
Базы Данных

Семинар 10

Описание семинара: MPP системы и колоночное хранение

Массивно-параллельные системы обработки (massively parallel processing, MPP).

Хотя архитектуры отдельных поставщиков могут варьироваться, массивно-параллельная обработка — наиболее развитой, проверенный и широко используемый механизм хранения и анализа больших объемов данных. Так что же собой представляет массивно-параллельная архитектура и что в ней особенного?

При использовании массивно-параллельной архитектуры данные разделяются на фрагменты, обрабатываемые независимыми центральными процессорами (CPU) и хранящиеся на разных носителях. Это похоже на загрузку разных фрагментов данных на несколько объединенных в сеть персональных компьютеров. Таким образом устраняется ограничение, обусловленное наличием одного центрального сервера с одним процессором и диском. Данные в массивно-параллельной системе распределяются по нескольким дискам, управляемым процессорами разных серверов.

В чем преимущество такой архитектуры?

Представьте себе движение по шестиполосному шоссе. Если эти шесть полос сойдутся в одну, пусть даже на коротком участке дороги, движение будет сильно затруднено. Если шесть полос остаются открытыми на всем протяжении пути от отправной точки до места назначения, то поездка будет гораздо более комфортной. В часы пик на дороге могут возникать пробки, но они будут меньше и очень скоро рассосутся. В случае с традиционной архитектурой базы данных в процессе обработки существует по крайней мере несколько точек, в которых количество полос сокращается до одной. Одной полосы может быть достаточно, только если объем движения небольшой. Именно это делает архитектуру MPP незаменимой для анализа больших объемов данных: она позволяет всем полосам оставаться открытыми на протяжении всего процесса.



Рассмотрим пример из мира баз данных. Традиционная база данных будет опрашивать терабайтную таблицу по одной строке за раз. Однако при использовании массивно-параллельной системы с 10 обрабатывающими устройствами данные разбиваются на 10 независимых фрагментов по 100 гигабайт. Это означает, что одновременно выполняется 10 запросов. При необходимости в большей вычислительной мощности и более высокой скорости просто добавьте дополнительные обрабатывающие устройства.

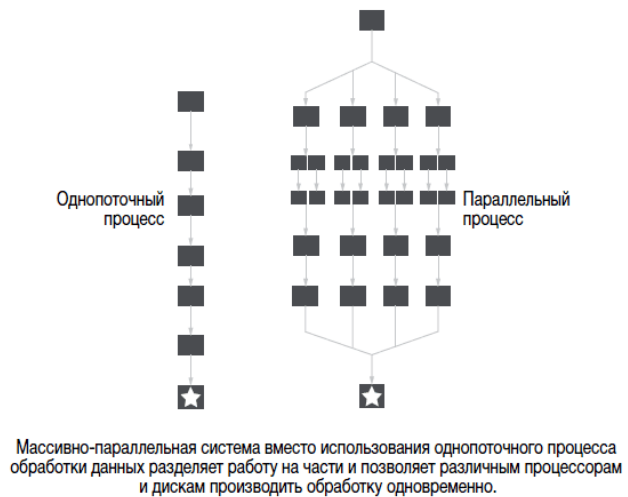
Разделяйте работу!

При использовании массивно-параллельной системы данные распределяются по разным процессорам и дискам. Подумайте о десятках или сотнях персональных компьютеров, каждый из которых содержит фрагмент большого набора данных. Это обеспечивает значительно более быстрое выполнение запроса, поскольку вместо одного большого запроса одновременно обрабатывается множество мелких независимых запросов.

Если бы система в нашем примере включала 20 обрабатывающих устройств, то вместо 10 фрагментов по 100 гигабайт одновременно обрабатывалось бы 20 независимых фрагментов по 50 гигабайт, что увеличило бы производительность. Дело несколько усложняется, когда выполнение запроса требует перемещения данных между процессорами, однако массивно-параллельные системы очень быстро справляются и с этой задачей (см. рис. 4.4).

При использовании MPP-систем данные не хранятся в одном месте, что облегчает восстановление в случае выхода оборудования из строя.

Эти системы также предусматривают инструменты управления ресурсами для управления процессорами и дисковым пространством, оптимизаторы запросов и другие средства, которые облегчают и повышают эффективность использования систем. Более подробное изложение этой темы выходит за рамки данной книги.



Аппаратная архитектура на примере VERTICA

Рассмотрим Vertica на уровне кластера. Эта СУБД обеспечивает массивно-параллельную обработку данных (MPP) в распределенной вычислительной архитектуре — «shared-nothing» — где, в принципе, любая нода готова подхватить функции любой другой ноды. Основные свойства:

- отсутствует единая точка отказа,
- каждый узел независим и самостоятелен,
- отсутствует единая для всей системы точка подключения,
- узлы инфраструктуры дублируются,
- данные на узлах кластера автоматически копируются.

