Big data and data storages

Оглавление

Базовые понятия и их соотношения. Предпосылки возникновения. Свойства. Примеры.

В 2001 году в попытке охарактеризовать и визуализировать вероятные изменения в будущем, Дуглас Лэйни из META Group (сейчас Gartner) предложил три измерения, которые характеризуют проблемы и возможности: объем данных, скорость и разнообразие, известные как 3V (Volume, Velocity, Variety) больших данных. Таким образом, по данным Gartner

«Большие данные» - это объемные, быстро доступные и разнообразные информационные ресурсы, которые требуют рентабельных, инновационных форм обработки информации для лучшего понимания и принятия решений.

Это определение намеренно субъективно и включает в себя гибкое определение того, насколько большим должен быть объем данных, чтобы считаться большими данными. Например, большими данными оперируют Amazon и Google и это сильно отличается от больших данных для компаний среднего размера в сфере страхования или телекоммуникаций.

Как следствие такой недосказанности, появилось множество разных определений больших данных, но в целом они говорят о таких данных, размер которых превышает способность типовых СУБД собирать, хранить, управлять и анализировать их. А также анализ и обработки которых может иметь большую ценность для бизнеса.

Годами добавлялись новые V в описания больших данных, что видно из таблицы ниже. Новые V:

Veracity – достоверность

Validity – действительность

Volatility – изменчивость

Value – ценность

Visualization – визуализация

Vulnerability – уязвимость

Variability - изменчивость

3 Vs	Volume	Vast amount of data that has to be captured, stored, processed and displayed
	Velocity	Rate at which the data is being generated, or analyzed
	Variety	Differences in data structure (format) or differences in data sources themselves (text, images, voice, geospacial data)
5 Vs	Veracity	Truthfulness (uncertainty) of data, authenticity, provenance, accountability
	Validity	Suitability of the selected dataset for a given application, accuracy and correctness of the data for its intended use
7 Vs	Volatility	Temporal validity and fluency of the data, data currency and availability, and ensures rapid retrieval of information as required
	Value	Usefulness and relevance of the extracted data in making decisions and capacity in turning information into action
10 Vs	Visualization	Data representation and understandability of methods (data clustering or using tree maps, sunbursts, parallel coordinates, circular network diagrams, or cone trees)
	Vulnerability	Security and privacy concerns associated with data processing
	Variability	the changing meaning of data, inconsistencies in the data, biases, ambiguities, and noise in data

Раньше корпорации имели дело со статическими централизованно хранимыми данными, собранными из различных источников, с появлением Интернета и облачных сервисов, облачные вычисления стремительно обгоняют традиционные. Таким образом, большие наборы данных – лог файлы, посты в социальных сетях, потоки «кликов» - больше не находятся на центральном сервере или в фиксированном месте. Чтобы справиться с большими объемами данных, необходимы продвинутые аналитические инструменты, которые могут обрабатывать и хранить миллиарды байтов данных в реальном времени с сотнями тысяч транзакций в секунду.

Когда говорят о больших данных – обычно говорят об экосистеме больших данных, которая включает в себя:

- Источники данных.
- Средства организации управления данными (интеграция, хранение и обработка данных).
- Средства анализа данных, BI (business intelligence) системы, KD (knowledge discovery) системы.

Примеры подобных экосистем с их основными характеристиками представлены в таблице ниже.

Цель этого модуля - познакомиться с определениями, методами, инструментами, фреймворками и решениями для обработки больших данных, начиная с самого процесса извлечения информации, через обработку и представления знаний, к хранению, визуализации, осмысления данных и созданию практических приложений. Будут рассмотрены вопросы проблемы хранения и управления данными, коммуникации, вычислений. Как получить полезную информацию из наборов данных, которые слишком громоздкие и сложные для обработки многими традиционными или классическими методами.

Facebook	Facebook (2018) has more than two billion users on **millions of servers**, running thousands of **configuration changes** every day involving trillions of configuration checks [310]
LinkedIn	It takes a lot of horsepower to support LinkedIn's 467 million members worldwide (in 2017), especially when you consider that each member is getting a **personalized experience** and a **web page** that includes only their contacts. Supporting the load are some 100,000 servers spread across **multiple data centers** [215]
Alibaba	The 402,000 **web-facing computers** that Alibaba hosts (2017) from China-allocated IP addresses would alone be sufficient to make Alibaba the second largest **hosting company** in the world today [321]
Google	There's no official data on how many servers there are in Google's data centers, but Gartner estimated in a July 2016 report that Google at the time had 2.5 **million servers**. Google data centers process an average of 40 million searches per second, resulting in 3.5 billion searches per day and 1.2 **trillion searches per year**, Internet Live Stats reports [390]
Amazon	... an estimate of 87 AWS datacenters in total and a range of somewhere between 2.8 and 5.6 million servers in Amazon's **cloud** (2014) [301]
Twitter	Twitter (2013) now has 150M worldwide **active users**, handles 300K queries per second (QPS) to generate timelines, and a firehose that churns out 22 MB/s. Some 400 million tweets a day flow through the system and it can take up to 5 min for a tweet to flow from Lady Gaga's fingers to her 31 million followers [197]

Примеры больших данных для организаций:

Хотя каждая организация будет иметь свой собственный набор требований к данным для обработки больших данных, вот несколько примеров:

● **Данные о погоде** - существует множество данных о погоде, предоставляемых правительственными агентствами по всему миру, научными организациями и потребителями. То, что мы слышим по телевидению или радио, является ключевым аналитическим показателем эффективности (KPI) температуры и прогнозируемых условий, основанным на нескольких факторах.

● **Данные контрактов** - существует множество типов контрактов, которые организация выполняет каждый год, и с каждым из них связано множество обязательств.

● **Данные о рабочей силе** – изменяющаяся (состав) рабочая сила порождает ряд проблем, которые необходимо решить организациям.

● **Данные технического обслуживания** - записи технического обслуживания оборудования, машин, систем, не связанных с компьютером, и т. д.

● **Данные финансовой отчетности** - отчеты о деятельности компании.

● **Данные о соответствии** - финансовые, медицинские, медико-биологические, больницы и многие другие агентства, которые хранят данные о соответствии для своих корпораций.

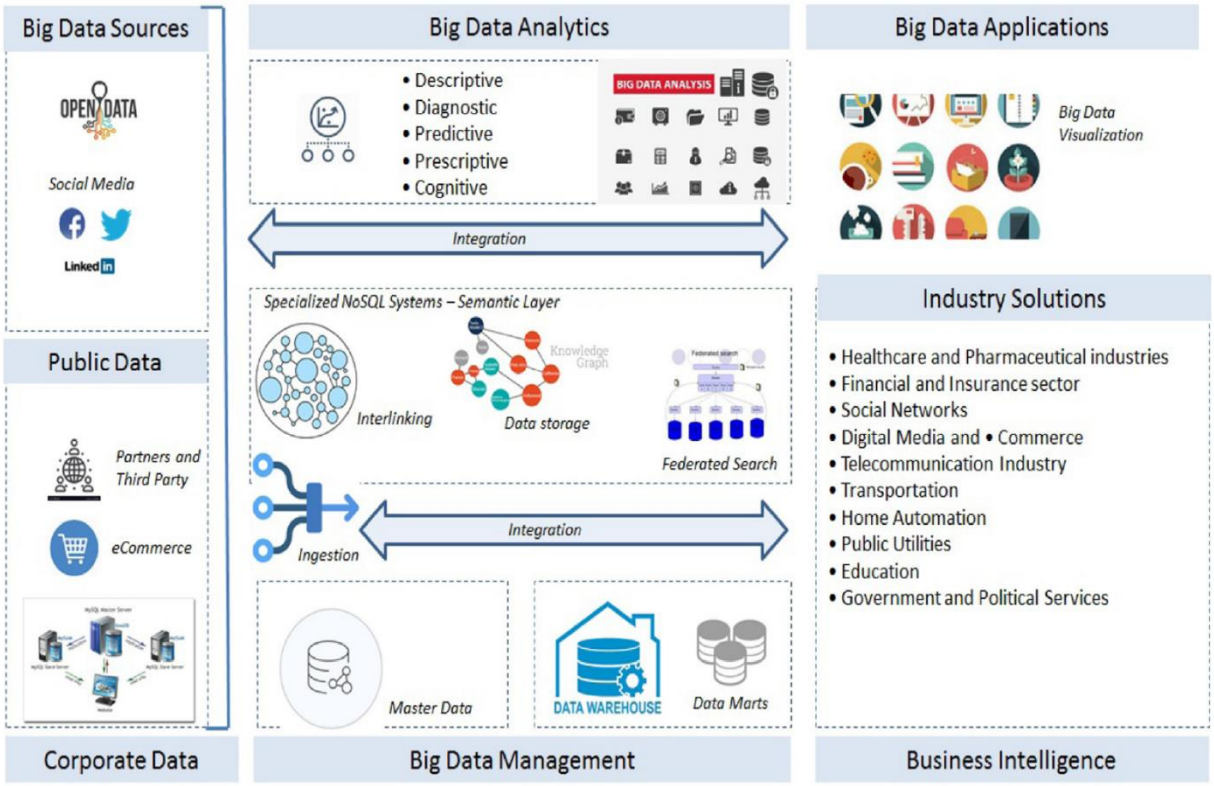
● **Данные клинических испытаний** - фармацевтические компании хотели минимизировать жизненный цикл обработки данных клинических испытаний и управлять им с помощью обработки на основе правил; это возможность для больших данных.

● **Обработка записей врачей по диагностике и лечению** - еще одна ключевая область скрытых идей и значение для управления болезненным состоянием и упреждающей диагностики; ключевая возможность машинного обучения.

● **Контракты** - каждая организация ежегодно заключает множество типов контрактов и должна обрабатывать и анализировать содержание контрактов вместе с показателями для измерения рисков и штрафов.

Источники данных

В современной экосистеме данных слой источников данных состоит из как частных, так и общедоступных источников данных - см. левую часть рис.



.

Типы источников данных:

1. **Направленные данные (Directed data)** – реальные данные разнообразных информационных систем, таких как: видеонаблюдения, аэрофотосъемки и пр.
2. **Автоматизированные данные (Automated data)** - В автоматизированных системах данные генерируются автоматически. Зачастую, такие данные также автоматически и автономно обрабатываются и анализируются и могут быть использованы для автоматизированного управления процессами и явлениями.
3. **Данные систем автоматизированного мониторинга (Automated surveillance)**
4. **Цифровые устройства (Digital devices)** - камеры, видео, телескопы, устройства GPS, различные формы медицинского оборудования, мобильные телефоны, кабельные и спутниковые приемники, которые генерируют данные, касающиеся как они используются (например, время, местоположение, вызываемый абонент / просматриваемый канал), с соответствующими данные собираются в виде журналов, которые передаются третьим сторонам (например, производителям устройств и поставщики услуг).
5. **Считываемые данные (Sensed data)** – данные датчиков и исполнительных механизмов.
6. **Данные сканеров (Scan data)** – данные различных сканеров идентификаторов устройств (штрих-коды, магнитные метки и пр.)
7. **Данные взаимодействий (Interaction data)** – данные о поведении пользователей при взаимодействии с устройствами и программами.

8. **Данные, предоставленные добровольцами (Volunteered data)** – OpenStreetMap, Википедия и пр.
9. **Данные транзакций (Transactions)** – данные об онлайн покупках, статистика поисковых запросов, кликов и пр.
10. **Данные социальных сетей (Social media)** – Facebook, VK, LinckedIn и пр.
11. **Данные управления своим здоровьем (Sousveillance)** - это самоконтроль и управление личным здоровьем и жизнью посредством цифровых технологий (например, оборудование для фитнеса, носимые компьютеры), которые записывают медицинские данные в отношении отдельного человека
12. **Краудсорсинг (Crowdsourcing)** - привлечение к решению тех или иных проблем инновационной производственной деятельности широкого круга лиц для использования их творческих способностей, знаний и опыта по типу субподрядной работы на добровольных началах с применением информационных технологий
13. **Гражданская наука (Citizen science)** - это особая форма краудсорсинга, в которой «сообщества или сети граждан действуют как наблюдатели в какой-то области науки. В этом случае люди генерируют, бесплатно подготавливают и обрабатывают эмпирические наблюдения и измерения явлений, которые имеют реальную ценность как данные для «правильной» науки.

Примеры источников открытых данных для разных доменов:

- **Facebook API Graph** — это основной инструмент для загрузки данных на платформу Facebook и их получения оттуда. Он представляет собой API на базе HTTP, с помощью которого приложения могут программным путем запрашивать данные, публиковать новые истории, управлять рекламой, загружать фото и выполнять множество других задач. Название API Graph подчеркивает связь этого API с "социальным графом" — системой представления информации на Facebook. API Graph состоит из следующих элементов:

    - **узлы** — отдельные "объекты" (например, Пользователь, Фото, Страница или Комментарий);
    - **границы контекста** — связи между подборкой объектов и отдельным объектом, например фото на Странице или комментарии к фото;
    - **поля** — данные об объекте, например день рождения пользователя или название Страницы.

  С помощью узлов можно получать данные о конкретном объекте, границы контекста позволяют собирать подборки объектов, связанные с отдельным объектом, а поля помогают получать данные об отдельном объекте или о каждом объекте в подборке.
- **Open Corporates** - одна из крупнейших открытых баз данных компаний в world и хранит сотни миллионов наборов данных практически в любой стране. Сервис представляет REST API для доступа к данным, доступные форматы вывода данных – JSON и XML.
- **Global Financial Data** - рекомендуется для аналитиков, которым требуется большие объемы данных для широких исследовательских нужд. Эти данные позволяет исследователям изучать взаимодействие между различными рядами данных, секторами и видами данных. В API поддерживает R и Python, поэтому данные можно напрямую загружать в разрабатываемое приложение.
- **Open Street Map** - это карта мира, созданная людьми, которые могут бесплатно использовать открытая лицензия. Он поддерживает картографические данные на тысячах веб-сайтов, мобильных приложениях и устройствах.
- **The National Centers for Environmental Information (NCEI)** - отвечает за хостинг и предоставление доступа к одному из наиболее значимых мировых архивов, с обширными океаническими, атмосферными и геофизическими данными.

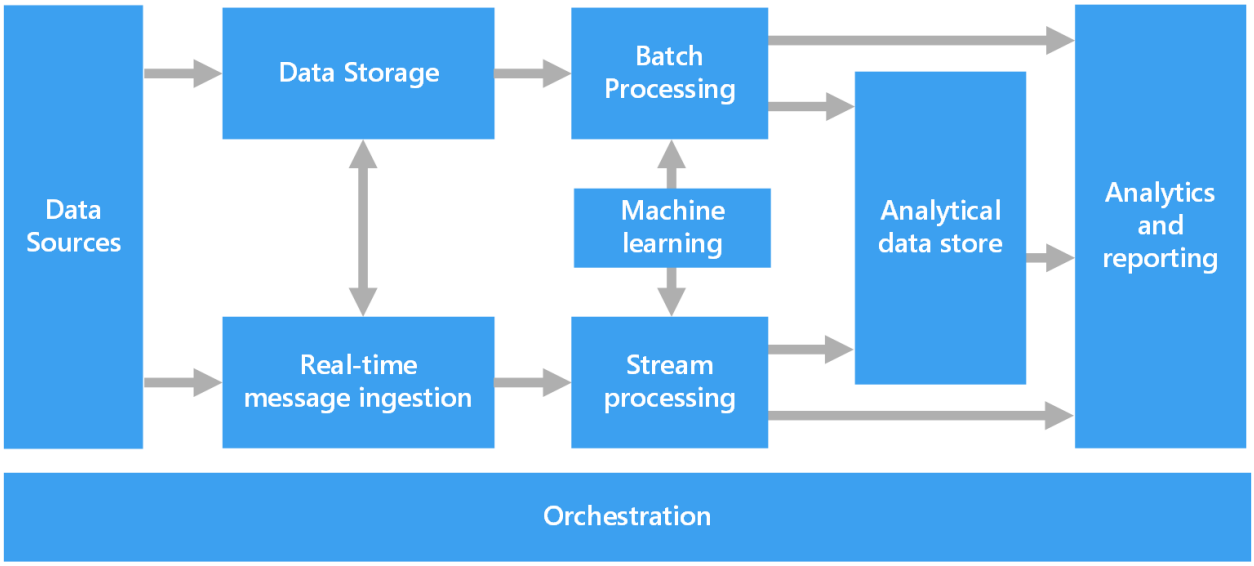
- **DBPedia** - это семантическая версия Википедии. Это решение помогло таким компаниям, как Apple, Google и IBM в поддержке проектов искусственного интеллекта.

Классификация источников данных с точки зрения характера предоставляемых ими данных и методов их обработки.

1. **Текстовые данные.** Методы обработки: Preprocessing, Named Entity Recognition (NER), Entity Linking (EL), Relation Extraction (RE), Joint tasks
2. **Размеченные данные** (например, лог файлы). Методы обработки: Wrapper-based extraction, Web table extraction, Deep Web crawling
3. **Структурированные данные** (таблицы, деревья, базы данных, графы знаний). Методы обработки: Mapping from tables, Mapping from trees, Mapping from other knowledge graphs

Компоненты архитектуры для обработки больших данных

На схеме ниже показаны логические компоненты, которые входят в архитектуру для обработки больших данных. Отдельные решения могут не содержать все компоненты в этой схеме.



- **Источники данных.** Все решения для обработки больших данных начинаются с одного или нескольких источников данных. Примеры приведены ниже:
  - Хранилища данных приложений, например реляционные базы данных.
  - Статические файлы, которые создаются приложениями, например файлы журнала веб-сервера.
  - Источники данных с передачей в режиме реального времени, например устройства Интернета вещей.
- **Хранилище данных.** Данные для пакетной обработки обычно хранятся в распределенном хранилище файлов, где могут содержаться значительные объемы больших файлов в различных форматах. Этот тип хранилища часто называют озером данных. Такое хранилище можно реализовать с помощью Azure Data Lake Store или контейнеров больших двоичных объектов в службе хранилища Azure.
- **Пакетная обработка.** Так как наборы данных очень велики, часто в решении обрабатываются длительные пакетные задания. Для них выполняется фильтрация, статистическая обработка и другие процессы подготовки данных к анализу. Обычно в эти задания входит чтение исходных файлов, их обработка и запись выходных данных в новые файлы. Варианты: выполнение заданий U-SQL в Azure Data Lake Analytics, использование пользовательских заданий Hive, Pig или Map/Reduce в кластере HDInsight Hadoop и применение программ Java, Scala или Python в кластере HDInsight Spark.
- **Прием сообщений в реальном времени.** Если решение содержит источники в режиме реального времени, в архитектуре должен быть предусмотрен способ сбора и сохранения сообщений в режиме реального времени для потоковой обработки. Это может быть простое хранилище данных с папкой, в которую входящие сообщения помещаются для обработки. Но для приема сообщений многим решениям требуется хранилище, которое можно использовать в качестве буфера. Такое хранилище должно поддерживать обработку с горизонтальным масштабированием, надежную доставку и другую семантику

очереди сообщений. Эта часть архитектуры потоковой передачи часто называется потоковой буферизацией. Варианты: Центры событий Azure, Центр Интернета вещей и Kafka.

- **Потоковая обработка.** Сохранив сообщения, поступающие в режиме реального времени, система выполняет для них фильтрацию, статистическую обработку и другие процессы подготовки данных к анализу. Затем обработанные потоковые данные записываются в выходной приемник. Azure Stream Analytics предоставляет управляемую службу потоковой обработки на основе постоянного выполнения запросов SQL для непривязанных потоков. Кроме того, для потоковой передачи можно использовать технологии Apache с открытым кодом, например Storm и Spark Streaming в кластере HDInsight.
- **Хранилище аналитических данных.** Во многих решениях для обработки больших данных данные подготавливаются к анализу. Затем обработанные данные структурируются в соответствии с форматом запросов для средств аналитики. Хранилище аналитических данных, используемое для обработки таких запросов, может быть реляционной базой данных типа Kimball, как можно увидеть в большинстве традиционных решений бизнес-аналитики (BI). Кроме того, данные можно представить с помощью технологии NoSQL с низкой задержкой, такой как HBase или интерактивная база данных Hive, которая предоставляет абстракцию метаданных для файлов данных в распределенном хранилище. Azure Synapse Analytics — это управляемая служба для хранения больших объемов данных в облаке. HDInsight поддерживает Interactive Hive, HBase и Spark SQL, которые также можно использовать, чтобы предоставлять данные для анализа.
- **Анализ и создание отчетов.** Большинство решений по обработке больших данных предназначены для анализа и составления отчетов, что позволяет получить важную информацию. Чтобы расширить возможности анализа данных, можно включить в архитектуру слой моделирования, например модель таблицы или многомерного куба OLAP в Azure Analysis Services. Также можно включить поддержку самостоятельной бизнес-аналитики с использованием технологий моделирования и визуализации в Microsoft Power BI или Microsoft Excel. Анализ и создание отчетов также может выполняться путем интерактивного изучения данных специалистами по их анализу и обработке. Для таких сценариев многие службы Azure поддерживают функции аналитического блокнота, например Jupyter, который позволяет пользователям применять свои навыки работы с Python или R. Для крупномасштабного изучения данных можно использовать Microsoft R Server (отдельно или со Spark).
- **Оркестрация.** Большинство решений для обработки больших данных состоят из повторяющихся рабочих процессов, во время которых преобразуются исходные данные, данные перемещаются между несколькими источниками и приемниками, обработанные данные загружаются в хранилища аналитических данных либо же результаты передаются непосредственно в отчет или на панель мониторинга. Чтобы автоматизировать эти рабочие процессы, вы можете использовать технологию оркестрации, такую как фабрика данных Azure или Apache Oozie и Sqoop.

Потоковая передача против пакетной обработки

Организации используют облачные технологии и большие данные для продвижения культуры принятия решений на основе данных, модернизации центров обработки данных и улучшения структуры традиционных информационных технологий. Облачные вычисления во многих аспектах изменили отрасль анализа данных.

В частности, облачные технологии обеспечивают правильную структуру управления данными. Однако здесь возникает проблема ускорения обработки больших объемов данных. Пакетная

обработка и потоковая обработка - два основных решения для более быстрой обработки больших наборов данных.

Обзор пакетной обработки

При пакетной обработке большие объемы данных обрабатываются пакетом или группой в течение определенного временного интервала. Система пакетной обработки запрограммирована так, что после подачи набора файлов данных в качестве входных данных она обрабатывает данные, а затем создает набор файлов данных в качестве выходных данных.

Вам нужно дождаться накопления определенного количества необработанных данных, прежде чем запускать процесс извлечения, преобразования, загрузки (ETL) в пакетной обработке. ETL представляет три функции базы данных (извлечение, преобразование, загрузка), которые извлекают данные из базы данных, преобразуют данные, выполняя конкатенации, вычисления и загружая данные в другую систему хранилища данных.

Обычно при пакетной обработке данные доступны для анализа через час или несколько дней. Пакетный ETL запускается по заданному расписанию, например, каждые 24 часа. Вы также можете настроить систему на запуск пакетного ETL, когда данные достигнут определенного предела.

