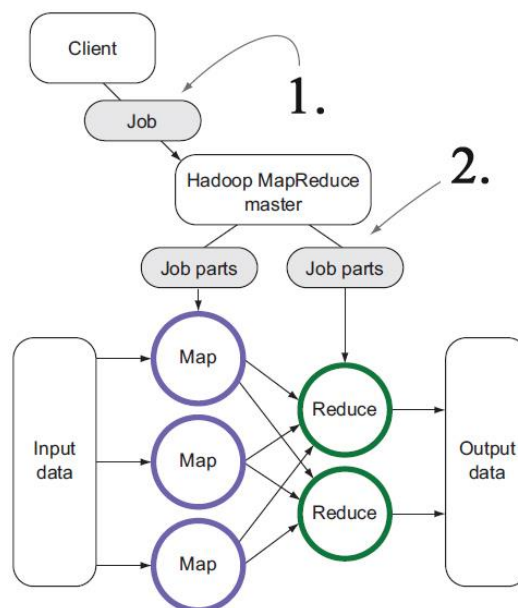


Рисунок 6.6 – Распараллеливание задачи MapReduce

На рис. 6.6 показаны различные этапы разделения задачи на уменьшенные подзадачи типа map и reduce:

1. Клиент запускает на выполнение работу MapReduce.
2. MapReduce выполняет декомпозицию работы на map- и reduce-подзадачи, а также распределяет их по slave-узлам кластера, где они и будут выполняться.

Задача программиста сводится к разработке map- и reduce-функций, причем map-функция на выходе отдает пары ключ/значение, а reduce-функция использует их для получения результата.

Map-функция получает на вход пары ключ/значение, которые представляют логические записи из источника данных. Если источник файловый, то под записью понимается строка из файла, а если источник таблица – строка таблицы.

С помощью map-функции получают ноль и более выходных пар ключ/значение на каждую входную пару. Например, если map-функция является функцией фильтрации, то возврат будет в виде множества пар, удовлетворяющих условию фильтрации. Или, это может быть операция демультиплексирования, когда входная пара порождает на выходе несколько пар.

С появлением YARN в Hadoop 2, MapReduce был переработан и выполняется в качестве YARN-приложения; также используется название MapReduce 2 (или MRv2). С точки зрения разработчика MapReduce в Hadoop 2 работает в большей степени без изменений, а код, написанный для Hadoop 1, будет выполняться и в Hadoop версии 2.

6.2.1 Вторичная сортировка

Прежде всего, столбцовые базы данных отличаются отсутствием строгих требований к схеме базы данных; можно легко добавлять новые столбцы при появлении дополнительных данных. В обычном столбцовом хранилище заранее определяются семейства столбцов, а не столбцы. Семейство столбцов – это набор столбцов, сгруппированных в пакет. Столца в одном пакете логически связаны, хотя это не является обязательным требованием. Колонки семейства хранятся физически совместно и пользователь, как правило, объединяет в семейство несколько столбцов с одинаковыми характеристиками. Могут существовать различные ограничения на количество столбцов в семействе, но определение минимального

количества колонок в пакете позволяет сделать схему более гибкой. В нашем примере достаточно определить три семейства: name, location и preferences.

В столбцовых базах данных семейство столбцов – это аналог столбца в РСУБД. Оба элемента определяются заранее, перед добавлением данных. Столбцы в РСУБД предполагают задание типа хранимых значений, что не является обязательным в столбцовых базах: они могут содержать любое количество столбцов, содержащих любые типы значений.

Каждая строка в столбцовых базах хранится только для тех столбцов, в которых для нее присутствуют корректные значения. NULL-значения не хранятся в базе. На рис. 6.7 показано как изменится схема данных при использовании семейств столбцов.

	name	location	preferences
	first name=>"...", last name=>"..."	zip=>"..."	d/r=>"...", veg/non-veg=>"..."

Рисунок 6.7 – Таблица в столбцовой базе данных

Кроме хорошего механизма хранения разреженных данных, столбцовые базы данных предоставляют возможность работы с данными в историческом разрезе. Поэтому постоянно развивающиеся данные будут храниться с использованием структуры, представленной на рис. 6.8.

	time	name	location	preferences
	t9	first name=>"...", last name=>"..."	zip=>"..."	d/r=>"...", veg/non-veg=>"..."
	t8			
	t7			
	t5			

Рисунок 6.8 – Хранение исторических данных

На физическом уровне данные не хранятся как единая таблица, но расположены как набор семейств столбцов. Набор столбцов спроектирован для масштабируемой архитектуры и легко может оперировать миллионами столбцов и миллиардами строк. Поэтому, одна таблица часто разделяется на множество машин. Строковый ключ идентифицирует запись в столбцовой базе. Строки упорядочены и объединены в пакеты, содержащие непрерывные значения, отражающие изменение данных во времени.

Рисунок 6.9 демонстрирует физическую схему хранения таблицы.

row-key	time	name	row-key	time	location
	t9			t9	
	t8			t8	
	t7				
	t5				

row-key	time	preferences
	t9	
	t7	

Рисунок 6.9 – Разделение таблицы на физическом уровне

Обычное решение при использовании столбцовой базы данных – применение кластера. Запуск подобного решения на единственной машине осуществляется для целей тестирования. Каждая база данных на основе семейства столбцов рассчитана на определенную типовую топологию кластера и обладает своим механизмом развертывания, но все подобные решения предполагают некоторый типовой сценарий.

6.2.2 Итоговая сортировка

Обычно, в таблице РСУБД определяется несколько столбцов, иногда – десятки. Внутри таблицы хранятся тысячи записей. В особых случаях таблицы могут хранить миллионы строк, но такое размещение может существенно повлиять на производительность и требуется применение

специализированных механизмов оптимизации, таких как нормализация и индексирование.

В процессе использования таблицы может возникнуть ситуация, когда требуется добавить несколько столбцов, например «Street Address» и «Food Preferences». Как только добавлены новые столбцы, они не содержат значений. В ячейках будет множество NULL-значений. Такая ситуация часто встречается на практике. В этом случае, таблица будет иметь вид, представленный на рис. 6.10.

First Name	Middle Name	Last Name	Street Address	ZIP	D/R	Veg/ Non-Veg

Рисунок 6.10 – Разраженный набор данных

Предположим, что данные в таблице изменяются во времени и нам необходимо хранить различные версии данных. В таком случае можно использовать структуру напоминающую трехмерную таблиц Excel, где одним из измерений выступает время. Рисунок 6.11 демонстрирует 3-D-набор данных.

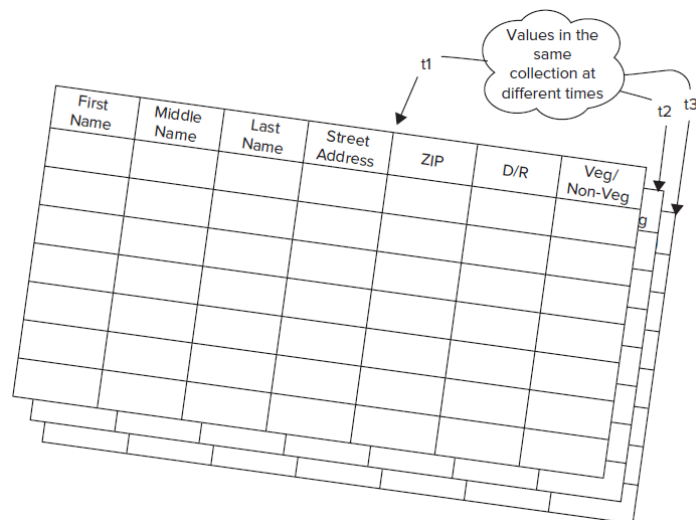


Рисунок 6.11 – 3-D-структура данных

Несмотря на то, что пример достаточно простой, вы можете догадаться насколько сильно увеличится количество пустых ячеек если в определенный момент времени все данные из таблицы будут удалены. Хранение большого количества пустых ячеек может быть сопряжено с серьезными проблемами. Точнее, проблемы связаны с особенностями РСУБД.