Кластер без проблем линейно масштабируется. Мы просто ставим сервера в полку и подключаем их через графический интерфейс. Помимо серийных серверов, возможно развертывание на виртуальные машины. Что можно добиться с помощью расширения?

- Увеличения объема для новых данных
- Увеличение максимальной рабочей нагрузки
- Повышение отказоустойчивости. Чем больше нод в кластере, тем меньше вероятность выхода кластера из строя из-за отказа, а следовательно, тем ближе мы к обеспечению доступности 24/7.

Периодически ноды нужно вынимать из кластера для обслуживания. Еще довольно распространенный кейс в крупных организациях — сервера сходят с гарантии и переходят из продуктивной в какую-нибудь тестовую среду. На их место встают новые, которые находятся на гарантии производителя. По итогам всех этих операций нужно выполнять ребалансировку. Это процесс, когда данные перераспределяются между нодами — соответственно перераспределяется рабочая нагрузка. Это требовательный к ресурсам процесс, и на кластерах с большим объемом данных он может сильно снизить производительность. Чтобы этого избежать, нужно выбрать сервисное окно — время, когда нагрузка минимальна, и в этом случае пользователи этого не заметят.

Проекции

Для понимания, как хранятся данные в Vertica, требуется разобраться с одним из основных понятий — проекцией.

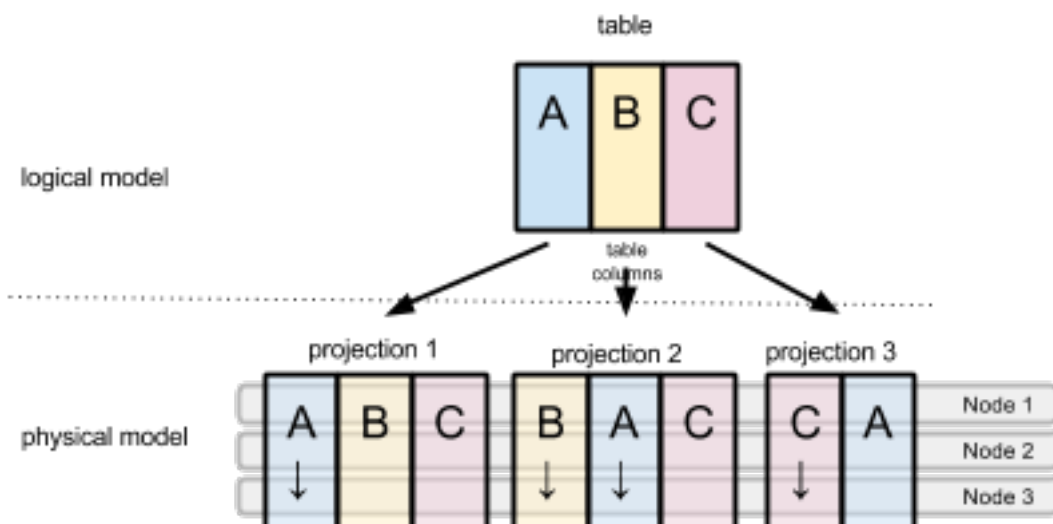
Логические единицы хранения информации — это схемы, таблицы и представления. Физические единицы — это проекции. Проекции бывают нескольких типов:

- суперпроекции (Superprojection),
- запрос-ориентированные проекции (Query-Specific Projections),
- агрегированные проекции (Aggregate Projections).

При создании любой таблицы автоматически создается суперпроекция, которая содержит все колонки нашей таблицы. Если нужно ускорить какой-то из регулярных процессов, мы можем создать специальную запрос-ориентированную проекцию, которая будет содержать, допустим, 3 столбца из 10.

Для ускорения предназначен и третий тип — агрегированные проекции. Не буду вдаваться в их подклассы — это не очень интересно. Сразу хочу предупредить, что постоянно решать свои проблемы с выполнением запросов через создание новых проекций не стоит. В конечном итоге кластер начнет тормозить.

Создавая проекции, нужно оценивать, хватает ли нашим запросам суперпроекций. Если мы все-таки хотим поэкспериментировать, добавляем строго по одной новой проекции. При возникновении проблем так будет проще найти первопричину. Для больших таблиц следует создавать сегментированную проекцию. Она разбивается на сегменты, которые распределяются по нескольким нодам, что повышает отказоустойчивость и минимизирует нагрузки на одну ноду. Если таблички маленькие, то лучше делать несегментированные проекции. Они полностью копируются на каждую ноду, и производительность таким образом увеличивается. Оговорюсь: в терминах Vertica «маленькая» таблица — это примерно 1 млн строк.

