Распределённые файловые системы для больших данных

Для того, чтобы обрабатывать большие данные, их нужно где-то хранить. Вертикальное масштабирование (scale up): покупаем всё бо'льшие диски. Ясно, что это не всегда возможно/целесообразно. Горизонтальное масштабирование (scale out): берём несколько серверов и объединяем в систему. Об этом и пойдёт речь в курсе.

Родоначальник – GFS (Google File System). Есть статья, где люди из Google объясняют, как они реализовали на имеющемся железе ФС, удовлетворяющую их нуждам для Больших Данных (просто распределённые ФС уже были, а вот для обработки Больших Данных не было).

Специфика BigData

- Большие файлы (>> 1 GB)
- Commodity hardware (обычные сервера, которые производятся и покупаются в больших количествах, и надёжностью особой не отличаются).
- "Write once, read many"
  - в основном — последовательные чтения
  - append
  - нет потребности в random write
  - не удалямем данные, а записываем новые (то есть, в серединку файла нельзя записать что-нибудь, придётся создать новый файл и удалить старый).

Эти требования связаны со спецификой Google: робот скачивает интернет, потом его обрабатывают, в основном – последовательно, в основном – чтения.

Архитектура

- Master server, хранящий список файлов и их метаданные
- Сами файлы разбиты на блоки(чанки). Блоки хранятся на Chunk servers: обычные сервера (commodity hardware) с обычной local Linux FS. A Master Server собирает это всё в единую ФС, раздаёт chunk-серверам команды, следит за работоспособностью. Запросы приходят на Master-сервер, а данные отдаются напрямую с chunk-серверов (иначе master-сервер будет бутылочным горлышком).
- Данные и мметаданные хранятся раздельно

GFS был в виде статьи и закрытой реализации от Google. Yahoo на тот момент написал HDFS (Hadoop Distributed File System) на Java и выкатил в OpenSource. Master server == Name node. Chunk server == Data node. chunk == block

TODO: скопипастить со слайда Роли компонент.

- Namenode – NN (master server)
  - хранит metadata в памяти (для быстрого доступа)
  - поддерживает список доступных Datanode
- Datanode – DN (chunk server)
  - хранит блоки (чанки) в локальной FS
  - оповещает Namenode о состоянии (heartbeat, раз в 3-5 секунд)
  - оповещает Namenode о списке своих блоков (примерно каждые полчаса)
  - получает инструкции от Namenode

Размер блока В обычных ФС обычно 4 Кб. Зачем вообще разбивать на блоки? Почти любой файл влезет на большой диск.

Если блок слишком большой, то:

1. Чем больше блок – тем больше фрагментация.

2. Хуже параллелизм.

Если блок слишком маленький, то:

1. Namenode хочет хранить всё в памяти. Каждый блок. Не влезет, и операции будут не очень быстрыми (если и влезет).
2. С каждым блоком хранится чексумма, и она создаёт накладные расходы.
3. Чаще всего читаем последовательно. Последовательное чтение быстро. Если блоки слишком маленькие, мы этим преимуществом не воспользуемся. Надо, чтобы время seek'а было мало по сравнению со временем последовательного чтения
4. Обращение к Datanode тоже занимает время.

Проблема: очень маленькие файлы (меньше размера блока). Физического ограничения, что меньше размера блока создавать файлы нельзя, нет. Поэтому мелкий файл будет занимать столько места на диске, сколько он и есть (+метаданные), но будет занимать отдельный блок на Namenode. Если таких мелких файлов очень много:

1. трата оперативной памяти Namenode и повышенная нагрузка на него
2. хуже скорость чтения

Вывод: лучше склеивать мелкие файлы.

Обычно размер блока 64 MB, 128 MB.

Отказоустойчивость

Решение – репликация.

Считаем, что места у нас много, поэтому можем не применять хитрых алгоритмов, а просто дублировать информацию.

Фактор репликации – число копий каждого блока. Обычно 3. Копии распределяются по нодам с учётом топологии сети.

Видео про дата-центр

Например, плохо дублировать файл на серверах одной и той же стойки, ибо у всей стойки может сломаться маршрутизатор.

Чтение из HDFS

TODO: Вставить картинку Namenode ещё проверяет права доступа, наличие такого файла, папки, и проч.

Запись в HDFS

Вставить картинку

Клиент передаёт данные на одну Datanode, на которую ему сказал писать Namenode. Потом datanode'ы между собой разбираются, как делать репликацию. После того, как репликация будет произведена, клиенту приходит подтверждение.

Если процессе записи datanode падает, ack пакет не приходит вовремя, клиентская библиотека сообщает об этом Namenode, Namenode выдаёт новый идентификатор блока (чтобы старый блок, если он частично записался, был помечен как мусор и удалён), и запись начинается снова. Если, например, данные были записаны на 2 ноды, а третья упала, Namenode даёт приказ продублировать с какой-то из уже записанных нод, а клиенту говорят, что запись завершена.

Заметим: общение между нодами происходит внутри кластера, где соединение довольно быстрое.

Недостатки:

1. нет random reads/writes
2. не подходит для low-latency приложений (например, key-value хранилища, для систем реального времени, для случаев, когда пользователю требуется быстрый ответ (например, открыть документ))

3. Проблема мелких файлов (картинки хранить не получится)
4. Namenode – единая точка отказа. Поэтому Namenode берётся дорогой и отказоустойчивой, питание, сеть и прочее дублируются, а сама она окружается теплом и заботой. Но лучше резервировать.

Secondary Namenode

TODO: вставить картинку

Сервер, хранящий реплику какого-то состояния Namenode. Namenode помимо непосредственных изменений памяти пишет лог, и периодически пишет слепки памяти (например, можно выключить её с сохранением состояния и включить потом). Но передавать слепки памяти накладно, поэтому при помощи какой-нибудь системы ротации логи пачками пересылаются на Secondary Namenode, где поддерживается такое же состояние (с точностью до последнего пакета). Secondary Namenode пишет свои слепки памяти периодически, и может их отправлять на Namenode, чтобы следующий старт был с более актуального состояния (и заодно Namenode не надо этим заниматься).

Если Namenode упала надолго, вручную вводится в строй Secondary Namenode как горячая замена (слегка отстающая, впрочем).

Если primary и secondary стоят физически рядом, можно просто подключать диск по NFS, чтобы автоматически все действия primary дублировались на secondary.

Нужно заметить, что эта система не очень приспособлена именно для горячей замены. Для этого есть другие.

Например, федерация Namenode: разбиваем на несколько Namenode, каждая из которых обслуживает свой раздел (напр., /home, /data). Вообще, в данном курсе такое не рассматривается.

Клиенты HDFS

- Java API
- CLI
- C (libhdfs)
- HTTP (WebHDFS)
- NFS
- FUSE (Filesystem in Userspace)
- Python (pip hdfs использует WebHDFS)

Command Line Interface

- hdfs dfs -<command> (то же, что hadoop fs -<command>)
- hdfs dfs –help
- hdfs dfs -ls <remote>
- hdfs dfs -put <local> <remote>
- hdfs dfs -get <remote> <local>
- hdfs dfs -mv <remote> <remote>
- hdfs dfs -cp <remote> <remote>*
- hdfs dfs -cat <remote>; hdfs dfs -text <remote>
- hadoop distcp hdfs://namenode1/file1 hdfs://namenode1/file2

Замечание: -cp работает через клиента, т. е. медленно. hadoop distcp производит копирование внутри кластера (или даже одной ноды).

Лайфхак: можно так составить конфигурационный файл, чтобы Hadoop считал Ваш компьютер удалённым сервером. Полезно для отладки.