6.3 Примеры

Hadoop имеет высокий уровень использования в IT-компаниях, данная технология начинает все шире внедряться в различных секторах рынка, включая производство, правительственные учреждения, учреждения здравоохранения.

Компания Facebook использует Hadoop, Hive и HBase для хранения данных и обслуживания приложений в режиме реального времени. Кластеры хранилища Facebook заполнены петабайтами данных, содержат тысячи узлов, используют отдельные кластеры реального времени на основе HBase для отправки сообщений и аналитических вычислений в реальном времени.

Yahoo! использует Hadoop для машинного обучения, аналитических задач, поискового ранжирования, в системах анти спама, оптимизации бизнес процессов, в ETL-системах и др. Система объединяет более 40 000 серверов с установленным Hadoop с 170 ПБ данных. Yahoo! также запустил первую крупномасштабную YARN-систему с 4000 узлов.

Twitter является главным новатором в области Big Data, компания внесла весомый вклад в развитие проекта Hadoop путем развития таких проектов как Scalding, Scala API для Cascading, Summingbird.

eBay, Samsung, Rackspace, J.P. Morgan, Groupon, LinkedIn, AOL, Spotify, и StumbleUpon, а также другие известные компании также заинтересованы в развитии Hadoop. Microsoft проводит совместные исследования с Hortonworks для развития технологии Hadoop на платформе Windows.

Компания Google в своей публикации, посвященной MapReduce, отметила, что использует Caffeine (собственную версию MapReduce) для создания индексов из исходных данных.

Ежедневно количество организаций, использующих Hadoop, растет и если практически все передовые компании мира используют данную технологию. Очевидно, что технология Hadoop постоянно изменяется, продолжается внедрение данной технологии для решения различных задач.

Высокая доступность и безопасность часто указываются в качестве основных проблем Hadoop. Многие из проблем были решены в Hadoop 2. Рассмотрим некоторые недостатки Hadoop.

Предприятия, использующие Hadoop, имели проблемы с доступностью и безопасностью. В Hadoop 1, все мастер-процессы являются одиночными, а это означает, что сбой мастер-процесса вызывает сбой всего кластера. В Hadoop 2, HDFS рассчитан на высокую готовность системы, а переработка архитектуры MapReduce с заменой на YARN упразднило критическое место технологии. Безопасность Hadoop представляет собой еще одну область, обладающую различными недостатками.

Заключение

В этом разделе пособия рассматривается понятие NoSQL. Приводятся исторический обзор развития технологии и освещаются основные механизмы NoSQL. Кроме основ данного подхода рассмотрены дистрибутивы, области применения. Одновременно выявляется, что NoSQL – это не решение всех проблем в сфере управления данными, и, безусловно, подходу свойственны определенные недостатки. Большинство решений NoSQL хорошо масштабируются при увеличении объемов обрабатываемых данных и работают на множестве узлов кластера. Распределенная обработка больших данных является сложной задачей, и нуждается в новых специальных методах.

Современное поколение разработчиков пришло в NoSQL из области реляционных баз данных, осознавая изменения, отличия этих подходов. Разработчик должен хорошо изучить NoSQL-технологии прежде чем принимать решение об их использовании в реальных проектах.

Контрольные вопросы

1. Опишите наиболее известные NoSQL-паттерны.
2. Поясните различные стратегии объединения.
3. Опишите объединение типа Map-side.
4. Что такое срез данных при объединении типа reduce-side?
5. Что такое sampling? Для чего он используется?

7. MONGODB

Базы данных принимают самые разные формы. Предметные указатели в книгах, каталожные карточки, когда-то стоявшие в библиотеках, - это тоже базы данных, как и специально структурированные текстовые файлы, излюбленные программистами былых времен, работавшими на языке Perl. Но, наверное, чаще всего базы данных ассоциируются с изощренными реляционными СУБД, лежащими в основе огромного количества современных программных систем, теми самыми СУБД, на которых делаются состояния. Реляционные базы, с их идеализированной третьей нормальной формой и выразительным SQL-интерфейсом, все еще вызывают уважение старой гвардии - и вполне заслуженно.

Но несколько лет назад, разрабатывая веб-приложения, я мечтал опробовать только появляющиеся на свет альтернативы правящим в мире реляционным базам. Открыв для себя MongoDB, я сразу понял - это как раз то, что мне надо. Мне понравилась идея использовать JSON-подобные структуры для представления данных. Язык JSON прост, интуитивно понятен и удобен для человека. И тот факт, что язык запросов в MongoDB также основан на JSON вселяет в пользователя этой новой СУБД ощущение комфорта и гармонии. На первом месте стоит интерфейс. А наличие таких привлекательных средств, как простая репликация и сегментирование, только подстегивает интерес к пакету.

7.1 Основы MongoDB

MongoDB - это система управления базами данных, «заточенная» под веб-приложения и инфраструктуру Интернета. Модель данных и стратегия их постоянного хранения спроектированы для достижения высокой пропускной способности чтения и записи и обеспечивает простую масштабируемость с автоматическим переходом на резервный ресурс в случае отказа. Сколько бы

узлов ни требовалось приложению - один или десятки, - MongoDB сумеет обеспечить поразительно высокую производительность. Того, кто раньше мучился с масштабированием реляционных баз, эта новость обрадует. Но не всем приложениям необходим крупный масштаб. Быть может, вам всегда хватало одного сервера базы данных. Зачем тогда использовать MongoDB?

Как выясняется, привлекательность MongoDB объясняется в первую очередь не стратегией масштабирования, а интуитивно понятной моделью данных. Учитывая, что с помощью документо-ориентированной модели можно представить развитые иерархически организованные структуры данных, часто оказывается' возможно обойтись без присущих реляционным СУБД сложностей, связанных с соединением нескольких таблиц. Пусть, например, вы моделируете товары для сайта интернет-магазина. В полностью нормализованной базе данных информация об одном товаре может быть разбросана по десяткам таблиц. Чтобы получить его представление в интерактивной оболочке СУБД, придется написать сложный SQL-запрос с кучей соединений.

Поэтому разработчики, как правило, обращаются к дополнительным программам, когда хотят собрать разрозненные данные в нечто осмысленное. С другой стороны, в документной модели большая часть информации о товаре может быть представлена в виде одного документа. В оболочке MongoDB, построенной на основе языка JavaScript, нетрудно получить полное представление о товаре в виде иерархически организованной JSON-подобной структуры¹. К ней можно предъявлять запросы, ей можно манипулировать. Средства составления запросов в MongoDB специально спроектированы для работы со структурированными документами, но так, чтобы пользователь, имеющий опыт работы с реляционными базами, располагал сравнимой выразительной мощностью. К тому же, сегодня большинство разработчиков пишут на объектно-ориентированных языках, поэтому им нужно такое хранилище, которое было бы проще отобразить на объекты. В случае MongoDB объект, определенный на языке программирования, сохраняется «как есть» - без

дополнительной сложности, приносимой системами объектно-реляционного отображения.

7.1.1 Особенности MongoDB

База данных в значительной мере определяется своей моделью данных. В этом разделе мы рассмотрим документную модель данных, а затем покажем, какие особенности MongoDB позволяют эффективно работать с этой моделью. Мы также обсудим вопросы промышленной эксплуатации MongoDB, уделив особое внимание средствам репликации и стратегии горизонтального масштабирования.

Документная модель данных.

