САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Реализация уровня Snapshot Isolation для хранилища данных HBase

Дипломная работа студента 544 группы

Василик Дмитрия Николаевича

Научный руководитель	…… / подпись /	д.ф.-м.н., проф. Новиков Б.А.
Рецензент	…… / подпись /	к.ф.-м.н., доц. Барашев Д.В.
“Допустить к защите” заведующий кафедрой,	…… / подпись /	д.ф.-м.н., проф. Терехов А.Н.

Санкт-Петербург
2010

SAINT PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Software Engineering Chair

Implementing Snapshot Isolation for HBase

by

Dmitry Vasilik

Master's thesis

Supervisor	……	Professor B.A. Novikov
Reviewer	……	Associated Professor D.V. Barashev
"Approved by" Head of Department	……	Professor A. N. Terekhov

Saint Petersburg
2010

Содержание

Введение

Одним из частых сценариев развития таких веб-проектов как Facebook, YouTube, MySpace и подобных им, является такой вариант развития событий: веб-приложение создается с использованием одной из не дорогостоящих RDBMS в качестве слоя данных (например, MySQL). По мере того как проект становится популярным, база данных становится больше и больше, запросы выполняются всё дольше и дольше.

Поскольку для большинства подобных веб-сервисов нет необходимости в ACID транзакциях, компаниями, встретившими подобную проблему, был создан целый ряд узко специализированных продуктов, которые показывали блестящую производительность по сравнению с популярными RDBMS. Высокая производительность становится результатом удаления излишних накладных расходов. Эти накладные расходы не имеют никакого отношения к SQL, они связаны скорее с реляционной моделью данных[1] и классической реализацией ACID. Позже класс подобных систем был назван NoSQL[2].

В 2007 году появилась работа [6], в который был проведен подробный анализ того какие компоненты современной OLTP системы могут быть слишком «дорогими» в смысле процессорного времени, затрачиваемого в них, на примере *объектной* СУБД Shore. Выбрав подмножество контрольных задач в бенчмарке TPC-C, авторы произвели замеры для исходной реализации Shore. Исходная реализации Shore в их тестовой конфигурации показывала пропускную способность 640 транзакций в секунду. Далее авторы этой статьи итеративно удаляли из Shore различную функциональность и производили новые измерения, пока они не остались с минимальным количеством кода «ядра», который позволял обрабатывать до 12700 транзакций в секунду. Авторы последовательно избавились от:

- логирования;
- работы с блокировками, необходимыми для реализации ACID (в Shore использовался протокол two-phase locking);
- сложных механизмов работы с индексами, сделав всю систему однопоточной;
- управления буфером страниц памяти (этот буфер необходим в первую очередь для работы с диском).

NoSQL системы зачастую не содержат одной или двух из этих компонент (или содержат их функциональность в «облегченном» виде), поскольку имеют узкую специализацию и разрабатываются в рамках других требований, чем классические реляционные СУБД.

Так Google BigTable имеет блокировки только на уровне строки, не имеет дополнительных индексов и имеет принципиально отличающуюся от реляционных схем организацию информации, что влечет другие способы работы с диском. Она безусловно не подходит для классических задач OLTP, но находит свое применение в других областях, где снижены требования к надежности и очень велик объем

1 Речь идет о том, что RDBMS работают с нормализованной схемой данных, о накладных расходах при выполнении операции JOIN.

2 NoSQL является негативным названием (оно построено на отрицании), из-за чего подвергается критике. NoSQL можно трактовать не только как «Не SQL», но и как «Не только SQL». Последняя трактовка более популярна внутри NoSQL сообщества.

хранимых данных.

Существует довольно много продуктов, которые попадают в категорию NoSQL, при этом обладая самой разной архитектурой и дизайном. NoSQL системы преимущественно предоставляют простой API, простейшую схему данных (многие из них являются schema-free), и ослабленные гарантии согласованности, например, гарантии событийной согласованности. В большинстве случаев в NoSQL системах не поддерживаются связи между таблицами и операции JOIN, так что полное описание объекта должно хранится в одной строке таблицы. Именно благодаря крайне простой схеме данных и событийной согласованности становится возможным «элегантное» масштабирование, кроме того это позволяет минимизировать время отклика.

Термин событийной согласованности вводится в контексте репликации данных. Событийная согласованность гарантирует, что по прошествии некоторого времени после последнего изменения, все реплики каждого элемента данных в системе будут согласованы. Иначе говоря, все клиенты по прошествии этого интервала времени обнаружат одно и то же значение элемента данных - гарантируется только непротиворечивость реплик.

