

# Big Data'13

## Лекция VI: NoSQL и согласованность

Дмитрий Барашев  
bigdata@barashev.net

Computer Science Center

28 марта 2013

Этот материал распространяется под лицензией

**Creative Commons "Attribution - Share Alike" 3.0**

<http://creativecommons.org/licenses/by-sa/3.0/us/deed.ru>

# Сегодня в программе

CAP теорема

Модели согласованности

Percolator

# Сегодня в программе

CAP теорема

Модели согласованности

Percolator

# САР теорема

- ▶ Из этих трех вещей можно выбрать только две:
  - ▶ **C**onsistency (согласованность)
  - ▶ **A**vailability (доступность)
  - ▶ **P**artition tolerance (устойчивость к разделению)

# САР теорема

- ▶ Из этих трех вещей можно выбрать только две:
  - ▶ **C**onsistency (согласованность)
  - ▶ **A**vailability (доступность)
  - ▶ **P**artition tolerance (устойчивость к разделению)

Это теорема, но не догма

# Согласованность

- ▶ Согласованность в СУБД: выполнение всех ограничений
- ▶ Согласованность в распределенной системе: во всех вычислительных узлах в один момент времени данные не противоречат друг другу
- ▶ Модель согласованности: набор правил, в обмен на соблюдение которых система дает какие-то гарантии согласованности
- ▶ Бывают разные модели согласованности

# Доступность

- ▶ Любой запрос, полученный работоспособным узлом, должен получить ответ, содержащий запрошенные данные
  - ▶ Ответ «ой, у нас модем сломался, приходите завтра» ответом не считается



# Доступность

- ▶ Любой запрос, полученный работоспособным узлом, должен получить ответ, содержащий запрошенные данные
  - ▶ Ответ «ой, у нас модем сломался, приходите завтра» ответом не считается
- ▶ Считается ли система из 10 идентичных read-only серверов в одном ЦОД доступной после попадания в ЦОД ядерной бомбы?

# Устойчивость к разделению

- ▶ Потеря сообщений (частичная или полная) между компонентами системы не влияет на работоспособность системы.
- ▶ Речь в основном о сбоях в сети
- ▶ Самая спорная концепция

## Устойчивость к разделению: критика

- ▶ Сбой одного узла – тоже разделение, ведь сообщения то к нему не ходят
  - ▶ не совсем так: он ведь и на запросы не отвечает, то есть не является работоспособным

## Устойчивость к разделению: критика

- ▶ Сбой одного узла – тоже разделение, ведь сообщения то к нему не ходят
  - ▶ не совсем так: он ведь и на запросы не отвечает, то есть не является работоспособным
- ▶ Абсолютно надежных сетей не задерживающих сообщения не бывает!
  - ▶ но если у двух машин нет общих атомных часов то и о 100% согласованности говорить сложно

## Устойчивость к разделению: критика

- ▶ Сбой одного узла – тоже разделение, ведь сообщения то к нему не ходят
  - ▶ не совсем так: он ведь и на запросы не отвечает, то есть не является работоспособным
- ▶ Абсолютно надежных сетей не задерживающих сообщения не бывает!
  - ▶ но если у двух машин нет общих атомных часов то и о 100% согласованности говорить сложно
- ▶ Окей, но ведь в любом случае нельзя просто так игнорировать разделение!
  - ▶ можно например сразу выключить всю систему

# Выбор СР

- ▶ Выбираем устойчивость к распаду и согласованность, жертвуем доступностью
- ▶ Гарантируем некую сильную модель согласованности и будем отказывать в обслуживании некоторых запросов в случае распада
  - ▶ только запросы на чтение (системы с синхронной репликацией)
  - ▶ обслуживать доступ к данным, целиком находящимся внутри одной сетевой компоненты (шардированные системы)

# Выбор AP

- ▶ Выбираем устойчивость к распаду и доступность, жертвуем согласованностью
- ▶ Гарантируем что все запросы будут обслужены, но в случае распада произойдет рассогласованность
  - ▶ пользователи из одной компоненты связности не увидят изменения, сделанные в другой компоненте
  - ▶ разные версии обновлений данных в разных компонентах