Модель данных MongoDB является документо-ориентированной. Для тех, кто не знаком с идеей документа в контексте баз данных, продемонстрировать ее проще всего на примере (рис. 7.1).

```
{ _id: ObjectID('4bd9e8e17cefd644108961bb'),
  title: 'Adventures in Databases',
  url: 'http://example.com/databases.txt',
  author: 'msmith',
  vote_count: 20,
  tags: ['databases', 'mongodb', 'indexing'],
  image: {
    url: 'http://example.com/db.jpg',
    caption: '',
    type: 'jpg',
    size: 75381,
    data: "Binary"
  },
  comments: [
    { user: 'bjones', text: 'Interesting article!' },
    { user: 'blogger', text: 'Another related article is at http://example.com/db/db.txt' }
  ]
}
```

1 Tags stored as array of strings

2 Attribute points to another document

3 Comments stored as array of comment objects

Рисунок 7.1 – Представление документа

В листинге 7.1 приведен пример документа, представляющего статью на социальном новостном сайте (вспомните о Digg). Как видите, документ - это по существу набор, состоящий из имен и значений свойств. Значение может быть представлено простым типом, например: строки, числа и даты. Но может

быть также массивом и даже другим документом (2) С помощью таких конструкций можно представлять весьма сложные структуры данных. Так, в нашем примере имеется свойство tags (1) - массив, в котором хранятся ассоциированные со статьей теги. Но еще интереснее свойство comments (3), которое ссылается на массив документов, содержащих комментарии.

Сделаем небольшую паузу и сравним это с представлением тех же данных в стандартной реляционной базе. На рис. 7.2 изображен типичный реляционный аналог. Поскольку таблицы по сути своей плоские, для представления связей типа один-ко-многим необходимо несколько таблиц. Мы начинаем с таблицы posts, в которой хранится основная информация о каждой статье. Затем создаем еще три таблицы, каждая из которых содержит поле post_id, ссылающееся на исходную статью. Эта техника разнесения данных объекта по нескольким таблицам называется нормализацией. Среди прочего, нормализованный набор данных характеризуется тем, что каждый элемент данных хранится только в одном месте.

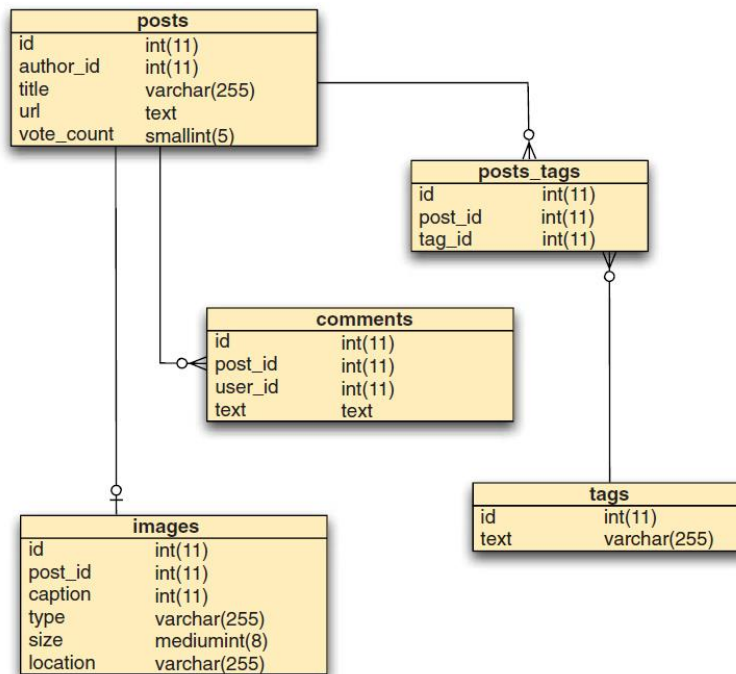


Рисунок 7.2 – Реляционная модель

Но за строгую нормализацию приходится платить. И самое главное то, что впоследствии данные нужно как-то собрать. Чтобы показать статью, необходимо выполнить соединение таблиц post и tags. А комментарии либо

запрашивать по отдельности, либо тоже включить их в соединение. В конечном итоге ответ на вопрос, нужна ли строгая нормализация, зависит от вида моделируемых данных. Я еще вернусь к этой теме в главе 4. Здесь же важно отметить, что документо-ориентированная модель позволяет естественно представить данные в агрегированной форме, то есть работать в объектом, как с единым целым: все данные, относящиеся к статье, от комментариев до тегов, хранятся в одном объекте базы данных.

Вероятно, вы обратили внимание, что документы не только позволяют представлять данные со сложной структурой, но и не нуждаются в заранее определенной схеме. В реляционной базе данных строки хранятся в таблице. У каждой таблицы имеется строго определенная схема, описывающая, какие столбцы и типы данных допустимы. Если окажется, что необходимо добавить еще одно поле, то таблицу надо будет явно изменить. В MongoDB документы группируются в коллекции - контейнеры, не налагающие на данные какую-либо схему. Теоретически у каждого входящего в коллекцию документа может быть своя структура, но на практике документы в одной коллекции похожи друг на друга. Например, у всех документов в коллекции `posts` имеются поля `title`, `tags`, `comments` и т. д.

Отсутствие предопределенной схемы несет с собой некоторые преимущества. Во-первых, структуру данных определяет код приложения, а не база. Это позволяет ускорить разработку на ранних этапах, когда схема часто изменяется. Во-вторых, и это куда важнее, безсхемная модель позволяет представить данные с переменным набором свойств. Представьте, к примеру, каталог товаров к интернет-магазину. Заранее неизвестно, какие у товара будут атрибуты, поэтому приложение должно как-то адаптироваться к подобной изменчивости. Традиционно в базе данных с фиксированной схемой эта задача решается с помощью паттерна сущность-атрибут-значение³, как изображено на рис. 7.3. Здесь показан один раздел модели данных, применяемой в Magento, каркасе с открытым кодом для электронной торговли.

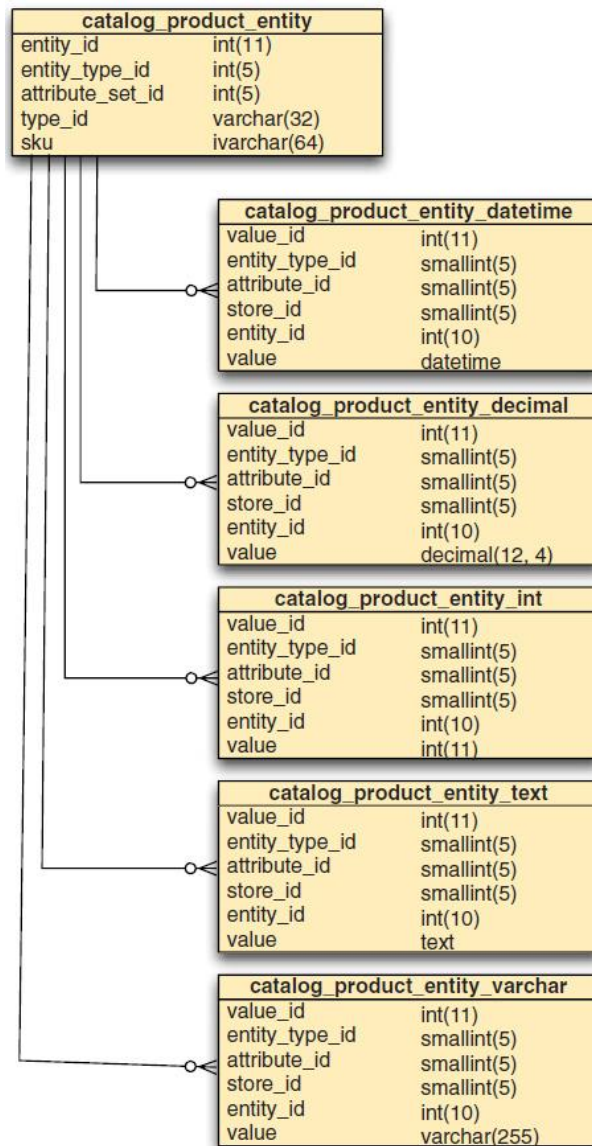


Рисунок 7.3 – Часть схемы проекта Magento, написанного на PHP

Обратите внимание на ряд таблиц, отличающихся только в одном поле `value`, которое в каждой таблице имеет свой тип. Такая структура позволяет администратору определять новые типы товаров вместе с их атрибутами, но ценой возрастания сложности. Представьте себе, какой запрос нужно было бы написать в оболочке MySQL, если бы потребовалось выбрать или обновить товар, смоделированный подобным образом; операции соединения таблиц оказались бы необычайно сложными. А при моделировании в виде документа никакого соединения не требуется, и добавлять новые атрибуты можно динамически.