Событийная согласованность в NoSQL системах приводит к сложной модели программирования прикладных приложений. Приложения, использующие хранилища данных с событийной согласованностью, должны быть готовы к тому, чтобы корректно обрабатывать аномалии, возникающие при работе с данными, например, аномалию потерянных обновлений. Для большинства задач могут быть созданы решения над событийной согласованностью, но зачастую это усложняет разработку и требует большей квалификации программистов. Из-за более сложной модели программирования ПО становится дороже, а также увеличивается его время выхода на рынок.

Таким образом, событийная согласованность NoSQL систем, являясь одной из их сильных сторон, предоставляет разработчикам, строящим свои приложения на базе этих систем, значительные сложности, которые не могут появиться при использовании реляционных СУБД.

В случае базы данных, поддерживающей транзакции, разработка сильно упрощается, поскольку программисту больше не нужно заботится о конкурентности доступа к данным, т.е. о всех эффектах, к которым может привести конкурентное исполнение потоков программы или даже параллельная работа нескольких её экземпляров с экземплярами других программ. Таким образом, поддержка транзакций в базе данных предоставляет возможность разработки прикладных программ так, как если бы они исполнялись строго последовательно относительно доступа к данным. Значительная доля сложности всей системы снимается с приложения и переходит на СУБД.

Целью данной дипломной работы является реализация в HBase возможности работы с транзакциями. При этом важно, чтобы система сохранила свою привлекательность для разработчиков приложений, в том числе, время отклика должно оставаться приемлемо низким.

Проект HBase — один из представителей NoSQL, система представляет собой распределенное хранилище данных с открытым исходным кодом. HBase построена над Hadoop Distributed File System согласно идеям, высказанным в работе [6], описывающей Google Bigtable.

Протоколы реализации строгой согласованности не могут удовлетворить требованию низких задержек, либо имеют высокий уровень обрывов транзакций по протокольным причинам. Одним из известных компромиссов между предоставляемыми гарантиями и производительностью является уровень изоляции Snapshot Isolation, предложенный в статье [5]. Snapshot Isolation гарантирует, что при чтении транзакция увидит только те данные, которые были зафиксированы до её начала. Свойство «первый фиксирующийся выигрывает» гарантирует сериализуемость по записи, т.к. транзакция не может зафиксироваться, если данные в её множестве записи были перезаписаны в промежутке времени от начала транзакции до её окончания. Таким образом, моделируется ситуация, в которой транзакция работала бы в «снимке» данных, созданном в момент её начала. Важно, что на этом уровне изоляции операции чтения никогда не блокируются.

В данной работе будет предложен неблокирующий протокол реализации этого уровня изоляции, а также описание его имплементации в хранилище данных HBase.

Глава 1. Обзор уровней изоляции

1.1. Введение

Существует обширный спектр уровней согласованности транзакций и протоколов их реализаций. Реализация протокола обычно оценивается при помощи двух метрик:

- пропускной способности (throughput), т.е. числа зафиксированных транзакций в секунду;
- времени отклика, которое определяется как промежуток между временем начала транзакции и её успешным окончанием.

Другой важной характеристикой протокола является количество обрывов транзакций по протокольным причинам.

В этой работе приоритет будет отдаваться времени отклика, т.е. в случае если возникает вопрос: «заблокировать транзакцию на время или оборвать?», ответом всегда будет последнее.

Логично предположить, что, чем строже гарантии, предоставляемые уровнем изоляции, тем более «тяжелыми» являются протоколы реализации. Конечно, такое наивное предположение не всегда выполняется, тем не менее, выбирая уровень изоляции, приходится искать компромисс между гарантиями, которые он предоставляет, и эффективностью протокола, его реализующего.

1.2. Обзор протоколов, реализующих сериализуемость по конфликтам

Самые сильные гарантии предоставляет уровень сериализуемости по конфликтам. Протоколы реализации этого уровня изоляции делятся на три класса: блокирующие протоколы, неблокирующие и гибридные.

Блокирующие протоколы являются дорогостоящими по своей природе, но это может окупаться тем, что альтернативные протоколы могут показывать очень большой процент обрывов транзакций по протокольным причинам. В работе [6], упомянутой выше, по результатам измерений компонента LockManager в СУБД Shore использовала очень небольшой процент числа инструкций процессора — 16,3% (процент невелик относительно управления пулом страниц, который занимал 34,6%), тем не менее в данной работе мы не будем рассматривать протоколы, использующие замки для синхронизации как слишком медленные для случая распределенных СУБД.