MapReduce

Были действия,которые совершают над данными. Для каждого конкретного случая писали свой велосипед. Google же заметил сходство и написал универсальную реализацию. Можно заметить, что функции напоминают одноимённые функции высших порядков из функционального программирования.

1. map

   $map : (k, v) \to [(k', v')]$. На вход подаётся одна запись, на выходе 0 или более записей вида (ключ, значение).

   Поскольку функция применяется поэлементно к входным данным и может выполняться параллельно.

2. reduce $reduce : (k, [v]) \to [(k', v')]$.

   Принимает на вход все пары (ключ, значение) с фиксированным ключом. На выход подаёт 0 или более записей вида (ключ, значение).

Терминология:

- Job — задача на кластере
- Task — map и reduce стадии
  - Map — столько, сколько splits во входных данных
  - Reduce — столько, сколько указал программист
- Attempt – попытка выполнения конкретного task
  - Несколько попыток на каждый task в случае неудачи
  - Speculative execution — запуск несколько attempts одного task: кто быстрее (можно запретить)

Workflow

- На вход подаются файлы $file_1, \ldots, file_n$. JobTracker в зависимости от самих файлов и указанного количества редьюсеров определяет, как надо их разбить на Splits.
- Input Splitter берёт и разбивает файлы на сплиты.

  По умолчанию размер сплита берётся dfs.blocksize, округлённый как надо. Разделение на Splits не должно испортить формат файла. Например,

  - Текстовый файл дробить можно только по строчкам.
  - Архивы bz2 можно дробить только по сжатым блокам.
  - Архивы gz тоже состоят из сжатых блоков, но их границы неразличимы, и приходится за сплит брать весь файл.

    Обычно один сплит — одна Map задача.
- Record Reader читает записи из сплита и передаёт их функции Map.
- Map применяется. Результат пока хранится в памяти. Потом он сбрасывается (spill) на диск.

  Результат map может быть больше, чем входные данные. Где-то было написано, что выход map пишется в буфер, а когда буфер заполнится, он сбрасывается (spill) на диск. В конце каждый такой файл подаётся partitioner'y.
- Combiner (optional)
  Если после Map создаётся очень много записей, которые накладно передавать по сети, их можно отсортировать на той же машине, на которой выполнялся Map, и применить к ним Combiner, который должен уменьшить количество данных.

  В частности, если Reduce коммутативна и ассоциативна, можно использовать её и как Combiner.

  Важно: Combiner не должен менять типа входных данных, т. к. реальности он может применяться 0, 1 или более раз, то есть не факт, что строго один.

  Логика применения Combiner сложна, но можно сказать, что он применяется к результату map, хранящемуся в памяти, который предварительно сортируют.

В большой схеме написано, что эти spilled фрагменты позже сливаются в один файл, и к нему тоже применяется комбайнер.

- Partition
  Partitioner распределяет получившиеся записи по r редьюсерам. По умолчанию запись (k, v) уходит на $hash(k)\%r$ редьюсер. В частности, так записи с одинаковым ключом обязательно попадут на один редьсюсер.

  Тут важно, чтобы хеширование было более-менее равномерным: если много записей отправятся на один и тот же редьюсер, он будет работать долго.

- Shuffle
  Данные передаются по сети к своим редьюсерам.

- Merge
  Куски, пришедшие с разных мапперов, собираются вместе и сортируются с применением Sort Comparator, чтобы значения с одинаковыми ключами шли подряд.

  Существует техника Secondary Sort. Куски собираются вместе и сортируются с помощью Grouping Comparator. После для каждой группы сортировка запускается с Sort Comparator. Зачем это нужно — см. в билете про Join'ы.

- Reduce
  Ко всем записям применяется функция Reduce.

- Выход подаётся в OutputCollector, который потом пишет всё через RecordWriter в выходной файл данного Reduce.

Что касается терминологии: кажется, всё до Shuffle относят к map-стадии, а reduce и далее — к reduce-стадии.

Пример: Лог веб-сервера. Хотим посчитать количество уникальных посетителей, сколько раз каждый пользователь посетил сайт. Различать пользователей можно по uid, который есть в логе.

1. map:
   Ключ — по сути номер строки в файле, но он нам не нужен, мы его игнорируем.

   Выделяем из строки лога uid.

   $f : line- > (uid, NULL)$

2. shuffle&sort
   Сортируем по uid.

3. reduce В сортированном списке количество уникальных элементов считается просто: количество смен ключа при последовательном чтении. $g : uid, []_n \to uid, n$ $g : uniq - c$ – утилита unix.

```
cat access.log | ./get_uid | sort | uniq.c
```

Сравнение MPI и MapReduce

MPI: Общее хранилище на SAN(Storage Area Network): данные хранятся отдельно, обрабатываются отдельно, любой узел имеет доступ к данным в равной мере CPU-intensive (дефицитный ресурс — процессор).

Минусы MPI.

Плюс: Задачи решаются за прогнозируемое время.

... Вместо простой разбивки на блоки фиксированного размера вводится разбивка на сплиты: блоки, учитывающие внутреннее строение файла (например, для текстового файла – по строкам). Поэтому сплит может быть больше по размеру, чем блок.

При отказе оборудования запускаем ту же задачу на другом сервере. Желательно, на том, где данные уже есть.

14.11.17

Небольшие примеры

- grep (docid, content) → (docid, content'), content' только из тех строк, которые содержат искомую фразу
  На стадии map ищем нужную подстроку, reduce тождественный.

- group by: (key, record) → (key, [record])
  map тождественный, sort тогда просто сортирует по ключу, reduce просто собирает куски с одинаковым ключом (они будут идти подряд) в группы.
- word count map разбивает предложение на слова. Слово — ключ, значение — количество (то есть, 1). reduce собирает количество одинаковых слов (они тоже подряд, как в предыдущем примере). Можно применить combiner, такой же, как reducer.
- Комбайнеры:
  - reduce=sum
    combine=sum
  - reduce=avg
    Выдаём (в том числе тогда маппер надо поменять) сумму и количество. Тогда можно восстановить правильное среднее.
  - reduce=median
    Можем сжать повторяющиеся значения в пары (количество, ключ) Но комбайн плохо всё равно работает.
- In-Mapper Combiner
  Если входные данные достаточно специфичны, можем улучшить свой маппер.
  - Агрегация в каждой итерации в mapper
    Пример: text → [(word, 1)], агрегируем по wird: получаем [(word', n)], где все word' — уникальные
  - Агрегация для нескольких итераций в mapper
    Пример аналогичный.
  - Сортировка по количеству Надо просто поменять местами ключ и значение (количество встреч и само слово)
  - Обратный индекс. Input: (docid, content) Output: (term, [docid, ...]), где term — слово из content

    То есть подаются документы, и надо будет по слову возвращать названия всех документов, в которых оно встретилось.

    map: (docid, content) → [(term, docid), ...]

    reduce: [(term, docid), ...] → (term, [docid, ...])

    combine = reduce

    Заметим, что в [docid, ...] могут быть повторы.
- term frequency Input: (docid, content) Output: (term, [(docid, tf), ...]), где term — слово из content, tf — term frequency (сколько раз term встретился в данном документе)

  map: (docid, content) → [⟨(term, docid), 1⟩, ...] reduce: [⟨(term, docid), 1⟩, ...] → ⟨(term, docid), sum⟩

  То есть, за ключ хотим взять пару, но распределять по редьюсерам мы должны только по первому элементу! Это можно сделать, указав кастомный Partitioner.
- Join
  Объединить два датасета A и B по некоторому признаку.
  - Reduce-side Join

    Map добавляет в key метку tag: A или B. partitioner учитывает только key, без tag. Группировка тоже только по key, но сортировка по паре (key, tag). reduce реализует логику объединения.