Ad hoc запросы. Говорят, что система поддерживает произвольные запросы, если не

требуется заранее определять, какие запросы разрешены. Реляционная база данных таким свойством обладает; она честно выполнит любой правильно записанный SQL-запрос, сколько бы условий он ни содержал. Если вы работали только с реляционными СУБД, то, возможно, считаете произвольные запросы чем-то само собой разумеющимся. Однако не все базы данных поддерживают динамические запросы. Например, хранилища ключей и значений можно опрашивать только по одной оси: ключу. Как и многие другие системы, хранилища ключей и значений жертвуют гибкостью запросов в обмен на простоту модели масштабируемости. При проектировании MongoDB в частности ставилась задача по возможности сохранить выразительную мощь запросов, которая считается принципиально важной особенностью реляционных СУБД.

Принцип построения запросов в MongoDB мы рассмотрим на простом примере статей и комментариев. Пусть необходимо найти все статьи, помеченные тегом politics, за которые проголосовало более 10 посетителей. SQL-запрос для решения этой задачи выглядел бы так:

```
SELECT * FROM posts
INNER JOIN posts_tags ON posts.id = posts_tags.post_id
INNER JOIN tags ON posts_tags.tag_id == tags.id
WHERE tags.text = 'politics' AND posts.vote_count > 10;
```

Эквивалентный запрос в MongoDB формулируется путем задания документа-образца. Условие «больше» обозначается специальным ключом \$gt.

```
db.posts.find({'tags': 'politics', 'vote_count': {'$gt': 10}});
```

Отметим, что в этих запросах предполагаются разные модели данных. SQL-запрос опирается на строго нормализованную модель, в которой статьи и теги хранятся в разных таблицах, тогда как в запросе для MongoDB считается, что теги хранятся внутри документа, описывающего статью. Однако же в

обоих случаях имеется возможность формулировать запрос с произвольной комбинацией атрибутов, что и составляет смысл понятия произвольных запросов.

Выше уже отмечалось, что не все базы данных поддерживают произвольные запросы - в угоду простоте модели. Так, хранилище ключей и значений можно опрашивать только по первичному ключу. С точки зрения запроса, значение, на которое указывает ключ, непрозрачно. Единственный способ включить в запрос дополнительный атрибут, в данном случае количество голосов, - написать специальный код, который вручную построит записи, где первичным ключом будет счетчик голосов, а значением - список первичных ключей документов, за которые подано соответствующее количество голосов. Если вы примените такой подход к хранилищу ключей и значений, то вас обвинят в трюкачестве. Хотя для небольших наборов данных это, возможно, и будет работать, но, вообще говоря, упаковывание нескольких индексов в физически единственный индекс - мысль не слишком удачная. К тому же, хешированные индексы, применяемые в хранилищах ключей и значений, не поддерживают запросы по диапазону, которые, скорее всего, необходимы для поиска с участием количества голосов.

Если вы раньше работали с реляционными базами данных, в которых произвольные запросы считаются нормой, то возьмите на за метку, что MongoDB обеспечивает аналогичную гибкость. Оценивая различные технологии баз данных, имейте в виду, что не все они поддерживают произвольные запросы, и если такая возможность необходима, то MongoDB может оказаться подходящим вариантом. Но одних лишь произвольных запросов недостаточно. Когда объем данных достигает определенного уровня, для обеспечения приемлемой эффективности становятся необходимы индексы. Наличие подходящих индексов увеличивает скорость выполнения запросов и сортировки на несколько порядков; следовательно, любая система, поддерживающая произвольные запросы, должна также поддерживать вторичные индексы.

7.1.2 Сервер и инструментальные средства MongoDB

СУБД MongoDB написана на языке C++ и активно разрабатывается компанией 10gen. Проект компилируется во всех основных операционных системах, включая Mac OS X, Windows и почти все дистрибутивы Linux. На сайте mongodb.org выложены откомпилированные двоичные файлы для каждой платформы. MongoDB - проект с открытым исходным кодом, распространяемый на условиях лицензии GNU-AGPL. Исходный код находится в открытом бесплатном доступе на сайте GitHub, в него часто включаются дополнения, предлагаемые сообществом. Но руководство проектом в целом осуществляет группа разработки сервера, состоящая из сотрудников компании 10gen, и подавляющее большинство обновлений в системе управления версиями исходит от нее.

The core server.

Исполняемый файл сервера базы данных называется `mongod` (`mongodb.exe` в Windows). Процесс `mongod` взаимодействует с клиентами по специальному протоколу, получая команды через сетевой сокет. Все используемые им файлы по умолчанию хранятся в каталоге `/data/db6`.

Сервер `mongod` может работать в нескольких режимах, самый употребительный - член набора реплик. Поскольку репликация рекомендуется, то обычно конфигурируется набор, состоящий из двух реплик и процесса-арбитра, расположенного на третьем сервере.

Если MongoDB работает в режиме автосегментирования, то процессы `mongod` в каждом сегменте конфигурируются как наборы реплик, а «сбоку» находятся специальные серверы метаданных, называемые конфигурационными серверами. Для отправки запросов конкретным сегментам используется также отдельный маршрутный сервер `mongos`.

Настройка процесса `mongod` проста по сравнению с другими СУБД, например MySQL. Конечно, можно определить номера портов и пути к каталогам данных, но для настройки собственно базы данных параметров не

так много. Оптимизация базы данных, которая в большинстве реляционных СУБД подразумевает манипуляции с кучей параметров, управляющих распределением памяти и другими аспектами работы сервера, превратилась в нечто сродни черной магии. Но при проектировании MongoDB было решено, что с управлением памятью операционная система справится лучше, чем администратор базы данных или разработчик приложения. Поэтому файлы данных проецируются на виртуальную память системы с помощью системного вызова `mmap()`. Тем самым забота об управлении памятью передается ядру ОС. Ниже я еще вернусь к вызову `mmap()`, а пока отмечу лишь, что почти полное отсутствие конфигурационных параметров - продуманная стратегия, а не ошибка.

The JavaScript shell.

Командная оболочка MongoDB - это инструмент для администрирования базы и манипулирования данными с помощью команд на языке JavaScript. Исполняемый файл `mongo` загружает оболочку и устанавливает соединение с указанным процессом `mongod`. Оболочка по своим возможностям сравнима с оболочкой MySQL с тем отличием, что язык SQL не используется. Вместо этого команды по большей части записываются в виде выражений языка JavaScript. Вот, например, как выбрать базу данных и вставить простой документ в коллекцию `users`:

```
> use mongodb-in-action
> db.users.insert({name: "Kyle"})
```

Первая команда, определяющая, какая база данных будет использоваться, знакома любому пользователю MySQL. Вторая команда - выражение JavaScript для вставки простого документа. Чтобы увидеть результат вставки, можно выполнить простой запрос.

Database drivers.

Если сама идея языкового драйвера вызывает у вас кошмарные ассоциации с низкоуровневыми драйверами устройств, расслабьтесь. Пользоваться драйверами MongoDB просто. Разработчики постарались

предоставить API, согласующийся с идиомами конкретного языка, обеспечив в то же время относительное единообразие интерфейса. Так, все драйверы реализуют общий набор методов для вставки документа в коллекцию, однако представление самого документа обычно выглядит естественно для данного языка. Например, в Ruby это будет хеш, в Python - словарь, а в Java, где нет аналогичного примитива на уровне языка, документ представляется с помощью специального класса конструктора документов, который реализует интерфейс `LinkedHashMap`.