## Выбор AP

- ▶ Выбираем устойчивость к распаду и доступность, жертвуем согласованностью
- ▶ Гарантируем что все запросы будут обслужены, но в случае распада произойдет рассогласованность
  - ▶ пользователи из одной компоненты связности не увидят изменения, сделанные в другой компоненте
  - ▶ разные версии обновлений данных в разных компонентах
- ▶ Тем не менее, хочется чтоб в конце концов система пришла в согласованное состояние



# Выбор СА

- ▶ Однозначно работает если система нераспределенная
- ▶ В распределенной кажется неразумным поведением. По Закону Мерфи, сеть обязательно упадет и нельзя это игнорировать
  - ▶ никто впрочем не употребляет «игнорировать» по отношению к согласованности

# Выбор СА

- ▶ Однозначно работает если система нераспределенная
- ▶ В распределенной кажется неразумным поведением. По Закону Мерфи, сеть обязательно упадет и нельзя это игнорировать
  - ▶ никто впрочем не употребляет «игнорировать» по отношению к согласованности
- ▶ Система может просто прекратить работу при неустранимом распаде.

# Выбор СА

- ▶ Однозначно работает если система нераспределенная
- ▶ В распределенной кажется неразумным поведением. По Закону Мерфи, сеть обязательно упадет и нельзя это игнорировать
  - ▶ никто впрочем не употребляет «игнорировать» по отношению к согласованности
- ▶ Система может просто прекратить работу при неустранимом распаде.
- ▶ И это, возможно, будет случаться не так часто

# Сегодня в программе

CAP теорема

Модели согласованности

Percolator

# ACID vs BASE

- ▶ **А**томарность, **С**огласованность, **И**золированность и **Д**олговечность
- ▶ **B**asically **A**vailable, **S**oft-state, **E**ventual consistency

# Спектр моделей согласованности

- ▶ Строгая согласованность (strong consistency)
  - ▶ все операции чтения возвращают данные, записанные последней из предыдущих операций записи, вне зависимости от того, в каком узле они сделаны
- ▶ Слабая согласованность (weak consistency)
  - ▶ операции чтения необязательно возвращают последнее записанное значение
  - ▶ восстановление согласованности требует выполнения каких-то условий
  - ▶ *окно несогласованности (inconsistency window)* - интервал между обновлением и гарантированной его видимостью всем читателям

# Согласованность в конечном счете

- ▶ Eventual consistency
- ▶ Любое обновление через некоторое время распространится и станет доступным для чтения при отсутствии последующих обновлений

# Согласованность в конечном счете

- ▶ Eventual consistency
- ▶ Любое обновление через некоторое время распространится и станет доступным для чтения при отсутствии последующих обновлений

eventual consistency is so eventual



# Бонусы к согласованности в конечном счете

- ▶ «Что запишешь то прочтешь» (Read Your Own Writes)
  - ▶ клиент может читать свои собственные обновления немедленно, вне зависимости от того, на каком узле они сделаны
- ▶ Сессионная согласованность
  - ▶ что запишешь то прочтешь, но в рамках одной сессии
- ▶ Причинная согласованность
  - ▶ если процессу сообщили о записи то он увидит измененное значение
- ▶ Монотонное чтение
  - ▶ номер версии объекта, читаемого процессом, не уменьшается

## В практическом применении

- ▶ Монотонное чтение + сессионная согласованность – хороший практический вариант

# Модель распространения обновлений

- ▶ Синхронные и асинхронные
- ▶ Асинхронное распространение
  - ▶ обновление принимается узлом, записывается локально и подтверждается
  - ▶ фоновый процесс рассылает обновления другим узлам
  - ▶ разумеется не гарантирует строгую согласованность
- ▶ Синхронное распространение
  - ▶ запись подтверждается только если ее приняли и подтвердили  $W$  узлов
  - ▶ клиент может ждать
  - ▶ при некоторых условиях может обеспечить строгую согласованность

## Немного синхронной математики

- ▶ Пусть  $N$  узлов хранят реплику объекта, запись идет на  $W$  узлов, чтение с  $R$  узлов
- ▶ Если  $W + R > N$  то множества узлов записи и чтения пересекаются и можно обеспечить строгую согласованность
  - ▶ если есть два MySQL сервера с синхронной репликацией то  $N = 2, W = 2, R = 1$
- ▶ Если  $W + R \leq N$  то множества могут не пересечься
  - ▶ два MySQL сервера с асинхронной репликацией:  $N = 2, W = 1, R = 1$