Обзор потоковой обработки

Обработка в потоке, данные обрабатываются, как только они поступают на уровень хранения, в отличие от пакетной обработки, когда вам нужно дождаться накопления данных. Сгенерированные данные обрабатываются в субсекундных таймфреймах.

Для конечных пользователей обработка данных происходит в режиме реального времени. Поскольку это операция без сохранения состояния, обработка данных включает в себя только простой расчет или преобразование.

Потоковая обработка запрашивает непрерывный поток данных и быстро определяет условия в течение ограниченного времени. Системы потоковой обработки получают данные о действиях, происходящих в режиме реального времени, таких как клики на веб-страницах, показания датчиков, транзакции электронной коммерции, сообщения в социальных сетях и многое другое.

Потоковая обработка используется для обнаружения сложных мировых проблем и обеспечения разумного ответа для лучшего результата. Методы машинного обучения для профилактического обслуживания основаны на потоковой обработке. Профилактическое обслуживание в режиме реального времени обеспечивает быстрое обнаружение и классификацию развивающихся неисправностей, а также их местоположения.

Использование сложной технологии обработки событий позволяет отслеживать состояние систем в режиме реального времени. Таким образом, выявление причин неисправностей при оценке состояния технических компонентов. Эта информация обеспечивает основу для инициирования различных мероприятий по техническому обслуживанию в рамках планирования технического обслуживания.

Плюсы и минусы пакетной обработки

Плюсы

- Пакетная обработка наиболее применима при работе с большими объемами данных или транзакциями.
- Обработка данных происходит самостоятельно.
- Это экономичный метод обработки данных.

Минусы

- Пользователи должны тщательно подготовить входные данные, предназначенные для пакетной обработки, прежде чем запускать их на компьютере.
- Проблемы с данными, сбои программы и ошибки, возникающие во время пакетной обработки, могут остановить весь процесс. К ним относятся незначительные ошибки в данных, такие как опечатки в датах.

Плюсы и минусы потоковой обработки

Плюсы

- При потоковой обработке данные всегда актуальны. Таким образом, организация может реагировать на проблемы и события в кратчайшие сроки.
- Данные обновляются в режиме реального времени, чтобы помочь организациям выявлять закономерности и анализировать возможные угрозы или возможности.
- Случаи задержки при выполнении потоковой обработки минимальны.

Минусы

- Потоковая обработка - это дорого и сложно.
- Информационный аудит при потоковой обработке является сложной задачей.

Пакетная обработка используется в:

- Системы расчета заработной платы для расчета заработной платы всех сотрудников. Системы расчета заработной платы собирают, хранят все связанные данные и обрабатывают счет как пакет в конце определенного периода, скажем, месяца.
- Системы биллинга компаний-операторов связи для обработки миллионов записей о звонках и расчета тарифов на услуги связи.
- Системы банковских счетов для ежемесячных выписок для всех владельцев счетов.
- Обрабатывающие отрасли для ежедневного отчета о деятельности линейки продуктов.

Потоковая обработка используется в:

- Решения для обнаружения мошенничества с транзакциями в Интернете для мгновенного реагирования. Платформы потоковой обработки обнаруживают аномалии в транзакциях электронной торговли в режиме реального времени и предотвращают завершение мошеннических транзакций.
- Приложения, требующие непрерывного вывода входящих данных, таких как анализ настроений в социальных сетях, показания датчиков и фондовый рынок.

Заключение

Как потоковая, так и пакетная обработка имеют свои преимущества и недостатки в зависимости от конкретного проекта. Организации, которые хотят оставаться гибкими, как правило, чаще используют потоковую обработку, в то время как те, которые определены устаревшими системами, в основном используют пакетную обработку.

Наиболее важным фактором для групп обработки данных является гибкость. У разных проектов разные требования, и эта статья подготовила команды, чтобы найти лучшее решение для обработки данных для каждого варианта использования.

Лямбда архитектура

Вычисление произвольных функций на произвольном наборе данных в реальном времени - серьезная проблема. Не существует единого инструмента, обеспечивающего полное решение. Вместо этого вы должны использовать различные инструменты и методы для построения полноценной системы больших данных. Основная идея лямбда-архитектуры состоит в построении систем больших данных в виде ряда уровней, как показано на рисунке 1.6. Каждый уровень удовлетворяет подмножеству свойств и основывается на функциональности, предоставляемой нижележащими слоями. Все начинается с уравнения query = function (all data). В идеале вы можете запускать функции на лету, чтобы получать результаты. К сожалению, даже если бы это было возможно, для этого потребовалось бы огромное количество ресурсов и было бы неоправданно дорого.

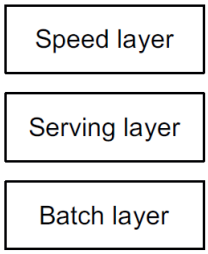


Figure 1.6 Lambda Architecture

Представьте, что вам нужно читать петабайтный набор данных каждый раз, когда вы хотите ответить на запрос о чьем-то текущем местоположении. Наиболее очевидный альтернативный подход - предварительное вычисление функции запроса. Назовем предварительно вычисленную функцию запроса пакетным представлением. Вместо того, чтобы вычислять запрос на лету, вы читаете результаты из предварительно вычисленного представления. Предварительно вычисленное представление индексируется, чтобы к нему можно было получить доступ с помощью случайных чтений. Эта система выглядит так:

batch view = function (all data)

query = function (batch view)

В этой системе вы запускаете функцию для всех данных, чтобы получить пакетное представление. Затем, когда вы хотите узнать значение запроса, вы запускаете функцию в этом пакетном представлении. Пакетное представление позволяет очень быстро получить из него нужные значения без необходимости сканировать все в нем. Поскольку это обсуждение несколько абстрактно, позвольте обосновать его примером. Предположим, вы создаете приложение для веб-аналитики и хотите запросить количество просмотров страницы для URL-адреса за любой диапазон дней. Если бы вы вычисляли запрос как функцию всех данных, вы бы просканировали набор данных на предмет просмотров страниц для этого URL в пределах этого временного диапазона и вернули бы количество этих результатов. Подход пакетного просмотра вместо этого запускает функцию для всех просмотров страниц, чтобы предварительно вычислить индекс от ключа [url, day] до подсчета количества просмотров страниц для этого URL за этот день. Затем, чтобы разрешить запрос, вы извлекаете все значения из этого представления для всех дней в пределах этого временного диапазона и суммируете счетчики, чтобы получить результат. Этот подход показан на рисунке 1.7. Должно быть ясно, что в этом подходе, описанном выше, чего-то

не хватает. Очевидно, что создание пакетного представления будет операцией с высокой задержкой, поскольку она запускает функцию для всех имеющихся у вас данных. К тому времени, когда он закончится, будет собрано много новых данных, которые не представлены в пакетных представлениях, а запросы будут устаревать на много часов. Но давайте пока проигнорируем эту проблему, потому что мы сможем ее исправить. Давайте представим, что запросы могут устареть на несколько часов, и продолжим изучение идеи предварительного вычисления пакетного представления, запустив функцию для всего набора данных.

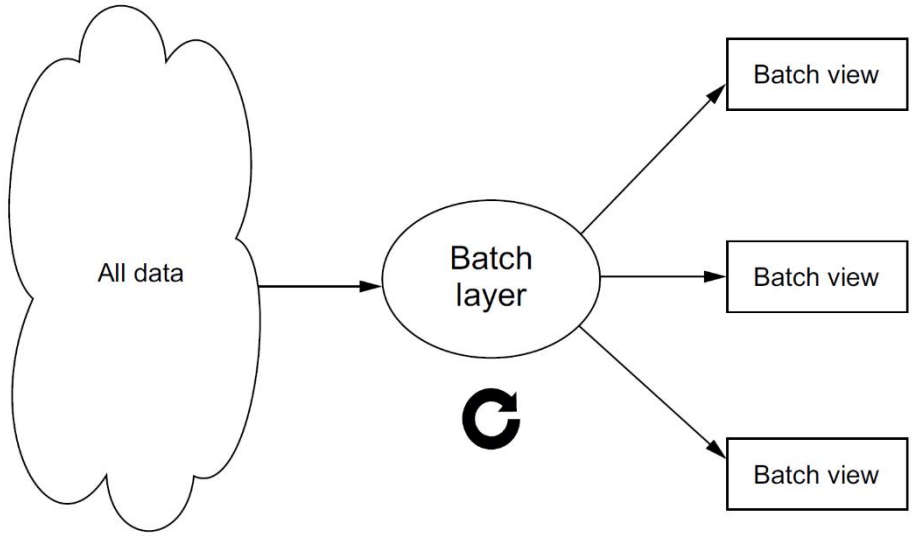


Figure 1.7 Architecture of the batch layer

Пакетный слой

Часть лямбда-архитектуры, которая реализует уравнение пакетное представление view = function(all data), называется пакетным уровнем. Пакетный слой хранит основную копию набора данных и предварительно вычисляет пакетные представления для этого основного набора данных (см. Рисунок 1.8). Главный набор данных можно рассматривать как очень большой список записей.

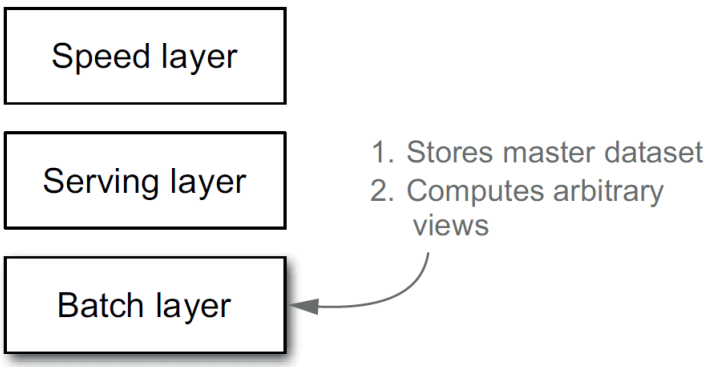


Figure 1.8 Batch layer

Пакетный уровень должен иметь возможность делать две вещи: хранить неизменяемый, постоянно растущий основной набор данных и вычислять произвольные функции для этого набора данных. Этот тип обработки лучше всего выполнять с помощью систем пакетной обработки. Hadoop - это канонический пример системы пакетной обработки, а Hadoop - это то, что мы будем использовать для демонстрации концепций пакетного уровня. Простейшую форму пакетного слоя можно представить в псевдокоде следующим образом:

```
function runBatchLayer():
    while(true):
        recomputeBatchViews()
```

Пакетный уровень запускается в цикле while (true) и постоянно пересчитывает пакетные представления с нуля. На самом деле пакетный уровень немного сложнее, но мы вернемся к этому позже в книге. На данный момент это лучший способ думать о пакетном слое.

Пакетный слой хорош тем, что его очень просто использовать. Пакетные вычисления написаны как однопоточные программы, и вы получаете параллелизм бесплатно. На пакетном уровне легко писать надежные, хорошо масштабируемые вычисления. Пакетный слой масштабируется за счет добавления новых машин. Вот пример вычисления пакетного слоя. Не беспокойтесь о понимании этого кода - суть в том, чтобы показать, как выглядит параллельная программа:

```
Api.execute(Api.hfsSeqfile("/tmp/pageview-counts"),
     new Subquery("?url", "?count")
         .predicate(Api.hfsSeqfile("/data/pageviews"),
             "?url", "?user", "?timestamp")
         .predicate(new Count(), "?count");
```

Этот код вычисляет количество просмотров страниц для каждого URL с учетом входного набора данных необработанных просмотров страниц. Что интересно в этом коде, так это то, что все проблемы параллелизма, связанные с планированием работы и объединением результатов, выполнены за вас. Поскольку алгоритм написан таким образом, его можно произвольно распределить в кластере MapReduce, масштабируя до любого количества доступных узлов. В конце вычисления выходной каталог будет содержать некоторое количество файлов с результатами.

Обслуживающий слой

Пакетный слой излучает пакетные представления в результате своих функций. Следующим шагом будет загрузка представлений куда-нибудь, чтобы их можно было запросить. Вот тут-то и появляется обслуживающий уровень. Обслуживающий уровень - это специализированная распределенная база данных, которая загружается в пакетном режиме и позволяет выполнять произвольное чтение из нее (см. Рисунок 1.9). Когда доступны новые пакетные представления, обслуживающий слой автоматически меняет их местами, чтобы были доступны более свежие результаты. База данных обслуживающего уровня поддерживает пакетные обновления и произвольные чтения. В частности, нет необходимости поддерживать произвольную запись. Это очень важный момент, поскольку случайные записи вызывают большую часть сложности в базах данных. Не поддерживая случайную запись, эти базы данных чрезвычайно просты. Эта простота делает их надежными, предсказуемыми, простыми в настройке и эксплуатации. ElephantDB, база данных уровня обслуживания состоит всего из нескольких тысяч строк кода.

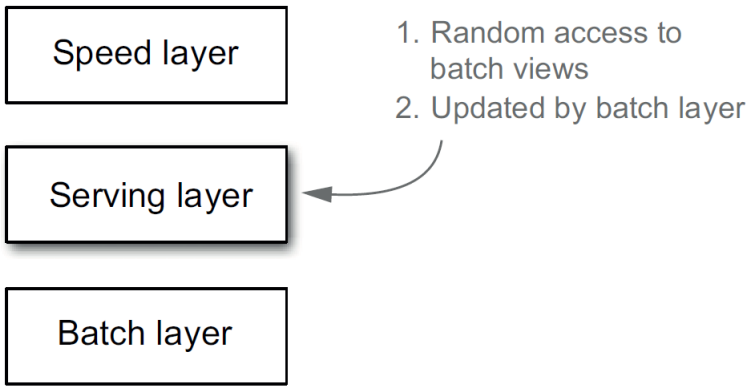


Figure 1.9 Serving layer

Пакетный и обслуживающий слои удовлетворяют почти всем свойствам

Уровни пакетной обработки и обслуживания поддерживают произвольные запросы к произвольному набору данных с учетом того, что запросы устареют на несколько часов. Новому фрагменту данных требуется несколько часов, чтобы распространиться через пакетный уровень на обслуживающий уровень, где он может быть запрошен. Важно отметить, что кроме обновлений с низкой задержкой, уровни пакетной обработки и обслуживания удовлетворяют всем свойствам, требуемым в системе больших данных. Давайте рассмотрим их один за другим:

■ **Надежность и отказоустойчивость** - Hadoop обрабатывает отказоустойчивость, когда машины выходят из строя. Уровень обслуживания использует внутреннюю репликацию, чтобы гарантировать доступность в случае отказа серверов. Уровни пакетной обработки и обслуживания также устойчивы к человеческим сбоям, потому что в случае ошибки вы можете исправить свой алгоритм или удалить неверные данные и пересчитать представления с нуля.

■ **Масштабируемость** - и пакетный, и обслуживающий уровни легко масштабируются. Это полностью распределенные системы, и масштабировать их так же просто, как добавлять новые машины.

■ **Обобщение.** Описанная архитектура является максимально общей. Вы можете вычислять и обновлять произвольные представления произвольного набора данных.

■ **Расширяемость** - добавить новое представление так же просто, как добавить новую функцию в основной набор данных. Поскольку основной набор данных может содержать произвольные данные, можно легко добавлять новые типы данных. Если вы хотите настроить представление, вам не нужно беспокоиться о поддержке нескольких версий представления в приложении. Вы можете просто пересчитать весь вид с нуля.

■ **Специальные запросы** - пакетный уровень изначально поддерживает специальные запросы. Все данные удобно доступны в одном месте.

■ **Минимальное обслуживание**. Основным компонентом, который необходимо обслуживать в этой системе, является Hadoop. Hadoop требует некоторых административных знаний, но работать с ним довольно просто. Как объяснялось ранее, базы данных обслуживающего уровня просты, потому что они не выполняют случайную запись. Поскольку в базе данных обслуживающего уровня так мало движущихся частей, меньше всего может выйти из строя. Как следствие, гораздо меньше вероятность того, что что-то пойдет не так с базой данных уровня обслуживания, поэтому их легче поддерживать.

■ **Возможность отладки** - входные и выходные данные вычислений всегда будут выполняться на уровне пакетной обработки. В традиционной базе данных выход может заменить исходный вход, например, при увеличении значения. В пакетном и обслуживающем слоях входными данными является основной набор данных, а выходными данными - представления. Точно так же у вас есть входы и выходы для всех промежуточных шагов. Наличие входов и выходов дает вам всю информацию, необходимую для отладки, когда что-то идет не так.

Прелесть пакетного и обслуживающего слоев в том, что они удовлетворяют практически всем желаемым свойствам с помощью простого и понятного подхода. Здесь нет проблем с параллелизмом, и он легко масштабируется. Единственное отсутствующее свойство - обновления с низкой задержкой. Последний слой, скоростной, решает эту проблему.

Скоростной слой

Уровень обслуживания обновляется всякий раз, когда пакетный уровень завершает предварительное вычисление пакетного представления. Это означает, что единственные данные, не представленные в пакетном представлении, - это данные, поступившие во время выполнения предварительных вычислений. Все, что осталось сделать, чтобы иметь систему данных полностью в реальном времени, то есть иметь произвольные функции, вычисляемые на произвольных данных в реальном времени, - требуется компенсировать эти последние несколько часов данных. Это и есть цель слоя скорости. Как следует из названия, его цель - обеспечить представление новых данных в функциях запросов так быстро, как это необходимо для требований приложения (см. Рисунок 1.10). Вы можете думать о скоростном уровне как о пакетном уровне, поскольку он создает представления на основе данных, которые он получает. Одно большое отличие состоит в том, что уровень скорости смотрит только на последние данные, тогда как пакетный уровень просматривает все данные сразу. Еще одно большое отличие состоит в том, что для достижения минимально возможных задержек уровень скорости не просматривает сразу все новые данные. Вместо этого он обновляет представления в реальном времени по мере получения новых данных, а не пересчитывает представления с нуля, как это делает пакетный уровень. Слой скорости выполняет инкрементные вычисления вместо пересчета, выполняемого на уровне пакетной обработки.

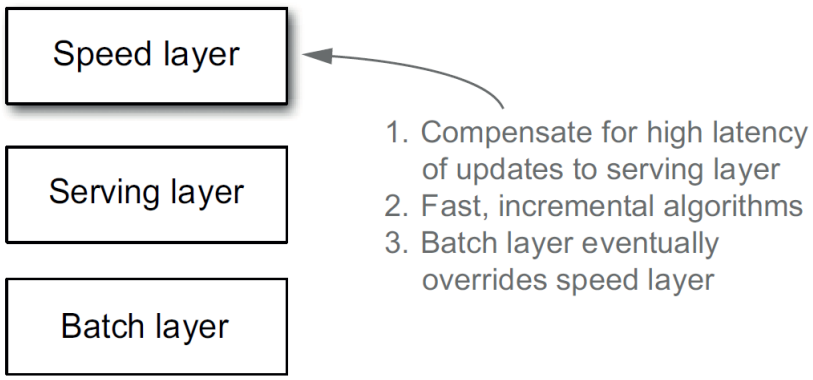


Figure 1.10 Speed layer

Мы можем формализовать поток данных на уровне скорости с помощью следующего уравнения:

realtime view = function(realtime view, new data)

Представление в реальном времени обновляется на основе новых данных и существующего представления в реальном времени. Лямбда-архитектура полностью описывается этими тремя уравнениями:

batch view = function(all data)

realtime view = function(realtime view, new data)

query = function(batch view. realtime view)

Наглядное представление этих идей показано на рисунке 1.11. Вместо того, чтобы разрешать запросы, просто выполняя функцию пакетного представления, вы разрешаете запросы, просматривая как пакетное представление, так и представление в реальном времени, и объединяя результаты вместе. Уровень скорости использует базы данных, которые поддерживают случайное чтение и случайную запись. Поскольку эти базы данных поддерживают произвольную запись, они на несколько порядков сложнее, чем базы данных, которые вы используете на обслуживающем уровне, как с точки зрения реализации, так и с точки зрения эксплуатации.