Because the drivers provide a rich, language-centric interface to the database, little abstraction beyond the driver itself is required to build an application. This contrasts notably with the application design for an RDBMS, where a library is almost certainly necessary to mediate between the relational data model of the database and the object-oriented model of most modern programming languages. Still, even if the heft of an object-relational mapper isn't required, many developers like using a thin wrapper over the drivers to handle associations, validations, and type checking.

На момент написания этой книги компания IOgen официально поддерживала драйверы для языков C, C++, C#, Erlang, Haskell, Java, Perl, PHP, Python, Scala и Ruby - и этот перечень постоянно расширяется. Если вам нужна поддержка для какого-то другого языка, то очень может быть, что уже имеется драйвер, написанный сообществом. А если такового не окажется, то на сайте mongodb.org опубликованы спецификации для разработки драйверов. Поскольку все официально поддерживаемые драйверы активно применяются в производственных системах и распространяются на условиях лицензии Apache, то авторы новых драйверов не испытывают недостатка в хороших примерах. Начиная с главы 3, я буду описывать, как работают драйверы и как ими пользоваться при написании программ.

Командные утилиты:

1. `mongodump` и `mongoexport` – стандартные утилиты резервного копирования и восстановления базы данных, `mongodump` сохраняет данные во

внутреннем формате BSON и потому применяется главным образом для создания резервных копий. Достоинством этой программы является возможность использования для снятия горячих резервных копий, из которых впоследствии можно восстановить базу данных с помощью утилиты `mongorestore`.

2. `mongoexport` и `mongoimport` – эти утилиты экспортируют и импортируют файлы в форматах JSON, CSV и TSV, поэтому они полезны, когда требуется представить данные в одном из широко распространенных форматов, `mongoimport` также удобна для начальной загрузки больших наборов данных, хотя стоит отметить, что перед импортом часто бывает желательно подправить модель данных, так чтобы можно было в полной мере задействовать все преимущества MongoDB. В таких случаях проще импортировать данные с помощью специально написанной программы на одном из поддерживаемых языков.

3. `mongosniff` – анализатор протокола для просмотра команд, посылаемых серверу базы данных. По сути дела транслирует передаваемые команды в формате BSON в понятные человеку команды оболочки..

4. `mongostat` – аналог `iostat`; постоянно опрашивает MongoDB и операционную систему для выдачи полезной статистики, в том числе количества операций (вставки, выборки, обновления, удаления и т. п.) в секунду, объема выделенной виртуальной памяти и числа подключений к серверу.

7.1.3 MongoDB и прочие базы данных

Количество имеющихся на рынке СУБД растет лавинообразно, и сравнивать одну с другой становится все труднее. К счастью, большинство из них попадает в одну из нескольких категорий. В последующих разделах я опишу простые и более изощренные хранилища ключей и значений, реляционные и документные базы данных и сопоставлю их с MongoDB.

Таблица 7.1 – Семейства баз данных

	Пример	Модель данных	Масштабирование	Применение
Простые хранилища ключей и значений	Memcached	Ключ-значение, где значением может быть произвольный двоичный объект.	Разные. Memcached может масштабироваться на несколько узлов, представляя всю доступную память в виде единого монолитного хранилища данных.	Кэширование, веб-приложения.
Развитые хранилища ключей и значений	Cassandra, Project Voldemort, Riak	Разные. В Cassandra используется структура ключ-значение, называемая столбцом. В Voldemort хранятся двоичные объекты.	Согласованное в конечном счете аспределение по узлам, обеспечивающее высокую доступность и простую отработку отказа.	Вертикальные приложения с высокой пропускной способностью (веб-каналы активности, очереди сообщений). Кэширование. Веб-приложения.
Реляционные СУБД	Oracle database, MySQL, PostgreSQL	Таблицы.	Вертикальное масштабирование. Ограниченная поддержка кластеризации и ручного сегментирования.	Системы, нуждающиеся в транзакциях (банковские и финансовые приложения) или в языке SQL. Нормализованная модель данных.

Назначение следует из названия - индексирование значение по ключу. Типичное применение - кэширование. Пусть, например, требуется кэшировать HTML-страницы, генерируемые приложением. В этом случае ключом может быть URL-адрес, а значением - сама HTML-страница. Отметим, что с точки зрения хранилища значение представляет собой непрозрачный массив байтов. Нет ни предопределенной схемы, как в реляционных базах данных, ни какого-либо понятия о типе данных. Это налагает естественное ограничение на операции, выполняемые хранилищами ключей и значений: можно только поместить новое значение, а затем получить или удалить его по ключу. Столь простые системы обычно работают очень быстро и хорошо масштабируются.

Из всех простых хранилищ ключей и значений наиболее широкое распространение получило memcached. Memcached хранит все данные в оперативной памяти, то есть приносит долговечность в жертву быстродействию. Хранилище распределенное - узлы memcached, размещенные на нескольких серверах, могут работать как единое хранилище данных, устраняя тем самым сложности, связанные с отслеживанием состояния кэша на разных машинах.

По сравнению с MongoDB простые хранилища ключей и значений типа memcached часто демонстрируют более высокую скорость операций чтения и записи. Но, в отличие от MongoDB, они редко используются в качестве основного хранилища данных. Они больше подходят на роль придатков, например, слоя кэширования поверх традиционной СУБД или простого слоя хранения для таких недолговечных структур, как очереди задач.

Простую модель ключей и значений можно развить, сделав ее пригодной для обработки более сложных схем чтения-записи или для предоставления улучшенной модели данных. Примером такого развитого хранилища ключей и значений может служить система Dynamo, разработанная компанией Amazon и описанная в широко известном документе «Dynamo: Amazon's Highly Available Key-value Store». Задача Dynamo - предложить базу данных, которая могла бы продолжать работу в условиях сетевых сбоев, отключения питания в центре обработки данных и других подобных отказов.

Таким образом, требуется, чтобы система при любых обстоятельствах обеспечивала чтение и запись, а это значит, что данные должны автоматически реплицироваться на несколько узлов. Если какой-то узел выходит из строя, то обслуживание пользователя системы, быть может, клиента Amazon, положившего что-то в корзину, не будет прервано. Dynamo располагает средствами разрешения неизбежных конфликтов, возникающих, когда системе разрешено записывать одни и те же данные на несколько узлов. В то же время Dynamo легко масштабируется. Поскольку в системе нет главного узла - все узлы.

7.1.4 Области применения MongoDB

MongoDB подходит на роль основного хранилища данных для веб-приложений. Даже в простом веб-приложении имеется много моделей данных для представления пользователей, сеансов, специфических данных, зачек и разрешений, не говоря уже о собственно предметной области. Все это прекрасно ложится как на табличную структуру, предлагаемую реляционными СУБД, так и на модель коллекций и документов в MongoDB. А поскольку с помощью документа можно представить сложные структуры данных, то количество коллекций обычно оказывается меньше, чем количество таблиц, необходимых для моделирования тех же данных в полностью нормализованной реляционной модели. Кроме того, с помощью динамических запросов и вторичных индексов можно без труда реализовать большинство запросов, знакомых пользователям SQL. Наконец, по мере роста приложения MongoDB предлагает простой механизм масштабирования.

На практике MongoDB подтвердила способность справляться со всеми аспектами приложения - от основных задач предметной области до таких вспомогательных аспектов, как протоколирование и аналитика в реальном времени. Примером может служить новостной сайт The Business Insider (TBE), который применяет MongoDB в качестве основного хранилища данных с января 2008 года. TBE обслуживает более миллиона уникальных просмотров страниц в день. Интересно, что помимо обработки основного содержимого сайта (статьи, комментарии, пользователи и т. д.), MongoDB также обрабатывает и хранит аналитические данные в режиме реального времени. Эти данные используются TBE для динамической генерации карт распределения интереса, на которых отражается кликабельность (CTR) различных новостей. Объем данных на сайте пока не настолько велик, чтобы требовалось сегментирование, но наборы реплик тем не менее используются для гарантированного перехода на резервный ресурс в случае отказа.

