

Распределённые файловые системы для больших данных

Для того, чтобы обрабатывать большие данные, их нужно где-то хранить. Вертикальное масштабирование (scale up): покупаем всё бо́льшие 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 удаляем 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. А 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 метаданные 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'a 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.

1. $\text{map } f : T \rightarrow [S] = [(k, v)]$. Функция применяется поэлементно к входным данным и может выполняться параллельно.

Выходные данные аннотируются ключом.

2. `shuffle & sort`

Данные перераспределяются по редьюсерам так, чтобы значения с одним ключом обязательно попали к одному редьюсеру, и при этом сами ключи были отсортированы.

3. $\text{reduce } g : k, [v] \rightarrow [(k', v')]$.

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`: выделяем из строки лога `uid`. $f : \text{line} \rightarrow (\text{uid}, \text{NULL})$
2. `shuffle&sort` Сортируем по `uid`.
3. `reduce` В отсортированном списке количество уникальных элементов считается просто: количество смен ключа при последовательном чтении. $g : \text{uid}, [v]_n \rightarrow \text{uid}, n$ $g : \text{uniq} - c - \text{утилита unix}$.

```
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

групп на стадии `map` ищем, всё.

`group by map` ничего, `sort` сортирует по ключу, `reduce` просто собирает куски с одинаковым ключом в группы.

`reduce=sum` Выдаём промежуточно сумму. `reduce=avg` Выдаём сумму и количество. Тогда можно восстановить правильное среднее. `reduce=median` можем сжать повторяющиеся значения в пары (количество, ключ) Но комбайн плохо всё равно работает.

Обратный индекс В `map` к слову приписываем количество его встреч, свели задачу к предыдущей.

В целом кажется, что в **MapReduce** довольно годно написано.

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
 - Shared Memory
 - * Система состоит из вычислительных узлов, объединённых в сеть
 - * Вычислительные узлы не имеют общей памяти и взаимодействуют только через передачу сообщений
 - Message Passing
 - * 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

wait-free Независимо от того, какие сообщения приходят тебе и доходят ли твои сообщения до других, ты за константное время совершаешь прогресс.

УТВ. $MP \Leftrightarrow SM$

- • $MP \Rightarrow SM$ Будем использовать адресное пространство вычислительных узлов как общее адресное пространство. Операции `store()` и `load()` будем эмулировать через отправку сообщения соответствующему узлу
- $SM \Rightarrow MP$ Разобьём 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. $A \Leftrightarrow S$

- • $S \Rightarrow A$ Синхронная модель — особый случай асинхронной. Т. о., эмуляция не требуется.
- $A \Rightarrow S$ Для MP систем нам потребуется специальный примитив `Synchronizer`, который позволит эмулировать тактовость синхронной системы. Для SM систем мы просто можем использовать барьеры.

◀

Формальное определение: АМР

Система состоит из n процессов p_i , расположенных на n процессорах. C_{ij} — канал связи между процессами и m_{ij} — сообщение от p_i для p_j . 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 не блокируют процесс.

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

Заданы три отношения порядка на множестве событий: Линейный порядок на событиях процесса p_i : $H_i = (\bigcup_x e_i^x \rightarrow i)$. отношение порядка \xrightarrow{msg} между отправкой и получением сообщения: $\text{send}(m) \xrightarrow{msg} \text{receive}(m)$ Их

замыканием будет отношение $H = (\bigcup e, \rightarrow) e_i^x \rightarrow e_j^y \Leftrightarrow \begin{cases} e_i^x \xrightarrow{i} e_j^y \\ e_i^x \xrightarrow{msg} e_j^y \\ \exists e_k^z : e_i^x e_k^z \rightarrow e_k^z \wedge e_k^z \rightarrow e_j^y \end{cases}$

Синхронизатор

Синхронизатор используется для эмуляции синхронной системы поверх асинхронной. При помощи синхронизатора будем эмулировать методы $\text{send}(m)$, $\text{receive}(m)$, 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 отправляет 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

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

Пусть два синхронизатора с разницей 2. Тогда

- 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. ◀

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

Модель отказов

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

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

- • 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
 - 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.

Формально:

- **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 принимается при помощи заранее известной функции, например $\min(\text{values})$ или $\text{majority}(\text{values})$

Разрешаем f отказов. Потоки отправляют свои значения. Когда получит $n-f$ сообщений, выбирает f из $n-f$ чисел детерминированной функцией, например, минимальных. И среди них хотя бы одно будет общим.

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

Теор 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