Самым привлекательным из неблокирующих протоколов для случая распределенного хранилища данных является протокол упорядочивания по временной метке (TO — Timestamp Ordering). Менеджер транзакции присваивает каждой транзакции временную метку ts(T), которую наследуют все операции этой транзакции. Конфликтующие операции упорядочиваются в истории согласно временным меткам. Менеджер транзакций отправляет операции менеджеру данных, который их незамедлительно исполняет. Исключение составляют операции, которые пришли «поздно» (над элементом данных уже была выполнена операция с большим значением временной метки), в этом случае менеджер транзакций обрывает

соответствующую транзакцию. К сожалению, в распределенном случае протокол должен показывать большое количество таких обрывов «поздно» пришедших операций. Кроме того фиксация транзакции T1, прочитавшей значение, записанное транзакцией T2, может быть задержана до фиксации T2, что может приводить к слишком большому времени задержки.

Одним из самых теоретически простых неблокирующих протоколов является протокол тестирования графа сериализуемости. В его основу положен алгоритм поиска циклов в графе сериализуемости. Если такой цикл найден после добавления в граф ребра, соответствующего операции транзакции T, транзакция обрывается. Этот протокол слишком непрактичен для реализации из-за поиска в графе циклов, который в худшем случае является квадратичным от числа вершин и должен происходить достаточно часто.

В предположении, что конфликтующие операции появляются достаточно редко, эффективны протоколы, которые не задерживают исполнение операций, но перед фиксацией изменений транзакции T производят проверку обновлений. Такие протоколы называются оптимистическими. Исполнение транзакции делится на три фазы: во время первой фазы транзакция производит все свои шаги, записывая обновления в собственное (приватное) рабочее пространство, во время второй фазы, перед фиксацией, производится проверка являются ли обновления корректными в смысле сериализуемости по конфликтам с уже зафиксированными транзакциями, во время третьей фазы, в случае, если шаги транзакции успешно прошли валидацию, приватные обновления транзакции записываются в базу данных. Вторая и третья фаза должны происходить в единой критической секции, в частности из-за этого этот протокол может давать большее время отклика, чем блокирующие протоколы в случае частого возникновения конфликтов. Поскольку мы не предполагаем, что конфликты должны возникать редко, этот уровень изоляции не является подходящим.

Для баз данных, предоставляющих возможность доступа к нескольким версиям одного объекта, были разработаны протоколы, порождающие истории из класса MVSR. Уровень изоляции, реализуемый этими протоколами, слабее сериализуемости по конфликтам.

Протокол мультиверсионного упорядочивания по временной метке является наследником классического TO, и таким образом наследует и его главный недостаток: в распределенном случае количество обрывов по протокольным причинам может быть велико.

Мультиверсионные протоколы проверки графа сериализуемости и блокирующие мультиверсионные протоколы также не подходят.

1.3. Обзор промежуточных уровней изоляции

В стандарте ANSI SQL 92 был введен ряд определений уровней изоляции. Каждый из четырех уровней изоляции определялся набором феноменов, которые не могут наблюдать транзакции. Позже в статье [5] была показана несостоятельность этих определений, которая проистекала из многозначности определений феноменов. Каждый из феноменов можно воспринимать в двух смыслах: его непосредственном смысле и в расширенном. Расширенная интерпретация определяет феномены, которые могут привести к аномалиям, тогда как непосредственная определяет фактическую аномалию. Кроме того, авторы показали что набор феноменов не

является полным для определения сериализуемости как уровня изоляции, который позволяет избежать всех этих феноменов. В данной работе мы приводим набор феноменов в их расширенной интерпретации, данной в [5]:

P0 (грязная запись): Транзакция T1 обновляет элемент данных, после этого другая транзакция T2 обновляет этот элемент данных до того как T1 фиксируется или откатывается.

P0: w1[x]...w2[x]...((c1 или a1) и (c2 или a2) в произвольном порядке)

P1 (грязное чтение): Транзакция T1 обновляет элемент данных, затем другая транзакция T2 читает этот элемент данных до выполнения фиксации или отката T1.

P1: w1[x]...r2[x]...((c1 или a1) и (c2 или a2) в произвольном порядке)

P2 (не повторяющееся чтение): Транзакция T1 читает элемент данных. Другая транзакция T2 затем обновляет или удаляет этот элемент данных до фиксации или обрыва T1.

P2: r1[x]...w2[x]...((c1 или a1) и (c2 или a2) в произвольном порядке)

P3 (фантом): Транзакция T1 читает множество элементов данных, удовлетворяющих некоторому предикату. Транзакция T2 создает элемент данных, который удовлетворяет этому предикату, до фиксации или обрыва T1.