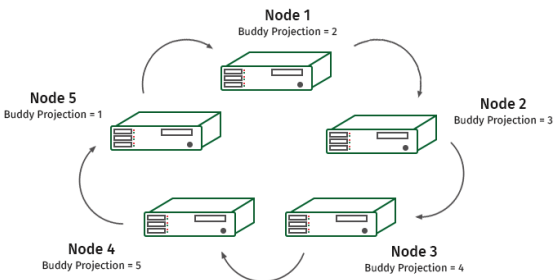
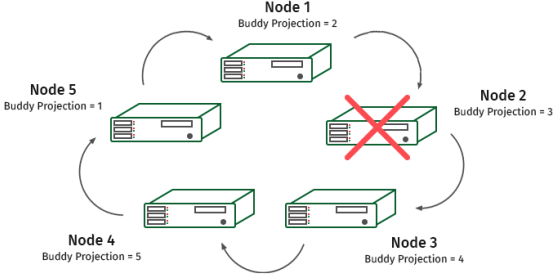
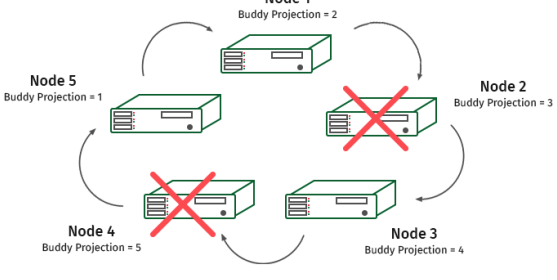
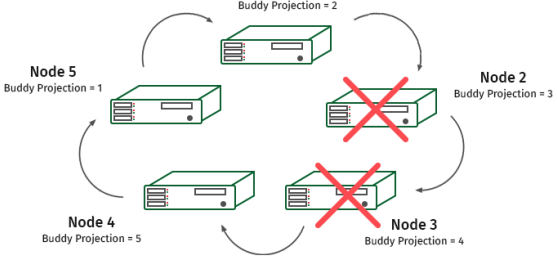


Отказоустойчивость

Отказоустойчивость в Vertica реализована при помощи механизма K-Safety. Он довольно простой с точки зрения описания, но сложный с точки зрения работы на уровне движка. Им можно управлять с помощью параметра K-Safety — он может иметь значение 0, 1 или 2. Этот параметр задает количество копий сегментированных проекционных данных.

Копии проекций называются buddy projections. Я попытался перевести это словосочетание через Яндекс-переводчик и получилось что-то вроде «проекции-кореша». Гугл предлагал варианты и интересней. Обычно данные проекции называют партнерскими или соседними, по их функциональному назначению. Это проекции, которые просто хранятся на соседних нодах и таким образом резервируются. У несегментированных проекций нет buddy projections — они копируются полностью.

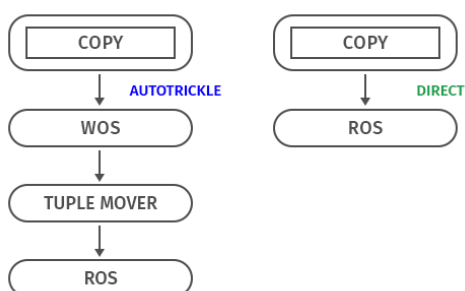
Как это работает?

<p>Рассмотрим кластер из пяти машин. Пусть K-safety у нас равен 1.</p> <p>Ноды пронумерованы, а под ними написаны партнерские проекции, которые хранятся на них.</p>	
<p>Предположим, у нас отключилась одна нода. Что будет?</p> <p>Нода 1 содержит дружественную проекцию ноды 2. Поэтому на ноде 1 подрастет нагрузка, но работать кластер не перестанет.</p>	
<p>А теперь такая ситуация:</p> <p>Нода 3 содержит проекцию ноды 4, ноды 1 и 3 будут перегружены.</p>	
<p>Усложняем задачу.</p> <p>K-Safety = 2, отключим две соседние ноды. Здесь будут перегружены ноды 1 и 4 (нода 2 содержит проекцию ноды 1, а нода 3 содержит проекцию ноды 4).</p>	

Логическое хранение данных

В Vertica есть области хранения данных, оптимизированные для записи, области, оптимизированные для чтения, и механизм Tuple Mover, который обеспечивает перетекание данных из первой области во вторую.

При использовании операции COPY, INSERT, UPDATE мы автоматически попадаем в WOS (Write Optimized Store) — область, где данные не оптимизированы для чтения и сортируются только при запросе, хранятся без сжатия или индексирования. Если объемы данных слишком велики для области WOS, то с помощью дополнительной инструкции DIRECT их стоит писать сразу в ROS. Иначе WOS будет переполнен, и у нас случится сбой.