    Но как отличить A от B на стадии map? В Java можно получить имя файла. В Streaming есть переменная окружения `mapreduce_map_input_file`.

    Это называется
  - Secondary Sort Сортировка по value на входе reduce.

    Как реализуется? Сортировка в памяти внутри reduce. value-to-key conversion: (key, value) → (⟨key, subkey⟩, value). Grouping Comparator — правило группировки ключей для передачи в Reduce — группирует по key. SortComparator — сортирует по (key, subkey).
  - Map-side join

    Если датасет B маленький, и можно поместить его в память на каждом маппере, то можно всё сделать в маппере. Редьюсер не нужен.

  - Distributed Cache Содержит файлы, необходимые для работы конкретному task. По одной копии на ноду, т. е. менять их нельзя. Файлы указываются в параметрах запуска:
    * files
    * archives (будут разархивированы)
    * libjars — будут добавлены в CLASSPATH
- Bucket side join Если входные данные подготовлены так, что датасет A разбит на диапазоны ключей, то map достаточно читать в память только те ключи из B, которые ему могут встретиться: для этого либо B должен быть заранее разбит на такие же диапазоны, либо мы читаем все, но оставляем только нужные.

  Если A и B — результаты MapReduce задачи, можно подготовить такое разбиение на диапазоны.

Hadoop Comparator. Можно задать свой компаратор (разумеется, только для ключей).

Леонов, Отказоустойчивые распределённые системы

Надо смириться с тем, что отказы в распределённых системах будут происходить, и планировать их.

Модель распределённой системы

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

— Leslie Lamport

Для дальнейшей работы нам нужна модель, на базе которой мы будем работать. Интуиция подсказывает, что в системе должны быть как-то представлены работающие процессоры и средство коммуникации. Осталось определиться со свойствами.

Существует две степени свободы:

- коммуникация: Shared memory vs Message Passing
  - Message Passing
    * Система состоит из вычислительных узлов, объединённых в сеть
    * Вычислительные узлы не имеют общей памяти и взаимодействуют только через передачу сообщений
  - Shared Memory
    * Процессоры взаимодействуют через доступ к общей памяти
- выполнение: Synchronous vs Asynchronous
  - Synchronous
    * Вычисления происходят тактами
    * Узлы имеют локальные монотонные часы, часы несогласованы, но есть ограничение на отклонение часов друг от друга
    * (message passing) Задержки передачи в сети конечные и укладываются в 1 логический такт
    * (shared memory) Все обращения одним процессором к общей памяти укладываются в 1 логический такт
  - Asynchronous
    * Узлы имеют локальные монотонные часы, часы несогласованы, отклонение между часами неограничено
    * (message passing) Задержки передачи в сети конечные, но неограниченные
    * (shared memory) Задержки доступа к общей памяти непредсказуемы, но конечны

Например, гарантию wait-free для AMP можно записать так:

Независимо от того, какие сообщения приходят тебе и доходят ли твои сообщения до других, ты за константное время (число твоих собственных шагов) совершаешь прогресс (прогресс — не получить или отправить сообщение (это лишь средство коммуникации), а, может, взять там лок какой-то, сделать запись в БД или что-то такое).

Утв. MP $\Leftrightarrow$ SM

► • MP ⇒ SM Будем использовать адресное пространство вычислительных узлов как общее адресное пространство. Операции store() и load() будем эмулировать через отправку сообщения соответствующему узлу ("прочитать такую-то ячейку" и "записать это в такую-то ячейку").

• SM ⇒ MP Разобъём общее адресное пространство на N непересекающихся множеств. В множестве i зарезервируем N буферов B_{ij} и на базе буферов реализуем SPSC-очереди. Отправку сообщения процессом i процессу j будем эмулировать через запись в очередь B_{ij}. Получение сообщения — через чтение из очереди B_{ij}.

◄

Утв. A ⇔ S

► • S ⇒ A Синхронная модель – особый случай асихронной. Т. о., эмуляция не требуется.

• A ⇒ S Для MP систем нам потребуется специальный примитив Synchronizer, который позволит эмулировать тактовость синхронной системы. Для SM систем мы просто можем использовать барьеры.

◄

Формальное определение: AMP

Система состоит из n процессов p_i, расположённых на n процессорах. C_{ij} — канал связи между процессами и m_{ij} — сообщение от p_i для p_j. Канал связи — просто множество, т. е. сообщение достаётся произвольное, порядок не гарантируется. Вычисления асинхронные, т.е. процесс в случайный момент времени выполняет внутренний шаг алгоритма или производит отправку/получение сообщения. Глобальных часов нет, отправка и получение сообщения неблокирующие.

Выполнение процесса p_i состоит из набора упорядоченных событий e_i^x. События бывают 3 типов: send(m), receive(m) или внутреннее событие.

Заданы три отношения порядка на множестве событий:

• Линейный порядок на событиях процесса p_i: $H_i = (\bigcup\limits_x e_i^x \to i)$.

• Отношение порядка $\xrightarrow[msg]{}$ между отправкой и получением сообщения: send(m) $\xrightarrow[msg]{}$ receive(m)

• Их замыканием будет отношение $H = (\bigcup e, \to)$ $e_i^x \to e_j^y \Leftrightarrow \begin{bmatrix} e_i^x \xrightarrow{i} e_j^y \\ e_i^x \xrightarrow[msg]{} e_j^y \\ \exists e_k^z : e_i^x \to e_k^z \wedge e_k^z \to e_j^y \end{bmatrix}$

Синхронизатор

Синхронизатор используется для эмуляции синхронной системы поверх асинхронной. При помощи синхронизатора будем эмулировать методы send(m), receive(m), а также генерацию событий начала нового такта.

Потребуем, чтобы в рамках одного эмулируемого такта процесс p_i отправлял процессу p_j всегда ровно одно сообщение:

1. Если эмулируемый процесс p_i не отправлят p_j ни одного сообщения, то будем генерировать пустое сообщение.
2. Если эмулируемый алгоритм p_i отправляет несколько сообщений p_j, то отправим одно сообщение — комбинацию исходных.

Номер такта:

• Синхронизатор начинает в состоянии такт 0.

• Когда синхронизатор получает сообщения от всех соседей, он переходит в такт x + 1

Утв. Простой синхронизатор корректен

► • Синхронизатор в такте x может получать сообщения от соседей находящихся только в тактах x или x + 1

Пусть получили сообщение от процессора в такте $x - 1$. Но мы могли перейти в такт x только получив все сообщения от процессоров в такте $x - 1$. Противоречие.

• Разница в тактах между любыми двумя синхронизаторами не больше 1.

Пусть есть синхронайзер в такте $x+2$ и синхронайзер в такте x. Тогда первый для перехода в такт $x+2$ должен был получить все сообщения от процессоров в такте $x+1$, но т. к. есть процессор в такте x, это невозможно. Противоречие. ◀

Зам. Всё описанное верно только для систем без отказов. Дальше мы увидим, что реализация синхронзатора при наличии отказов может быть невозможна. Т.о., синхронные системы имеют бóльшие возможности, чем асинхронные.

Модель отказов

Отказывать могут процессы или сеть. Мы ограничимся рассмотрением отказов процессов, значительную часть отказов сети можно свести к отказм процессов.

Будем называть систему f-failure tolerant, если при отказе до f процессов система продолжает удовлетворять заявленным свойствам.