Гибкая разработка

Как бы вы относились к движению за гибкую разработку, трудно отрицать, что приложение желательно создавать быстро. Немало коллективов, в частности *Shutterfly* и *The New York Times*, выбрали *MongoDB* отчасти за то, что она позволяет разрабатывать приложения быстрее, чем при использовании реляционных СУБД. Одна из причин, лежащая на поверхности, заключается в отсутствии фиксированной схемы, что позволяет не тратить время на разработку схемы, ее распространение и внесение в нее изменений.

Кроме того, меньше времени уходит на втискивание реляционного представления данных в объектно-ориентированную модель, на борьбу с капризами системы ORM и на оптимизацию сгенерированного ей кода. Поэтому *MongoDB* часто используется в проектах с коротким циклом разработки, над которыми трудятся гибкие команды среднего размера.

Аналитика и протоколирование.

Я уже упоминал, что *MongoDB* хорошо приспособлена для аналитики и протоколирования, и число приложений, в которых эта СУБД используется для таких целей, быстро растет. Нередко солидные компании с прочной репутацией начинают вторжение в мир *MongoDB* со специальных приложений, относящихся в аналитике. В качестве примеров приведу *GitHub*, *Disqus*, *Justin.tv* и *Gilt Groupe*. Для аналитических приложений *MongoDB* привлекательна своим быстродействием и двумя важными особенностями: атомарным обновлением и ограниченными коллекциями (*capped collections*). Атомарное обновление позволяет клиенту эффективно увеличивать счетчики и помещать новые значения в массив. Ограниченные коллекции, часто применяемые для протоколирования, имеют фиксированный размер, то есть старые элементы автоматически удаляются в порядке FIFO. Хранение протоколируемых данных в базе, а не в файловой системе упрощает администрирование и существенно расширяет возможности запросов. Теперь вместо использования *gper* или специализированной утилиты поиска в журнале пользователь может написать запрос на знакомом языке запросов *MongoDB*.

Кэширование.

Благодаря модели данных, позволяющей представлять объекты целиком, и повышенной скорости выполнения запросов MongoDB можно использовать вместо более традиционной связки MySQL и memcached. Например, вышеупомянутый сайт TBE смог отказаться от memcached в пользу непосредственной генерации страниц из данных, хранящихся в MongoDB.

7.2 Оболочка MongoDB Shell

В этой

главе мы с помощью оболочки MongoDB изучим основные концепции этой СУБД на ряде примеров. Вы научитесь создавать, читать, обновлять и удалять (create, read, update, delete - CRUD) документы и по ходу дела познакомитесь с языком запросов MongoDB. Кроме того, мы немного поговорим об индексах базы данных и об их применении для оптимизации запросов. И закончим главу рассмотрением некоторых простых административных команд, после чего я покажу, как можно получать справку по ходу работы. Можете считать эту главу одновременно более подробным изложением идей, упомянутых во введении, и практическим занятием по наиболее употребительным способам использования оболочки MongoDB.

7.2.1 Sell Basics

Если вы выполнили инструкции, приведенные в приложении А, то сейчас на ваш компьютер установлен работоспособный экземпляр MongoDB. Убедитесь, что процесс mongod работает, а затем запустите оболочку MongoDB:

```
./mongo
```

Если все нормально, то на экране появится приглашение, как на рис. 7.4. В заголовке отображается версия MongoDB, с которой вы работаете, и сведения о текущей базе данных. Если вы хоть немного знакомы с языком

JavaScript, то можете сразу же приступить к вводу кода и исследованию возможностей оболочки. В противном случае читайте дальше, и вы узнаете, как вставить свой первый документ.

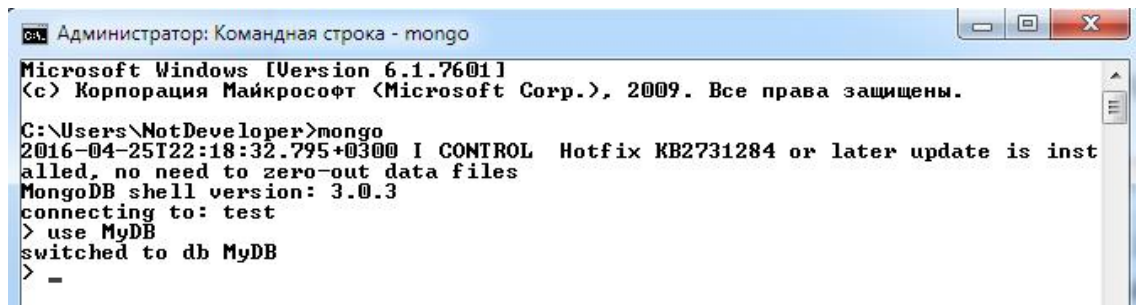


Рисунок 7.4 – Запуск оболочки MongoDB

Если при запуске не указана база данных, то по умолчанию оболочка выбирает базу test. Но чтобы во всех последующих упражнениях оставаться в одном и том же пространстве имен, переключимся на базу данных MyDB.

Возможно, у вас возник вопрос, как можно переключиться на базу данных tutorial, не создав ее явно. На самом деле, создавать базу данных необязательно. Базы данных и коллекции создаются при вставке первого документа. Такое поведение согласуется с принятым в MongoDB динамическим подходом к данным - точно также, как структуру документов не нужно определять заранее, так и отдельные коллекции и базы можно создавать прямо во время выполнения. Это упрощает и ускоряет процесс разработки и позволяет динамически назначать пространства имен, что очень часто бывает полезно. Тем не менее, если вас пугает возможность непреднамеренного создания базы данных или коллекции, то в большинстве драйверов имеется строгий режим, предотвращающий такого рода случайные ошибки.

Но пора уже создать первый документ. Так как мы работаем с JavaScript-оболочкой, то документы представляются в формате JSON (JavaScript Object Notation). Простейший документ, описывающий одного пользователя, мог бы выглядеть так:

```
{username: "Jones"}
```

Этот документ содержит одну пару ключ-значение для хранения имени пользователя Jones. Чтобы сохранить этот документ, нужно указать коллекцию. Коллекция users вполне подойдет:

```
> db.users.insert({username: "Jones"})
```

После ввода этого предложения может наблюдаться небольшая задержка. В это время на диске создается база данных tutorial и коллекция users. Задержка обусловлена созданием начальных файлов для того и другого.

Если вставка завершилась успешно, то ваш первый документ сохранен. Чтобы убедиться в этом, можно выполнить простой запрос:

```
> db.users.find()
```

Ответ будет выглядеть примерно так:

```
> db.users.find<>
{ "_id" : ObjectId<"571e6f5e0a89f929f77a3d16">, "username" : "Jones" }
> -
```

Рисунок 7.5 – Выполнение простого запроса

Обратите внимание, что в документ добавлено поле `_id`. Можете считать его значение первичным ключом документа. В любом документе MongoDB обязательно присутствует поле `_id`, и если в момент создания его значение не задано, то MongoDB генерирует специальный идентификатор объекта. На вашей консоли отобразится не тот же идентификатор, что в примере выше, но он гарантированно будет уникален среди всех значений `_id` в данной коллекции - единственное непрекаемое требование к этому полю.

Добавим нового пользователя:

```
> db.users.save({username: "Michel"})
```

Имея в коллекции более одного документа, мы можем попробовать более сложные запросы. Как и раньше, можно запросить все документы в коллекции:

```
> db.users.count()
```

```
2
```

Но можно также передать методу `find` простой селектор запроса. Селектором запроса называется документ, с которым сравниваются все

документы в коллекции. Чтобы найти все документы, в которых поле username равно jones, нужно задать такой селектор:

```
> db.users.find()
{ _id : ObjectId("4bf9bec50e32f82523389314"), username : "Jones" }
{ _id : ObjectId("4bf9bec90e32f82523389315"), username : "Michel" }
```

Селектор запроса {username: "jones"} возвращает все документы, в которых имя пользователя содержит строку jones - документообразец буквально сравнивается со всеми хранящимися в коллекции документами:

```
> db.users.find({username: "jones"})
{ _id : ObjectId("4bf9bec90e32f82523389315"), username : "Jones" }
```

Для обновления нужно задать по меньшей мере два аргумента. Первый определяет, какие документы обновлять, второй - как следует модифицировать отобранные документы. Существует два способа модификации; в этом разделе мы рассмотрим направленную модификацию (targeted modification) - одну из наиболее интересных и уникальных особенностей MongoDB.