По истечении заданного в настройках времени данные из WOS перетекают в ROS (Read Optimized Store) — оптимизированную, ориентированную на чтение структуру дискового хранилища. В ROS хранится основная часть данных, здесь она сортируется и сжимается. Данные в ROS разделены на контейнеры хранения. Контейнер представляет собой набор строк, созданных операторами трансляции (COPY DIRECT), и хранится в определенной группе файлов.

Вне зависимости от того, куда записаны данные — в WOS или в ROS — они доступны сразу. Но из WOS чтение идет медленнее, потому что данные там не сгруппированы.

Tuple Mover — это инструмент-уборщик, который выполняет две операции:

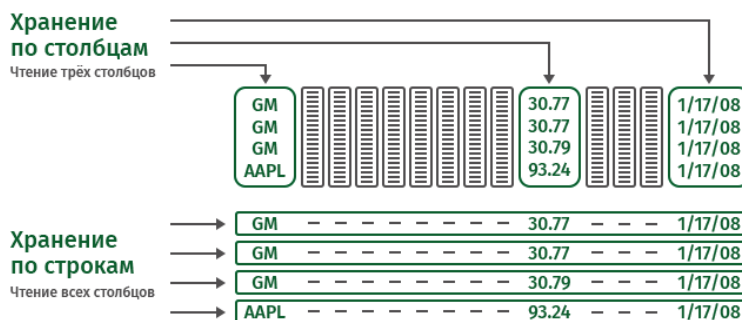
- Moveout — сжимает и сортирует данные в WOS, перемещает их в ROS и создает для них в ROS новые контейнеры.
- Mergeout — подметает за нами, когда мы используем DIRECT. Мы не всегда способны грузить столько информации, чтобы получались большие ROS-контейнеры. Поэтому он периодически объединяет небольшие контейнеры ROS в более крупные, очищает данные с пометкой на удаление, работая при этом в фоновом режиме (по времени, заданному в конфигурации).

В чем выгода колоночного хранения?

Если мы читаем строки, то, например, для выполнения команды

```
SELECT 1,11,15 FROM table1
```

нам придется читать всю таблицу. Это огромный объем информации. В данном случае колоночный подход выгоднее. Он позволяет считать только три нужных нам столбца, экономя память и время.



Что такое Parquet

Это бинарный, колоночно - ориентированный формат хранения данных, изначально созданный для экосистемы hadoop. Разработчики уверяют, что данный формат хранения идеален для big data (неизменяемых).



Parquet ведёт себя как неизменяемая таблица или БД. Значит для колонок определён тип, и если вдруг у вас комбинируется сложный тип данных (скажем, вложенный json) с простым (обычное строковое значение), то вся система разрушится. Например, возьмём два события и напомним их в формате Json:

```
{
  "event_name": "event 1",
  "value": "this is first value",
}

{
  "event_name": "event 2",
  "value": {"reason": "Ok"}
}
```

В parquet-файл записать их не получится, так как в первом случае у вас строка, а во втором сложный тип. Хуже, если система записывает входной поток данных в файл, скажем, каждый час. В первый час могут прийти события со строковыми value, а во второй — в виде сложного типа. В итоге, конечно, получится записать parquet файлы, но при операции merge schema вы наткнётесь на ошибку несовместимости типов.

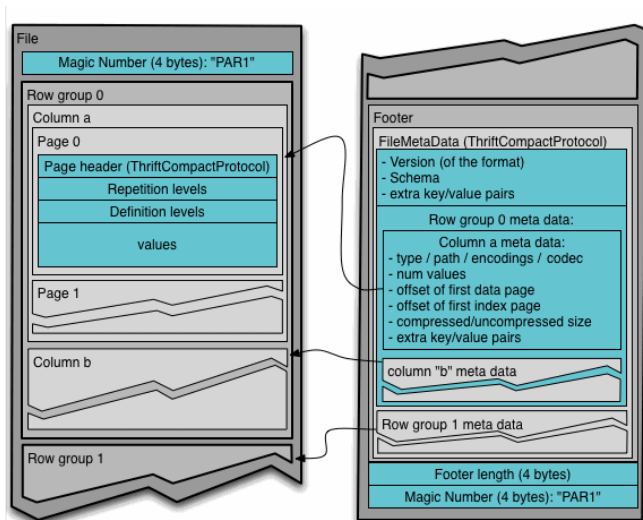
Как выглядит структура Parquet файлов

Parquet является довольно сложным форматом. Примечательно, что свои корни этот формат пустил даже в разработки Google, а именно в их проект под названием Dremel — об этом уже упоминалось на Хабре. Если коротко, Parquet использует архитектуру, основанную на “уровнях определения” (definition levels) и “уровнях повторения” (repetition levels), что позволяет довольно эффективно кодировать данные, а информация о схеме выносится в отдельные метаданные. При этом оптимально хранятся и пустые значения.