- Доброкачественные (benign) отказы
  - Fail-stop: После наступления события e_i^x процесс p_i перестаёт выполнять какие-бы то ни было действия. Дополнительно, остальные процессы могут узнать об этом событии при помощи дополнительной абстракции.
  - Crash: После события e_i^x процесс p_i перестаёт выполнять любые действия. Механизма извещения других процессов не предусмотрено.
  - Omission: Работающий процесс можеть терять полученные сообщения до их обработки или забывать отправлять некоторые сообщения. Различают особые случаи: send omission и receive omission.
- Злокачественные (malign) отказы
  - Византийский (Byzantine) отказ: процессы с византийским отказом могут демонстрировать произвольное поведение. Чаще всего под этим понимается поведение злоумышленника, получившего полный контроль над одним из процессов и пытающегося навредить системе.
- Как имитировать потерю сообщений сетью через отказы процессов?

  Omission отказы симулируют потерю пакетов сетью. Византийские отказы симулируют изменение (искажение) пакетов и создание сообщений (никем не отправленных).
- Какая форма отказов наиболее частая?

  Из теоремы ниже следует, что византийские отказы наиболее часты, т. к. все остальные являются их частным случаем. Но если смотреть именно без вложений, то, скорее всего, Crash или Omission.
- Назовите пример системы, устойчивой к византийскому поведению.

  Bitcoin. Кажется, что если бы она была неустойчива, то кто-нибудь бы её таки взломал и разбогател.

Утв. В приведённой классификации одни отказы являются частным случаем других: Fail-stop $\subset$ Crash $\subset$ (Send omission | Receive omission) $\subset$ General omission $\subset$ Byzantine

▶
- Fail-stop отказы являются частным случаем Crash отказов, т.к. подразумевают наличие абстракции нотификации об отказе
- Crash является частным случаем General omission отказов, т.к. может быть описан потерей всех сообщений
- Потеря сообщений входит в понятие «произвольное поведение», поэтому Omission отказы являются частным случаем византийских.

◀

Консенсус

Постановка задачи:

- Процессы
  - Отказы В системе из n процессов разрешим отказывать не более чем f процессам Запретим процессам подниматься после падения. Если мы, например, докажем невозможность чего-нибудь, то скорее всего, это будет верно и в системе, где процессы могут возобновлять работу.
  - Тип отказов Будем разрешать только один тип отказа (например, crash)

  - Процесс может упасть в середине логического шага и (например) отправить часть сообщений из планируемых
  - В асинхронной системе невозможно понять, сломался ли получатель сообщения, или оно ещё не доставлено
- Сеть
  - Сеть надёжна, т.е. не теряет сообщения
  - Между любой парой процессов есть связанность
- Консенсус
  - Значением консенсуса будем считать boolean или любой другой тип

Система состоит из N процессов, и не более f процессов могут испытать отказ класса ψ. Каждый процесс получает на вход значение, все исправные процессы должны выбрать одно число из предложенных, удовлетворяя трём свойствам:

- **Agreement** Все исправные процессы выбирают одно и то же число
- **Validity** Процессы выбирают число из предложенных
- **Termination** Каждый процесс рано или поздно примет решение

В случае отсутствия отказов задача решается тривиально:

- Процесс рассылает всем своё значение
- Когда процесс знает значения всех остальных процессов, он принимает решение
- Решение приниматеся при помощи заранее известной функции, например min(values) или majority(values)

Разрешаем f отказов.

- Процесс рассылает всем своё значение.
- Когда процесс получит n-f сообщений, выбирает f из n-f чисел детерминированной функцией, например, минимальных.
- И среди этих чисел хотя бы одно будет общим.

Эквивалентные определения консенсуса

The Byzantine agreement problem/Terminating Reliable Broadcast

Есть выделенный процесс G, у которого есть начальное значение I. И есть n процессов p_i.

- Agreement Все исправные процессы решают одно и то же значение
- Validity Если G исправен, то все исправные значения решают значение I
- Termination Каждый исправный процесс рано или поздно примет какое-то решение

The interactive consistency problem

Есть N процессов, каждый из которых имеет начальное значение $\vec{V}$. Все исправные процессы должны выбрать вектор значений $\vec{V}$ такой,что:

- Agreement Все исправные процессы решают один и тот же вектор $\vec{V}$
- Validity Если процесс i — исправен, то $\vec{V}[i] = v_i$, или любое другое число в случае отказа p_i
- Termination Каждый исправный процесс рано или поздно примет решение

Разрешимость консенсуса

Пусть n — общее число процессов, и допускается падение f из них. Тогда условия на разрешимость консенсуса:

тип отказа	SMP, SSM	AMP, ASM
без отказов	разрешим	разрешим
Crash	разрешим при f <= n - 1	не разрешим
Byzantine	разрешим при f <= (n - 1) / 3	не разрешим

Отказоустойчивые распределённые системы

Объяснения шуток

- Что смешного в фразе "Руководство компании™ демонстрирует византийскую форму отказа"?
  Руководство компании принимает спонтанные решения.
- Почему не смешна фраза "Алексей демонстрирует признаки византийского отказа"?
  Алексей один, ему некому вредить.
- Синхронные вычисления требуют глобальных часов для управления старта тактами?
- Приведите пример SSM (Synchronous shared memory) системы
  SSM — та модель, которую используют при написании многопоточных программ. Точки синхронизации — барьеры — создают общие для всех потоков часы.
- Приведите пример SMP (Synchronous message passing) системы Распределённая база данных с синхронизатором. Синхронизатор берёт всё асинхронное на себя, а остаётся синхронная часть.

Теорема Фишера, Линч и Патерсона

Teop FLP impossibility. В асинхронной системе, построенной на передаче сообщений (AMP), не существует алгоритма консенсуса, допускающего Crash хотя бы одного процесса.

Зам. • Мы допускаем отказ типа Crash, т.е. докажем невозможность для любого более широкого класса отказов

- Фактически мы докажем, что ни один процесс не примет решения. Это более сильное утверждение, теорема будет из него следовать.
- Если разрешать только отказы типа Fail-Stop, то консенсус можно решить при некотором числе упавших процессов.
- В синхронной системе за такт сообщение обязано прийти, так что если этого не произошло, то мы гарантированно детектируем отказ. То есть по предыдущему пункту, консенсус можно построить.

Пререквизиты

- Будем считать, что процессы выбирают из множества $\{0, 1\}$.
- Будем считать, что сеть надёжна. Т. е., все сообщения будут рано или поздно доставлены исправным процессам.

CAP-теорема

Постановка

Рассмотрим распределённую систему, реализующую, например, базу данных, и потребуем от неё 3 свойства:

- Consistency
  Под консистентностью будем понимать линеаризуемость. Распределённая структура данных должна быть линеаризуемой.
- Availability
  Доступность означает, что любой запрос, полученный исправным процессом, должен быть обработан. Ограничений на время обработки нет, но система должна переживать серьёзные отказы сети.
- Partition tolerance
  Устойчивость к сетевому разделению подразумевает, что произвольное число сообщений может теряться. Или процессы могут разделиться на две группы, между которыми не будет связанности.

Teop. В асинхронной распределённой системе невозможно имплементировать базу данных, удовлетворяющую CAP свойствам.

▶ • Пусть это не так, и в системе есть хотя бы 2 процесса.

- Разобъём процессы на две непересекающиеся группы $\{G_1, G_2\}$.
- Пусть v_0 — начальное состояние базы.

- Пусть все узлы связны.
  - Рассмотрим выполнение α_1, в котором была единственная запись в группе $\{G_1\}$, и пусть мы записали v_1.
  - Рассмотрим выполнение α_2, в котором было единственное чтение в группе $\{G_2\}$, и оно могло вернуть только v_0.
- Потребуем потери сообщений между $\{G_1, G_2\}$ и рассмотрим выполнение $\alpha = \alpha_1\alpha_2$. Для процессов G_2 выполнение α неотличимо от α_2, т. е. чтение по-прежнему возвращает v_0.
- Выполнение α нарушает свойство Consistency, т. к. история этого выполнения нелинеаризуема: чтение α_2 не увидело результат записи α_1, то есть эти вызовы не упорядочены.