Предположим, что пользователь smith решил указать свою страну проживания. Сделать это можно следующим образом:

```
> db.users.update({username: "Jones"}, {$set: {country: "Russia"}})
```

Здесь мы просим MongoDB найти документ, в котором поле username равно smith, и записать в свойство country значение Canada. Если теперь запросить этот документ, то мы увидим, что он обновился:

```
> db.users.find({username: "Jones"})
{"_id" : ObjectId("4bf9ec440e32f82523389316"),
"country" : "Russia", username : "Jones"}
```

Разовьем этот пример. Данные представляются в виде документов, которые, как мы видели в главе 1, могут содержать сложные структуры данных. Предположим, что, помимо профиля, пользователь желает хранить списки своих любимых вещей. Представление такого документа могло бы выглядеть так:

```
> db.users.update({username: "smith"}, {$unset: {country: 1}})
```

Ключ `favorites` указывает на объект, содержащий два других ключа, указывающих на списки любимых городов и фильмов. Можете ли вы придумать, как с помощью того, что вам уже известно, модифицировать исходный документ о пользователе `smith` так, чтобы он принял такой вид? Нам должен сразу прийти оператор `$set`. Отметим, что в этом случае мы практически полностью переписываем документ, и `$set` ничего не имеет против: (Рисунок 7.6):

```
{
  username : "Jones",
  favorites : {
    cities : ["Moscow", "Stavropol"],
    movies : ["Crazy Cops", "The War", "12"]
  }
}
```

Рисунок 7.6 – Complex Document

Точка между `favorites` и `movies` означает, что нужно найти ключ `favorites`, который указывает на вложенный объект с ключом `movies`, а затем сравнить значение этого вложенного ключа с указанным в запросе. Этот запрос вернет оба документа.:

```
> db.users.update({username : "Jones"}, {"$set": {favorites : {movies : ["12", "The War", "Crazy! Crazy!"]}}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

Рисунок 7.7 – Update complex document

Если не задавать никаких параметров, то операция удаления `remove` удалит из коллекции все документы. Так, чтобы избавиться от коллекции `foo`, нужно выполнить такую команду.:

```
> db.users.remove()
```

You often need to remove only a certain subset of a collection's documents, and for that, you can pass a query selector to the `remove()` method. If you want to remove all users whose favorite city is `Stavropol`, the expression is pretty straightforward:

```
> db.users.remove({"favorites.cities": "Stavropol"})
```

Отметим, что операция `remove()` не уничтожает саму коллекцию, а лишь удаляет из нее документы. Можете считать ее аналогом команд `SQL DELETE` и `TRUNCATE`. Чтобы уничтожить коллекцию вместе со всеми построенными над ней индексами, используйте метод `drop()`:

```
> db.users.drop()
```

Создание, чтение, обновление и удаление - основные операции в любой базе данных. Если вы читали внимательно, то теперь можете самостоятельно поэкспериментировать с операциями `CRUD` в `MongoDB`. В следующем разделе мы продолжим изучение операций выборки, обновления и удаления, познакомившись с вторичными индексами.

7.2.2 Создание индексов и применение их в запросах

Индексы обычно создаются для повышения скорости выполнения запросов. К счастью, оболочка `MongoDB` позволяет создавать индексы без всякого труда. Если ранее вам не доводилось работать с индексами базы данных, то из этого раздела вам станет ясно, зачем они нужны. А если у вас уже есть опыт работы с индексами, то вы узнаете, как просто они создаются в `MongoDB` и как можно профилировать выполнение запросов с помощью метода `explain()`.

Индексировать коллекцию имеет смысл, только когда в ней много документов. Поэтому добавим 200 000 простых документов в коллекцию `numbers`. Поскольку оболочка `MongoDB` одновременно является интерпретатором `JavaScript`, то сделать это несложно: (Рисунок 7.8).

```
> for (i=0; i<200000; i++){db.numbers.save({num:i})}
WriteResult({ "nInserted" : 1 })
> db.numbers.count()
200000
> _
```

Рисунок 7.8 – Создание коллекции документов

200 000 документов - это немало, так что не удивляйтесь, если на выполнение команды уйдет несколько секунд. По завершении вы можете с помощью парочки запросов убедиться, что все документы на месте::

```

> db.numbers.find()
{ "_id" : ObjectId("571e773c0a89f929f77a3d17"), "num" : 0 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d18"), "num" : 1 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d19"), "num" : 2 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d1a"), "num" : 3 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d1b"), "num" : 4 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d1c"), "num" : 5 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d1d"), "num" : 6 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d1e"), "num" : 7 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d1f"), "num" : 8 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d20"), "num" : 9 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d21"), "num" : 10 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d22"), "num" : 11 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d23"), "num" : 12 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d24"), "num" : 13 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d25"), "num" : 14 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d26"), "num" : 15 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d27"), "num" : 16 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d28"), "num" : 17 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d29"), "num" : 18 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d2a"), "num" : 19 }
Type "it" for more
>

```

Рисунок 7.9 – Выполнение выборки

Команда `count()` показывает, что вставлено 200 000 документов. Второй запрос выводит первые 20 результатов. Чтобы показать следующую порцию, выполните команду `it`:

```

Type "it" for more
> it
{ "_id" : ObjectId("571e773c0a89f929f77a3d2b"), "num" : 20 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d2c"), "num" : 21 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d2d"), "num" : 22 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d2e"), "num" : 23 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d2f"), "num" : 24 }

```

Рисунок 7.10 – Запрос подмножества

Команда `it` просит оболочку вернуть следующий результирующий набор. Имея набор документов приличного размера, попробуем выполнить несколько запросов. Вас уже не удивит, что для поиска документа по его атрибуту `num` достаточно такого простого запроса:

```

> db.numbers.find({num : 500})
{ "_id" : ObjectId("571e773c0a89f929f77a3f0b"), "num" : 500 }

```

Более интересны запросы по диапазону, для которых предназначены специальные операторы `$gt` и `$lt`. Мы уже встречались с ними в главе 1 (`gt` означает `greater than` [больше], а `lt` - `less than` [меньше]). Вот как запросить документы, для которых значение `num` больше 199 995:

```
> db.numbers.find({num : {"$gt" : 199995}})
{ "_id" : ObjectId("571e77730a89f929f77d4a53"), "num" : 199996 }
{ "_id" : ObjectId("571e77730a89f929f77d4a54"), "num" : 199997 }
{ "_id" : ObjectId("571e77730a89f929f77d4a55"), "num" : 199998 }
{ "_id" : ObjectId("571e77730a89f929f77d4a56"), "num" : 199999 }
>
```

Рисунок 7.11 – Использование оператора \$gt

Как видите, с помощью простого JSON-документа можно сформулировать сложный запрос по диапазону, как на языке SQL. \$gt и \$lt - всего два из многочисленных ключевых слов, используемых в языке запросов MongoDB; в последующих главах мы встретим много других примеров.

Разумеется, от таких запросов мало толку, если они выполняются неэффективно. В следующем разделе мы впервые задумаемся об эффективности и начнем изучать имеющиеся в MongoDB средства индексирования.

Если вы долго работали с реляционными базами данных, то, наверное, знакомы с командой SQL explain. Она описывает путь выполнения запроса и позволяет выявить медленные операции, показывая, какие индексы были использованы. В MongoDB тоже имеется вариант explain с аналогичной функциональностью. Чтобы понять, как эта команда работает, применим ее к одному из предыдущих запросов. Выполните такую команду::

```
> db.numbers.find( { num: {"$gt": 199995 } } ).explain()
```

Результат должен выглядеть примерно так, как показано в листинге ниже.