## Вариации

- ▶ Системы, ориентированные на согласованность, будут устанавливать  $W = N, R = 1$ 
  - ▶ и будут отказывать в записи если какой-то из  $W$  узлов упадет
- ▶ Системы, ориентированные на доступность, будут устанавливать  $W = 1$  и разные варианты  $R$ 
  - ▶ интересный вариант  $W = 1, R = N, W + R > N$ : чтение всегда может выбрать последнюю версию, но узел с ней может упасть
- ▶ Системы, допускающие запись в разные узлы, скорее всего захотят  $W \geq \frac{N+1}{2}$ 
  - ▶ иначе могут получиться разные версии объекта
- ▶ Если  $W + R \leq N$  то нет особого смысла иметь  $R > 1$ 
  - ▶ согласованности нет все равно, зачем же читать несколько реплик

# Привязка сессии к серверу

- ▶ Поддержка RYOW и сессионной согласованности
- ▶ Пока сессия жива, запросы прилетают к одному и тому же серверу
- ▶ Сложнее делать балансировку нагрузки и устойчивость к сбоям

# Версионирование данных

- ▶ Если допускается слабая согласованность то появляются разные версии данных
- ▶ Нужна модель поведения, которая позволит обработать эти разные версии и сойтись к единой

# Оптимистичные блокировки

- ▶ Каждый элемент данных имеет свою временную метку
- ▶ Метки читаются вместе с данными
- ▶ При записи транзакция должна убедиться, что метки имеющихся у нее данных совпадают с актуальными на момент записи
- ▶ Если это не так, транзакция не принимается
- ▶ Если метки совпали то производится запись и метки записанных элементов обновляются
  - ▶ сильно помогает атомарная операция Compare-And-Swap (CAS)
- ▶ Много приложений: ETag и If-Match заголовки в HTTP, редактирование страниц в Wikipedia, etc.



# Мультиверсионный протокол

- ▶ У каждого элемента есть временная метка и каждой метке соответствует ревизия элемента
- ▶ У каждой транзакции есть временная метка
- ▶ Метки монотонно увеличиваются
- ▶ Транзакция с меткой  $t_i$  читает элементы с меткой  $t_j < t_i, \nexists k : t_j < t_k < t_i$
- ▶ При записи обновленные данные получают метку от транзакции

# Векторные часы

- ▶ У каждого значения на каждом узле есть своя временная метка
- ▶ Каждый узел поддерживает вектор известных ему значений меток других узлов
- ▶ Получается матрица, где
  - ▶ строка  $i$  - сведения узла  $i$  о часах других узлов
  - ▶ диагональный элемент  $V_{ii}$  - актуальные показания часов узла  $i$
  - ▶ столбец  $j$  - сведения других узлов о часах узла  $j$

## Векторные часы: правила

- ▶ Операция на узле  $i$  увеличивает значение  $v_{ii}$
- ▶ Если узел  $i$  посылает сообщение узлу  $j$ , он увеличивает показания своих часов и посылает весь свой вектор вместе с сообщением
- ▶ Если узел  $j$  принимает сообщение от узла  $i$ , он увеличивает показания своих часов и сравнивает векторы.

$$V_i > V_j, \text{ if } \forall k V_i[k] > V_j[k]$$

Если  $V_i > V_j$  то выигрывает значение отправителя, если  $V_j > V_i$  то выигрывает значение получателя, если никто не больше то значит конфликт

# Векторные часы: пример

- ▶ Начальное состояние

	1	2	3
1	5	13	0
2	5	14	1
3	4	14	1

- ▶ Изменение на узле 2

	1	2	3
1	5	13	0
2	5	<b>15</b>	1
3	4	14	1

## Векторные часы: пример

- ▶ Узел 2 звонит узлу 1:  $V_2 > V_1$

	1	2	3
1	<b>5</b>	<b>15</b>	<b>1</b>
2	5	15	1
3	4	14	1

- ▶ Узел 3 звонит узлу 1:  $V_1 > V_3$

	1	2	3
1	<b>5</b>	<b>15</b>	<b>1</b>
2	5	15	1
3	4	14	1

# Преимущества

- ▶ Не требуются синхронизированные часы
- ▶ Не требуется глобального порядка номеров версий