◀

Упражнения

Придумать системы, удовлетворяющие двум свойствам.

- CP

  Экстремальный вариант: Не реагируем на записи, всегда возвращаем начальное состояние.

  Нормальный вариант: Храним реплики в каждом процессе. Пробуем прочитать большинство значений в других процессах. Если не получилось достучаться до большинства, то чтобы гарантировать Partition Tolerance говорим, что операция не успешна (можно, так как нет Availability). Если получилось, то возвращаем самое частое значение. Для записи аналогично: пытаемся записать в большинство, если не получилось, то сообщаем о провале операции.

- AP

  Будем держать локальную базу данных и не общаться. Каждый поток возвращает своё значение. Будет доступность и устойчивость к партицированию, но consistency не будет.

- CA

  Выделяем сервер, и пусть все узлы обращаются к нему как клиенты. Будет consistency и availability, но если не можем соединиться с сервером, всё плохо.

Критика CAP-теоремы

Свойства, требуемые в теореме, слишком сильны. Линеаризуемость вообще самое крутое, что можно потребовать от многопоточной (многопроцессорной) структуры данных.

CAP теорема подразумевает модель, слишком пессимизирующую реальные сети.

Оказывается, ослабив требования, можно решить задачу. Например, задача t-Connected Consistency.

Разница между FLP и CAP

Решение задачи CAP требует, чтобы любой исправный процесс правильно обслуживал запросы даже в случае, когда он не получает никаких сообщений. «Упавший» узел в FLP не должен достигать консенсуса, так как на него все забили, а в CAP требуется, чтобы отделенный процесс (по сути — «упавший») все равно должен каким-то образом поддерживать актуальное состояние всей системы.

ACID

Свойства баз данных

Базы данных (не обязательно распределённые) обычно имеют более одного клиента. Так или иначе, эти клиенты конкурируют за доступ к базе. Клиент работает с базой данных, выполняя операции, операции группируются в транзакции. База данных должна давать клиентам гарантии, описывающие, как они будут влиять друг на друга.

Наиболее популярным из стандартов для баз данных стал ACID, определяющий 4 свойства БД.

- Atomicity — Атомарность
  Либо все операции транзакции когда-нибудь будут применены, либо ни одна операция транзакции не будет применена. Т.е. отказы, потеря питания или ошибки выполнения не должны на это влиять.
  - Когда другие клиенты увидят результат транзакции?
  - Увижу ли я результат своей транзацкии сразу?
  - Будет ли порядок для разных клиентов одинаковым?
- Consistency — Согласованность
  Гарантирует, что результатом транзакции будет корректное состояние базы данных.
  - Под корректным состоянием тут понимается условие соблюдения правил базы данных, например, требования целостности, триггеров или каскадных откатов транзакций. Consistency из ACID совсем не то же, что в CAP.
- Isolation — Изолированность
  Гарантирует, что конкурентное выполнение транзакций приведёт базу данных в состояние, идентичное последовательному их применению.
  - Когда записи транзакции увидят другие клиенты?
  - Увижу ли я собственные незакоммиченные записи?
  - Различают разные степени изоляции, т.к. это одно из наиболее тяжёлых свойств

1. Serializable
2. Repeatable reads
3. Read committed
4. Read uncommitted
5. Durability — Устойчивость
   Гарантирует, что применённая транзакция останется в истории базы данных навсегда.

Степени изоляции транзакций

Уровни изоляции принято описывать через наборы возможных влияний параллельных транзакций.

	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	возможно	возможно	возможно
Read Committed	нет	возможно	возможно
Repeatable Read	нет	нет	возможно
Serializable	нет	нет	нет

Dirty reads

Чтение может вернуть изменения, вызванные транзакцией, которая впоследствии не подтвердится.

Transaction 1 Transaction 2

```
/* Query 1 */
SELECT age FROM users WHERE id = 1;
/* will read 20 */
                      /* Query 2 */
                      UPDATE users SET age = 21 WHERE id = 1;
                      /* No commit here and ROLLBACK later*/
/* Query 1 */
SELECT age FROM users WHERE id = 1;
/* will read 21 */
```

Non-repeatable reads

Повторное чтение может вернуть другое состояние строки.

Transaction 1 Transaction 2

```
/* Query 1 */
SELECT * FROM users WHERE id = 1; /* прочитает одно значение */
                        /* Query 2 */
                        UPDATE users SET age = 21 WHERE id = 1;
                        COMMIT;
/* Query 1 */
SELECT * FROM users WHERE id = 1; /* прочитает другое значение в той же транзакции */
COMMIT;
```

Phantom reads

Два идентичных запроса возвращают разный набор строк.

Transaction 1 Transaction 2

```
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30; /* Вернет одни строки. */
                        /* Query 2 */
                        INSERT INTO users(id,name,age)
                        VALUES ( 3, 'Bob', 27 );
                        COMMIT;
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30; /* Вернет еще и новую строку в той же транзакции. */
COMMIT;
```

CP системы

1. Master/Slave — традиционная терминология
2. Leader/Follower — политкорректная терминология
3. Primary/Secondary — из мира БД; на Secondary находится не очень свежая копия Primary

1. Master всегда один
2. Процессы могут падать (crash) и восстанавливаться (recovery)
3. Синхронизация значений (например, нода вернулась, и нужно синхронизировать данные на ней)
4. инкрементальные обновления (транзакции упорядочены, и их нужно применять в том порядке, в котором они пришли)
5. Идемпотентность (повторное применение одной и той же транзакции ничего не поменяет; если мы вдруг думаем, что транзакция могла не примениться, применяем её ещё раз для подстраховки, и уверены, что плохо не станет).
6. Несколько транзакций одновременно.
7. FIFO для транзакций (от мастера к фолловерам) и FIFO для клиентов (от клиента к мастеру)

Производительность:

- Latency — задержка, которую имеет система.
- Throughput — количество операций, которые она может одновременно выполнять.

Например, для сетей (вспомним):

- Throughput — количество байт, которые можно в пике прососать.
- Latency — задержка между отправкой и получением подтверждения.

У нас же, применительно к транзакциям:

- Throughput — количество транзакций, которые можно обрабатываать одновременно.
- Latency — время обработки одной транзакции.

ZAB(ZooKeeper Atomic Broadcast)

Есть Master, и есть Follower. Как Master'у отдавать приказы Follower'ам? Через ZAB. ZAB реплицирует тразакции.

Гарантии:

1. инкрементальные (т. е. транзакции зависимы, и ZAB должен зависимостям удовлетворять)

2. инкрементальный префикс.

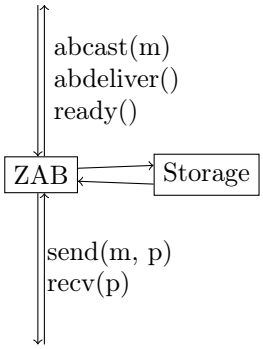
3. Доставка at least once (можно больше, т. к. они идемпотентные).

Главный процесс ZAB состоит из двух модулей. Мастера нужно периодически выбирать, и ещё в любой момент должно быть можно узнать, кто текущий мастер.

1. Leader oracle (знает, есть ли master и где его найти)

2. Leader election (руководит выборами нового master'а)

С сетью они не общаются напрямую, а обращаются к ZAB. Он реализует им абстракцию сети, давая повышенные гарантии, и только под ним находится настоящая сеть. ZooKeeper реализует стек протоколов/технологий (как TCP).



Методы ZAB

- abcast(m) — метод — отправка сообщения всем нодам
- abdeliver() — callback — через этот метод ZAB доставляет сообщение программе
- ready() — callback
- send(m, p)
- recv(p)

Журнал

ZAB хранит журнал.