```
{
  "cursor" : "BasicCursor",
  "nscanned" : 200000,
  "nscannedObjects" : 200000,
  "n" : 4,
  "millis" : 171,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : { }
}
```

Рисунок 7.12 – Результат команды explain()

Изучение распечатки, выданной explain () показывает, что для

возврата всего четырех результатов (п) серверу пришлось просканировать всю коллекцию из 200 000 (nscanned) документов. Тип курсора BasicCursor подтверждает, что для формирования результирующего набора индексы не использовались. Столь большая разница между количеством просмотренных и возвращенных документов свидетельствует о неэффективности выполнения запроса. В реальном приложении, когда и коллекция больше, и сами документы объемнее, время выполнения запроса окажется существенно больше 171 миллисекунды, как в данном примере.

Этой коллекции явно не хватает индекса. Построить индекс по ключу num можно с помощью метода ensure index (). Введите такую команду:

```
> db.numbers.ensureIndex({num: 1})
```

Как и для любой другой операции MongoDB, например выборки или обновления, методу ensureIndex () передается документ, определяющий, по каким ключам следует индексировать. В данном случае документ {num: 1} говорит, что над коллекцией numbers нужно построить индекс по ключу num в порядке возрастания. Убедиться в том, что индекс действительно построен, позволит метод get Indexes ():

```
> db.numbers.getIndexes()
[
  {
    "name" : "_id_",
    "ns" : "tutorial.numbers",
    "key" : {
      "_id" : 1
    }
  },
  {
    "_id" : ObjectId("4bfc646b2f95a56b5581efd3"),
    "ns" : "tutorial.numbers",
    "key" : {
      "num" : 1
    },
    "name" : "num_1"
  }
]
```

Рисунок 7.13 – Результат команды getIndexes()

Теперь над этой коллекцией построено два индекса. Первый - по стандартному ключу _id - автоматически строится для любой коллекции;

второй - по ключу `num` - мы только что создали сами. Если сейчас выполнить тот же запрос с помощью метода `explain()`, то будет заметна ощутимая разница во времени выполнения, что отражено в листинге ниже.

```
> db.numbers.find({num: {"$gt": 199995 }}).explain()
{
  "cursor" : "BtreeCursor num_1",
  "indexBounds" : [
    [
      {
        "num" : 199995
      },
      {
        "num" : 1.7976931348623157e+308
      }
    ]
  ],
  "nscanned" : 5,
  "nscannedObjects" : 4,
  "n" : 4,
  "millis" : 0
}
```

Рисунок 7.14 – Результат команды `explain()`

Теперь, когда используется индекс по ключу `num`, запрос просматривает только пять документов. В результате общее время выполнения сократилось со 171 мс до менее чем 1 мс.

Заключение

Мы рассмотрели много вопросов. Подведем итоги. MongoDB - документо-ориентированная СУБД с открытым исходным кодом. Будучи спроектирована с учетом требований к данным и масштабируемости, предъявляемым современными веб-приложениями, MongoDB поддерживает динамические запросы и вторичные индексы, быстрое атомарное обновление и сложные способы агрегирования, репликацию с автоматической отработкой отказов и сегментирование, обеспечивающее масштабирование по горизонтали.

Мы также познакомимся с JavaScript-оболочкой MongoDB, которая весьма полезна для интерактивной работы с базой.

Контрольные вопросы

1. В каком случае оправдан выбор MongoDB?
2. Каковы основные компоненты MongoDB?
3. Опишите различия MongoDB и прочими типами СУБД.
4. Опишите стратегии применения MongoDB.
5. Что такое MongoDB Shell? Какие типы операций поддерживает MongoDB shell?
6. Какой оператор используется для создания документов?
7. Какой оператор используется для получения количество документов в коллекции?
8. Поясните механизм инициализации коллекции в MongoDB.
9. Опишите значение терминов: collection, document, index и document-oriented database.

ЗАКЛЮЧЕНИЕ

В пособии были рассмотрены множество теоретических аспектов Big Data. Подведем итоги. MongoDB - документо-ориентированная СУБД с открытым исходным кодом. Будучи спроектирована с учетом требований к данным и масштабируемости, предъявляемым современными веб-приложениями, MongoDB поддерживает динамические запросы и вторичные индексы, быстрое атомарное обновление и сложные способы агрегирования, репликацию с автоматической обработкой отказов и сегментирование, обеспечивающее масштабирование по горизонтали.

Мы познакомились с документной моделью данных на практике и продемонстрировали типичные операции MongoDB. Вы научились создавать индексы и с помощью команды `explain ()` убедились, что индексы повышают производительность выполнения запросов. Вы также умеете получать информацию о базах данных и коллекциях, имеющихся в системе, и знаете всё об «умной» коллекции `$cmd`. А если потребуется помощь, то у вас найдется парочка трюков, которые помогут выбраться на правильную дорогу. Многому можно научиться, работая в оболочке MongoDB, но ничто не сможет заменить опыт создания реального приложения.

СПИСОК ЛИТЕРАТУРЫ

Основная литература

1. Лэм, Ч. Hadoop в действии. / Чак Лэм. – М.: ДМК Пресс, 2012. – 424 с. – ISBN 978-5-94074-785-7.
2. Бэнкер, К. MongoDB в действии, пер. А. Слинкин / Кайл Бэнкер. – М.: ДМК Пресс, 2014. – 394 с. – ISBN 978-5-94074-831-1, 978-1-93518-287-0, 978-5-97060-057-3.
3. Фаулер, М. NoSQL: новая методология разработки нереляционных баз данных. Пер. с англ. / М. Фаулер, Дж. Садаладж. – М.: Вильямс, 2013. – 192 с.
4. Venner, J. Pro Hadoop. 2nd edition. / Jason Venner. – Apress; 2nd ed. (September 9, 2014). – 444 p.
5. Holmes, A. Hadoop in Practice / Alex Holmes. – Manning Publications; 2nd ed. (October 12, 2014). – 512 p.

Дополнительная литература

6. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google File System”; pub. 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October 2003. URL: <http://labs.google.com/papers/gfs.html>
7. Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”; pub. OSDI’04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December 2004. URL: <http://labs.google.com/papers/mapreduce.html>
8. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. “Bigtable: A Distributed Storage System for Structured Data”; pub. OSDI’06:

Seventh Symposium on Operating System Design and Implementation, Seattle, WA, November 2006. URL: <http://labs.google.com/papers/bigtable.html>.

9. Mike Burrows. “The Chubby Lock Service for Loosely-Coupled Distributed Systems”; pub. OSDI’06: Seventh Symposium on Operating System Design and Implementation, Seattle, WA, November 2006. URL: <http://labs.google.com/papers/chubby.html>.

10. <http://adoop.apache.org>.

11. White, T. Hadoop: The Definitive Guide. 4th edition. / Tom White. – O'Reilly Media; 4th ed. (April 11, 2015). – 756 p. ISBN-10: 1491901632.

12. Marz, N. Big Data: Principles and best practices of scalable realtime data systems. / Nathan Marz. – Manning Publications; 1st ed. (May 10, 2015). – 328 p.

Основы NoSQL СУБД

УЧЕБНОЕ ПОСОБИЕ

Автор

Николаев Евгений Иванович

Редактор, техническая правка,
компьютерная верстка:

Подписано в печать _____ г.

Формат 60х84 1/16 Усл.п.л. – 4,25. Уч.-изд.л. – 3,4.

Бумага газетная. Печать офсетная. Заказ № ____ Тираж ____ экземпляров

ФГАОУ ВПО «Северо-Кавказский федеральный университет»

355000, г. Ставрополь, ул. Пушкина, 1

Издательство СКФУ