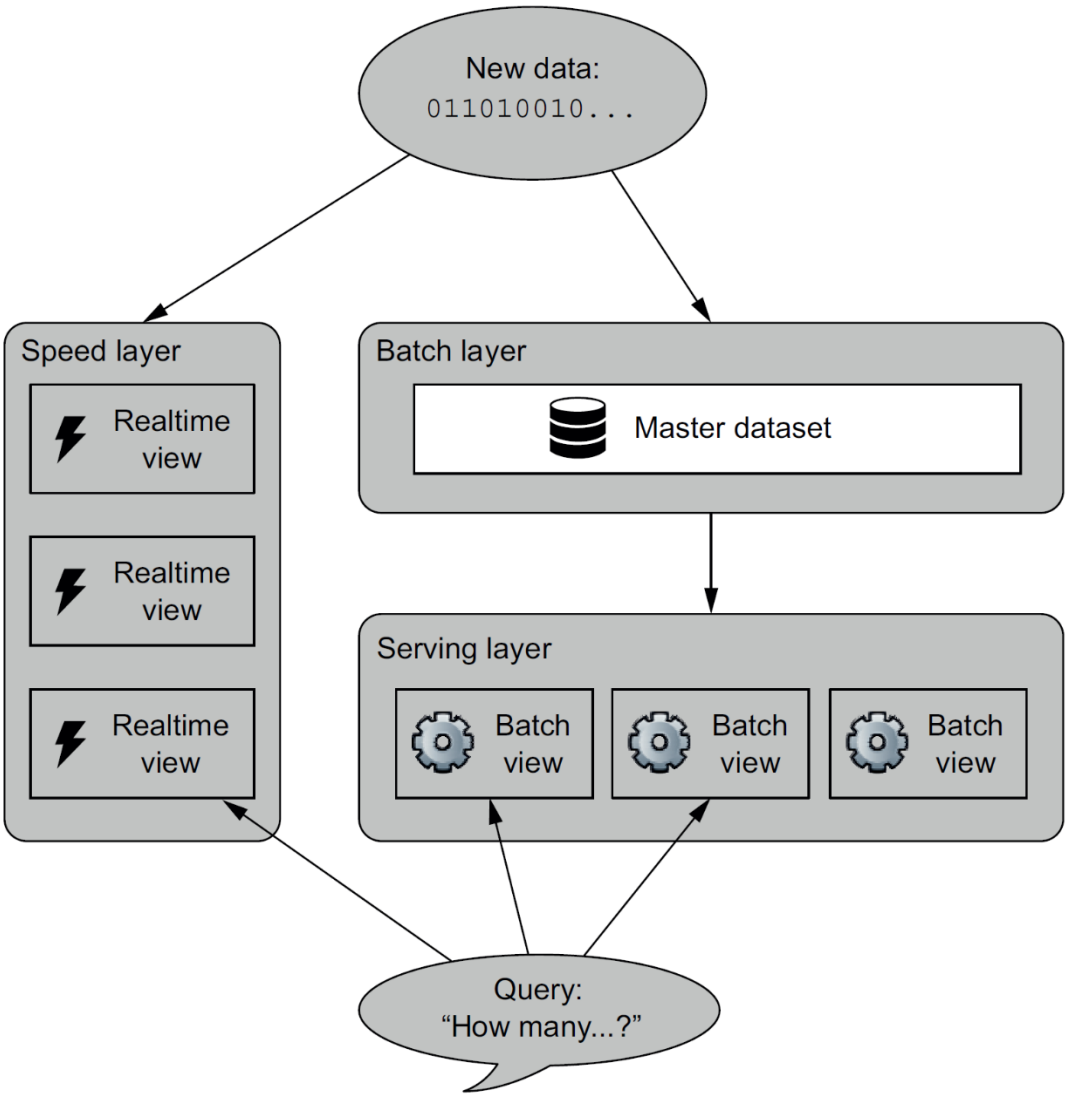


Figure 1.11 Lambda Architecture diagram

Преимущества лямбда-архитектуры заключаются в том, что после того, как данные проходят через пакетный уровень на обслуживающий уровень, соответствующие результаты в представлениях в реальном времени больше не нужны. Это означает, что вы можете отказаться от фрагментов

изображения в реальном времени, поскольку они больше не нужны. Это замечательный результат, потому что уровень скорости намного сложнее, чем слои пакетов и обслуживания. Это свойство лямбда-архитектуры называется изоляцией сложности, что означает, что сложность переносится на уровень, результаты которого являются временными. Если что-то пойдет не так, вы можете сбросить состояние для всего слоя скорости, и все вернется в нормальное состояние в течение нескольких часов. Продолжим пример создания приложения веб-аналитики, которое поддерживает запросы о количестве просмотров страниц в течение определенного диапазона дней. Напомним, что пакетный уровень производит пакетные просмотры от [url, day] до количества просмотров страницы. Слой скорости сохраняет свое собственное отдельное представление [url, день] и количество просмотров страниц. В то время как пакетный уровень пересчитывает свои представления, буквально подсчитывая просмотры страниц, уровень скорости обновляет свои представления, увеличивая счетчик в представлении всякий раз, когда он получает новые данные. Чтобы разрешить запрос, вы запрашиваете и пакетное представление, и представление в реальном времени, если это необходимо, чтобы удовлетворить заданному диапазону дат, и суммируете результаты, чтобы получить окончательный счет. Необходимо проделать небольшую работу, чтобы правильно синхронизировать результаты, но мы расскажем об этом в следующей главе. Некоторые алгоритмы сложно вычислить постепенно. Разделение уровня пакета / скорости дает вам возможность использовать точный алгоритм на уровне пакета и приблизительный алгоритм на уровне скорости. Пакетный слой многократно переопределяет уровень скорости, поэтому приближение корректируется, и ваша система демонстрирует свойство конечной точности. Например, подсчет количества уникальных посетителей может быть сложной задачей, если наборы уникальных посетителей становятся большими. На уровне пакета легко произвести уникальный подсчет, потому что вы просматриваете все данные сразу, но на уровне скорости вы можете использовать набор Hyper-Log Log в качестве приближения. В итоге вы получите лучшее из двух миров: производительности и надежности. Система, которая выполняет точные вычисления на уровне пакетной обработки и приблизительные вычисления на уровне скорости, демонстрирует конечную точность, потому что пакетный уровень корректирует то, что вычислено на уровне скорости. Вы по-прежнему получаете обновления с низкой задержкой, но поскольку уровень скорости является временным, сложность достижения этого не влияет на надежность ваших результатов. Неустойчивый характер уровня скорости дает вам гибкость, позволяющую действовать очень агрессивно, когда дело доходит до компромиссов для производительности. Конечно, для вычислений, которые могут быть выполнены точно поэтапно, система полностью точна.

Каппа архитектура

В блоге 2014 года Джей Крепс точно ввел термин «архитектура каппа», указав на подводные камни архитектуры лямбда и предложив потенциальную эволюцию программного обеспечения. Архитектура Kappa упрощает архитектуру Lambda, удаляя пакетный уровень и заменяя его потоковым слоем. Чтобы понять, как это возможно, нужно сначала понять, что пакет - это набор данных с началом и концом (ограниченным), в то время как поток не имеет ни начала, ни конца и является бесконечным (неограниченным). Поскольку пакетная обработка является ограниченным потоком, можно сделать вывод, что пакетная обработка является подмножеством потоковой обработки. Следовательно, результаты уровня пакетной обработки Lambda также могут быть получены с помощью механизма потоковой передачи. Это упрощение сводит архитектуру к единому механизму потоковой передачи, способному принимать необходимые объемы данных для обработки как пакетной обработки, так и обработки в реальном времени. Общая сложность

системы значительно уменьшается с архитектурой Карра. См. Рисунок 2:

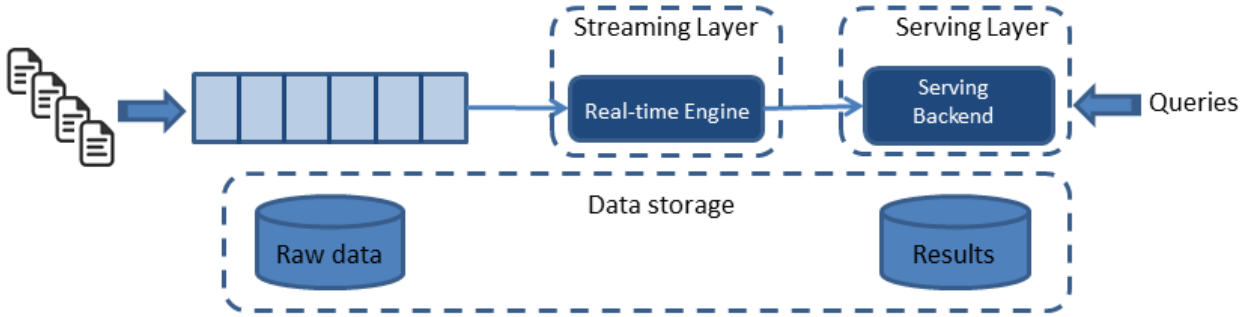


Рисунок 2. Каппа-архитектура. Рисунок любезно предоставлен Игнасио Муласом Виелой и Николасом Сейветом.

По сути, в архитектуре Карра есть четыре основных принципа:

- Все является потоком: пакетные операции становятся подмножеством потоковых операций. Следовательно, все можно рассматривать как поток.
- Неизменяемые источники данных: необработанные данные (источник данных) сохраняются, а представления производятся, но состояние всегда можно пересчитать, поскольку исходная запись никогда не изменяется.
- Единая аналитическая структура: принцип краткости и простоты (KISS). Требуется единый аналитический движок. Код, обслуживание и обновления значительно сокращены.
- Функциональность воспроизведения: вычисления и результаты могут развиваться путем воспроизведения исторических данных из потока.

Для соблюдения четвертого принципа конвейер данных должен гарантировать, что события останутся в порядке от генерации до приема. Это очень важно для гарантии согласованности результатов, так как это гарантирует детерминированные результаты вычислений. Двойное выполнение одних и тех же данных в вычислении должно привести к одинаковому результату.

Однако эти четыре принципа накладывают ограничения на построение аналитического конвейера.

Хранилища данных

В результате анализа существующих и будущих технологий хранения данных был сделан ряд выводов о технологиях хранения данных. Стало очевидно, что хранилище больших данных стало обычным бизнесом и что масштабируемое технологии хранения данных достигли уровня предприятия, позволяющего управлять виртуально неограниченные объемы данных. Об этом свидетельствует широкое использование решения на основе Hadoop, предлагаемые такими поставщиками, как Cloudera (2014), Hortonworks (2014) и MapR (2014), а также производители различных NoSQL БД, в частности те, которые используют технологии хранения в памяти и технологию хранения столбцов.

По сравнению с традиционными системами управления реляционными базами данных, которые полагаются на построчное хранилище и дорогостоящие стратегии кэширования - эти новые технологии хранения больших данных предлагают лучшую масштабируемость при меньшей сложности эксплуатации и меньших затратах.

Несмотря на эти достижения, улучшающие производительности, масштабируемости и удобства использования технологий хранения, есть еще значительный неиспользованный потенциал для развития технологий хранения больших данных:

- Возможность трансформации общества и бизнеса в разных секторах: технологии хранения больших данных являются ключевым фактором для расширенной аналитики, которая имеет потенциал для преобразования общества и способом принятия ключевых бизнес-решений. Это особенно важно в традиционно не связанных с ИТ секторах, таких как энергетика. Хотя эти секторы сталкиваются с нетехническими проблемами, такими как нехватка квалифицированных экспертов по большим данным и законодательные барьеры, тем не менее новые технологии хранения данных имеют потенциал для создания новой аналитики, создающей ценность, в различных отраслях промышленности.
- Отсутствие стандартов - серьезное препятствие: история создания и развития NoSQL основана на решение конкретных технологических задач, которые приводят к целому ряду различных технологий хранения данных. Большой выбор в сочетании с отсутствием стандартов для запроса данных затрудняют обмен данными между хранилищами, поскольку требуется разрабатывать конкретный код приложения для определенного хранилища.
- Проблемы открытой масштабируемости в хранилищах данных на основе графов: обработка данных, основанная на графовых структурах данных полезна во все большем количестве приложений. Это позволяет лучше улавливать семантикой и сложными отношениями с другими фрагменты информации, поступающие из большого количества различных источников данных, и имеет потенциал для повышения общей ценности, которая может быть получена путем анализа данных. Хотя для этой цели все чаще используются графовые базы данных, все еще составляет сложность эффективно распределить структуру данных на основе графа по вычислительным узлам.
- Конфиденциальность и безопасность отстают: хотя есть несколько проектов и решения, которые касаются конфиденциальности и безопасности, защиты людей, защита их данных отстает от технологических достижений в области систем хранения данных. Требуются серьезные исследования, чтобы лучше понять, как данные могут использоваться не по назначению, как необходимо защитить конфиденциальные данные и интегрировать их в хранилище больших данных.

Технологии хранения данных

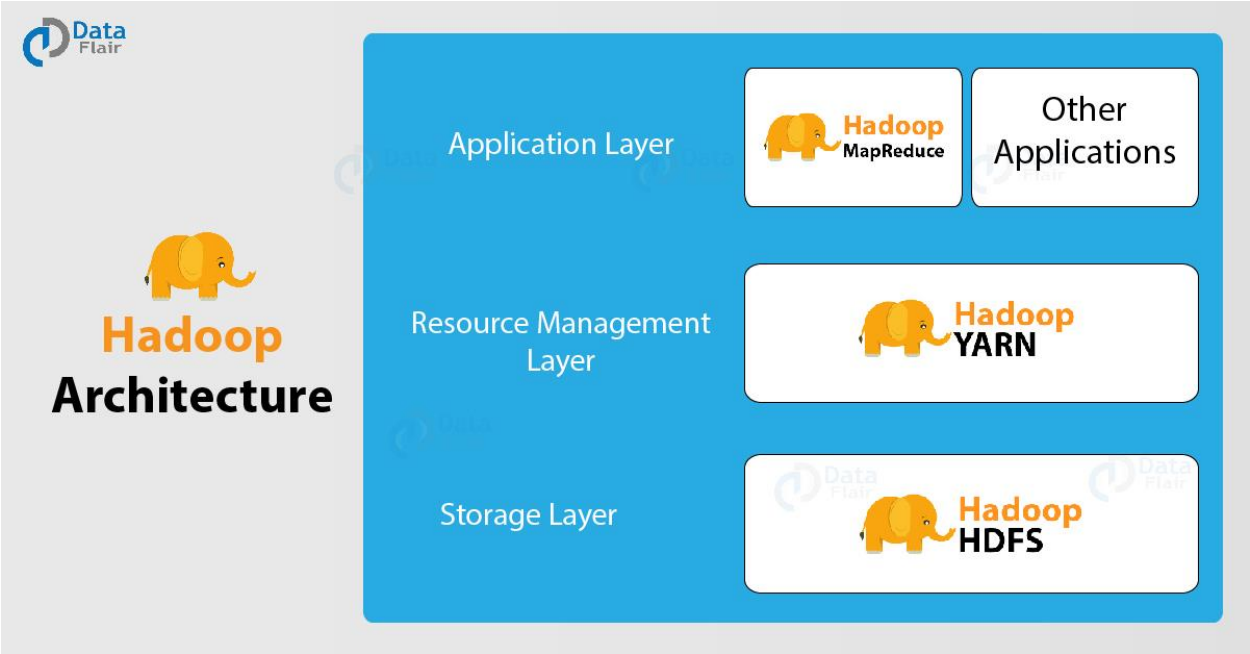
В последнее десятилетие мы имеем дело с взрывным ростом объемов данных (Turner et al., 2014) и переходом от вертикального к горизонтальному масштабированию хранилищ данных, которые в свою очередь отошли от традиционной реляционной модели данных. Эти подходы обычно приносят в жертву такие свойства, как согласованность данных, чтобы поддерживать быстрые ответы на запросы при увеличивающемся объеме данных. Хранилища больших данных используют аналогичные традиционным хранилищам системы управления, например как решение онлайн-обработки транзакций (OLTP) и хранилища структурированных или полуструктурированных данных. Особые преимущества в обработке неструктурированные и полуструктурированные данных состоят в их большом размере.

Далее будут даны оценки текущего состояния технологий хранилищ данных, которые способны обрабатывать большие объемы данных и идентифицировать связанные с хранилищами данных тенденции. Ниже приведены различные типы систем хранения:

- **Распределенные файловые системы**: файловые системы, такие как файловая система Hadoop (HDFS). (Швачко и др. 2010) предлагают возможность хранить большие объемы неструктурированных данных надежным способом на обычном оборудовании. Хотя есть файловые системы и с более высокой производительностью, HDFS является неотъемлемой частью Hadoop framework, которая уже достигла уровня де-факто стандарта. Она была разработана для больших файлов данных и хорошо подходит для быстрой загрузки данных и массовой обработки.
- **Базы данных NoSQL:** Возможно, самое важное семейство хранилищ больших данных. технологии - это системы управления базами данных NoSQL. Базы данных NoSQL используют модели данных из-за пределов реляционного мира, которые не обязательно придерживаются транзакционные свойства атомарности, согласованности, изолированности и долговечности (ACID).
- **Базы данных NewSQL:** современная форма реляционных баз данных, предназначенная для обеспечения сравнимой масштабируемости с базами данных NoSQL при сохранении транзакционных гарантий, предоставляемых традиционными системами баз данных.
- **Платформы запросов к большим данным**: технологии, обеспечивающие фасады запросов к хранилищами больших данных, такими как распределенные файловые системы или базы данных NoSQL. Основной выигрыш - обеспечение высокоуровневого интерфейса, например через SQL подобный запрос и достижения низких задержек запросов.

Hadoop HDFS

Теперь Hadoop стал популярным решением для удовлетворения потребностей современного мира. При разработке Hadoop учитываются различные цели. Это отказоустойчивость, обработка больших наборов данных, локальность данных, переносимость между разнородными аппаратными и программными платформами и т. Д. В этом блоге мы подробно рассмотрим архитектуру Hadoop. Кроме того, мы увидим диаграмму архитектуры Hadoop, которая поможет вам лучше понять ее.



Что такое архитектура Hadoop?

Hadoop имеет топологию главный-подчиненный. В этой топологии у нас есть один главный узел и несколько подчиненных узлов. Функция главного узла состоит в том, чтобы назначать задачу различным подчиненным узлам и управлять ресурсами. Подчиненные узлы выполняют фактические вычисления. Ведомые узлы хранят реальные данные, тогда как на ведущем у нас есть метаданные. Это означает, что он хранит данные о данных. Что представляют собой метаданные, которые мы увидим через мгновение?

Hadoop – детальная архитектура

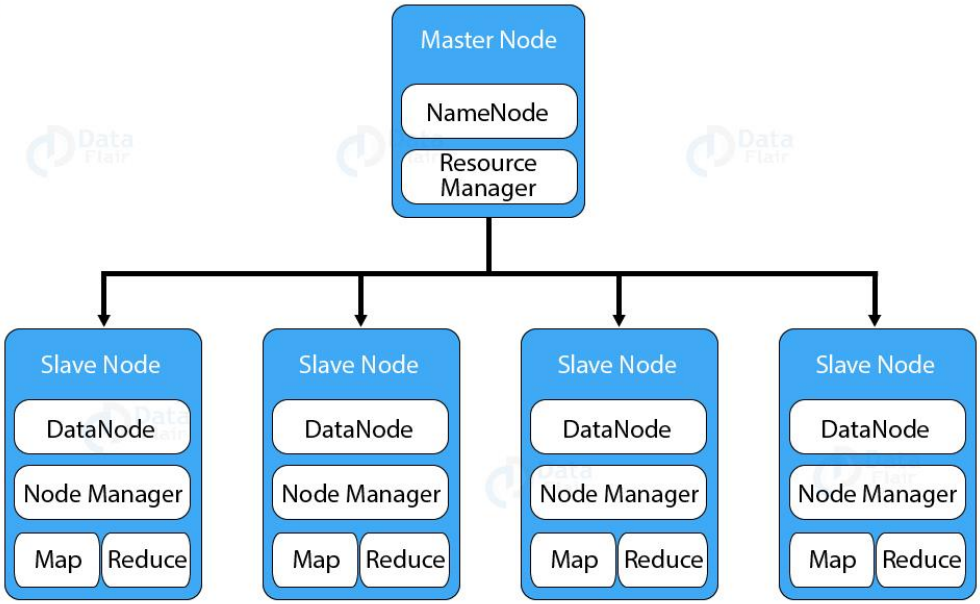
Архитектура Hadoop состоит из трех основных уровней. Это: - HDFS (Распределенная файловая система Hadoop)

1. HDFS

HDFS означает распределенную файловую систему Hadoop. Он обеспечивает хранение данных Hadoop. HDFS разбивает единицу данных на более мелкие единицы, называемые блоками, и сохраняет их распределенным образом. У него работают два демона. Один для главного узла - NameNode и другой для подчиненных узлов - DataNode.

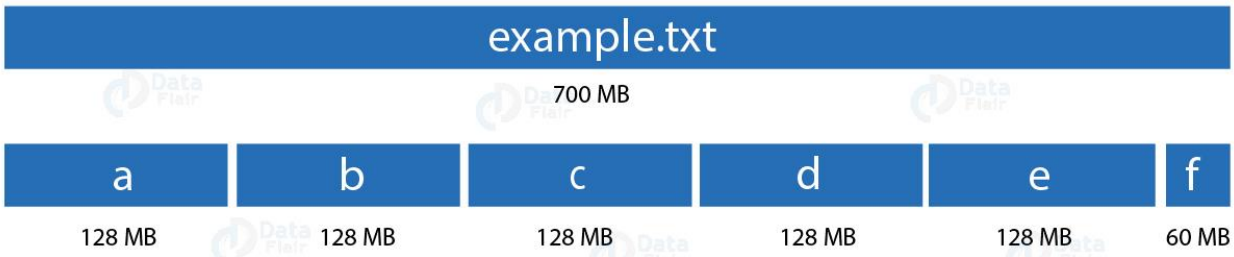
a. NameNode и DataNode

HDFS имеет архитектуру Master-Slave. Демон под названием NameNode работает на главном сервере. Он отвечает за управление пространством имен и регулирует доступ клиента к файлам. Демон DataNode работает на подчиненных узлах. Он отвечает за хранение фактических бизнес-данных. Внутри файл разбивается на несколько блоков данных и сохраняется на группе подчиненных машин. Namenode управляет модификациями пространства имен файловой системы. Это такие действия, как открытие, закрытие и переименование файлов или каталогов. NameNode также отслеживает отображение блоков на DataNodes. Этот DataNodes обслуживает запросы чтения / записи от клиента файловой системы. DataNode также создает, удаляет и реплицирует блоки по запросу из NameNode.



b. Блок в HDFS

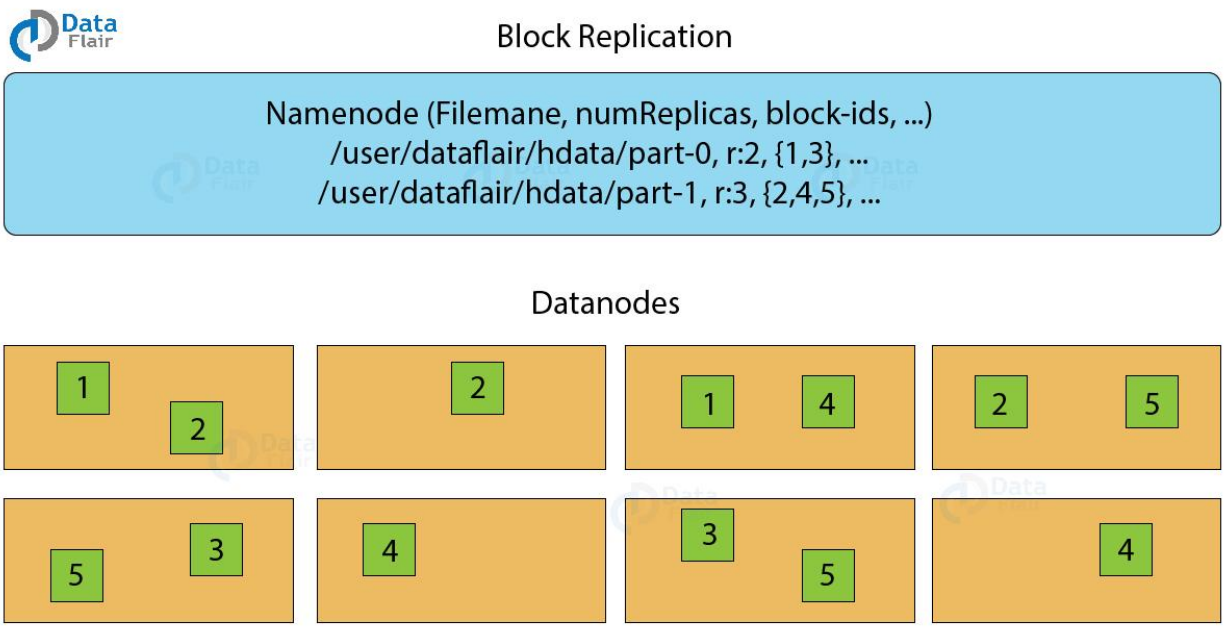
Блок - это не что иное, как наименьшая единица хранения в компьютерной системе. Это наименьшее непрерывное хранилище, выделенное для файла. В Hadoop размер блока по умолчанию составляет 128 или 256 МБ.



Подбирать размер блока нужно очень внимательно. Чтобы объяснить, почему это так, давайте возьмем пример файла размером 700 МБ. Если размер нашего блока составляет 128 МБ, то HDFS делит файл на 6 блоков. Пять блоков по 128 МБ и один блок по 60 МБ. Что будет, если размер блока будет 4 КБ? Но в HDFS у нас были бы файлы размером от терабайтов до петабайт. С размером блока 4 КБ у нас будет много блоков. Это, в свою очередь, создаст огромные метаданные, которые перегрузят NameNode. Следовательно, мы должны разумно выбирать размер блока HDFS.