В журнал записываются транзакции.

Транзакция $t = \langle v, z \rangle$ (value, индекс), где $z = \langle e, c \rangle$ (epoch, counter). Будем писать $c = count(z), e = epoch(z)$. Эпоха связана со сменой leader'а. counter — просто монотонный внутри эпохи счётчик. Получается, сам index тоже монотонный, и даёт линейный порядок.

Q — кворум, множество (или количество) процессов, на которые нужно записать данные, чтобы запись считалась успешной. Ждать только одного бессмысленно (он может упасть), ждать всех — тоже (кто-то мог упасть, а мы не узнаем и будем его ждать).

Пусть собрали Q_1, крашнулись, восстановились, собрали Q_2. Если $Q_1 \cap Q_2 \neq \emptyset$, мы гарантированно увидим изменения. Значит, кворум, если $|Q| > \frac{n}{2}$ процессов.

ZAB на Master по сути две службы:

1. Query-Agent, на него приходят записи, делает abcast.
   По сути ему надо просто проверить предикат (если когда делаем set, указываем версию, то надо проверить, что она совпадает). Далее он делает abcast, и все Slave когда-то получат abdeliver.

2. Data, на него приходят чтения, делает abdeliver.

ZAB на Slave ещё проще: там только Data, и он делает только abdeliver.

Надо заботиться о том, чтобы abdeliver на разных репликах делал транзакции в одном и том же порядке.

Наши требования к ZAB (Consistency):

1. Integrity
   если какой-то процесс делает $abdeliver(< v, z >)$, то $\exists$ процесс, который сделал $abcast(< v, z >)$ (т. е. сообщения не берутся из воздуха).

2. Total Order
   Пусть $\Pi = p_1, \ldots, p_n$ — множество процессов (считаем, что фиксировано).

Зам. Пусть $\rho_1, \ldots, \rho_e, \rho_{e+1}, \ldots$ — лидеры эпох $1, \ldots, e, e+1, \ldots$. При этом один и тот же процесс мог быть лидером в разные эпохи: $\rho_i = p_j \wedge \rho_k = p_j$. Тогда говорим $\rho_e < \rho_{e'}$, если $e < e'$. То есть, если бы сравнивали по номерам процессов, могло быть плохо, а если сравниваем по номерам эпох, то всё хорошо.

Пусть $\exists p_i$, который сделал $abdeliver(v, z), abdeliver(v', z'), \langle v, z\rangle < \langle v', z'\rangle$. Пусть $\exists p_j$: сделал $abdeliver(\langle v', z'\rangle) \Rightarrow$ p_j сделал $abdeliver(< v, z >)$, причём раньше $abdeliver(< v', z' >)$.

3. Agreement
   Если $\begin{cases} p_i \text{ сделал } abdeliver(< v, z >) \\ p_j \text{ сделал } abdeliver(< v', z' >) \end{cases}$,
   то $\begin{bmatrix} p_i \text{ сделал } abdeliver(< v', z' >) \\ p_j \text{ сделал } abdeliver(< v, z >) \end{bmatrix}$.

Зам. Из TO следует, что если один поток получает оба сообщения, то другой поток тоже долен получить оба.

То есть, наборы сообщений, доставляемые разным потокам, одинаковы.

4.

(a) Local order. Пусть была эпоха e, $abcast(v, \langle e, c_1\rangle)$ сделан перед $abcast(v', \langle e, c_2\rangle)$. Если $\exists p_i$, который сделал $abdeliver(v', < e, c_2 >)$, то p_i сделал $abdeliver(v, < e, c_1 >)$.

To есть, в пределах одной эпохи *deliver* происходят в порядке *broadcast* от Master'a.

(b) Global Order. Хотим сохранить свойство из Local Order при смене эпох.

Зам. Попробуем определить так: Пусть $abcast(v, \langle e, c\rangle)$ перед $abcast(v', \langle e', c'\rangle)$, $e \neq e'$. Если был $abdeliver(v', \langle e', c'\rangle)$, то был $abdeliver(v, \langle e, c\rangle)$. Пусть у нас текущий master сделал *abcast* и сразу умер. Новый leader сделает другой *abcast*, а предыдущий не будет виден.

$\exists p_i$, который сделал $\begin{cases} abdeliver(v', < e', c' >) \\ abdeliver(v, < e, c >) \end{cases}$ $\Rightarrow$ если процесс p_k доставил оба, то $abdeliver(v, < e, c >)$ предшествует $abdeliver(v', < e,' c' >)$.

5. Консистентность префикса
   TODO: вставить картинку с 22:39.

   Пусть I_e — префикс транзакций от начала до конца эпохи e, тогда $abdeliver(\langle v, z\rangle)$ предшествует $abcast(\langle v', z'\rangle)$, если $epoch(z) < epoch(z')$.

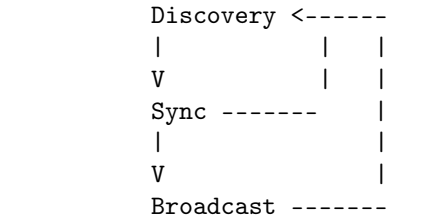
Зам. У нас из курса параллелочек есть Happens Before — транзитивное замыкание Program Order и synchronizes-with. Оно логично переносится на модель Message Passing и называется Causal Order. Почему просто не использовать его?

У нас есть отказы, для которых мы сделали эпохи и лидеров. Но эпохи ломают HB. Пусть поток был лидером и сделал *bcast*, но он никому не дошёл, была смена лидера, потом лидером был кто-то другой, а потом опять первый поток. Тот первый *bcast* потерялся.

См. 00094.MTS 16:34

Local Order + Global Order = Primary Order $\neq$ Causal Order

Алгоритм



Будем считать, что Leader Election и Leader Oracle у нас уже как-то реализованы, и к ним можно просто обращаться.

Обозначения:

- $f.p$ — последняя эпоха, на которую согласился follower
- $f.a$ — последний лидер
- $h(f)$ — история (журнал)
- $f.ZxID = \langle e, c \rangle$ — номер последней транзакции (у нас это пара, в реализации оно записывается в одно число)
- Q — кворум
- I_e — инкрементальный префикс e
- β_e — транзакции эпохи e

Discovery

- F.1
  - `CEPOCH(e)` -> Leader
- L.2
  - Лидер пытается понять, что творится на кластере после предыдущего лидера.
  - Собирает Q (включая себя туда), узнаёт, какие эпохи у фолловеров.
  - Отправляет `NEWEPOCH(`e'`)`, где $e' >$ номеров эпохи всех нод, у которых он узнал.
  - Если кворум не набирается, лидер ждёт до таймаута, снимает с себя полномочия и уходит в Discovery.
- F.3 Получают `NEWEPOCH(`e'`)`.
  - $e' < e \Rightarrow$ мы получили сообщение от какого-то левого процесса, который вряд ли может быть лидером. Уходим обратно в DISCOVERY.
  - $e' = e$:
    - f.p = e
    - Отправляет `ACK-E(`h_f`)`
    - Переходит в `SYNC`
- L.4
  - Собирает Q для `ACK-E(`h_{f_i}`)`. Эти два кворума Q должны быть одинаковые, то есть мы ждём во второй раз все те процессы, которые мы набрали в первый.
  - Не набирается $\Rightarrow$ складывает с себя полномочия и уходит.
  - Строит $I_e = \{z \in h_{f_I} \mid z' \leqslant max(z(h_{f_i}))\}$ (то есть выбирает фолловера с самой длинной историей и начинает отсчёт с неё (берёт всё, что было раньше её конца)).
  - Переходит в `SYNC`

SYNC

- L.1
  - Отправляет `NEWLEADER`$(e', I_{e'})$ в Q.
- F.2
  - atomic{
    $f.a = \rho_{e'}$
    $h_f = I_{e'}$ — отправляет просто весь журнал, в реальности можно отправлять только отсутствующую часть.
    }
  - Отправляет `ACK-LD` — согласие на лидерство L.
- L.3
  - Собирает кворум Q из тех, кто прислал `ACK-LD`.
  - Отправляет `COMMIT` в Q.
- F.4
  - Получает `COMMIT`.
  - Делает $abdeliver(\langle v, z\rangle)$ в порядке z для $\langle v, z\rangle$, для которых ещё не было $abdeliver$
  - В этот момент лидер становится полноценным лидером и начинает принимать и обрабатывать запросы.