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.

Состояние системы Введём понятие "Состояния системы" GS (Global State). В GS войдёт состояние процессов. Мы договаривались, что работа процесса состоит из набора атомарных событий e_i^x . События бывают трёх типов: внутренние, отправка сообщения $send(p_i, m)$ и получение сообщения $recieve(p_j)$.

Состояние сети можно описать как множество пересылаемых сообщений. Вызов процессом p_i функции $recieve(p_j)$ удаляет из множества случайное сообщение от p_i к p_j или ничего не делает.

Валентность состояния Определим функцию $v(GS)$, возвращающую значение консенсуса для состояния GS . Функция может вернуть $\{0\}$, $\{1\}$ или $\{0, 1\}$ в случае неопределённости. **Валентностью системы** будем называть результат функции $v(GS)$. Т.е. система может находиться в трёх состояниях: *0-valent*, *1-valent* или *bivalent*.

Вход алгоритма Через $\vec{I} = \{in_0, in_1, \dots, in_n\}$ обозначим вектор начальных значений in_i для процесса p_i .

Положим $\vec{I}_0 = \{0, 0, \dots, 0\}$, $\vec{I}_1 = \{1, 1, \dots, 1\}$.

Теор. Существует бивалентное начальное состояние

- От противного. Пусть бивалентного состояния не существует.

- Ввод \vec{I}_0 даст 0-валентное состояние, \vec{I}_1 — 1-валентное.
- Выпишем входы $0 \dots 0, 10 \dots 0, 110 \dots 0, 1 \dots 1$, которые из I_0 приводят к I_1 . Среди них есть два соседних \vec{I}_a и \vec{I}_b с разной валентностью 0 и 1 соответственно.
- Очевидно, что \vec{I}_a и \vec{I}_b отличаются одним битом, и пусть этот бит принадлежит процессу p_i .

Если существует алгоритм консенсуса, то:

- При отказе процесса p_i после ввода \vec{I}_a система должна решить 0
- При отказе процесса p_i после ввода \vec{I}_b система должна решить 1

Но это невозможно, т.к. после отказа p_i состояние системы будет неразличимо для остальных процессов. ◀

Теор. Пусть есть бивалентное состояние системы C и есть сообщение $e = (p, m)$, применимое к этой конфигурации. Пусть D — множество состояний, достижимых из C без применения e . И пусть $H = e(D) = \{e(E) : E \in D\}$ — множество состояний, в которые переходят состояния D после применения e . Тогда H содержит хотя бы одно бивалентное состояние.

- От противного. Пусть все состояния H одновалентные, т.е. или 0-валентные или 1-валентные.

Пусть E_0 — 0-валентная конфигурация, достижимая из C . Она существует, т.к. C — бивалентна. Если $E_0 \in D$, то $F_0 = e(E_0)$ и $F_0 \in H$. Если $E_0 \notin (D \cup H)$, то существует $F_0 \in (H)$, из которого E_0 достижимо. $v(E_0) = v(F_0)$, т.к. мы предположили, что у C нет бивалентных потомков в (H) . В любом случае $F_0 \in H$ и $v(F_0) = 0$, что и требовалось доказать.

Случай 1-валентной конфигурации доказывается аналогично. ◀

Назовём конфигурации **соседними**, если их отделяет доставка одного сообщения. Несложно показать, что существуют две соседние конфигурации C_0 и C_1 , такие что $D_i = \{e(C_i) \mid i \in \{0, 1\}\}$ и D_i — i -валентная. Для определённости положим $C_1 = e'(C_0)$ и $e' = (p', m')$. Рассмотрим два случая: $p = p'$ и $p \neq p'$.

Консенсус (задача о византийских генералах). Есть процессы и числа. Из всех предложенных чисел нужно выбрать одно. В ситуации с отказом число должно быть предложено не отказавшим процессом.

Сортировать надо не лексикографически (контрпример 01 10). Говорят, что достаточно выписать n состояний $0 \dots 0, 10 \dots 0, 110 \dots 0, 1 \dots 1$, которые из I_a приводят к I_b .

САР-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.

Теор. В асинхронной распределённой системе невозможно имплементировать базу данных, удовлетворяющую САР свойствам.

- • Пусть это не так, и в системе есть хотя бы 2 процесса.
- Разобьём процессы на две непересекающиеся группы $\{G_1, G_2\}$.
 - Пусть v_0 — начальное состояние базы.
 - Рассмотрим выполнение α_1 , в котором была единственная запись в группе $\{G_1\}$, и пусть мы записали v_1 . Потребуем потери сообщений между $\{G_1, G_2\}$
 - Рассмотрим выполнение α_2 , в котором было единственное чтение в группе $\{G_2\}$, и оно могло вернуть только v_0
 - 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 атомарности.

◀

Terminated Есть распределённая БД, надо, чтобы все узлы, реализующие её, получили значение, которое нужно записать

Есть цепочка банков, переводим с одного счёта на другой через всю цепочку.

Упражнения

Придумать системы, удовлетворяющие двум свойствам.

СР – не реагируем на записи, всегда возвращаем начальное состояние. АР – Будем держать локальную базу данных и не общаться. Каждый поток возвращает своё. СА – Выделяем сервер, и пусть все узлы обращаются к нему как клиенты.

Критика САР-теоремы

САР теорема подразумевает модель, слишком пессимизирующую реальные сети.

Линеаризуемость — сильное свойство атомарных объектов.

Новая формулировка

Разрешим процессам иметь локальные часы, идущие с одинаковой скоростью, но несогласованные. Введём верхнюю границу на время доставки сообщения t_{msg} .

Такая система называется частично-синхронной.

t-Connected Consistency

Будем называть такой косистентностью ситуацию, когда выполняются два свойства:

- Выполнение, во время которого не было потери сообщений — атомарное
- При выполнении с потерями существует частичный порядок P , такой что:
 1. P упорядочивает все операции записи и упорядочивает все операции чтения относительно записи
 2. Чтение возвращает значение последней записи относительно P или начальное значение, если нет предыдущей записи в P
 3. P сохраняет порядок операций чтений и записей, обработанных каждым узлом
 4. Если операции α и β разделяет интервал дольше $2t_{msg}$, в котором не было потерь, и α завершилась до него, а β началась после, то операции упорядочены в P

Реализация Есть выделенный процесс C , хранящий оригинал данных. `read` на узле A обрабатывается через запрос последней версии данных у узла C . Если удалось получить ответ за $2t_{msg}$, то запоминаем новое значение и возвращаем результат. Если нет, то возвращаем последнее известное значение. `write` на узле A тоже проксируется (пересылается) на C , но в случае таймаута вернёт клиенту подтверждение, а сам узел будет периодически перепосылать запрос на запись нового значения. `write` на узле C присваивает изменению номер и запоминает. Периодически C перепосылает всем узлам свежее значение данных.

Теор. Предложенный алгоритм является t-Connected Consistent

- • Атомарность на промежутке без потерь

В ситуации отсутствия потерь точкой линеаризации можно выбрать обработку запроса узлом C . Таким образом мы зададим глобальный порядок на всех операциях за этот интервал времени.

- Пункты 1, 2

Узел C задаёт глобальный порядок на всех операциях записи, т.к. он присваивает им номера. Операции чтения упорядочим сразу после соответствующей записи.

- Пункт 3

Достигается обработкой запросов в порядке получения.

- Пункт 4

Рассмотрим 4 случая, в которых две операции разделял интервал $T > t_{msg}$, в котором не было потерь.

- `write` и следующий за ним `read`
после `write` прошло достаточно времени, чтобы C успел узнать об этой записи, присвоить ей номер, анонсировать её (или последующую запись), и узел обработавший вторую операцию X должен был успеть получить этот анонс
- `write` и следующий за ним `write` после `write` прошло достаточно времени, чтобы C успел узнать об этой записи и присвоить ей номер.
- `read` и следующий за ним `read` первый `read` упорядочивается после записи φ . Мы знаем, что φ и второй `read` упорядочены, т.е. второй `read` не мог вернуть значения моложе записи φ
- `read` и следующий за ним `write` Аналогично ситуации `read read` второй `write` упорядочится позже



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 транзакции 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
<i>/* Query 1 */</i>	
SELECT * FROM users WHERE id = 1;	<i>/* Query 2 */</i>
	UPDATE users SET age = 21 WHERE id = 1;
	COMMIT;
<i>/* Query 1 */</i>	
SELECT * FROM users WHERE id = 1;	
COMMIT;	

Phantom reads

Два идентичных запроса возвращают разный набор строк.

Transaction 1	Transaction 2
<i>/* Query 1 */</i>	
SELECT * FROM users	
WHERE age BETWEEN 10 AND 30;	<i>/* Query 2 */</i>
	INSERT INTO users(id,name,age)
	VALUES (3, 'Bob', 27);
	COMMIT;
<i>/* Query 1 */</i>	
SELECT * FROM users	
WHERE age BETWEEN 10 AND 30;	
COMMIT;	

СР 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 обрабатывать одновременно.
- Latency — время обработки одной транзакции.

ZAB(ZooKeeper Atomic Broadcast)

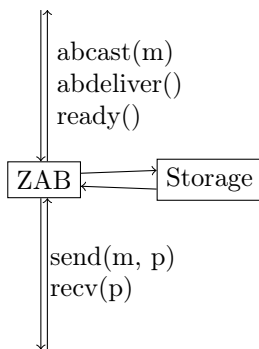
Есть Master, и есть Follower. Как Master'у отдавать приказы Follower'ам? Через ZAB. ZAB реплицирует транзакции. Гарантии:

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'a)

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'a. 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*($\langle v, z \rangle$), то \exists процесс, который сделал *abcast*($\langle v, z \rangle$) (т. е. сообщения не берутся из воздуха).

2. Total Order

Пусть $\Pi = p_1, \dots, p_n$ — множество процессов (считаем, что фиксировано).

Зам. Пусть $\rho_1, \dots, \rho_e, \rho_{e+1}, \dots$ — лидеры эпох $1, \dots, e, e+1, \dots$. 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*($\langle v, z \rangle$), причём раньше *abdeliver*($\langle v', z' \rangle$).

3. Agreement

Если $\begin{cases} p_i \text{ сделал } abdeliver(\langle v, z \rangle) \\ p_j \text{ сделал } abdeliver(\langle v', z' \rangle) \end{cases}$,
то $\begin{cases} p_i \text{ сделал } abdeliver(\langle v', z' \rangle) \\ p_j \text{ сделал } abdeliver(\langle v, z \rangle) \end{cases}$.

Зам. Из ТО следует, что если один поток получает оба сообщения, то другой поток тоже должен 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*($v', \langle e, c_2 \rangle$), то p_i сделал *abdeliver*($v, \langle e, c_1 \rangle$).

То 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 $\begin{cases} abdeliver(v', \langle e', c' \rangle) \\ abdeliver(v, \langle e, c \rangle) \end{cases} \Rightarrow abdeliver(v, \langle e, c \rangle)$ предшествует *abdeliver*($v', \langle e', c' \rangle$).

1. 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 Casual 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 Casual Order

Алгоритм

Discovery <-----
| | |