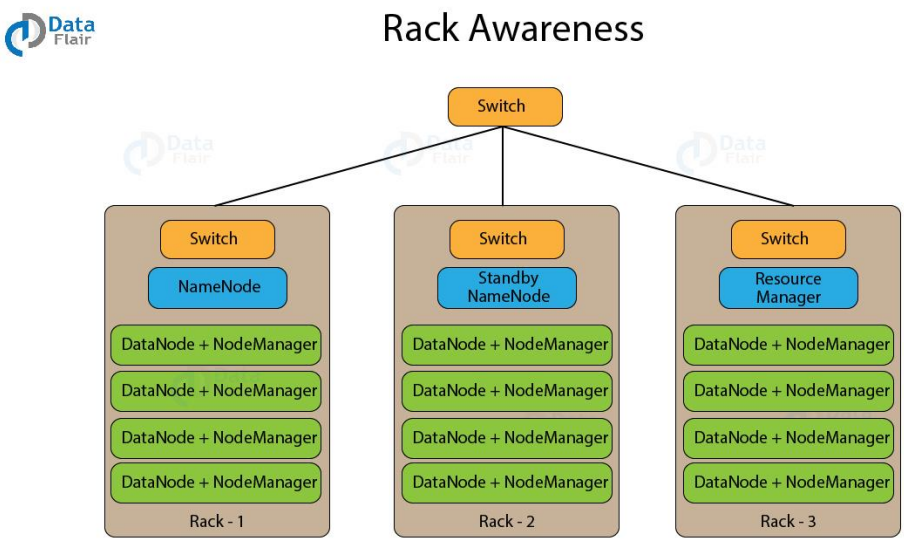
c. Управление репликацией

Для обеспечения отказоустойчивости HDFS использует метод репликации. При этом он делает копии блоков и сохраняет их на разных узлах данных. Коэффициент репликации определяет, сколько копий блоков будет сохранено. По умолчанию это 3, но мы можем настроить любое значение.



На рисунке выше показано, как работает метод репликации. Предположим, у нас есть файл размером 1 ГБ, тогда с коэффициентом репликации 3 для него потребуется 3 ГБ общего хранилища. Для поддержания фактора репликации NameNode собирает отчет о блоках со всех DataNode. Каждый раз, когда блок реплицируется недостаточно или избыточно, NameNode соответственно добавляет или удаляет реплики.

d. Что такое осведомленность рэка(стойки)?



Стойка содержит множество машин DataNode, и таких стоек в производстве несколько. HDFS следует алгоритму распознавания стойки для размещения реплик блоков распределенным образом. Этот алгоритм распознавания стойки обеспечивает низкую задержку и отказоустойчивость. Предположим, что настроен коэффициент репликации 3. Теперь алгоритм распознавания стойки поместит первый блок в локальную стойку. Остальные два блока останутся на другой стойке. По возможности в одной стойке не может храниться более двух блоков.

NoSQL базы данных

Базы данных NoSQL предназначены для обеспечения масштабируемости, часто за счет снижения согласованности. По сравнению с реляционными базами данных они часто используют нестандартные низкоуровневые запросы, интерфейсы, которые затрудняют их интеграцию в существующие приложения, ожидающие SQL интерфейс. Отсутствие стандартных интерфейсов затрудняет переключение между хранилищами. Базы данных NoSQL можно различить по используемым моделям данных:

Хранилища "ключ-значение"

Хранилища "ключ-значение" позволяют хранить данные без схемы. Объекты данных могут быть полностью неструктурированными или структурированными, и доступ к ним возможно получить по ключу. Поскольку схема не используется, даже не обязательно, чтобы объекты данных совместно использовали одинаковую структуру.

Одна из наиболее популярных БД такого типа – Redis.

Redis (от англ. remote dictionary server) — резидентная система управления базами данных класса NoSQL с открытым исходным кодом, работающая со структурами данных типа «ключ — значение». Используется как для баз данных, так и для реализации кэшей, брокеров сообщений. Ориентирована на достижение максимальной производительности на атомарных операциях (заявляется о приблизительно 100 тыс. SET- и GET-запросов в секунду на Linux-сервере начального уровня). Написана на Си, интерфейсы доступа созданы для большинства основных языков программирования. В период 2010—2013 годов разработка системы спонсировалась компанией VMware, с мая 2013 года, после реорганизаций в федерации EMC — VMware, проект передан в Pivotal. С июня 2015 года основной спонсор проекта — компания Redis Labs, специально основанная для коммерциализации Redis, в неё же перешёл основной разработчик продукта — Сальваторе Санфилиппо.

Концепция и архитектура

Кластер Redis Enterprise состоит из идентичных узлов, которые развернуты в центре обработки данных или распределены по локальным зонам доступности. Архитектура Redis Enterprise состоит из пути управления (показан в синем слое на Рисунке 1 ниже) и пути доступа к данным (показан в красном слое на Рисунке 1 ниже).

- Путь управления включает диспетчер кластера, прокси и безопасный REST API / UI для программного администрирования. Короче говоря, диспетчер кластера отвечает за организацию кластера, размещение сегментов базы данных, а также за обнаружение и устранение сбоев. Прокси-сервер помогает масштабировать управление подключениями.
- Путь доступа к данным состоит из главного и подчиненного сегментов Redis. Клиенты выполняют операции с данными на главном сегменте. Главные сегменты поддерживают подчиненные сегменты, используя репликацию в памяти для защиты от сбоев, которые могут сделать главный сегмент недоступным.

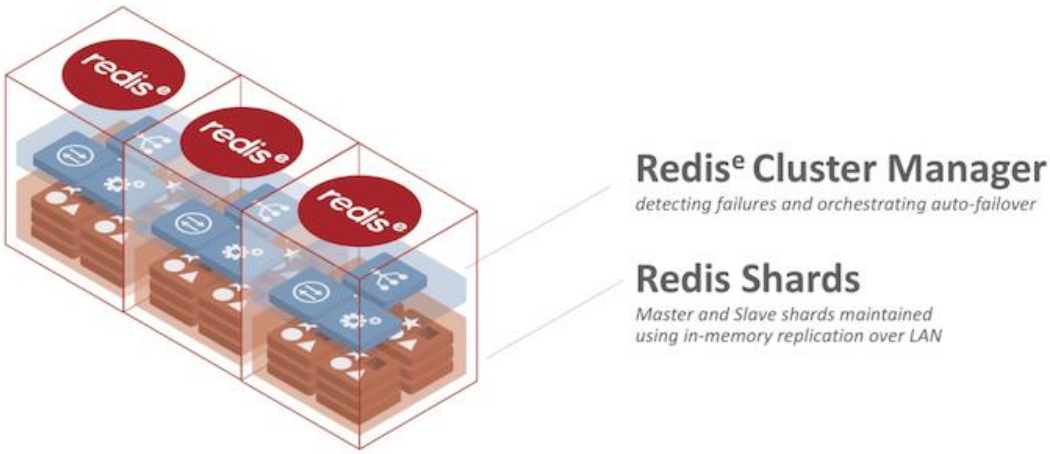


Рис. 1. Узлы Redis Enterprise Nodes с синим слоем, представляющим путь управления, и красными плитками, представляющими путь доступа к данным с Redis в качестве сегментов.

Высокая доступность с Redis Enterprise

Redis Enterprise использует репликацию в памяти для поддержки главной и подчиненной реплик. Redis Enterprise поставляется с различными сторожевыми таймерами, которые обнаруживают и защищают от многих типов сбоев. При сбоях, таких как сбой узла, сети или процесса, из-за которых главная реплика становится недоступной, Redis Enterprise автоматически превращает подчиненную реплику в главную реплику и прозрачно перенаправляет клиентское соединение на новую главную реплику. Помимо внутрикластерной репликации, Redis Enterprise также имеет встроенную репликацию на основе WAN для развертываний Redis в нескольких центрах обработки данных. Механизмы репликации на основе WAN в Redis Enterprise разработаны для защиты от полного сбоя центра обработки данных или более широких сетевых сбоев.

Масштабирование баз данных

Каждый кластер Redis Enterprise может содержать несколько баз данных. В Redis базы данных представляют данные, принадлежащие одному приложению, клиенту или микросервису. Redis Enterprise поддерживает масштабирование до 100 баз данных на кластер, что обеспечивает гибкие и эффективные модели с несколькими арендаторами.

Каждая база данных может содержать несколько или много шардов Redis. Шардинг прозрачен для приложений Redis. Главные сегменты в базе данных обрабатывают операции с данными для заданного подмножества ключей. Количество сегментов в базе данных настраивается и зависит от требований к пропускной способности приложений. Базы данных в Redis Enterprise можно преобразовать в большее количество сегментов Redis для масштабирования пропускной способности при сохранении задержек менее миллисекунды. Перешардинг осуществляется без простоев.

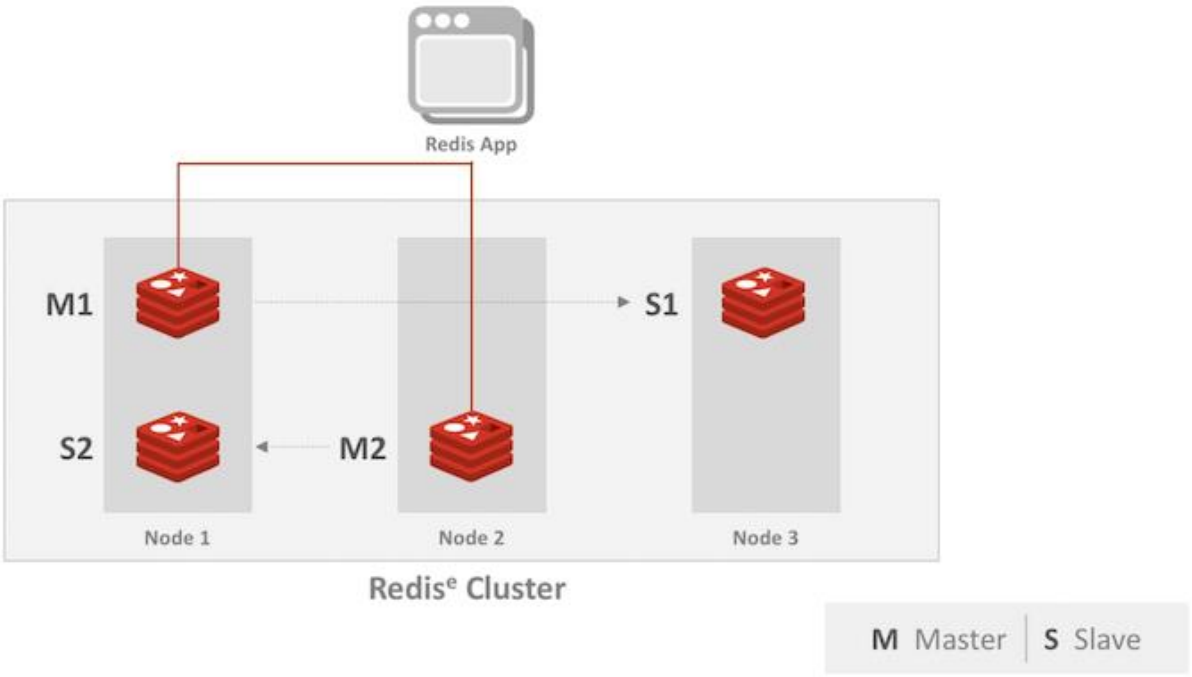


Рис. 2 Redis Enterprise размещает главную (M) и ведомую (S) реплики в отдельных узлах, стойках и зонах и использует репликацию в памяти для защиты данных от сбоев.

В Redis Enterprise у каждой базы данных есть квота ОЗУ. Квота не может превышать пределы ОЗУ, доступной на узле. Однако с Redis Enterprise Flash оперативная память расширяется до локального флэш-накопителя (SATA, твердотельные накопители NVMe и т. Д.). Общая квота базы данных может использовать как оперативную память, так и флэш-накопитель. Администратор может выбрать соотношение RAM и Flash и настроить его в любое время в течение срока службы базы данных без простоев.

В Redis on Flash вместо того, чтобы хранить все ключи и данные для данного шарда в ОЗУ, менее часто используемые значения помещаются во флэш-память. Если приложениям требуется доступ к значению во флэш-памяти, Redis Enterprise автоматически переносит это значение в ОЗУ. В зависимости от используемого флеш-оборудования, приложения испытывают немного большую задержку при возврате значений в оперативную память из флеш-памяти. Однако последующие обращения к одному и тому же значению происходят быстро, если значение находится в ОЗУ.

Надежность данных с Redis Enterprise

Redis Enterprise имеет два варианта надежности:

Долговечность на основе диска: Redis Enterprise по-прежнему поддерживает надежную копию на диске. Как и в случае с дисковыми системами, этот путь ввода-вывода размещается на более медленном и надежном сетевом запоминающем устройстве. Базы данных Redis предоставляют настраиваемые параметры для поддержки этой надежной копии и поддержания ее в актуальном состоянии с частыми периодическими операциями записи вплоть до каждой операции записи.

Надежность на основе репликации: Redis Enterprise также поддерживает реплику, подчиненный сегмент, для обеспечения надежности. Эта реплицированная надежность защищает от отказов узла, стойки или зоны. Replicated-durability обеспечивает лучшую производительность записи по сравнению с записью в сетевое хранилище. Это означает, что при незапланированном прерывании более вероятно, что ваша реплика будет более актуальной по сравнению с вашей

устойчивой копией на диске. Чтобы в полной мере воспользоваться реплицированной надежностью, Redis предоставляет команду WAIT. WAIT гарантирует, что запись может ждать подтверждения, пока несколько реплик не подтвердят эту запись. Это гарантирует, что запись, подтвержденная с помощью WAIT на репликах, будет надежной, даже если узел загорится и никогда не вернется в кластер.

Хранилища столбцов

«СУБД, ориентированная на столбцы, - это система управления базами данных (СУБД), которая хранит таблицы данных как столбцы данных, а не строки данных, как в большинстве реляционных СУБД». Такие базы данных обычно разреженные, распределенные и представляются в виде многомерных отсортированных карт данных, в которых данные индексируются триплетом ключ строки, ключ столбца и метка времени. Значение представлено в виде непрерывного строкового типа данных. Доступ к данным осуществляется по семействам столбцов, то есть по набору связанных ключей столбцов, которые эффективно сжимают разреженные данные в столбцах. Семейства столбцов создаются до того, как данные могут быть сохранены, и ожидается, что их количество будет небольшим. Количество столбцов наоборот не ограничено. В принципе, хранилища столбцов менее подходит, когда необходимо получить доступ ко всем столбцам. Однако на практике это бывает редко, что приводит к превосходной производительность хранилищ столбцов.

В качестве примеров рассмотрим гибридную СУБД Кассандра, совмещающую подходы хранения столбцов и key-value.

Apache Cassandra – это нереляционная отказоустойчивая распределенная СУБД, рассчитанная на создание высокомасштабируемых и надёжных хранилищ огромных массивов данных, представленных в виде хэша. Проект был разработан на языке Java в корпорации Facebook в 2008 году, и передан фонду Apache Software Foundation в 2009. Эта СУБД относится к гибридным NoSQL-решениям, поскольку она сочетает модель хранения данных на базе семейства столбцов (ColumnFamily) с концепцией key-value (ключ-значение).

МОДЕЛЬ ДАННЫХ APACHE CASSANDRA

Модель данных Cassandra состоит из следующих элементов:

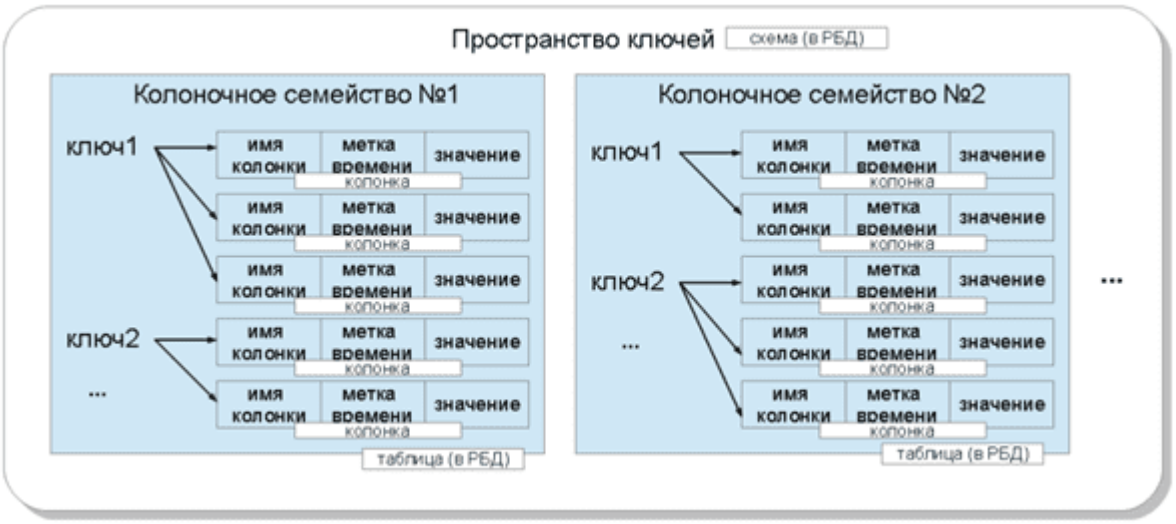
- столбец или колонка (column) – ячейка с данными, включающая 3 части – имя (column name) в виде массива байтов, метку времени (timestamp) и само значение (value) также в виде байтового массива. С каждым значением связана метка времени — задаваемое пользователем 64-битное число, которое используется для разрешения конфликтов во время записи: чем оно больше, тем новее считается столбец. Это учитывается при удалении старых колонок.
- строка или запись (row) – именованная коллекция столбцов;
- семейство столбцов (column family) – именованная коллекция строк;
- пространство ключей (keyspace) – группа из нескольких семейств столбцов, собранных вместе. Оно логически группирует семейства столбцов и обеспечивает изолированные области имен.

Также стоит отметить понятия сравнителя (comparator), задаваемого для имени столбца и валидатора (validator) для значений и ключей. Comparator определяет, какие байтовые значения допустимы для имён колонок и как их упорядочить, а validator – для значений колонок и ключей. Если они не заданы, то Кассандра хранит значения и сравнивает их как байтовые строки

(BytesType) так как, по сути, они сохраняются внутри. Вообще, в рассматриваемой СУБД доступны следующие типы данных :

- BytesType: любые байтовые строки (без валидации);
- AsciiType: ASCII строка;
- UTF8Type: UTF-8 строка;
- IntegerType: число с произвольным размером;
- Int32Type: 4-байтовое число;
- LongType: 8-байтовое число;
- UUIDType: UUID 1-ого или 4-ого типа;
- TimeUUIDType: UUID 1-ого типа;
- DateType: 8-байтовое значение метки времени;
- BooleanType: два значения: true = 1 или false = 0;
- FloatType: 4-байтовое число с плавающей запятой;
- DoubleType: 8-байтовое число с плавающей запятой;
- DecimalType: число с произвольным размером и плавающей запятой;
- CounterColumnType: 8-байтовый счётчик.

Пространство ключей соответствует понятию схемы базы данных (database schema) в реляционной модели, а находящиеся в нем семейства столбцов – реляционной таблице. Столбцы объединяются в записи при помощи ключа (row key) в виде массива байтов, по значению которого столбцы упорядочены в пределах одной записи. В отличие от реляционных СУБД, в NoSQL модели возможна ситуация, когда разные строки содержат разное число колонок или столбцы с такими же именами, как и в других записях.



Можно сказать, конкретное значение, хранимое в Apache Cassandra, идентифицируется следующими привязками :

- к приложению или предметной области в пространстве ключей, что позволяет на одном кластере размещать данные разных приложений;
- к запросу в рамках семейства столбцов;
- к узлу кластера с помощью ключа, который определяет, на какие узлы попадут сохранённые колонки;
- к атрибуту в записи с помощью имени колонки, что позволяет в одной записи хранить несколько значений.

Структура данных при этом выглядит так:

Keyspace

ColumnFamily

Row

Key

Column

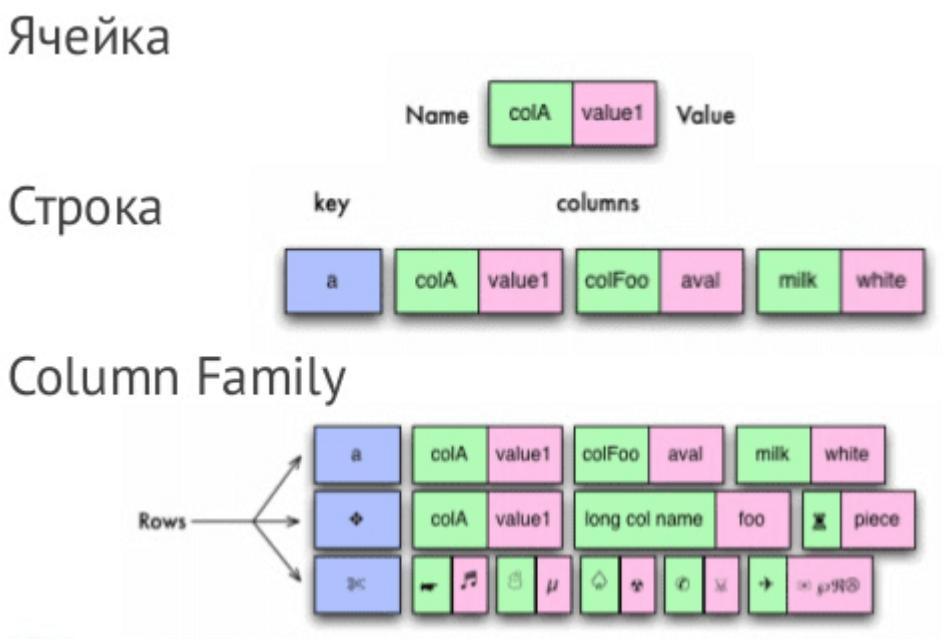
Name

Value

Column

…

…



АРХИТЕКТУРА

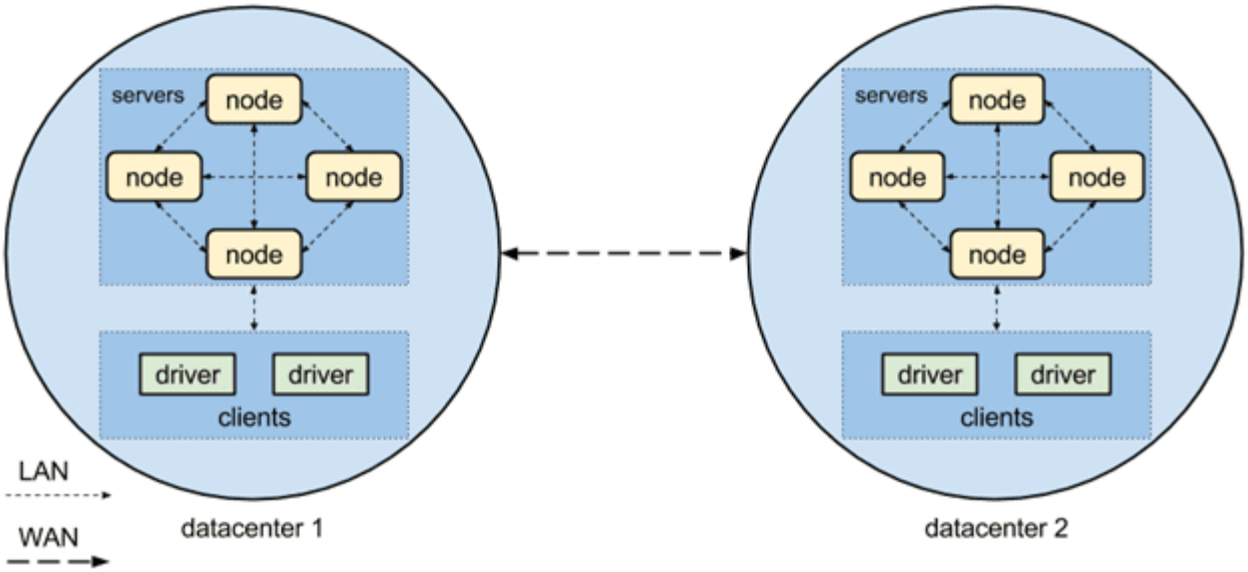
Apache Cassandra – это децентрализованная распределенная система, состоящая из нескольких узлов, по которым она распределяет данные. В отличие от многих других Big Data решений экосистемы Apache Hadoop (HBase, HDFS), эта СУБД не поддерживает концепцию master/slave (ведущий/ведомый), когда один из серверов является управляющим для других компонентов кластера.