Файлы имеют несколько уровней разбиения на части, благодаря чему возможно довольно эффективное параллельное выполнение операций поверх них:

- **Row-group** — это разбиение, позволяющее параллельно работать с данными на уровне Map-Reduce
- **Column chunk** — разбиение на уровне колонок, позволяющее распределять IO операции
- **Page** — Разбиение колонок на страницы, позволяющее распределять работу по кодированию и сжатию

Если сохранить данные в parquet файл на диск, используя самую привычную нам файловую систему, вы обнаружите, что вместо файла создаётся директория, в которой содержится целая коллекция файлов. Часть из них — это метайнформация, в ней — схема, а также различная служебная информация, включая частичный индекс, позволяющий считывать только необходимые блоки данных при запросе. Остальные части, или партиции, это и есть наши Row group.



Для интуитивного понимания будем считать **Row groups** набором файлов, объединённых общей информацией. Кстати, это разбиение используется HDFS для реализации data locality, когда каждая нода в кластере может считывать те данные, которые непосредственно расположены у неё на диске. Более того, row group выступает единицей Map Reduce, и каждая map-reduce задача в Spark работает со своей row-group. Поэтому worker обязан поместить группу строк в память, и при настройке размера группы надо учитывать минимальный объём памяти, выделяемый на задачу на самой слабой ноде, иначе можно наткнуться на OOM.

Column chunk (разбиение на уровне колонок) — оптимизирует работу с диском (дисками). Если представить данные как таблицу, то они записываются не построчно, а по колонкам.

Представим таблицу:

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

Тогда в текстовом файле, скажем, csv мы бы хранили данные на диске примерно так:

A1	B1	C1	A2	B2	C2	A3	B3	C3
----	----	----	----	----	----	----	----	----

В случае с Parquet:

A1	A2	A3	B1	B2	B3	C1	C2	C3
----	----	----	----	----	----	----	----	----

Благодаря этому мы можем считывать только необходимые нам колонки.

Из всего многообразия колонок на деле аналитику в конкретный момент нужны лишь несколько, к тому же большинство колонок остается пустыми. Parquet в разы ускоряет

процесс работы с данными, более того — подобное структурирование информации упрощает сжатие и кодирование данных за счёт их однородности и похожести.

Каждая колонка делится на страницы (**Pages**), которые, в свою очередь, содержат метайнформацию и данные, закодированные по принципу архитектуры из проекта Dremel. За счёт этого достигается довольно эффективное и быстрое кодирование. Кроме того, на данном уровне производится сжатие (если оно настроено). На данный момент доступны кодеки *snappy*, *gzip*, *lzo*.

Есть ли подводные камни?

За счёт “паркетной” организации данных сложно настроить их стриминг — если передавать данные, то полностью всё группу. Также, если вы утратили метайнформацию или изменили контрольную сумму для Страницы данных, то вся страница будет потеряна (если для Column chunk — то chunk потерян, аналогично для row group). На каждом из уровней разбиения строятся контрольные суммы, так что можно отключить их вычисления на уровне файловой системы для улучшения производительности.

Достоинства хранения данных в Parquet:

- Несмотря на то, что они и созданы для hdfs, данные могут храниться и в других файловых системах, таких как GlusterFs или поверх NFS
- По сути это просто файлы, а значит с ними легко работать, перемещать, бэкапить и реплицировать.
- Колончатый вид позволяет значительно ускорить работу аналитика, если ему не нужны все колонки сразу.
- Нативная поддержка в Spark из коробки обеспечивает возможность просто взять и сохранить файл в любимое хранилище.
- Эффективное хранение с точки зрения занимаемого места.
- Как показывает практика, именно этот способ обеспечивает самую быструю работу на чтение по сравнению с использованием других файловых форматов.

Недостатки:

- Колончатый вид заставляет задумываться о схеме и типах данных.
- Кроме как в Spark, Parquet не всегда обладает нативной поддержкой в других продуктах.
- Не поддерживает изменение данных и эволюцию схемы. Конечно, Spark умеет мерджить схему, если у вас она меняется со временем (для этого надо указать специальную опцию при чтении), но, чтобы что-то изменить в уже существующим файле, нельзя обойтись без перезаписи, разве что можно добавить новую колонку.
- Не поддерживаются транзакции, так как это обычные файлы а не БД.

Распределенные вычисления. Hadoop

Hadoop не так уж сложен, ядро состоит из файловой системы HDFS и MapReduce фреймворка для обработки данных из этой файловой системы.