BROADCAST

- L.1
  - Тот процесс p_i, который считает себя лидером, делает $ready(e')$ и сообщает Query-Agent, что он уже может давать новые пары $\langle v, z\rangle$.
- F.2
  - `COMMIT`(v, z)
  - делает $abdeliver(v', z')$ для таких z, что $z' \leqslant z$ и не было $abdeliver(v, z')$
- L.3
  - Отправляет $abcast(\langle v, z\rangle)$
  - Делает $abdeliver(z')$ для z', т. ч. $z' \leqslant z$ и не было $abdeliver(z')$.
- L.4
  - Если какой-то процесс p_j ожил и захотел вернуться:
    - Лидер получает от p_j `CEPOCH()`
    - Отправляет ему `NEWEPOCH`(e') и `NEWLEADER`$(I_e + \beta_{e'})$.
    - Когда получит от p_j `ACK-LD`, добавит его в кворум: $Q = Q \cap \{p_j\}$
  - Если какой-то процесс p_j умер:
    - Удаляем его из кворума.
    - Если кворум не набирается, складываем с себя полномочия и уходим в DISCOVERY.

Детали реализации

Иногда выделяют выборы лидера (Leader Election) в фазу 0. Чтобы снизить объём передаваемых по сети данных, можно всегда выбирать Leader с самой длинной историей, чтобы не отправлять это всё по сети.

Можем потерять что-то в журнале: Пусть лидер был выбран, получил какие-то данные, но упал, не успев их отправить. Был выбран новый лидер, а потом старый вернулся, попытался подключиться, и он должен выкинуть те записи, которые есть только у него.

Вместо `NEWLEADER` отправляется три разных вида сообщений: SNAP — узел очень отстал, поэтому не качать журнал, а качать snapshot (периодически делаются слепки состояния) и докачать отсутствующие в нём транзакции.

Верны ли свойства?

- Integrity — сообщениям просто неоткуда браться из воздуха.
- TO — вроде правда.
- abdeliver делаем только после того, как сообщение будет записано на кворум Follower'ов. Потерять можем только если не было *abdeliver*. Agreement — вроде тоже.
  - Local Order. Когда происходит COMMIT, доставляются все транзакции, не доставленные до той, о которой пришло подтверждение.
- Global Order. Верно.
- Консистентность префикса. В конце SYNC делаем COMMIT и только потом ready. Это и гарантирует.

AP системы

Есть пользователь, который хочет купить что-то в интернет-магазине. Его запрос о добавлении товара в корзину прилетел на какой-то фронтенд в каком-то дата-центре. Представим, что все наши дата-центры в огне, мы не смогли обработать запрос и вернули ошибку пользователю. Пользователь ушёл, маркетологи прибежали и настучали за недоступность сервиса.

Да, по CAP-теореме все запросы обработать не можем. Но маркетологи-то злые и не понимают. Что делать?

Какие требования есть к корзине?

1. Добавил в корзину — не значит купил. Выдавать ошибку не надо, просто надо сделать хоть что-то. Запрос где-то сохраняется, обработаем позже.
2. Корзина неконсистентна. Добавляем товары в корзину на разных вкладках, в разное время, и она может быть в разных состояниях.
3. Корзина всегда доступна на запись. Главное — записать, синхронизируем потом.

Это похоже на AP-систему.

Что хотим от системы?

1. Key-Value хранилище (реляционная БД не всегда нужна, для реализации корзины вполне достаточно key-value, где пользователь — ключ).
2. Шардирование (размазывать данные по узлам, не хранить всё в одном месте) & репликация.
3. Всегда доступны на запись.
4. Версионирование values (чтобы понимать, какой из наборов данных настоящий).
5. Децентрализована.
6. Масштабирование без деградаций.

Люди из Amazon написали это и назвали Amazon DynamoDB.

Наши хотелки в формальной форме

1. Формальное описание базы данных (Query language) key → value, один запрос — один ключ
2. ACID
   (a) Atomicity — атомарность всегда будет, ибо в транзакции только одна операция, и не о чем говорить
   (b) Consistent — не будет (если только очень слабая)
   (c) Isolated — нет
   (d) Durability — eventually (сколько сможем: если ноды упали, но вернулись, сможем получить с них данные)
3. Efficiency — 99.9% (перцентиль — распределение времён ответов, и смотрим на статистику для перцентили 99.9%)

4. Кастомизация — дать пользователю настраивать базу данных, tradeoff времени ответа и степени консистентности
5. AAA — Authentication, Authorization, Accounting — нет. Каждый сервис поднимает свою инсталляцию.
6. Symmetry — не должно быть выделенных узлов, играющих особую роль, поскольку их отказы будут критичны (т. е. хотим, чтобы узлы имели одинаковую функциональность). При записи узлы пишут с разной скоростью. Пусть они общаются между собой, делая вместе асинхронно лучше и быстрее (аналог P2P-сетей).
7. Гетерогенность — учитывать то, что у нас разное оборудование разной производительности.

Проблемы и решения

Проблема	Техника	Профит
Шардирование	Консистентные хэши	Инкрементальный рост
HA (Highly Available) на запись	Векторные часы и reconcilation на чтение	HA и Durability
Обработка отказов	Sloppy Quorums + Hinted Handoffs	Решили проблему
Recovery	Anti-entropy и Merkle Trees	Восстановление ...
Membership и Failure detection	gossip negotiation	Решили проблему

Шардирование

Строим БД.

Как у нас устроены key-value хранилища? Хэш-таблица или дерево? Хэши, конечно. Как вообще распределять и параллелить деревья?

Пользователь имеет id в диапазоне $[0, R]$. Пусть есть сервера $S_1, \ldots, S_N$. Каждому серверу сопоставляем случайное число $T_1, \ldots, T_N$. Данные из диапазона $[T_i, T_{i+1}]$ будем хранить на сервере S_i.

Пусть приходит новый сервер S_{N+1} с числом T_{n+1}. Вспомним обычную хэш-таблицу. Перехэширование полностью переписывало все данные. Мы не можем себе это позволить. Поэтому смотрим, куда попадает T_{N+1}, забираем часть данных от текущей дуги и отправляем на новый.

Итого, переезжает лишь $[\frac{R}{N}]$ данных. Это называется "консистентный хэш".

Проблемы:

1. Неравномерность распределения данных
2. Неравномерность использования оборудования

Решение: серверу сопоставляем не ключ, а последовательность ключей $T_{i,1}, \ldots, T_{i,m_i}$. Количество таких кусков m_i зависит от производительности сервера. Этот метод можно рассматривать как "виртуальные сервера". Итак, мы размазали данные более равномерно и учли производительность.

Репликация: D — число копий.

В нашей системе без репликации $R_0 \in T_1$. Сделаем $R_0 \in T_1, \ldots, T_D$.

При добавлении нового сервера нужно поддерживать это. То есть, на новый сервер записать все дубликаты, а с какого-то стереть.