# Сегодня в программе

CAP теорема

Модели согласованности

Percolator



# Проблема построения веб индекса

- ▶ Map-Reduce это прекрасно но он производит целый индекс или никакого
- ▶ Хочешь обновить индекс – запускаешь цепь Map-Reduce'ов и ждешь
- ▶ Это плохо подходит для новостей и блогов
- ▶ Хотелось бы обновлять индекс поддокументно

# Проблема подocumentного обновления

- ▶ Map-Reduce неприменим: ему нужен весь вход чтобы построить весь выход
- ▶ Наивное обновление индекса может нарушить ограничения целостности
  - ▶ например если есть несколько дубликатов страниц то в индексе должна быть страница с самым высоким рангом и отдельный список дубликатов
- ▶ Для высокогранулярного обновления нужны транзакции

# Проблемы с транзакциями

- ▶ DBMS либо разорвутся либо озолотятся
- ▶ Bigtable as is поддерживает атомарность записи одной строки
- ▶ Нужно что-то что обеспечивает ACID свойства для групп строк

# Percolator

- ▶ Percolator – протокол и библиотека реализующая ACID транзакции поверх Bigtable
- ▶ Это клиентская библиотека, не встроенная в Bigtable

# Действующие лица

- ▶ Bigtable и GFS
- ▶ Клиентские бинарники, использующие Percolator
- ▶ Сервер временных меток
- ▶ Легковесный сервис блокировок

# Что обещается и что нет

- ▶ Обещается
  - ▶ ACID семантика
  - ▶ Snapshot isolation
  - ▶ использование императивного языка
- ▶ Не обещается
  - ▶ быстрый отклик

## Схема работы

- ▶ При старте транзакция получает метку начала  $t_s$
- ▶ Транзакция читает данные с временной меткой  $t < t_s$  и записывает изменения локально
- ▶ Когда клиентский код решает подтвердить транзакцию
  - ▶ выполняется стадия блокировок двухфазного протокола с возможными исходами:
    - ▶ успех
    - ▶ обрыв из-за наличия более поздней подтвержденной записи
    - ▶ обрыв из-за наличия конкурирующих блокировок
  - ▶ транзакция получает метку коммита  $t_c$
  - ▶ выполняется стадия снятия блокировок и подтверждения записанных значений с новой меткой

# Где держать блокировки?

- ▶ Сервис блокировок должен:
  - ▶ переживать сбои
  - ▶ обеспечивать относительно низкое время отклика
  - ▶ обеспечивать относительно высокую пропускную способность



# Где держать блокировки?

- ▶ Сервис блокировок должен:
  - ▶ переживать сбои
  - ▶ обеспечивать относительно низкое время отклика
  - ▶ обеспечивать относительно высокую пропускную способность
- ▶ То есть наверное быть распределенным, реплицируемым, сбалансированным

# Где держать блокировки?

- ▶ Сервис блокировок должен:
  - ▶ переживать сбои
  - ▶ обеспечивать относительно низкое время отклика
  - ▶ обеспечивать относительно высокую пропускную способность
- ▶ То есть наверное быть распределенным, реплицируемым, сбалансированным
- ▶ Погодите, так это ж Bigtable

# Организация Percolator-строки в Bigtable

- ▶ Для каждого столбца данных **c** создаются:
  - c:data** Собственно данные, как подтвержденные, так и неподтвержденные
  - c:lock** Блокировка. Наличие записи означает что транзакция находится в какой-то из стадий двухфазного протокола
  - c:write** Метка последнего подтвержденного значения

## Пример транзакции

- ▶ Транзакция переводит 7 долларов со счета Алисы на счет Болванщика
- ▶ Начальное состояние

<i>Name</i>	<i>ts</i>	<i>bal:data</i>	<i>bal:lock</i>	<i>bal:write</i>
Алиса	6			@5
	5	\$10		
Болванщик	6			@5
	5	\$2		

## Получение блокировок

- ▶ Транзакция получила метку  $t_s = 7$  и сделала локальную работу
- ▶ Время подтвердаться. Требуем блокировки и записываем значения в *data*

<i>Name</i>	<i>ts</i>	<i>bal:data</i>	<i>bal:lock</i>	<i>bal:write</i>
Алиса	7	<b>\$3</b>	<b>prim</b>	@5
	6			
	5	\$10		
Болванщик	7	<b>\$9</b>	<b>prim@Алиса.bal</b>	@5
	6			
	5	\$2		