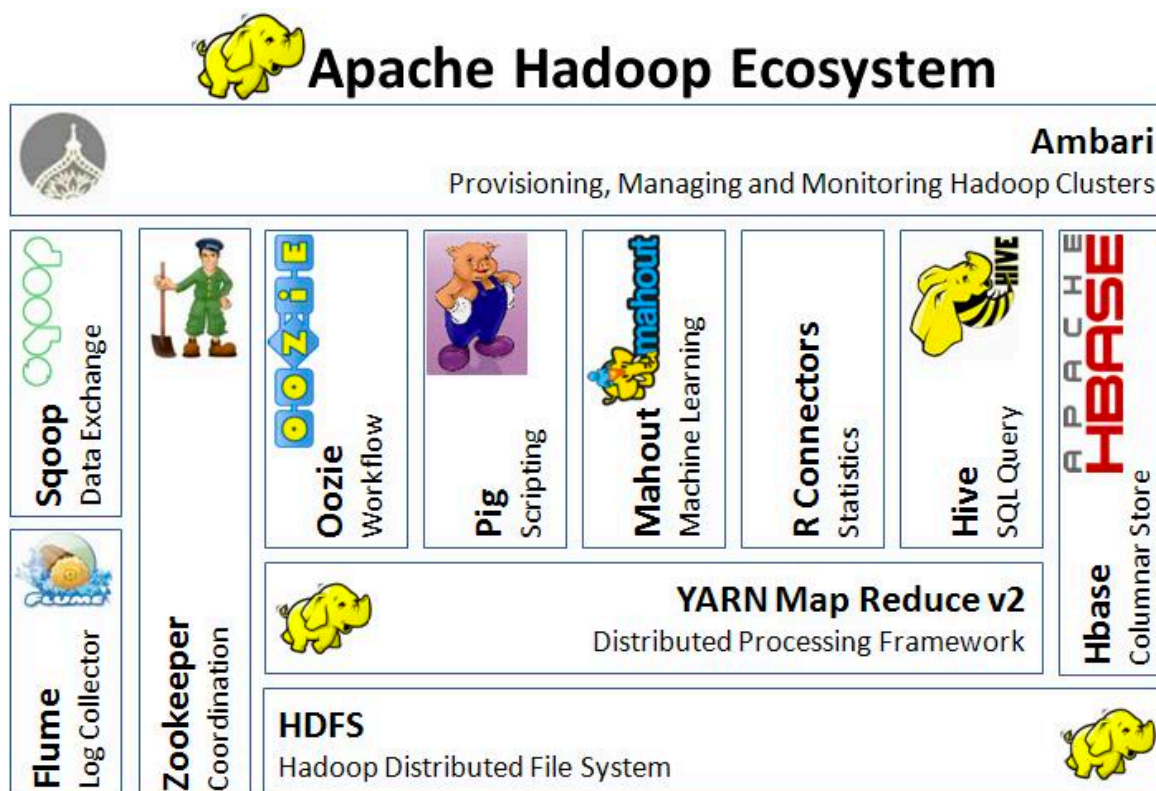
Hadoop следует использовать, если:

- Вычисления должны быть компонентными, другими словами, вы должны иметь возможность запустить вычисления на подмножестве данных, а затем слить результаты.
- Вы планируете обрабатывать большой объем неструктурированных данных — больше, чем можно уместить на одной машине (> нескольких терабайт данных).

Hadoop не следует использовать:

- Для некомпонуемых задач — например, для задач рекуррентных.
- Если весь объем данных уместается на одной машине. Существенно сэкономяте время и ресурсы.
- Hadoop в целом — система для пакетной обработки и не подходит для анализа в режиме реального времени (здесь на помощь приходит система Storm).

Архитектура HDFS и типичный Hadoop кластер



HDFS подобна другим традиционным файловым системам: файлы хранятся в виде блоков, существует маппинг между блоками и именами файлов, поддерживается древовидная структура, поддерживается модель доступа к файлам основанная на правах и т. п.

Отличия HDFS:

- Предназначена для хранения большого количества огромных (>10GB) файлов. Одним Следствие — большой размер блока по сравнению с другими файловыми системами (>64MB)
- Оптимизирована для поддержки потокового доступа к данным (high-streaming read), соответственно производительность операций произвольного чтения данных начинает хромать.
- Ориентирована на использование большого количество недорогих серверов. В частности, серверы используют JBOV структуру (Just a bunch of disk) вместо RAID — зеркалирование и репликация осуществляются на уровне кластера, а не на уровне отдельной машины.
- Многие традиционные проблемы распределенных систем заложены в дизайн — уже по дефолту все выход отдельных нод из строя является совершенно нормальной и естественной операцией, а не чем-то из ряда вон.

Hadoop-кластер состоит из нод трех типов: NameNode, Secondary NameNode, Datanode.

namenode — мозг системы. Как правило, одна нода на кластер (больше в случае Namenode Federation, но мы этот случай оставляем за бортом). Хранит в себе все метаданные системы — непосредственно маппинг между файлами и блоками. Если нода 1 то она же и является Single Point of Failure. Эта проблема решена во второй версии Hadoop с помощью Namenode Federation.