Для распределения элементов данных по узлам Кассандра использует последовательное хэширование, применяя хэш-алгоритм для вычисления хэш-значений ключей каждого элемента данных (имя столбца, ID строки и пр.). Диапазон всех возможных хэш-значений, т.е. пространство ключей, распределяется между узлами кластера так, что каждому элементу данных назначен свой узел, который отвечает за хранение и управление этим элементом данных.

Благодаря такой распределенной архитектуре, Кассандра предоставляет следующие возможности:

- распределение данных между узлами кластера прозрачно для пользователей – каждый сервер может принимать любой запрос (на чтение, запись или удаление данных), пересылая его на другой узел, если запрашиваемая информацию хранится не здесь;
- пользователи могут сами определить необходимое количество реплик, создание и управление которыми обеспечит Cassandra;
- настраиваемый пользователями уровень согласованности данных по каждой операции хранения и считывания;
- высокая скорость записи (около 80-360 МБ/с на узел) – данные записываются быстрее, чем считываются за счет того, что их большая часть хранится в оперативной памяти ответственного узла, и любые обновления сперва выполняются в памяти, а только потом – в файловой системе. Чтобы избежать потери информации, все транзакции фиксируются в специальном журнале на диске. При этом, в отличие от обновления данных, записи в журналы фиксации только добавляются, что исключает задержку при вращении диска. Кроме того, если не требуется полная согласованность записей, Cassandra записывает данные в достаточное число узлов без разрешения конфликтов несоответствия, которые разрешаются только при первом считывании.
- гибкая масштабируемость – можно построить кластер даже на сотню узлов, способный обрабатывать петабайты данных.

Таким образом, отсутствие центрального узла лишает Кассандру главного недостатка, свойственного системам master/slave, в которых отказывает весь кластер при сбое главного сервера (Master Node). В кластере Cassandra все узлы равноценны между собой и, если один из них отказал, его функции возьмет на себя какой-то из оставшихся. Благодаря такой децентрализации Apache Cassandra отлично подходит для географически распределенных систем с высокой доступностью, расположенных в разных датацентрах. Однако, при всех преимуществах такой гибко масштабируемой архитектуры, она обусловливает особенности операций чтения и записи, а также накладывает ряд существенных ограничений на использование этой СУБД в реальных Big Data проектах.



Базы данных документов

В отличие от значений в хранилище ключей и значений, документы структурированы. Однако, нет необходимости в общей схеме, в которой все документы должны соответствовать одной

структуре, как и в случае с записями в реляционных базах данных. Таким образом, базы данных документов называются хранилищами полуструктурированных данных. Аналогично хранищам ключ-значение, документы можно запрашивать с помощью уникального ключа. Однако возможно получить доступ к документам, запросив их внутреннюю структуру, например запрос всех документов, содержащих поле с указанным значением. Возможности интерфейса запросов обычно зависит от используемого формата кодирования данных. Общие кодировки включают XML или JSON.

В качестве примера будет рассмотрена СУБД MongoDB

Архитектура Nexus

Философия проектирования MongoDB сосредоточена на сочетании критических возможностей реляционных баз данных с инновациями технологий NoSQL. Наше видение состоит в том, чтобы максимально использовать работу, проделанную Oracle и другими за последние 40 лет, чтобы сделать реляционные базы данных такими, какими они являются сегодня. Вместо того, чтобы отказываться от десятилетий доказанной зрелости баз данных, MongoDB продолжает с того места, где они остановились, сочетая ключевые возможности реляционных баз данных с работой, проделанной первопроходцами Интернета для удовлетворения требований современных приложений.

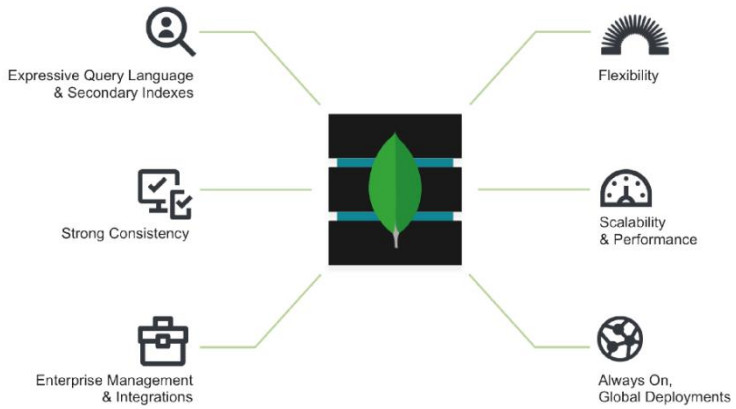


Рисунок 1: Архитектура MongoDB Nexus, сочетающая в себе лучшие реляционные технологии и технологии NoSQL

Реляционные базы данных надежно обслуживают приложения в течение многих лет и предлагают функции, которые остаются критически важными сегодня, когда разработчики создают приложения следующего поколения:

• Выразительный язык запросов и вторичные индексы. Пользователи должны иметь возможность получать доступ и управлять своими данные сложными способами для поддержки как операционных, так и аналитических приложений. Индексы играют критически важную роль в обеспечении эффективного доступа к данным, которые изначально поддерживаются базой данных, а не поддерживаются в коде приложения.

• Сильная консистенция. Приложения должны иметь возможность немедленно читать то, что было записано в базу данных. Гораздо сложнее создавать приложения на основе согласованной в конечном итоге модели, требующей от разработчика значительных усилий даже для самых опытных инженерных групп.

• Управление предприятием и интеграции. Базы данных - это всего лишь одна часть инфраструктуры приложений, и они должны легко вписываться в корпоративный ИТ-стек. Организациям нужна база данных, которую можно защищать, отслеживать, автоматизировать и интегрировать с существующей технологической инфраструктурой, процессами и персоналом, включая операционные группы, администраторов баз данных и аналитиков данных. Однако современные приложения предъявляют требования, не удовлетворяемые реляционными базами данных, и это привело к развитию баз данных NoSQL, которые предлагают:

• Гибкая модель данных. Базы данных NoSQL возникли для удовлетворения требований к данным, которые, как мы видим, доминируют в современных приложениях. Будь то документ, график, ключ-значение или широкий столбец, все они предлагают гибкую модель данных, что упрощает хранение и комбинирование данных любой структуры и позволяет динамически изменять схему без простоев или снижения производительности.

• Масштабируемость и производительность. Все базы данных NoSQL были созданы с упором на масштабируемость, поэтому все они включают в себя некоторую форму сегментирования или разделения. Это позволяет масштабировать базу данных на стандартном оборудовании, развернутом локально или в облаке, обеспечивая практически неограниченный рост с более высокой пропускной способностью и меньшей задержкой, чем у реляционных баз данных.

• Постоянное глобальное развертывание. Базы данных NoSQL предназначены для систем с высокой доступностью, которые обеспечивают единообразие и высокое качество работы пользователей во всем мире. Они предназначены для работы на многих узлах, включая репликацию для автоматической синхронизации данных между серверами, стойками и центрами обработки данных.

Предлагая эти инновации, системы NoSQL пожертвовали критически важными возможностями, которые люди привыкли ожидать от реляционных баз данных и полагаться на них. MongoDB предлагает другой подход. Благодаря архитектуре Nexus, MongoDB является единственной базой данных, которая использует инновации NoSQL, сохраняя при этом основу реляционных баз данных.

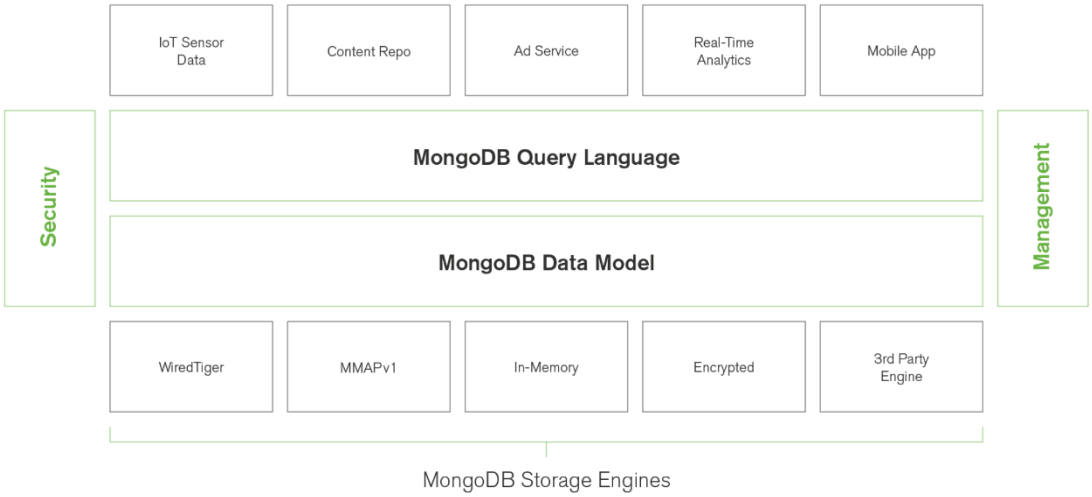


Рисунок 2: Гибкая архитектура хранилища, оптимизирующая MongoDB для уникальных требований приложений

Гибкая архитектура хранения MongoDB

MongoDB охватывает две ключевые тенденции в современных ИТ:

• Организации расширяют спектр приложений, которые они предоставляют для поддержки бизнеса.

• ИТ-директора рационализируют свои технологические портфели в соответствии со стратегическим набором поставщиков, которых они могут использовать для более эффективной поддержки своего бизнеса.

С помощью MongoDB организации могут удовлетворить различные потребности приложений, аппаратные ресурсы и схемы развертывания с помощью единой технологии баз данных. За счет использования гибкой архитектуры хранения MongoDB может быть расширен за счет новых возможностей и настроен для оптимального использования конкретных аппаратных архитектур. MongoDB уникальным образом позволяет пользователям комбинировать и сопоставлять несколько механизмов хранения в рамках одного развертывания. Эта гибкость обеспечивает более простой и надежный подход к удовлетворению различных потребностей приложений в данных. Традиционно для удовлетворения этих потребностей необходимо было управлять несколькими технологиями баз данных, используя сложный настраиваемый код интеграции для перемещения данных между технологиями и обеспечения согласованного и безопасного доступа. Благодаря гибкой архитектуре хранения MongoDB база данных автоматически управляет перемещением данных между технологиями подсистемы хранения с использованием собственной репликации. Такой подход значительно снижает сложность разработки и эксплуатации по сравнению с использованием нескольких различных технологий баз данных. Пользователи могут использовать один и тот же язык запросов MongoDB, модель данных, масштабирование, безопасность и операционные инструменты в разных частях своего приложения, каждая из которых работает на оптимальном механизме хранения.

MongoDB 3.2 поставляется с четырьмя поддерживаемыми механизмами хранения, каждый из которых может сосуществовать в одном наборе реплик MongoDB. Это упрощает оценку и переход между ними, а также оптимизацию в соответствии с требованиями конкретных приложений - например, объединение механизма в памяти для операций со сверхмалой задержкой и механизма на основе диска для обеспечения устойчивости. Поддерживаемые механизмы хранения включают:

• Механизм хранения WiredTiger по умолчанию. Для многих приложений детализированный контроль параллелизма и собственное сжатие WiredTiger обеспечат наилучшую всестороннюю производительность и эффективность хранения для самого широкого спектра приложений.

• Механизм зашифрованного хранилища, защищающий очень конфиденциальные данные, без дополнительных затрат на производительность или управление, связанных с шифрованием отдельной файловой системы. (Требуется MongoDB Enterprise Advanced).

• Механизм хранения In-Memory обеспечивает высочайшую производительность в сочетании с аналитикой в реальном времени для самых требовательных и чувствительных к задержкам приложений. (Требуется MongoDB Enterprise Advanced).

• Механизм MMAPv1, улучшенная версия механизма хранения, используемого в выпусках MongoDB до 3.x.

Модель данных MongoDB

Данные как документы MongoDB хранит данные как документы в двоичном представлении, называемом BSON (двоичный JSON). Кодировка BSON расширяет популярное представление JSON (JavaScript Object Notation) за счет включения дополнительных типов, таких как int, long, date и float. Документы BSON содержат одно или несколько полей, и каждое поле содержит значение определенного типа данных, включая массивы, двоичные данные и вложенные документы.

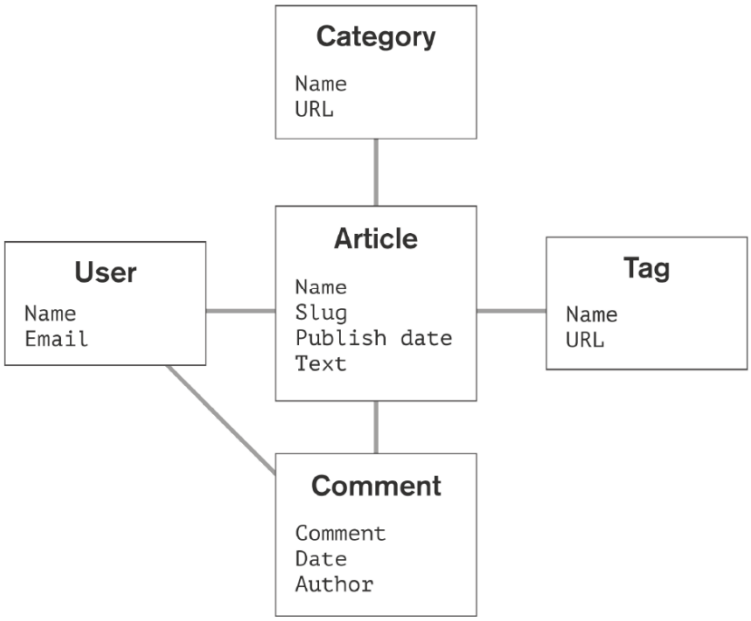


Рисунок 3: Пример реляционной модели данных для приложения для ведения блога

Документы, которые имеют схожую структуру, организованы в виде коллекций. Может быть полезно думать о коллекциях как о аналоге таблицы в реляционной базе данных: документы подобны строкам, а поля подобны столбцам. Например, рассмотрим модель данных для приложения для ведения блога. В реляционной базе данных модель данных будет состоять из нескольких таблиц. Чтобы упростить пример, предположим, что есть таблицы для категорий, тегов, пользователей, комментариев и статей.

В MongoDB данные могут быть смоделированы как две коллекции, одна для пользователей, а другая для статей. В каждом документе блога может быть несколько комментариев, несколько тегов и несколько категорий, каждая из которых выражена в виде встроенного массива.

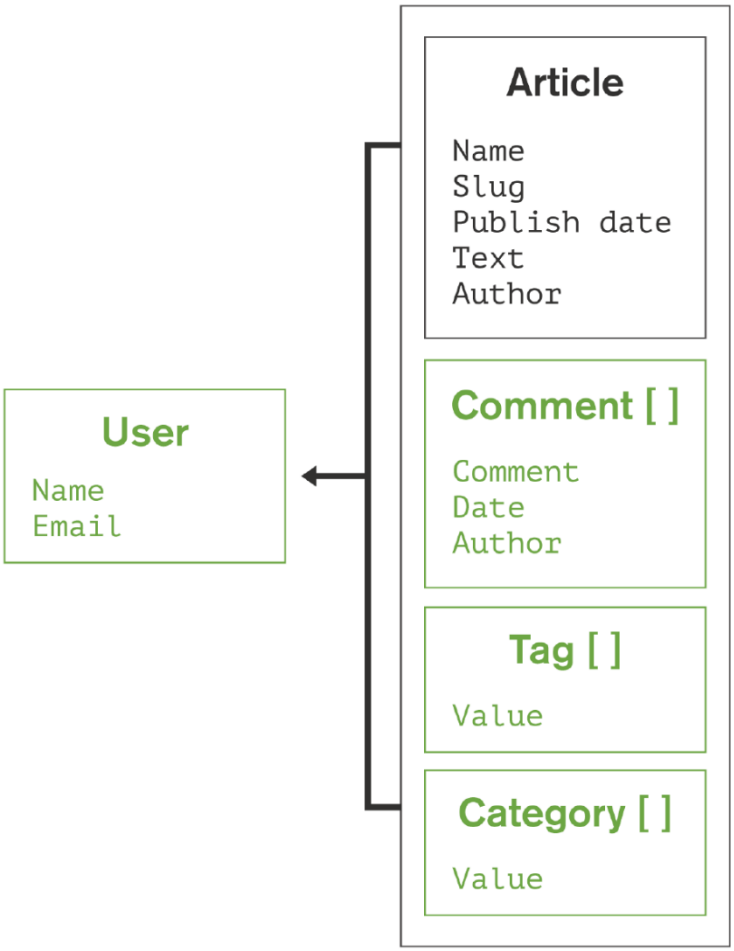


Рисунок 4: Данные как документы: проще для разработчиков, быстрее для пользователей.

Как показывает этот пример, документы MongoDB, как правило, содержат все данные для данной записи в одном документе, тогда как в реляционной базе данных информация для данной записи обычно распределена по множеству таблиц. В модели документа MongoDB данные более локализованы, что значительно снижает необходимость в отдельных таблицах. Результат - значительно более высокая производительность и масштабируемость на стандартном оборудовании, поскольку одно чтение из базы данных может получить весь документ, содержащий все связанные данные. В отличие от многих баз данных NoSQL, пользователям не нужно полностью отказываться от JOIN. Для дополнительной гибкости аналитики MongoDB сохраняет LEFT-OUTTER семантику JOIN с помощью оператора $ lookup, позволяя пользователям максимально эффективно использовать как реляционное, так и нереляционное моделирование данных.

Документы MongoDB BSON тесно связаны со структурой объектов на языке программирования. Это упрощает и ускоряет для разработчиков моделирование того, как данные в приложении будут сопоставляться с данными, хранящимися в базе данных.

Динамическая схема без компромиссов

Управление данными

Документы MongoDB могут различаться по структуре. Например, все документы, описывающие пользователей, могут содержать идентификатор пользователя и дату последнего входа в систему, но только некоторые из этих документов могут содержать идентификационные данные пользователя для одного или нескольких сторонних приложений. Поля могут отличаться от

документа к документу; нет необходимости декларировать структуру документов в системе - документы самоописательны. Если в документ необходимо добавить новое поле, это поле можно создать, не затрагивая все другие документы в системе, без обновления центрального системного каталога и без отключения системы.

Разработчики могут начать писать код и сохранять объекты по мере их создания. А когда разработчики добавляют дополнительные функции, MongoDB продолжает хранить обновленные объекты без необходимости выполнять дорогостоящие операции ALTER TABLE или, что еще хуже, переделывать схему с нуля.

Проверка документов

Динамические схемы обеспечивают большую гибкость, но также важно, чтобы элементы управления могли быть реализованы для поддержания качества данных, особенно если база данных поддерживает несколько приложений или интегрирована в более крупную платформу управления данными с потоками в вышестоящие и нижележащие системы. В отличие от баз данных NoSQL, которые возвращают применение этих элементов управления в код приложения, MongoDB обеспечивает проверку документов в базе данных. Пользователи могут принудительно проверять структуру документа, типы данных, диапазоны данных и наличие обязательных полей. В результате администраторы баз данных могут применять стандарты управления данными, в то время как разработчики сохраняют преимущества гибкой модели документов.

Дизайн - схема

Хотя MongoDB обеспечивает гибкость схемы, дизайн схемы по-прежнему важен. Разработчики и администраторы баз данных должны рассмотреть ряд тем, включая типы запросов, которые приложение должно будет выполнять, отношения между данными, способы управления объектами в коде приложения и то, как документы будут меняться с течением времени. Разработка схемы - обширная тема, выходящая за рамки этого документа. Дополнительные сведения см. В разделе «Рекомендации по моделированию данных».

Модель запроса MongoDB

Идиоматические драйверы

MongoDB предоставляет собственные драйверы для всех популярных языков программирования и фреймворков, чтобы сделать разработку естественной. Поддерживаемые драйверы включают Java, .NET, Ruby, PHP, JavaScript, node.js, Python, Perl, PHP, Scala и другие, а также более 30 драйверов, разработанных сообществом. Драйверы MongoDB разработаны так, чтобы быть идиоматическими для данного языка.

Одним из фундаментальных отличий от реляционных баз данных является то, что модель запросов MongoDB реализована как методы или функции в API определенного языка программирования, в отличие от полностью отдельного языка, такого как SQL. Это, в сочетании с близостью между моделью документа JSON MongoDB и структурами данных, используемыми в объектно-ориентированном программировании, упрощает интеграцию с приложениями. Полный список драйверов см. На странице драйверов MongoDB.

Взаимодействие с базой данных

MongoDB предлагает разработчикам и администраторам ряд инструментов для взаимодействия с базой данных, независимо от драйверов. Оболочка mongo - это многофункциональная интерактивная оболочка JavaScript, которая включена во все дистрибутивы MongoDB. Оболочка обеспечивает удобный способ запроса и обновления данных, а также выполнения административных операций. MongoDB Compass предоставляет графический пользовательский

интерфейс, который позволяет пользователям визуализировать свою схему и выполнять специальные запросы к базе данных - и все это без знания языка запросов MongoDB. Администраторы баз данных могут исследовать структуру документа, включая поля, значения и типы данных; определить, какие индексы могут быть подходящими, и определить, следует ли добавлять правила проверки документов для обеспечения согласованной структуры документа. Сложные запросы можно создавать и выполнять, просто выбирая элементы документа в пользовательском интерфейсе, а результаты просматриваются как графическими, так и графическими.