## Возможные неприятности

- ▶ Наличие записи в *bal:write* с меткой  $> 7$  – повод упасть
- ▶ Наличие записи с какой-либо меткой в **bal:lock** – повод задуматься

## Подтверждение

- ▶ Все блокировки получены и неподтвержденные значения записаны
- ▶ Время подтверждаться. Получаем метку коммита  $t_c$ , обновляем значения в столбце *write* и чистим блокировки, начиная с главной

<i>Name</i>	<i>ts</i>	<i>bal:data</i>	<i>bal:lock</i>	<i>bal:write</i>
Алиса	8			<b>@7</b>
	7	\$3		
	6			@5
	5	\$10		
Болванщик	7	<b>\$9</b>	<b>prim@Алиса.bal</b>	
	6			@5
	5	\$2		

# Главная блокировка

- ▶ Признак (не)подтвержденности транзакции
- ▶ Главная блокировка висит → транзакция неподтверждена и возможно еще пишет изменения
- ▶ Главная блокировка снята → транзакция записала все изменения по крайней мере в *data*
- ▶ Все остальные взятые блокировки показывают на главную



## Завершение подтверждения

- ▶ Продолжаем снимать блокировки и подтверждать значения

<i>Name</i>	<i>ts</i>	<i>bal:data</i>	<i>bal:lock</i>	<i>bal:write</i>
Алиса	8			<b>@7</b>
	7	\$3		
	6			@5
	5	\$10		
Болванщик	8			<b>@7</b>
	7	\$9		
	6			@5
	5	\$2		

## Возможные неприятности

- ▶ Транзакция в клиенте может упасть при завершении подтверждения
  - ▶ тогда кто-то может доделать ее работу. Например, следующая транзакция, которая обнаружит вторичную блокировку, но не найдет главной
- ▶ Транзакция может упасть между взятием блокировок и подтверждением. Тогда уже взятые замки будут мешать последующим транзакциям
  - ▶ вводится политика сборки мусора другими транзакциями
  - ▶ используется наличие главной блокировки как признак неподтвержденности транзакции (и возможного ее падения)
  - ▶ используется сторонний сервис блокировок для хранения абсолютного времени начала транзакции

# Операция чтения

- ▶ Чтение просит значения с меткой меньше  $t_s$
- ▶ Если чтение видит блокировку то оно ждет ее снятия
- ▶ Утверждение: чтение всегда будет получать подтвержденные значения с меткой меньше  $t_s$

## Корректность snapshot изоляции в Percolator

- ▶ пусть транзакция  $W$  с  $T_W < T_R$  пишет значения параллельно с  $R$
- ▶ так как  $T_W < T_R$  то значит сервис часов выдал метку  $W$  в том же запросе что и метку  $R$  или раньше
- ▶ значит  $W$  попросила метку раньше чем  $R$  ее получила
- ▶  $W$  обязана все заблокировать, прежде чем получить метку подтверждения
- ▶  $R$  обязана попросить метку старта прежде чем начать читать
- ▶ значит  $W$  заблокировала  $< W$  запросила метку подтверждения  $< R$  получила метку начала  $< R$  читает

# Результат внедрения Percolator

- ▶ Медиана времени обновления документа стала меньше в сотни раз
- ▶ Система стала проще в обслуживании
- ▶ Потребление ресурсов стало гораздо более гладким
  - ▶ хотя в целом и выросло

# Занавес




- ▶ CAP теорема считается верной, но кроме ее экстремальных случаев есть полутона
- ▶ Бывают разные модели согласованности
- ▶ Согласованности можно достичь даже внешними средствами

Эта презентация сверстана в

**PVPEERIA**

$\text{\LaTeX}$  в вашем браузере  
[alpha.papeeria.com](http://alpha.papeeria.com)

# Литература I

-  Seth Gilbert and Nancy Lynch.  
Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services.  
*ACM SIGACT News*, 33(2):51-59, 2002.
-  Daniel Peng and Frank Dabek.  
Large-scale incremental processing using distributed transactions and notifications.  
*In Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1-15. USENIX Association, 2010.
-  Michael Stonebraker.  
In search of database consistency.  
*Commun. ACM*, 53(10):8-9, October 2010.



# Литература II



Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha.

Nosql databases.

*Lecture Notes, Stuttgart Media University, 2011.*