Проблема:

- Учитываем виртуальность.
  Реальных репликаций может оказаться меньше. Токены принадлежат виртуальным серверам, а они могут быть на одном физическом.
- Учитываем физические условия. Если у нас есть два дата-центра, хочется хранить реплики в разных дата-центрах, чтобы не было failed domain — когда отвалился дата-центр, стойка, коммутатор, что-то ещё.

Надо разметить все наши сервера по этим признакам и выбирать места для дубликатов с учётом этого.

HA на запись

Durability — если прошла запись, то она сохранится, не потеряется.

API: $put(key, value, ctx)$ $get(key) -> [(v_1, ctx_1), \ldots, (v_n, ctx_n)]$

У нас может быть неконсистентность, поэтому иногда БД не сможет различить, что надо вернуть. Поэтому она возвращает несколько вариантов, в надежде на то, что пользовательская программа сможет найти нужное ей значение по контексту ctx.

Можно сделать ctx=версия (версионирование). Но проблема в линейности этого версионирования. Пусть прилетели 2 запроса $put(k, v_1, x)$ и $put(k, v_2, y)$ на два узла, между которыми не было соединения на момент обработки запроса. Когда связность вернётся, придётся решить, какая из записей настоящая.

Заставим решать проблему БД. Если выкинуть какой-то ключ (случайно или по записям системных часов), нарушаем Durability. Но есть случаи, когда БД сможет различить, тогда хорошо, знаем, что вернуть. Если не можем решить на уровне БД, заставляем решать пользовательское приложение.

Векторные часы (воспоминание из курса параллелочек).

1. int Пусть есть p_1 с часами c_1 и p_2 с часами c_2. При получении сообщения с меткой времени ctx процессом его часы $c_i = max(ctx, c_2) + 1$. Проблема в том, что много информации теряется.
2. vector Каждый процесс хранит часы каждого из других процессов в виде вектора, и ctx=этот вектор. Происходит событие — увеличиваем свои часы. При получении сообщения берём максимум из позиций, которые не свои, а свою увеличиваем на 1 (так обеспечиваем транзитивность).

   Могут быть проблемы, если вектора становятся длинными (чтоы понять, кто более новый, надо сортировать вектора, это за $N \log N$). Это решается ...
3. matrix Тоже есть, вспоминаем сами.

Как пользовательская программа может разрешать неоднозначности? Если в корзину можно только добавлять, то можно просто объединить списки продуктов и сказать потом базе, что на самом деле пользователь добавил всё вместе (сделать put нового значения). База сама поймёт по векторным часам, что то значение новое, а предыдущие несколько — старые. Это и есть reconcilation — обновление значений.

При этом при любом get база может смотреть, у кого старые значения, и говорить им, что надо обновить, чтобы поддерживать актуальное состояние.

Обработка отказов

Если $Q > \frac{N}{2}$, пересечение кворумов будет непусто, и потери данных не произойдёт. Будем собирать отдельные кворумы на чтение и на запись. Если $|Q_R| + |Q_W| > N$, последнюю запись всегда будет видно.

Sloppy Quorums означает, что N не фиксировано. Если сломались все ноды $S_1, \ldots, S_D$, пишем на S_{D+1} и на другие, до которых дотянемся. Но при этом если сразу же после такой записи сделаем чтение, записи можем не увидеть: если ноды поднялись, то будем читать с них, а на них нет информации (ещё не синхронизировались). То есть, можем получить старое значение, новое или оба сразу.

Владельцы чужих записей выставляют на не свои записи флаг foreign и периодически сообщают $T_1, \ldots, T_D$ (истинным хозяевам), что нужно их забрать (такое обращение называется hint). Это и есть Handoff (вспомним помогание в lock-free алгоритмах).

Считая, что пользователь не слишком быстрый, и не будет непрерывного потока операций с корзиной, это работает неплохо.

Recovery

Узел пропал и вернулся $\Rightarrow$ может восстановиться.

Анти-энтропия — сравнение всех реплик одного и того же блока и обновление их до актуальной версии. Допустим, упавший узел восстановился, на нём есть какие-то реплики, и надо проверить, актуальны ли они.

Пусть есть T_1, T_2 с общим Range R_0. В какой-то момент T_2 решает, что надо синхронизироваться. Обмениваться всеми данными по сети долго (может быть, они даже согласованы, тогда ничего делать не надо).

Merkle Trees — деревья хэшей. То есть от куска считается хэш, он делится пополам, от половинок тоже считается хэш, и так далее.

Обмениваемся хэшами корня дерева, если совпали, OK, если нет, обмениваемся хэшами на уровень ниже. Так можно вполне быстро понять, что именно поменялось. Это допускает асинхронную синхронизацию.

Это используется в Bitcoin, Ethereum, magnet-ссылках в торрентах.

Membership и Failure detection

Пусть есть T_1, T_2, T_3. Есть клиент C_1 и range ключей R_0. Тогда всегда C_0 обращается к T_1, а он потом распределяет на T_2, T_3. Если T_1 недоступен, все идут в T_2. Это гарантирует нам бóльшую упорядоченность, и у нас меньше мороки с синхронизацией.

Проблема: может возникнуть bottleneck. Пусть пришла куча клиентов на T_1. Он может стать перегружен. Мы теряем симметричность нод, у них появляются особые роли. Поэтому пусть клиент обращается к любому из T_1, T_2, T_3. Но клиент ничего не знает о range и внутреннем устройстве БД и не может выяснить, где T_1, T_2, T_3. Так пусть он пойдёт к любому узлу, а они уже сами распределят между собой. При этом синхронно выяснять кто есть кто мы не можем, ибо нода могла вообще партицироваться в тот момент и ни до кого не достучаться.

Как добавлять новые ноды? Опять же, синхронно это делать плохо.

Решение: добавлять и удалять ноды будет оператор. Новая нода будет просто рассказывать всем о своём существовании, а писать на неё данные, пока она не появится в топологии, никто не будет, и запросы на неё приходить тоже не будут. Если какой-то узел упал, мы думаем, что он поднимется, и не начинаем реплицировать данные, которые на нём были, на какой-то ещё узел.

Мы будем хранить версионированную топологию сети, которую будет менять оператор. Пусть оператор пришёл и сказал о новой топологии сети. Любому узлу. А узлы между собой должны распределить новую информацию. Узлы будут асинхронно рассказывать друг другу о новой топологии. В итоге все узнают о новости примерно за логарифм.

Можно ускорить. Возьмём несколько нод, пометим звёздочками. Будем говорить, что это узлы-информаторы, которые знают самые последние новости, и о новой топологии надо спрашивать сначала их. Потеряли в симметрии, но это внутренний контрольный трафик, он небольшой, и такая потеря в симметрии не страшна. Эти ноды называются seed nodes.

Как быстро мы работаем?

Для записи клиент ищет любую ноду, та ищет $|C_W|$ нод и пишет на них. $O(ping + flush * D)$

Чтение — то же самое, только собираем кворум на чтение и не тратим время на запись. $O(ping)$

Кастомизация: меняем значения. Обычно $|C_W|$ = 2-3, $|C_W|$ = 2-3, D = 3-4.

Консистентное хэширование

- Вариант S1 — см. выше. Проблемы: неодинаковые куски, неоднородные хэши (получаются "горячие участки").
- Вариант S2. Делаем кольцо, разбиваем на range-и $R_1, \ldots, R_n$ равномерно. Сервер генерирует себе токены $T_1, \ldots, T_{m_i}$. Токен попадает в какой-то range $\Rightarrow$ сервер берёт себе этот range. Если какие-то куски остались непокрытыми, надо было сгенерировать больше токенов. Промежуточная версия для перехода от S1 к S3, может работать даже медленнее S1.
- Вариант S3. Узел выбирает себе range'и исходя из загруженности этих range.