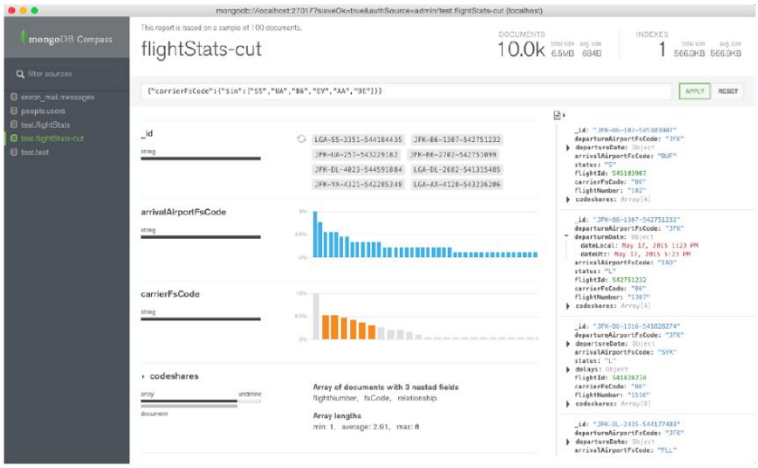


Рисунок 5: Интерактивное построение и выполнение запросов к базе данных

с помощью MongoDB Compass Запрос и визуализация данных В отличие от баз данных NoSQL, MongoDB не ограничивается простыми операциями «ключ-значение». Разработчики могут создавать многофункциональные приложения, используя сложные запросы, агрегаты и вторичные индексы, которые раскрывают ценность структурированных, полуструктурированных и неструктурированных данных. Ключевым элементом этой гибкости является поддержка MongoDB многих типов запросов. Запрос может возвращать документ, подмножество определенных полей в документе или сложные агрегаты для многих документов:

- Запросы «ключ-значение» возвращают результаты на основе любого поля в документе, часто первичного ключа.
- Запросы диапазона возвращают результаты, основанные на значениях, определенных как неравенства (например, больше, меньше или равно, между).
- Геопространственные запросы возвращают результаты на основе критериев близости, пересечения и включения, заданных точкой, линией, кругом или многоугольником.
- Запросы текстового поиска возвращают результаты в порядке релевантности на основе текстовых аргументов с использованием логических операторов (например, И, ИЛИ, НЕ).
- Запросы Aggregation Framework возвращают агрегаты значений, возвращаемых запросом (например, count, min, max, average, аналогично оператору SQL GROUP BY). На этапе поиска $ документы из отдельных коллекций могут быть объединены с помощью левой внешней операции JOIN.
- Запросы MapReduce выполняют сложную обработку данных, которая выражается в JavaScript и выполняется для данных в базе данных.

Визуализация данных с помощью инструментов бизнес-аналитики

Благодаря растущим требованиям пользователей к самообслуживанию аналитики, более быстрому обнаружению на основе оперативных данных в реальном времени и необходимости интеграции мультиструктурированных и потоковых наборов данных платформы бизнес-аналитики

и аналитики являются одними из самых быстрорастущих рынков программного обеспечения. С помощью MongoDB Connector for BI, включенного в MongoDB Enterprise Advanced, данные современных приложений можно легко анализировать с помощью стандартных отраслевых платформ бизнес-аналитики и аналитики на основе SQL. Бизнес-аналитики и специалисты по данным могут легко анализировать полу- и неструктурированные данные, управляемые в MongoDB, наряду с традиционными данными в своих базах данных SQL, используя те же инструменты бизнес-аналитики, которые используются на миллионах предприятий.

Индексирование

Индексы являются важным механизмом для оптимизации производительности и масштабируемости системы, обеспечивая при этом гибкий доступ к данным. Как и большинство систем управления базами данных, хотя индексы улучшают производительность некоторых операций на порядки, они несут связанные с этим накладные расходы на операции записи, использование диска и потребление памяти. По умолчанию механизм хранения WiredTiger сжимает индексы в ОЗУ, освобождая больше рабочего набора для документов.

MongoDB включает поддержку многих типов вторичных индексов, которые могут быть объявлены в любом поле документа, включая поля в массивах:

- Уникальные индексы. Указав индекс как уникальный, MongoDB отклонит вставку новых документов или обновление документа с существующим значением для поля, для которого был создан уникальный индекс. По умолчанию все индексы не уникальны. Если составной индекс указан как уникальный, комбинация значений должна быть уникальной.
- Составные индексы. Может быть полезно создавать составные индексы для запросов, которые задают несколько предикатов. Например, рассмотрим приложение, которое хранит данные о клиентах. Приложению может потребоваться поиск клиентов по фамилии, имени и городу проживания. С помощью составного индекса по фамилии, имени и городу проживания запросы могут эффективно находить людей со всеми тремя указанными значениями. Дополнительным преимуществом составного индекса является то, что можно использовать любое начальное поле в индексе, поэтому может потребоваться меньшее количество индексов по отдельным полям: этот составной индекс также оптимизирует запросы, ищущие клиентов по фамилии.
- Индексы массивов. Для полей, содержащих массив, каждое значение массива сохраняется как отдельная запись индекса. Например, документы, описывающие продукты, могут включать поле для компонентов. Если есть индекс в поле компонента, каждый компонент индексируется, и запросы в поле компонента могут быть оптимизированы с помощью этого индекса. Для создания индексов массива не требуется специального синтаксиса - если поле содержит массив, он будет проиндексирован как индекс массива.
- Индексы TTL. В некоторых случаях данные должны автоматически удаляться из системы. Индексы Time to Live (TTL) позволяют пользователю указать период времени, по истечении которого данные будут автоматически удаляться из базы данных. Обычно индексы TTL используются в приложениях, которые поддерживают скользящее окно истории (например, за последние 100 дней) для действий пользователя, таких как потоки кликов.
- Геопространственные индексы. MongoDB предоставляет геопространственные индексы для оптимизации запросов, связанных с местоположением в двухмерном пространстве, таких как проекционные системы для Земли. Эти индексы позволяют MongoDB оптимизировать запросы для документов, содержащих точки или многоугольник, наиболее близкие к данной точке или линии; которые находятся внутри круга, прямоугольника или многоугольника; или которые пересекаются с кругом, прямоугольник или многоугольник.

- Частичные индексы. Указав выражение фильтрации во время создания индекса, пользователь может указать MongoDB включать только те документы, которые соответствуют желаемым условиям, например, путем индексации только активных клиентов. Частичный баланс индексов обеспечивает выполнение запросов с низкой задержкой и снижает нагрузку на систему.
- Разреженные индексы. Разреженные индексы содержат записи только для документов, содержащих указанное поле. Поскольку модель данных документа MongoDB обеспечивает гибкость модели данных от документа к документу, некоторые поля обычно присутствуют только в подмножестве всех документов. Разреженные индексы позволяют использовать более мелкие и эффективные индексы, когда поля присутствуют не во всех документах.
- Индексы текстового поиска. MongoDB предоставляет специализированный индекс для текстового поиска, который использует расширенные языковые лингвистические правила для выделения корней, токенизации, чувствительности к регистру и стоп-слов. Запросы, использующие индекс текстового поиска, будут возвращать документы в порядке релевантности. В текстовый индекс можно включить одно или несколько полей.

Оптимизация запросов

MongoDB автоматически оптимизирует запросы, чтобы сделать оценку максимально эффективной. Оценка обычно включает выбор данных на основе предикатов и сортировку данных на основе предоставленных критериев сортировки. Оптимизатор запросов выбирает лучший индекс для использования, периодически выполняя альтернативные планы запросов и выбирая индекс с наилучшим временем ответа для каждого типа запроса. Результаты этого эмпирического теста хранятся в виде кэшированного плана запроса и периодически обновляются. Разработчики могут просматривать и оптимизировать планы, используя мощный метод объяснения и фильтры индекса. Пересечение индексов обеспечивает дополнительную гибкость, позволяя MongoDB использовать более одного индекса для оптимизации специального запроса во время выполнения.

Покрытые запросы

Запросы, которые возвращают результаты, содержащие только проиндексированные поля, называются покрытыми запросами. Эти результаты могут быть возвращены без чтения из исходных документов. С помощью соответствующих индексов можно оптимизировать рабочие нагрузки для использования преимущественно покрытых запросов.

Графовые базы данных

Графовые базы данных, такие как Neo4J (2015), хранят данные в формате графа. Такие хранилища хорошо подходят для хранения высокоассоциативных данных, таких как социальные сетевые графы. Особой разновидностью графовых баз данных являются хранилища триплетов (RDF), такие как AllegroGraph, Virtuoso, BlazeGraph которые специально предназначен для хранения триплетов RDF. Однако существующие технологии хранилищ триплетов не очень подходит для эффективного хранения действительно больших наборов данных.

Рассмотрим на примере RDF СУБД Blazegraph.

Архитектура базы данных RDF

В этом разделе мы определяем структуру описания ресурсов (RDF) и показываем, как база данных RDF реализована с использованием архитектуры больших данных. Bigdata реализует API уровня хранения и вывода (SAIL), который предоставляет подключаемый бэкэнд для платформы Sesame.

Однако модели оценки запросов и транзакций для bigdata значительно отличаются от моделей openrdf.

Структура описания ресурсов

Структура описания ресурсов (RDF) может пониматься как универсальная модель с гибкой схемой для описания метаданных и информации в форме графа. RDF представляет информацию в форме утверждений (троек или четверок). Каждая тройка означает ребро между двумя узлами в графе. Положение квадрата может использоваться для придания идентичности операторам (наш механизм происхождения основан на этом подходе) или для размещения операторов внутри именованного графа. RDF предоставляет некоторые базовые концепции, используемые для моделирования информации - операторы состоят из субъекта (URI или пустой узел), предиката (всегда URI), объекта (URI, пустого узла или буквенного значения) и контекста. (URI или пустой узел). URI используются для идентификации конкретного ресурса19, тогда как литеральные значения описывают константы, такие как символьные строки, и могут содержать либо языковой код, либо атрибут типа данных в дополнение к своему значению. RDF также предоставляет синтаксис на основе XML (называемый RDF / XML20) для обмена графами RDF.

Существует также теоретический слой над моделью RDF и синтаксисом обмена RDF / XML, который полезен для описания онтологий и для вывода. RDF Schema21 и OWL Ontology Web Language (OWL) представляют собой два таких основанных на стандартах уровня поверх RDF. Схема RDF полезна для описания иерархий классов и свойств. OWL - более выразительная модель. Конкретные конструкции OWL могут применяться для объединения и семантического выравнивания, такие как owl: equalClass и owl: equalProperty (для выравнивания схем) и owl: sameAs (для динамического связывания данных экземпляра вместе).

Между выразительностью и масштабом существует внутреннее противоречие, поскольку высокая выразительность требует больших вычислительных ресурсов и только усиливается по мере увеличения размера данных. Bigdata сосредоточился на масштабе, а не на выразительности. Схема базы данных для RDF Bigdata поддерживает три различных режима базы данных RDF: тройки, тройки с происхождением и квадраты. Эти режимы отражают небольшие вариации общей схемы базы данных. Абстрактно эта схема может быть концептуализирована как отношение лексикона и утверждения, каждое из которых использует несколько индексов. Совокупность этих индексов в совокупности представляет собой экземпляр базы данных RDF. Каждая база данных RDF идентифицируется своим собственным пространством имен. В экземпляре bigdata можно управлять любым количеством экземпляров базы данных RDF.

Словарный запас

Для управления значениями атрибутов переменной длины, произвольной мощностью значений атрибутов и отсутствием статической типизации, связанной с данными RDF, использовалось множество подходов. Bigdata использует комбинацию встроенных представлений для числовых и фиксированных литералов RDF со словарным кодированием URI и других литералов. Встроенное представление обычно на один байт больше, чем соответствующий примитивный тип данных, и налагает естественный порядок сортировки для соответствующего типа данных. Встроенные представления для xsd: decimal и xsd: integer используют кодировку переменной длины. URI, объявленные в словаре при создании экземпляра базы знаний, также встроены (в 2-3 байта). В зависимости от конфигурации пустые узлы обычно встраиваются. Как обсуждалось в другом месте, утверждения об утверждениях встроены как представление утверждения, которое они описывают.

Закодированные формы значений RDF известны как внутренние значения (IV). IV - это идентификаторы переменной длины, которые фиксируют различные различия, относящиеся как к

данным RDF, так и к тому, как база данных кодирует значения RDF. Каждый IV включает в себя байт флагов, который указывает тип значения RDF (URI, литерал или пустой узел), естественный тип данных значения RDF (Unicode, xsd: byte, xsd: short, xsd: int, xsd: long, xsd: float, xsd: double, xsd: integer и т. д.), захватывается ли значение RDF полностью встроенным представлением и является ли это типом данных расширения. Типы данных, определяемые пользователем, могут быть созданы с использованием байта расширения, который необязательно следует за байтом флагов. Встраивание используется для уменьшения шага в индексах операторов и минимизации необходимости материализовать значения RDF из индексов словаря при оценке SPARQL FILTER.

Лексика состоит из трех индексов:

- BLOBS - большие литералы и URI хранятся в индексе BLOBS. Ключ формируется из байта флагов, байта расширения, хэш-кода int32 литерала и счетчика конфликтов int16. Значение, связанное с каждым ключом, является Unicode-представлением значения RDF. Использование этого индекса помогает не допускать попадания очень больших литералов в индекс TERM2ID, где они могут привести к серьезному перекосу в размере страницы B + Tree. Компонент хэш-кода индекса BLOBS вводит значительный случайный ввод-вывод во время операций загрузки. Таким образом, использование индекса BLOBS ограничено литералами, длина строки которых превышает пороговое значение (256). Это лишь небольшой процент литералов в изученных нами наборах данных.

- TERM2ID - ключ является ключом сопоставления Unicode для литерала или URI. Значение - присвоенный уникальный идентификатор int64.

- ID2TERM - ключ является идентификатором (из индекса TERM2ID). Значение - это значение RDF.

При написании индексов лексики используется последовательный подход. Это позволяет производить запись в лексикон без глобальной блокировки в федерации. Необязательный полнотекстовый индекс отображает токены, извлеченные из значений RDF, во внутренние идентификаторы для этих значений RDF и может использоваться для выполнения поиска по ключевым словам в хранилище троек или четверок.

Индексы выписки

Отношение Утверждение моделирует Субъект, Предикат, Объект и, необязательно, Контекст для каждого утверждения. База данных RDF использует покрывающие индексы. Для каждой возможной комбинации переменных и констант в базовом тройном шаблоне (или четырехугольном шаблоне) существует кластеризованный индекс, который имеет хорошую локальность для этого шаблона доступа. Для тройного хранилища требуется 3 индекса операторов (SPO, POS и OSP). Для четырехъядерного хранилища требуется 6 индексов операторов (OCSP, SPOC, CSPO, PCSO, POCS и SPOC). В каждом случае имя индекса указывает способ, которым были заказаны субъект, предикат, объект и необязательный контекст для формирования ключей для индекса.

Обработка запросов SPARQL

Важно помнить об архитектурных различиях между масштабируемой архитектурой (включая журнал, WAR и кластер репликации высокой доступности) и горизонтально масштабируемой архитектурой (федерация). Сканирование индекса в масштабируемой архитектуре превращается в случайные операции ввода-вывода, поскольку индекс не находится в порядке ключей на диске. Однако сканирование индекса в масштабируемой архитектуре превращается в последовательные операции ввода-вывода, поскольку подавляющее большинство всех данных в кластере находится в файлах сегментов индекса только для чтения в порядке ключей на диске. Это архитектурное отличие означает, что кластер может более эффективно обрабатывать планы запросов, которые

выполняют непрерывное сканирование индекса. Однако, поскольку сканирование индекса превращается в случайный ввод-вывод в масштабируемой архитектуре, следует использовать либо множество шпинделей, либо твердотельные накопители, чтобы уменьшить время ожидания диска при вводе-выводе. В дополнение к внутренним ресурсам и возможностям для увеличения параллелизма, федерация имеет еще два архитектурных преимущества. Во-первых, горизонтально масштабируемая архитектура может использовать фильтр цветения перед каждым сегментом индекса. Это означает, что точечные тесты могут быть намного быстрее в кластере, чем на отдельной машине, поскольку правильные отклонения никогда не коснутся диска. Во-вторых, все узлы B + Tree в сегменте индекса находятся в одной непрерывной области на диске. Когда сегмент индекса открыт, узлы считываются с использованием одного устойчивого ввода-вывода. После этого чтение листа в сегменте индекса будет выполнять не более одного ввода-вывода.

RDF-запрос основан на шаблонах операторов. Тройной шаблон имеет общую форму (S, P, O), где S, P и O - либо переменные, либо константы в позициях Subject, Predicate и Object соответственно. Для четырехъядерного хранилища это обобщено как шаблоны, имеющие форму (S, P, O, C), где C - позиция контекста (или графика) и может быть либо пустым узлом, либо URI. Bigdata переводит SPARQL в абстрактное синтаксическое дерево (AST), которое довольно близко к синтаксису SPARQL, а затем применяет к этому AST серию оптимизаторов перезаписи. Эти оптимизаторы решают широкий спектр проблем, включая замену констант в план запроса, создание предложения WHERE и проекции для запроса DESCRIBE или CONSTRUCT, статический анализ переменных, выравнивание групп, исключение выражений или групп, которые, как известно, оцениваются как константа, обеспечивающая согласованность планов запросов с семантикой восходящей оценки SPARQL, переупорядочение объединений, присоединение ФИЛЬТРОВ в наиболее выгодных местах и т. д. Переписывание основано либо на полностью решаемых критериях, либо на эвристике, а не на поиске пространства возможных планы. Использование эвристики позволяет отвечать на запросы, содержащие 50–100 СОЕДИНЕНИЙ с очень малой задержкой - при условии, что соединения делают запрос выборочным в данных. Объединения переупорядочиваются на основе статического анализа запроса, распространения привязок переменных, быстрых оценок количества элементов для тройных шаблонов 25 и анализа распространения переменных в области видимости между подгруппами и подгруппами SELECT.

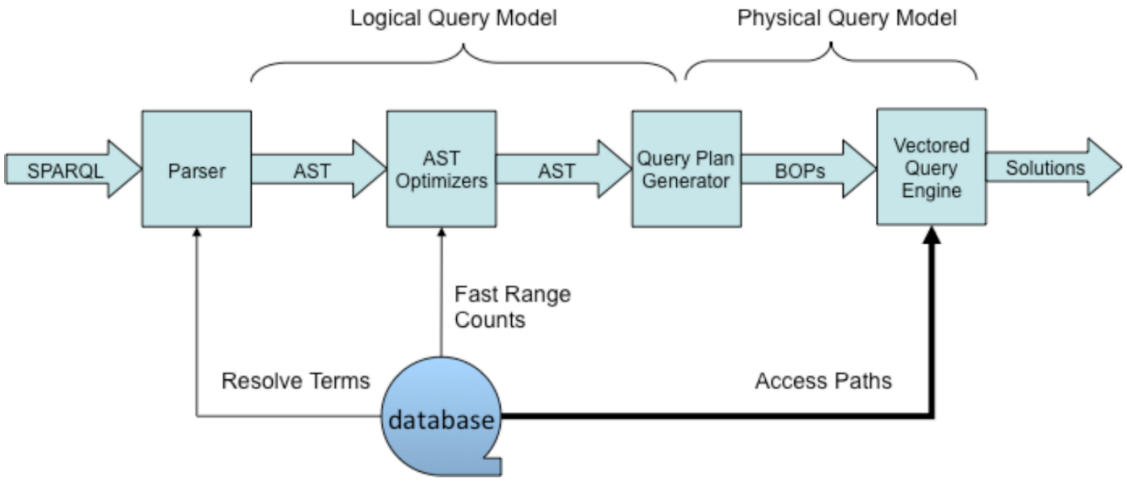


Рисунок 9: Выполнение запроса

После переписывания AST он переводится в физический план запроса. Каждый шаблон группового графа, уцелевший из исходного запроса SPARQL, будет смоделирован последовательностью физических операторов. Вложенные группы оцениваются с помощью хеш-соединений набора решений. Видимость переменных внутри групп и подзапросов соответствует правилам для области переменных для SPARQL (например, как если бы выполнялась оценка снизу

вверх). Для данной группы обычно существует последовательность требуемых объединений, соответствующая шаблонам операторов в исходном запросе. Также могут быть необязательные объединения, объединения подвыборки, объединения предварительно вычисленных именованных наборов решений и т. Д. Ограничения (ФИЛЬТРЫ) оцениваются, как только известно, что переменные, участвующие в ограничении, связаны, но не позднее конца группа. Многие ФИЛЬТРЫ SPARQL могут работать непосредственно с IV. Когда ФИЛЬТР требует доступа к материализованному значению RDF, план запроса включает дополнительные операторы, обеспечивающие материализацию объектов значения RDF перед их использованием.

План запроса передается на выполнение механизму векторных запросов. Механизм запросов поддерживает как масштабирование, так и оценку горизонтального масштабирования. Для масштабирования операторы несут дополнительные аннотации, которые указывают, должны ли они выполняться на контроллере запросов (где запрос был отправлен для выполнения), должны ли они быть сопоставлены с разделом индекса, на котором будет читаться путь доступа (для объединений) 26 и могут ли они работать в какой-либо службе данных в федерации. Последний оператор в плане запроса записывает в приемник, который опорожняется клиентом, отправившим запрос. Для горизонтального масштабирования в конце плана запроса добавляется оператор, чтобы гарантировать, что решения копируются обратно в контроллер запросов, где они доступны для клиента. Для всех остальных операторов промежуточные решения помещаются в рабочую очередь для целевого оператора. Механизм запросов управляет очередями работы для каждого оператора, планирует выполнение операторов и управляет перемещением данных в федерации.

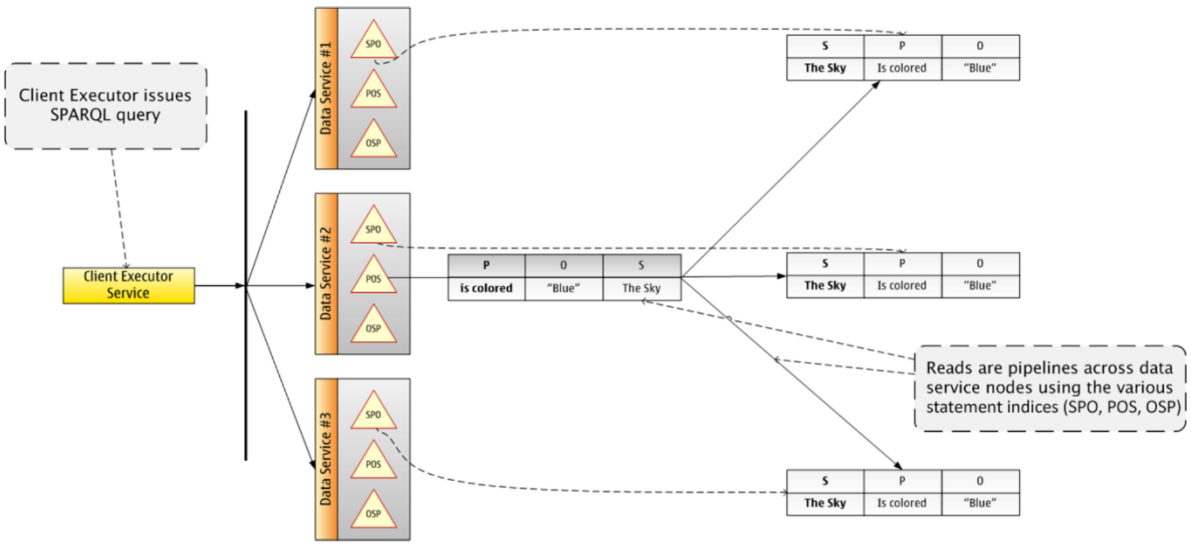


Рисунок 10: Иллюстрация выполнения конвейерного соединения в горизонтальном масштабе.