Secondary NameNode — 1 нода на кластер. Принято говорить, что «Secondary NameNode» — это одно из самых неудачных названий за всю историю программ. Действительно, Secondary NameNode не является репликой NameNode. Состояние файловой системы хранится непосредственно в файле fsimage и в лог файле edits, содержащим последние изменения файловой системы (похоже на лог транзакций в мире РСУБД). Работа Secondary NameNode заключается в периодическом мерже fsimage и edits — Secondary NameNode поддерживает размер edits в разумных пределах. Secondary NameNode необходима для быстрого ручного восстановления NameNode в случае выхода NameNode из строя.

В реальном кластере NameNode и Secondary NameNode — отдельные сервера, требовательные к памяти и к жесткому диску. А заявленное “commodity hardware” — уже случай DataNode.

DataNode — Таких нод в кластере очень много. Они хранят непосредственно блоки файлов. Нода регулярно отправляет NameNode свой статус (показывает, что еще жива) и ежечасно — репорт, информацию обо всех хранимых на этой ноде блоках. Это необходимо для поддержания нужного уровня репликации.

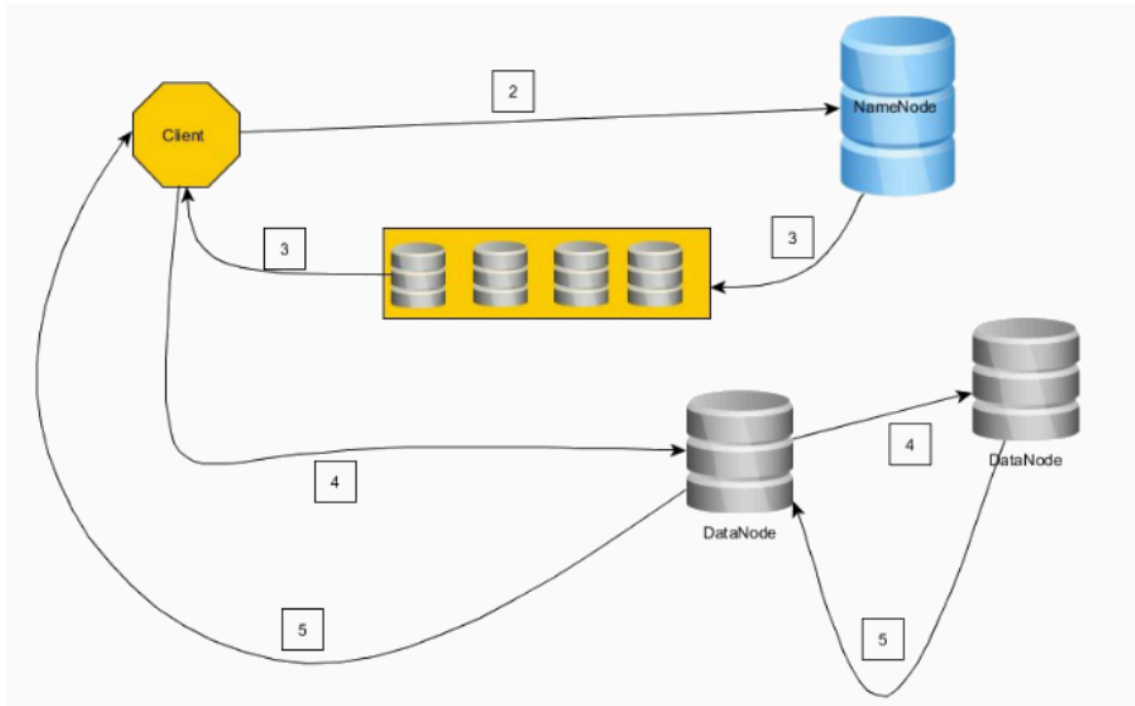
Посмотрим, как происходит запись данных в HDFS:

1. Клиент разрезает файл на цепочки блокового размера.
2. Клиент соединяется с NameNode и запрашивает операцию записи, присылая количество блоков и требуемый уровень репликации
3. NameNode отвечает цепочкой из DataNode.

4. Клиент соединяется с первой нодой из цепочки (если не получилось с первой, со второй и т. д. не получилось совсем — откат). Клиент делает запись первого блока на первую ноду, первая нода — на вторую и т. д.

5. По завершении записи в обратном порядке (4 -> 3, 3 -> 2, 2 -> 1, 1 -> клиенту) присылаются сообщения об успешной записи.

6. Как только клиент получит подтверждение успешной записи блока, он оповещает NameNode о записи блока, затем получает цепочку DataNode для записи второго блока и т.д.



Клиент продолжает записывать блоки, если сумеет записать успешно блок хотя бы на одну ноду, т. е. репликация будет работать по хорошо известному принципу «eventual», в дальнейшем NameNode обязуется компенсировать и таки достичь желаемого уровня репликации.