```

V          |  |
Sync -----|
|          |  |
V          |  |
Broadcast -----

```

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
 - $CEPOCH(e) \rightarrow \text{Leader}$
- L.2
 - Лидер пытается понять, что творится на кластере после предыдущего лидера.
 - Собирает Q (включая себя туда), узнаёт, какие эпохи у фолловеров.
 - Отправляет $Newepoch(e')$: $e' >$ номера 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 $I_e = \{z \in h_{f_i} \mid z' \leq \max(z(h_{f_i}))\}$ (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 $z' \leq z$ и не было $abdeliver(v, z')$

• L.3

– Отправляет $abcast(\langle v, z \rangle)$

– Делает $abdeliver(z')$ для z' , т. ч. $z' \leq z$ 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 $NEWPOCH(e')$ и $NEWLEADER(I_e + \beta_{e'})$.

* Когда получит от p_j ACK-LD, добавит его в кворум: $Q = Q \cup \{p_j\}$

– 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, обрабатываем 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) $key \rightarrow value$, 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
НА (Highly Available) на запись	Векторные часы и reconciliation на чтение	НА 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, \dots, S_N . Каждому серверу сопоставляем случайное число T_1, \dots, 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 $\lceil \frac{R}{N} \rceil$ данных. Это называется "консистентный хэш".

Проблемы:

1. Неравномерность распределения данных
2. Неравномерность использования оборудования

Решение: серверу сопоставляем не ключ, а последовательность ключей $T_{i,1}, \dots, T_{i,m_i}$. 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 $R_0 \in T_1, \dots, T_D$.

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.

НА на запись

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

API: $put(key, value, ctx) \rightarrow [(v_1, ctx_1), \dots, (v_n, ctx_n)]$

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 (вспоминание 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 (чтобы 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 reconciliation — 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, \dots, 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, \dots, 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, ОК, 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 партиципировать 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. $O(\text{ping} + \text{flush} * D)$

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

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

Консистентное хэширование

- Вариант S1 — см. выше. Проблемы: неодинаковые куски, неоднородные хэши (получаются "горячие участки").
- Вариант S2. Делаем кольцо, разбиваем на range-и R_1, \dots, R_n равномерно. Сервер генерирует себе токены T_1, \dots, 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 загрузки этих range.