Механизм запросов поддерживает параллелизм на нескольких уровнях:

Запросы с одновременным выполнением. Пул потоков в конечной точке SPARQL контролирует количество запросов, которые могут выполняться одновременно. Одновременное выполнение разных операторов в одном запросе. Параллелизм здесь не ограничен, чтобы избежать тупиковой ситуации. Параллелизм на этом уровне также помогает гарантировать, что рабочая очередь для каждого оператора остается заполненной, и служит для минимизации задержки для запроса. Одновременное выполнение одного и того же оператора в рамках одного запроса для разных фрагментов данных. Аннотация используется для ограничения параллелизма на этом уровне. Решения направляются каждому оператору. Некоторые операторы являются «сразу» и перед выполнением буферизуют все промежуточные решения. Например, при оценке сложного

необязательного параметра мы полностью буферизуем промежуточные решения в хеш-индексе перед запуском подгруппы. Другие операторы «заблокированы» - они будут буферизовать большие блоки данных в собственной куче, чтобы обрабатывать как можно больше данных при каждом выполнении - например, хеш-соединение против сканирования пути доступа. Однако многие операторы являются «конвейерными» - они будут выполняться для каждого фрагмента промежуточных решений.

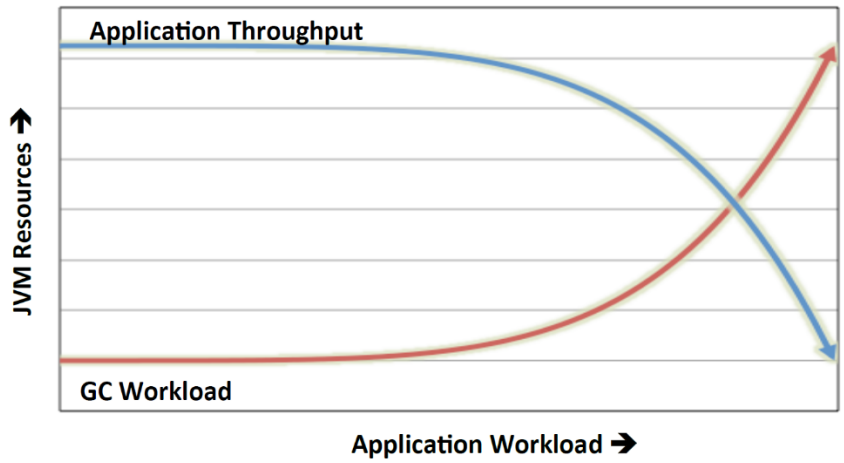
Bigdata отдает предпочтение конвейерному выполнению операторов, когда это возможно. Запросы SPARQL, включающие последовательность тройных шаблонов, транслируются с использованием вложенных индексных объединений и имеют очень низкую задержку для первого решения. Каждый путь доступа ограничен, поскольку решения проходят через механизм запросов. Ограниченные пути доступа исследуются с использованием привязок для каждого промежуточного решения. Это превращается в высоко локализованное чтение индекса B + Tree для этого пути доступа. Конвейерное выполнение также поддерживается для DISTINCT и для простых OPTIONAL (OPTIONAL, содержащего единственный тройной шаблон и без фильтров, которые требуют материализации привязок переменных к лексикону).

Планы запросов, включающие GROUP BY, ORDER BY, сложные OPTIONAL, EXISTS, NOT EXISTS, MINUS, SERVICE или sub-SELECT, имеют этапы, на которых не могут быть получены какие-либо выходные данные, пока все решения не будут вычислены до этой точки в плане запроса. Такие запросы могут иметь низкую задержку, пока объем данных невелик. Если вы хотите агрегировать или упорядочить подмножество данных, вы можете переместить часть запроса в подвыборку с помощью LIMIT, но оставьте агрегирование или предложение упорядочить по в родительском запросе. Подвыборка будет обработана конвейером, и оценка будет остановлена, как только будет достигнут предел. Затем родительский запрос может агрегировать или упорядочивать только данные из вложенного SELECT.

Планы запросов, включающие подгруппы (включая сложные OPTIONAL, MINUS и SERVICE), отрицание в фильтрах (EXISTS, NOT EXISTS) или подвыборки, обрабатываются аналогичным образом. В каждом случае оператор, строящий хеш-индекс, накапливает промежуточные решения. После того, как все промежуточные решения будут собраны, привязки для переменных, входящих в область действия, переносятся в подплан. Выходные решения из подплана затем объединяются по хэш-индексу и переносятся в оставшуюся часть родительского плана запроса. UNION обрабатывается оператором TEE. Решения для каждой стороны объединения разделены на сегменты плана запроса, которые выполняются одновременно.

Аналитический режим запроса

Инкрементальная компиляция и сложный анализ горячих точек во время выполнения приложений Java часто дает код со скоростью написанной вручную C. Однако куча управляемых объектов является известным слабым местом JVM. По мере увеличения скорости создания объектов и периода хранения объектов время рабочего цикла сборщика мусора увеличивается. В конце концов, сборка мусора начинает блокировать приложение на значительный период времени.



Чтобы решить эту проблему, bigdata предоставляет две реализации для каждой операции на основе хеш-индекса. Одна версия оператора использует классы коллекции JVM. Эти классы обеспечивают отличную производительность для небольших размеров коллекций и в некоторых случаях предлагают очень высокий уровень параллелизма. Другая версия оператора основана на структуре индекса HTree и MemStore (который поддерживается собственной кучей процесса C). Эти операторы масштабируются до очень больших наборов данных без каких-либо накладных расходов со стороны сборщика мусора. Для запросов выбора с малой задержкой производительность операторов собственной памяти близка к производительности версий JVM того же оператора. Однако, когда запросы должны материализовать большие сотни миллионов решений, классы коллекции JVM несут очень высокие издержки. Сборщик мусора стоит, но операторы собственной памяти изящно масштабируются. Если указан режим аналитического запроса, план запроса будет использовать операторы собственной памяти. Режим аналитического запроса может быть включен с помощью флажка в HTML FORM, параметр запроса URL (? Analytic) или подсказка запроса (hint: analytic) может использоваться для включения режима аналитического запроса.

Вывод и поддержание истины

Теория моделей RDF определяет различные следствия. Следствиями являются тройки, которые явно не указаны во входных данных, но база данных должна вести себя так, как если бы эти тройки присутствовали. Вообще говоря, есть два способа справиться с такими последствиями. Во-первых, они могут быть вычислены заранее, когда данные загружены и сохранены в базе данных вместе с явно заданными тройками. Этот подход известен как активное замыкание, потому что вы вычисляете замыкание теории модели по явным троек и материализуете эти выведенные тройки в базе данных. Основное преимущество активного закрытия состоит в том, что оно материализует все данные, как явные, так и предполагаемые, в базе данных. Это значительно упрощает планирование запросов и обеспечивает одинаково быстрые пути доступа для подразумеваемых и явных операторов. Стремительное закрытие может быть чрезвычайно эффективным, но все же может быть значительная задержка, особенно для очень больших наборов данных, поскольку время для вычисления закрытия часто примерно равно времени загрузки необработанных данных. Другой недостаток - это пространство, поскольку предполагаемые тройки хранятся в индексах, тем самым увеличивая размер набора данных на диске. Второй подход - материализовать выводы во время запроса. Это имеет то преимущество, что набор данных может быть запрошен, как только необработанные данные будут загружены, а требования к хранилищу соответствуют требованиям только для необработанных данных. Для этого существует множество методов, включая обратное рассуждение 30, 31, которое часто используется в системах Prolog, и

волшебные наборы, которые часто используются в системах регистрации данных. В SPARQL 1.1 пути к свойствам также можно использовать для встраивания выводов выбора в запросы.

База данных RDF, использующая стратегию активного закрытия, сталкивается с другой проблемой. Он должен поддерживать согласованное состояние базы данных, включая предполагаемые тройки, по мере добавления или удаления данных из базы данных. Эта проблема известна как сохранение истины. Для схемы RDF поддержание истины при добавлении новых данных тривиально. Однако это может стать довольно сложным при удалении данных, поскольку необходимо провести поиск, чтобы определить, являются ли выводы, уже находящиеся в базе данных, по-прежнему без отозванных утверждений. Еще раз, есть несколько способов справиться с этой проблемой. Одна крайность - отбросить выводы, удалить их из базы данных, а затем заново вычислить полное прямое закрытие оставшихся операторов. У этого есть все недостатки, связанные с нетерпеливым закрытием, и даже тривиальный отвод может привести к повторному вычислению всего закрытия. Во-вторых, поддержание истинности может быть достигнуто путем хранения цепочек доказательств в индексе33. Когда утверждение отозвано, вычисляются следствия этого утверждения, и для каждого такого следования проверяются цепочки доказательств, чтобы определить, доказано ли утверждение без отозванного утверждения. Однако хранение цепочек доказательств может быть обременительным. В-третьих, волшебные наборы снова предлагают эффективную альтернативу системе, использующей активное замыкание для предварительной материализации выводов. Вместо того, чтобы хранить цепочки доказательств, мы можем просто вычислить набор следствий для утверждений, которые должны быть отозваны, а затем отправить запросы к базе данных, в которой мы узнаем, доказаны ли эти утверждения.

Bigdata поддерживает гибридный подход, в котором активное закрытие выполняется для некоторых следствий схемы RDF, в то время как другие следствия материализуются только во время запроса. Такой подход не редкость для баз данных RDF. Кроме того, масштабируемая архитектура также поддерживает поддержание истины на основе хранения цепочек доказательств. Поддержание истинности недоступно при горизонтальном масштабировании, потому что все обновления должны быть сериализованы (выполняться в одном потоке), чтобы поддержание истинности имело четко определенную семантику.

NewSQL базы данных

Базы данных NewSQL - это современная форма реляционных баз данных, предназначенная для обеспечения масштабируемости, сравнимой с базами данных NoSQL при сохранении транзакционных гарантий, поддерживаемых традиционными системами баз данных. Такие СУБД обладают следующими характеристиками:

- SQL - это основной механизм взаимодействия с приложениями.
- Поддержка ACID для транзакций
- Механизм управления неблокирующим параллелизмом.
- Архитектура, обеспечивающая гораздо более высокую производительность на каждом узле.
- Горизонтально масштабируемая архитектура без совместного использования ресурсов, способная работать на большом количестве узлов без проблем.

Ожидается, что системы NewSQL примерно в 50 раз быстрее традиционных. СУБД OLTP. Например, VoltDB линейно масштабируется в случае несложных (одноразделных) запросов и

обеспечивает поддержку ACID. Масштабируется на десятки узлов, каждый из которых ограничен размером основной памяти.

CDN – Content Delivery Network

Основные термины

Прежде чем начать предметный разговор об особенностях CDN, определимся с основной терминологией.

CDN (Content Delivery Network) — это географически распределённая сетевая инфраструктура, обеспечивающая быструю доставку контента пользователям веб-сервисов и сайтов. Входящие в состав CDN серверы географически располагаются таким образом, чтобы сделать время ответа для пользователей сайта/сервиса минимальным.
Ориджин (origin) — сервер, на котором хранятся исходные файлы или данные, раздаваемые через CDN.
PoP (point of presence, точка присутствия) — кэширующий сервер в составе CDN, расположенный в определенной географической локации. Для обозначения таких серверов также используется термин edge.
Динамический контент — контент, генерируемый на сервере в момент получения запроса (либо изменяемый пользователем, либо загружаемый из базы данных).
Статический контент — контент, хранимый на сервере в неизменяемом виде (например, бинарные файлы, аудио- и видеофайлы, JS и CSS).

Немного истории и теории

Резкий рост Интернета в середине 1990-х привел к ситуации, что серверы стали с трудом выдерживать нагрузку. С серверами того времени (которые по техническим характеристикам иногда были слабее не самого производительного современного ноутбука) приходилось идти на разные ухищрения: погуглите, например, «иерархическое кэширование» и information superhighway — сейчас эти словосочетания используются разве что в статьях по истории интернет-технологий. Чтобы понять, как развивались технологии раздачи контента, сделаем небольшое теоретическое отступление.

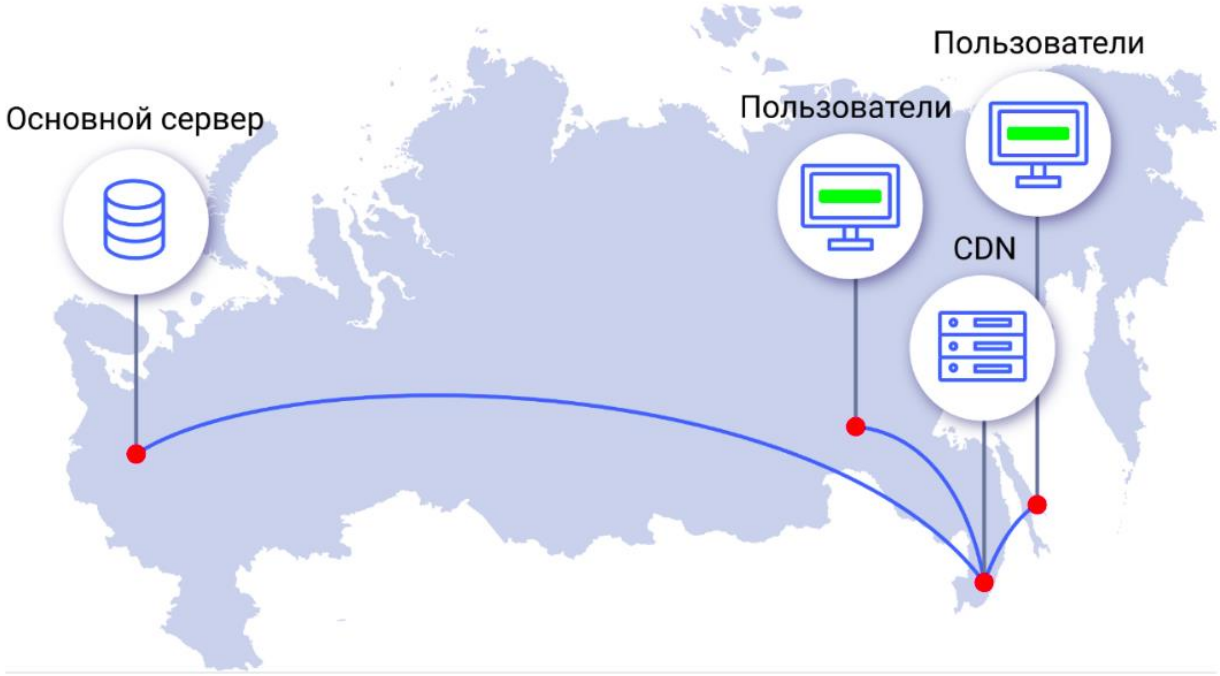
Обратим внимание: раздача статического и динамического контента связаны с разными типами нагрузки на сервер. В случае с динамическим контентом, генерация которого связана с обращениями к базе данных, важны быстродействие процессора и объём оперативной памяти.

Для раздачи статического контента, который в большинстве случаев оказывается очень «тяжелым» и который нужно загрузить очень быстро, важна в первую очередь скорость сети. Смысл технических решений для ускорения раздачи статики заключается в следующем: обеспечить горизонтальное масштабирование без сложных двусторонних синхронизаций с основным сервером.
Для снижения нагрузки владельцы веб-сервисов ещё в конце 1990-х годов начали раздавать статику и динамику с разных серверов. Крупные веб-проекты с огромной аудиторией, разбросанной по всему миру, начали размещать серверы со статикой в разных географических точках.
Тогда же, в конце 1990-х, стали появляться компании, у которых организация раздачи статики стала одним из основных направлений бизнеса. В 1998 году студент Массачусетского технологического института Дэниэл Левин и преподаватель математики Томсон Лейтон основали компанию Akamai. Ныне она является одним из крупнейших (если не самым крупным) CDN-провайдером в мире.
Уже в 2004 году CDN использовали более 3000 компаний; общий объем расходов на доставку контента составлял до 20 миллионов долларов в месяц.

Количество CDN во всём мире постоянно растет: соответствующие услуги предоставляют как крупные международные компании (например, Akamai, Amazon, Cloudflare), так и многочисленные региональные провайдеры (подробные обзоры).
CDN используется не только для раздачи статики в строгом смысле слова: распределение контента по многочисленным серверам в разных точках планеты помогает обеспечивать доступность в периоды пиковых нагрузок.

В течение последних 10-12 лет широкое распространение в Интернете получил еще один тип контента — стриминговый (многочисленные сервисы потокового аудио и видео, которые в наши дни имеют огромную популярность и миллионную, если не миллиардную, аудиторию). Раздача сегодня является еще одним распространенным сценарием использования CDN.

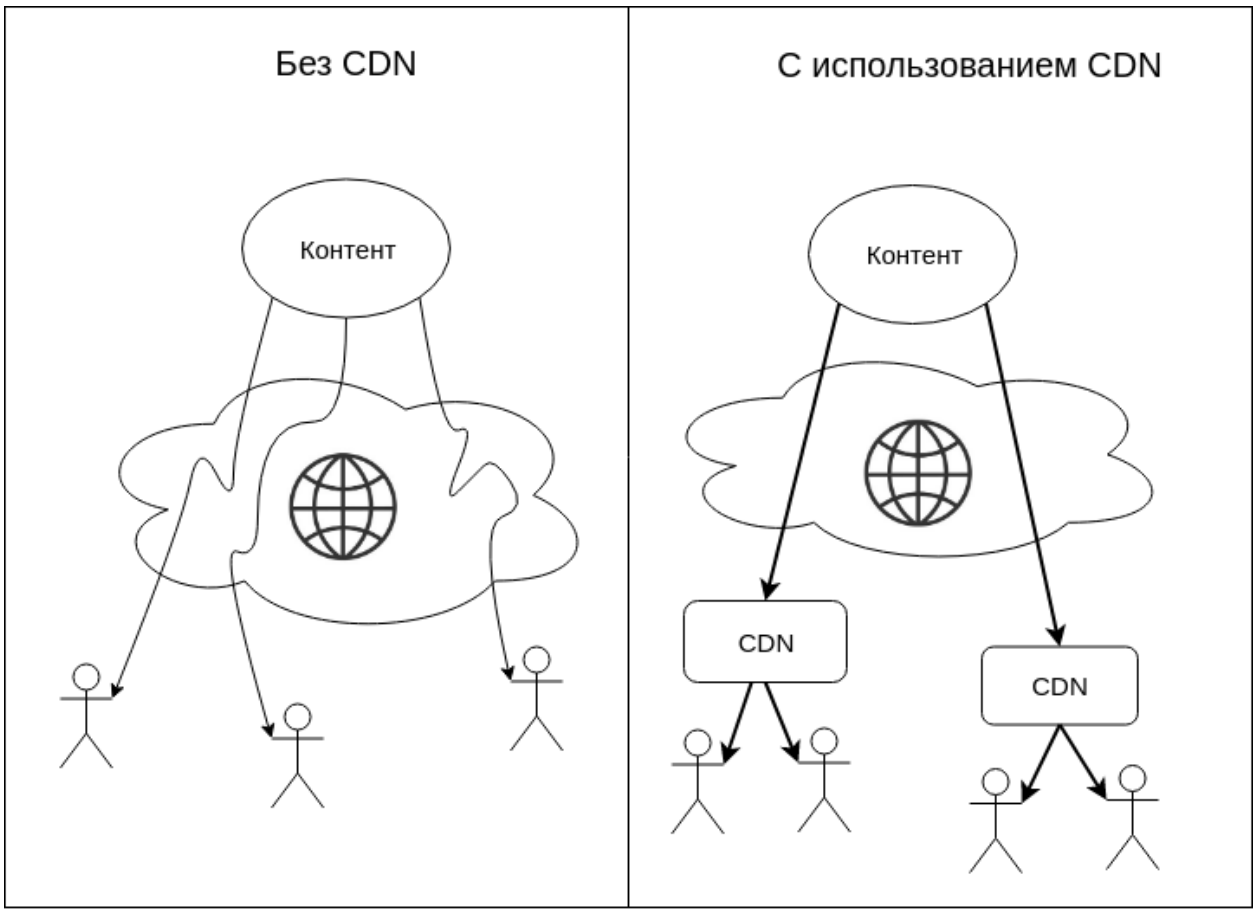
Рассмотрим принципы работы и особенности использования CDN более подробно.



Как работает CDN

Представим себе веб-сервис, которым пользуются люди на всей территории России. Основные серверы расположены в Санкт-Петербурге, а пользователи находятся в самых разных географических точках: скажем, в Краснодаре (2 604,2 км от Петербурга), Новосибирске (3 826,1 км), Иркутске (5 661, 7 км) или Владивостоке (9 602, 4 км). Чем дальше пользователь находится от оригинального сервера, тем больше время «оригинального» ответа. На заре Рунета, в самом начале 2000-х, жители Южно-Сахалинска или Петропавловска-Камчатского могли дожидаться полной загрузки простой веб-страницы полновесные 5, а то и все 10, минут.

При использовании CDN всё происходит по-другому: пользователь из Владивостока переадресуется к географически ближайшему кэширующему серверу в составе CDN, благодаря чему доставка статического контента происходит гораздо быстрее.



Для ускорения раздачи динамики при использовании CDN используются другие механизмы: CDN-провайдер за счет своей сети сокращает сетевой маршрут.
Ещё один интересный сценарий использования CDN — так называемый live-streaming: пользователи Интернета со всего мира могут в браузере (а иногда и в специальном приложении) смотреть или слушать трансляцию с мест событий. Устроено это так: один или несколько ориджин-серверов принимают с видеокамеры транслируемый поток, который сразу же ретранслируется на точки присутствия. Ориджин-серверы при этом контент клиентам не раздают. В состав стриминговых CDN входят также балансировщики нагрузки, перенаправляющие запросы к наименее загруженным на текущий момент edge-серверам.

Как организована раздача контента?

Как правило, для настройки раздачи статического контента через CDN необходимо выполнить следующие шаги:

Шаг 1: Вынести статику сайта на отдельный домен, например, static.example.com — это будет origin.

Шаг 2: Для работы через CDN создать домен вида cdn.example.com.

Шаг 3: Подключить CDN у провайдера. Для подключения владельцу веб-сервиса необходимо сообщить провайдеру следующее: домен, с которого он будет забирать статику — static.example.com; домен, с которого будет идти раздача — cdn.example.com.

Шаг 4: У своего DNS-регистратора настроить CNAME запись с cdn.example.com на домен CDN-провайдера, который CDN провайдер выделяет при подключении. Например, в CDN Selectel такой домен имеет вид 85e77c09-bc03-43bf-b8f3-9492ae33390f.selcdn.net, где 85e72c09-bc03-43bf-b8f3-9492ae33390f генерируется автоматически.

Шаг 5: На своем сайте изменить домен для статики, которую планируется раздавать через CDN, на cdn.example.com.