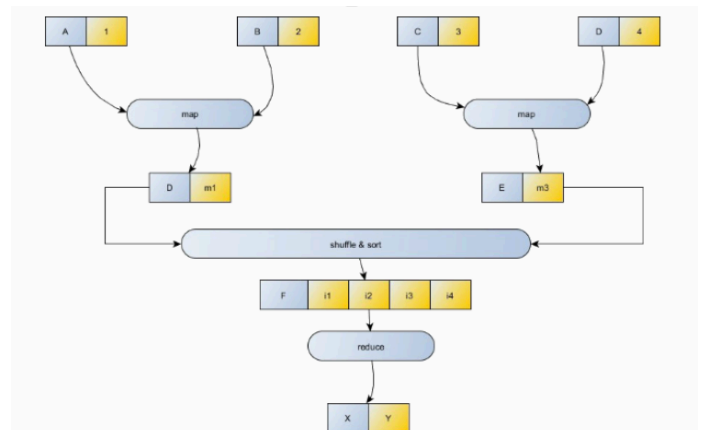
Завершая обзор HDFS и кластера, обратим внимание на еще одну замечательную особенность Hadoop'a — rack awareness. Кластер можно сконфигурировать так, чтобы NameNode имел представление, какие ноды на каких rack'ах находятся, тем самым обеспечив лучшую защиту от сбоев.

MapReduce

Единица работы job — набор map (параллельная обработка данных) и reduce (объединение выводов из map) задач. Map-задачи выполняют mapper'ы, reduce — reducer'ы. Job состоит минимум из одного mapper'a, reducer'ы опциональны. Здесь разобран вопрос разбиения задачи на map'ы и reduce'ы. Если слова «map» и «reduce» вам совсем непонятны, можно посмотреть классическую статью на эту тему.

Модель MapReduce

- Ввод/вывод данных происходит в виде пар (key, value)
- Используются две функции map: $(K1, V1) \rightarrow (K2, V2), (K3, V3), \dots$ — отображающая пару ключ-значение на некое множество промежуточных пар ключей и значений, а также reduce: $(K1, (V2, V3, V4, \dots, VN)) \rightarrow (K1, V1)$, отображающая некоторое множество значений, имеющих общий ключ на меньшее множество значений.
- Shuffle and sort нужна для сортировки ввода в reducer по ключу, другими словами, нет смысла отправлять значение $(K1, V1)$ и $(K1, V2)$ на два разных reducer'а. Они должны быть обработаны вместе.

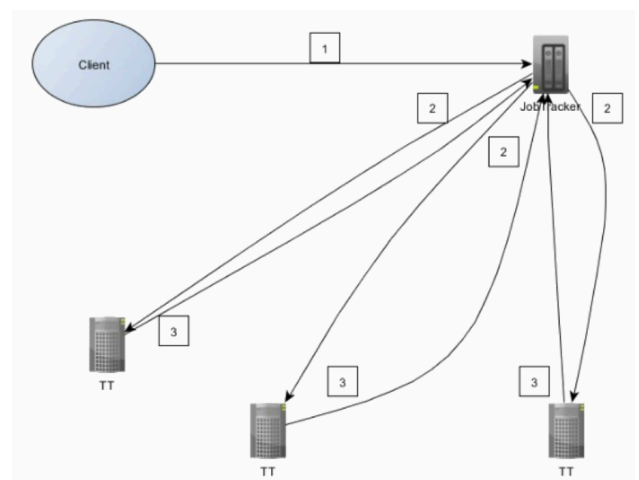


Посмотрим на архитектуру MapReduce

1. Для начала расширим представление о hadoop-кластере, добавив в кластер два новых элемента — JobTracker и TaskTracker. JobTracker непосредственно запросы от клиентов и управляет map/reduce задачами на TaskTracker'ах. JobTracker и NameNode разносится на разные машины, тогда как DataNode и TaskTracker находятся на одной машине.

Взаимодействие клиента и кластера выглядит следующим образом:

1. Клиент отправляет job на JobTracker. Job представляет из себя jar-файл.
2. JobTracker ищет TaskTracker'ы с учетом локальности данных, т.е. предпочитая те, которые уже хранят данные из HDFS. JobTracker назначает map и reduce задачи TaskTracker'ам
3. TaskTracker'ы отправляют отчет о выполнении работы JobTracker'у.



Неудачное выполнение задачи — ожидаемое поведение, провалившиеся таски автоматически перезапускаются на других машинах.

В Map/Reduce 2 (Apache YARN) больше не используется терминология «JobTracker/TaskTracker». JobTracker разделен на **ResourceManager** — управление ресурсами и **Application Master** — управление приложениями (одним из которых и является непосредственно MapReduce). MapReduce v2 использует новое API