Пользователь набирает в строке браузера адрес www.example.com, с которого он получает HTML-страницу. При этом весь статический контент, например, графические изображения, подгружается из CDN (с адреса cdn.example.com). Статический контент, предназначенный для раздачи, часто помещается в объектные хранилища (об этом мы писали еще шесть лет назад). Существует множество плагинов и расширений для популярных CMS (Wordpress, Joomla, Drupal, 1С Битрикс и других), с помощью которых можно настроить интеграцию с облачными сервисами хранения и раздачу статики через CDN. Веб-сервис после подключения CDN будет работать на том же оригинальном сервере. Кэшированные части сайта будут загружены на серверы CDN-сети. Система находит для пользователя ближайший сервер и максимально быстро загружает статику сайта с него.

Обратим внимание на один важный момент: серверы, входящие в состав CDN, не являются подобием файловых серверов, на которые контент размещается для последующего скачивания. CDN используются не для хранения контента, а для кэширования на основе конкретных алгоритмов.

Как CDN понимает, где находится ближайший кэширующий сервер?

Как правило, для подгрузки контента из CDN используются две популярные технологии: GeoDNS и AnyCast.
С помощью GeoDNS можно привязать к одному доменному имени несколько IP-адресов. В зависимости от географического положения (определяется по IP-адресу, с которого пришел запрос) пользователь перенаправляется на ближайший сервер. Об особенностях работы GeoDNS можно почитать в этой статье (на английском языке). При использовании технологии Anycast адреса общие, но маршрутизация происходит на «свои» серверы в пределах региона. При обращении к адресу www.example.com пользователь переадресуется на ближайшую точку присутствия. Провайдер пользователя получает несколько анонсов от разных сетей, в которых есть точка присутствия, и маршрутизатор провайдера выбирает из них самый близкий. Ответ аналогичным образом возвращается по наиболее короткому маршруту.

Как кэшируется контент?

Самой распространенной является схема **по первому обращению**: максимальное количество времени на загрузку затрачивает пользователь, обратившийся к оригинальному серверу первым. Все последующие пользователи будут получать данные, кэшированные на ближайшей к ним точке присутствия.

Здесь очень важна география: например, после обращения пользователя из Рио-де-Жанейро данные будут закэшированы на сервере, находящемся на территории Бразилии, что не решит проблемы со скоростью доступа для пользователей из Парижа или Лондона. Для преодоления ограничений, накладываемых этой схемой, используются технологии регионального извлечения: соседние серверы, входящие в состав CDN, забирают контент друг у друга, а не обращаются к оригинальному серверу. В большинстве CDN пользователь, отправивший запрос на получение статического контента, переадресуется к ближайшей точке присутствия и получает кэшированную версию этого контента с неё. Если ближайшая точка присутствия не сможет найти файлы, начнётся поиск по соседним точкам присутствия, откуда и будет перенаправлен ответ пользователю. В CDN Akamai эта процедура называется tiered distribution (на русский можно перевести как «многоуровневая раздача»).

Для чего используются CDN?

Чаще всего CDN используется для уменьшения времени отклика кэшированного контента, что, как

мы уже упоминали выше, уменьшает отток посетителей из-за медленной загрузки ресурса и тем самым сокращает возможные финансовые потери. Также CDN помогает снизить риск потери доступа к контенту из-за падения основного сервера. Контент будет доступен всё время, пока вы восстанавливаете работоспособность основного сервера.

Использование CDN существенно снижает нагрузку на основной сервер, что помогает решить проблему пиковых нагрузок. Современная CDN способна переживать очень большие нагрузки. В конце 2018 года компания Akamai заявила о рекордном объеме передаваемого через CDN трафика: 72 Тб/с. В наше время CDN активно используются также для раздачи стримингового контента.

О чем важно помнить при работе с CDN?

Как и любая технология, CDN обладает рядом особенностей. Самая первая проблема, с которой могут столкнуться использующие CDN веб-сервисы — это задержки кэширования. Вполне вероятна следующая ситуация: на основном сервере файл был изменён, а вот на кэширующих серверах он всё ещё будет лежать в неизмененном виде. Это особенно важно, когда через CDN распространяется часто обновляемый контент (фотографии с места событий, новые версии ПО и так далее)

Чтобы обеспечить доставку «свежего» контента в современных CDN имеется функция очистки кэша, то есть удаление контента из пула кэширования. Кроме того, владельцы сайтов и сервисов могут сами управлять настройками, используя заголовки-валидаторы (см. наши рекомендации на эту тему в опубликованной ранее статье). Еще одна сложность связана с блокировками: если по той или иной причине будут заблокированы сервисы, являющиеся вашими «соседями» по IP CDN-провайдера, вместе с вами может оказаться заблокированным и ваш сайт. Но и это проблема решаема: по запросу CDN-провайдеры могут изменить ваш IP-адрес.

Кому нужны CDN?

CDN нужны в первую очередь проектам с большой аудиторией в разных регионах или странах. Здесь всё ясно: снижение задержек, быстрая раздача контента и повышение уровня удобства, и, как следствие, больше довольных пользователей. CDN может пригодиться также разработчикам мобильных приложений: по статистике, пользователи часто отказываются продолжать работу с приложением из-за проблем со скоростью. В последнее время появились специальные технические решения, ориентированные на раздачу контента на мобильные устройства. Они так и называются — Mobile CDNs. Соответствующие услуги предлагают многие крупные CDN-провайдеры — например, Akamai или Amazon. Нужны CDN и проектам, ориентированным на распространение игрового, мультимедийного контента и стриминг (об этом уже было сказано выше).

Платформы запросов к большим данным

Платформы запросов к большим данным предоставляют фасады запросов поверх базовых хранилищ больших данных, которые упрощают запросы к базовым хранилищам данных. Обычно они предлагают SQL-подобный интерфейс запросов для доступа к данным, но различаются подходом и производительностью. Hive (Тусу и др., 2009) предоставляет абстракцию поверх Hadoop Distributed файловой системой (HDFS), которая позволяет запрашивать структурированные файлы с помощью SQL-подобного языка запросов. Hive выполняет запросы, переводя запросы в MapReduce задачи. Как следствие, запросы Hive имеют высокую задержку даже для небольших наборов данных. Преимущества Hive включают интерфейс запросов, подобный SQL, и гибкость для развития. Это возможно, поскольку схема хранится независимо от данных и данные проверяются только во время запроса. Этот подход называется «схемой при чтении» и сравним с

подходом «схема при записи» в базах данных SQL. Поэтому изменение схемы представляет собой сравнительно дешевую операцию. Колонна Hadoop

В отличие от Hive, Impala (Russel 2013) предназначена для выполнения запросов с низкими задержками. Он переиспользует те же метаданные и SQL-подобный пользовательский интерфейс, что и Hive, но использует собственный механизм распределенных запросов, позволяющий снизить задержки. Он также поддерживает HDFS и HBase в качестве базовых хранилищ данных.

Spark SQL (Шенкер и др., 2013) - еще одно средство обработки запросов с малой задержкой, которое поддерживает интерфейс Hive. В проекте утверждается, что «он может выполнять запросы Hive QL до 100 раз быстрее, чем Hive, без каких-либо изменений существующих данных или запросов. Это достигается путем выполнения запросов с использованием Фреймворк Spark (Захария и др., 2010), а не MapReduce Hadoop фреймворка.

Наконец, Drill - это реализация Dremel от Google с открытым исходным кодом (Melnik и другие. 2002), который похож на Impala, разработан как масштабируемая интерактивная система запросов вложенных данных. Drill предоставляет собственный SQL-подобный язык запросов DrQL, который совместим с Dremel, но разработан для поддержки других языков запросов, таких как язык запросов Mongo. В отличие от Hive и Impala, он поддерживает ряд источников данных без схемы, такие как HDFS, HBase, Cassandra, MongoDB и SQL базы данных.

Облачные хранилища

По мере роста популярности облачных вычислений растет и их влияние на большие данные. В то время как Amazon, Microsoft и Google используют собственные облачные платформы, другие компании, включая IBM, HP, Dell, Cisco, Rackspace и т. д., создают свое предложение вокруг OpenStack, платформы с открытым исходным кодом для создания облачных систем (OpenStack 2014). Согласно IDC (Grady 2013), к 2020 году 40% цифровой вселенной «будет «Затронуто» облачными вычислениями », и« возможно, до 15% данных будут сохранены в облаке».

Облако в целом, и особенно облачное хранилище, могут использовать как предприятия, так и конечные пользователи. Для конечных пользователей хранение их данных в облаке обеспечивает доступ из любого места и с любого устройства надежным способом. Кроме того, конечные пользователи могут использовать облачное хранилище как простое решение для онлайн-резервного копирования данных своих настольных компьютеров. по аналогии. Аналогично облачное хранилище для предприятий обеспечивает гибкий доступ из разных локаций и быстрое простое масштабирование емкости (Grady 2013), а также более низкие издержки на хранение и лучшую поддержку. Особенно это заметно когда потребности в корпоративных хранилищах меняются со временем вверх и вниз.

Технически облачные хранилища можно разделить на объектные и блочные хранилища. Хранилище объектов - это общий термин, описывающий подход к адресации и обработке дискретных единиц хранения, называемыми объектами. Напротив, данные блочного хранилища хранятся в томах, также называемых блоками. По словам Маргарет Роуз (2014b), «каждый блок действует как индивидуальный жесткий диск и обеспечивает произвольный доступ к битам и частям данных, таким образом такие хранилища, хорошо работают с такими приложениями, как базы данных. Помимо объектного и блочного хранилища, основные платформы обеспечивают поддержку хранилищ на основе реляционных и нереляционных баз данных, а также хранилищ в памяти и хранение очереди. Облачные хранилища имеют существенные различия, которые необходимо учитывать на этапе их выбора:

- Поскольку облачное хранилище является услугой, приложения, использующие такие хранилище, имеют меньший контроль над ним и может наблюдаться снижение

производительности в результате подключения к сети. Эти различия в производительности необходимо учитывать при проектировании и реализации приложений.

- Безопасность - одна из основных проблем, связанных с общедоступными облаками. В связи с этим, Технический директор Amazon прогнозирует, что через пять лет все данные в облаке будут зашифрованы по умолчанию.
- Многофункциональные облака, такие как AWS, поддерживают калибровку задержки, избыточности и уровни пропускной способности для доступа к данным, что позволяет пользователям найти правильный компромисс между стоимостью и качеством.

Еще одна важная проблема при рассмотрении облачного хранилища - поддерживаемая согласованность модели данных (и связанные с ней масштабируемость, доступность, устойчивость к разделам и задержка). Хотя Amazon Simple Storage Service (S3) поддерживает конечную согласованность, хранилище BLOB-объектов Microsoft Azure поддерживает высокую согласованность и в то же время высокая доступность по времени и устойчивость. Microsoft использует два уровня: (1) уровень потока, «который обеспечивает высокую доступность при разделении сети и других сбоях, и (2) разделительный слой, который« обеспечивает гарантию сильной согласованности.

Перспективные требования к хранению больших данных

Стандартизированные интерфейсы запросов

В среднесрочной и долгосрочной перспективе базы данных NoSQL значительно выиграют от стандартизации интерфейсов запросов, аналогичные SQL для реляционных баз. В настоящее время нет стандартов, существуют только для отдельных типов хранилищ NoSQL за пределами стандартных API де-факто для графовых базы данных и язык обработки данных SPARQL при поддержке поставщиков Triplestore. Другие базы данных NoSQL обычно предоставляют собственный декларативный язык или API, декларативные языки, как правило, отсутствуют. Хотя для некоторых категорий базы данных (ключ / значение, документ и т. д.) декларативная языковая стандартизация все еще отсутствует, есть попытки обсудить потребность в стандартизации. Например, исследовательская группа ISO / IEC JTC по большим данным недавно рекомендовал существующему комитету по стандартам ISO / IEC продолжить определение стандартных интерфейсов для поддержки нереляционных хранилищ данных. Определение стандартизованных интерфейсов позволит создавать уровень виртуализации данных, который обеспечит абстракцию разнородного хранилища данных, поскольку они обычно используются для больших данных.

Безопасность и конфиденциальность

Были проведены интервью с консультантами и конечными пользователями хранилищ больших данных, которые несут ответственность за безопасность и конфиденциальность, чтобы узнать их личное мнение и идеи. На основании этих интервью были определены будущие требования к безопасности и конфиденциальности при хранении больших данных.

Общие данные и социальные обычные данные, хранящиеся в больших количествах, будут подлежать свободному совместному использованию, а также к производной работе с ними, чтобы максимизировать преимущества больших данных. Сегодня, пользователи не знают, как большие данные обрабатывают их личные данные (прозрачность). Кроме того, юридические ограничения в отношении конфиденциальности и авторских прав в отношении больших данных в настоящее время отсутствуют. Очевидно, что большие данные позволяют проводить новую

аналитику на основе агрегированных данных из множества источников. Как этот подход влияет на личную информацию? Как могут правила и положения для перекомпоновки и производных данных быть применимы для работы с большими данными?

Отслеживание данных и происхождение.

Отслеживание и происхождение данных становится все более актуальным по двум причинам: (1) пользователи хотят понимать, откуда берутся данные, правильны ли они и заслуживают ли доверия, и что происходит с их результатами, и (2) хранение больших данных должно соответствовать правилам, поскольку большие данные входят в критически важные бизнес-процессы и цепочки создания стоимости. Поэтому хранилища больших данных должны поддерживать метаданные происхождения, обеспечивать контроль происхождения данных на всем протяжении цепочки обработки данных и предлагать удобные способы понимания и отслеживания использование данных.

Песочница и виртуализация

Песочница и виртуализация аналитики больших данных становится более важным в дополнение к контролю доступа. При больших масштабах аналитика больших данных выигрывает от совместного использования ресурсов. Однако нарушение безопасности общих аналитических компонентов приводит к компрометации криптографических ключей доступа и полный доступ к хранилищу. Таким образом, работа в области аналитики больших данных должна быть изолирована, чтобы предотвратить эскалацию нарушений безопасности и, следовательно, несанкционированные доступ к хранилищу.

Семантические модели данных

Множество разнородных источников данных увеличивает затраты на разработку, так как приложения требуют знания об индивидуальных форматах данных каждого источника. Возникающая тенденция - семантическая сеть. Множество исследовательских проектов занимаются всеми уровнями семантического моделирования и вычислений. Например, была выявлена потребность в семантических аннотациях для сектора здравоохранения ЕС. Требование к хранению данных, таким образом, состоит в том, чтобы поддерживать крупномасштабные хранилища и управлять семантическими моделями данных.

Новые парадигмы для хранения больших данных

Расширенное использование баз данных NoSQL

Базы данных NoSQL, в первую очередь графовые базы данных и хранилища столбцов, все чаще используется в качестве замены или дополнения к существующим реляционным СУБД. Например, требование использования семантических моделей данных и перекрестных ссылок данных с множеством разных данных и источников информации настоятельно требуют возможность хранить и анализировать большие объемы данных с использованием моделей на основе графов. Тем не менее, это требует преодоления ограничений текущих систем на основе графов, как было описано выше. Например, Джим Уэббер заявляет: «Графовые технологии собираются быть невероятно важным». В другом интервью Рикардо Баэса-Йетс, вице-президент Yahoo по исследованиям в Европе и Латинской Америке, также отмечает важность обработки крупномасштабных графовых данных. Microsoft в исследовательском проекте Тринити добился значительного прорыва в этой области. Платформа Тринити предназначена для хранения данных в памяти и распределенной обработки. Основываясь на своей возможности очень быстрого обходить граф, исследователи Microsoft представили новый подход к работе с графовыми запросами. Другие проекты включают граф знаний Google и поиск по графу Facebook, которые демонстрируют растущую релевантность и растущую зрелость графовых технологий.

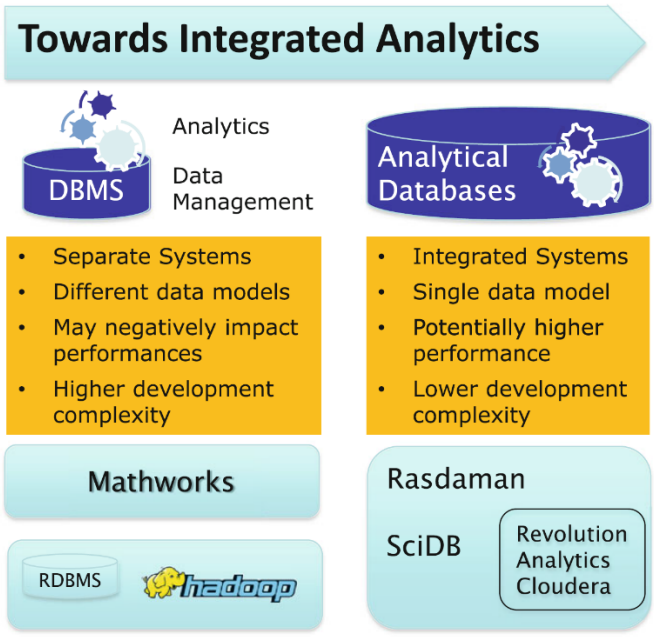
Хранение данных в памяти и хранилища столбцов

Многие современные высокопроизводительные базы данных NoSQL построены на основе хранения столбцов. Основное преимущество состоит в том, что в большинстве практических приложений используются только несколько столбцов для доступа к данным. Следовательно, хранение данных в столбцах позволяет обеспечить более быстрый доступ. Кроме того, столбцовые базы данных часто не поддерживают дорогостоящие присоединиться к операциям из реляционного мира. Вместо этого общий подход – использовать таблицы с одним широким столбцом, в котором хранятся данные на основе полностью денормализованной схемы. По словам Майкла Стоунбрейкера, «все поставщики SQL перейдут в хранилища столбцов, потому что они намного быстрее, чем обычные хранилища.

Высокопроизводительные базы данных в памяти, такие как SAP HANA, обычно сочетают методы in-memory с дизайном на основе столбцов. В отличие от реляционных систем, которые кэшируют данные в памяти, базы данных в памяти могут использовать такие методы, как антикэширование. Harizopoulos et al. показали, что большую часть времени для выполнения запроса тратится на административные задачи, такие как управление буфером и блокировки.

Конвергенция с аналитическими платформами

Было выявлено множество сценариев, требующих лучший анализ имеющихся данных для улучшения операций в различных секторах. Технически это означает повышенную потребность в сложной аналитике, выходящей за рамки простой агрегации и статистика. Стоунбрейкер отмечает, что потребность в сложных аналитика сильно повлияет на существующие решения для хранения данных. Поскольку аналитика для конкретного варианта использования является одним из наиболее важных компонентов, которые создают реальную ценность для бизнеса, становится все более важным наращивать эти аналитические возможности, удовлетворяющие требованиям к производительности, но также сокращающие общие сложность и стоимость.



Центр данных (The Data Hub)

Центральный хаб данных, который объединяет все данные на предприятии, - это парадигма, которая рассматривает управление всеми данными компании в целом, а не отдельными изолированными базами данных, управляемыми разными организационными подразделениями. Преимущество централизованных данных заключается в том, что данные можно анализировать

как единое целое, связывая различные наборы данных, принадлежащие компании, таким образом, это приводит к более глубокому пониманию данных. Типичные технические реализации основаны на основе системы Hadoop, которая может использовать HDFS или HBase для хранения интегрированного основного набора данных. С одной стороны, этот основной набор данных можно использовать в качестве достоверной информации и резервного копирования для существующих системы управления данными, но также он может обеспечить основу для расширенной аналитики, которая способна объединить ранее изолированные наборы данных. Такие компании, как Cloudera, используют эту парадигму для продвижения своего дистрибутива Hadoop. Уже существует множество вариантов использования концентраторов корпоративных данных. Пример из финансового сектора описан в следующем разделе.

Отраслевые примеры организации хранения больших данных

Финансовый сектор: централизованный концентратор данных

Как указано в описании отраслевых дорожных карт, финансовый сектор сталкивается с проблемами в связи с увеличением объемов данных и множеством новых источников данных, таких как социальные сети. Ниже описан вариант использования хранения больших данных для финансового сектора на основе краткого обзора решения Cloudera.

Финансовые продукты все в большей степени цифровизируются, включая онлайн-банкинг и торговлю. Поскольку онлайн- и мобильный доступ упрощает доступ к финансовым продуктам, существует рост уровня активности, ведущий к еще большему количеству данных. Потенциал больших данных в этом сценарии - использовать все доступные данные для построения точных моделей, которые могут помочь финансовому сектору лучше управлять финансовыми рисками. Согласно краткому описанию решения, компании имеют доступ к нескольким петабайтам данных. По словам Ларри Файнсмита, управляющий директор JPMorgan Chase, его компания хранит более 150 петабайт онлайн и использует Hadoop для обнаружения мошенничества. Во-вторых, новые источники данных увеличивают объем и разнообразие имеющихся данных. В частности, неструктурированные данные из веб-логов, социальных сетей, блогов и других новостных каналов могут помочь в управлении отношениями с клиентами, управлении рисками и, возможно, даже реализовать алгоритмическую торговлю. Собирая все данные вместе в централизованный центр данных, обеспечивается более подробная аналитика, которая может обеспечить конкурентоспособность. Однако, традиционные системы не могут угнаться за масштабом, стоимостью и громоздкой интеграцией традиционных процессов извлечения, преобразования и загрузки (ETL), используемые схемы с фиксированными данными и не могут обрабатывать неструктурированные данные. Однако хранилища больших данных очень хорошо масштабируются и могут обрабатывать как структурированные, так и неструктурированные данные.

Энергия: Измерение уровня устройства

В энергетическом секторе интеллектуальные сети и управление интеллектуальными счетчиками - это область, которая обещает высокие экономические и экологические преимущества. Как показано на следующем рисунке, внедрение возобновляемых источников энергии, таких как солнечные батареи, развернутые на домах могут вызвать нестабильность электросети. В настоящее время сетевые операторы мало осведомлены о последней миле до потребителей

энергии. Таким образом, они не могут должным образом реагировать на нестабильности, возникающие на краях энергосистемы (последняя миля). Анализируя данные интеллектуальных счетчиков, дискретизируемые с секундными интервалами, краткосрочное прогнозирование потребления энергии и управление спросом на питание таких устройств, как отопительные устройства и электрические автомобили становится возможным, таким образом стабилизируя сеть. При развертывании в миллионах домашних хозяйств объемы данных могут достигать петабайтного масштаба, что дает большую выгоду от новых хранилищ больших данных. В таблице показан объем данных только для необработанных данных, собранных за один день.

Проект Peer Energy Cloud (PEC) (2014 г.) - это проект, финансируемый государством, который продемонстрировал, как данные интеллектуальных счетчиков могут быть проанализированы и использованы для торговли энергией в окрестности, тем самым повышая общую стабильность энергосистемы. Более того, он успешно показал, что, собирая более мелкие детализированные данные, т.е. осуществляя мониторинг энергопотребления отдельных устройств в домашнем хозяйстве, точность прогнозирования энергопотребления домохозяйств может быть значительно лучше. По мере увеличения объемов данных становится все труднее обрабатывать данные с помощью устаревших реляционных баз данных.

Sampling rate	1 Hz
Record size	50 Bytes
Raw data per day and household	4.1 MB
Raw data per day for 10 Mio customers	~39 TB