P3: r1[P]...w2[y in P]... ((c1 или a1) и (c2 или a2) в произвольном порядке)

Каждый из вводимых уровней изоляции гарантирует, что феномен грязной записи не может появится в истории, удовлетворяющей этому уровню. Уровень, который ограничивается только этой гарантией, называется READ UNCOMMITTED. Это самый слабый уровень, он предоставляет минимальные разумные гарантии, т.к. если допустить P0 в истории, то восстановление становится невозможным из-за невозможности корректно определить операцию отката шага транзакции.

Второй уровень READ COMMITTED расширяет READ UNCOMMITTED и предлагает дополнительную гарантию того, что феномен грязного чтения не появится в историях. Этот уровень гораздо более удобен для разработчика, так как предоставляет хотя бы минимальные гарантии касательно чтения.

REPEATABLE READ в дополнение к свойствам READ COMMITTED запрещает феномен не повторяющегося чтения.

Уровень SERIALIZABLE запрещает все вышеупомянутые феномены.

Существует множество более сложных уровней изоляции, которые по силе[3] располагаются между READ COMMITTED и SERIALIZABLE.

Одним из таких уровней является Snapshot Isolation (SI). Транзакция, которая исполняется с уровнем изоляции Snapshot Isolation, читает данные из «снимка» зафиксированных данных, актуального на момент старта транзакции. Иначе говоря транзакция работает в собственном виртуальном пространстве, это касается и записи данных. Когда транзакция готова к фиксации, ей присваивается временная метка коммита, и в случае если в системе нет конкурирующих с ней транзакций, она фиксируется. Две транзакции называют конкурирующими, если их множества записи пересекаются и их временные интервалы перекрываются, т.е. нельзя сказать, что какая-то из них началась после фиксации второй.

3 Имеется в виду определение понятия «сильнее» для уровней изоляции, данное в [5].

Этот уровень сильнее, чем READ COMMITTED и предоставляет разработчикам приложений более простые и интуитивные гарантии. Простые и естественные предположения о согласованности данных для приложений делают результат выполнения транзакций для разработчика более предсказуемым, а уровень изоляции более удобным для разработки.

Еще одним преимуществом этого уровня изоляции является то, что он исключает возможность блокирования шага чтения.

Глава 2. Описание протокола

2.1. Определения

В качестве уровня изоляции будем рассматривать классический Snapshot Isolation (SI). Для того чтобы привести определение SI прежде необходимо дать определение мультиверсионной истории (из [1]).

Опр. 1. Пусть $T = \{t_1, \ldots, t_n\}$ — конечное множество транзакций, где каждая $t_i \in T$ имеет вид $t_i = (op_i, <_i)$, op_i обозначает множество операций t_i, $<_i$ обозначает их порядок, $1 \le i \le n$.

История для T есть пара $s = (op(s), <_s)$, такая что:

(a) $op(s) \subset (\bigcup_i op_i) \cup (\bigcup_i \{a_i, c_i\})$ и $\bigcup_i op_i \subset op(s)$, т.е. s состоит из объединения операций данных транзакции плюс терминирующей операции, которой является либо c_i(фиксация), либо a_i(обрыв), $\forall t_i \in T$;

(b) $(\forall i, 1 \le i \le n)$ $c_i \in op(s) \iff a_i \notin op(s)$, т.е. для каждой транзакции есть либо фиксация, либо обрыв в истории s, но не оба одновременно;

(c) $<_i \subset <_s$, т.е. все порядки транзакций содержатся в частичном порядке, определенном s;

(d) $(\forall i, 1 \le i \le n)$ $(\forall p \in op_i)$ $p <_s a_i$ или $p <_s c_i$, т.е. операция фиксации или обрыва всегда является последним шагом транзакции;

(e) каждая пара операций $p,q \in op(s)$, которые принадлежат различным транзакциям, являются операциями над одним объектом данных и хотя бы одна из них является операцией записи, упорядочены в s, так что $p <_s q$, либо $q <_s p$.

Опр. 2. Версионная функция h для истории s — функция, сопоставляющая каждой операции чтения s предыдущий шаг записи на том же элементе данных и которая является Id отображением на операциях записи s:

1. $h(r_i(x)) = w_j(x)$ для некоторого $w_j(x) <_s r_i(x)$ и $r_i(x)$ читает x_j,

2. $h(w_i(x)) = w_i(x)$ и $w_i(x)$ пишет x_i.

Опр. 3. Мультиверсионная история T — пара $m = (op(m), <_m)$, где $<_m$ — порядок на множестве op(m), и:

(a) $op(m) = h(\bigcup_i op(t_i))$ для некоторой версионной функции h,

(b) $\forall t \in T$, $\forall p,q \in op(t)$ выполняется следующее:

$$p <_t q \Rightarrow h(p) <_m h(q),$$

(c) если $h(r_j(x)) = w_i(x_i)$, $i \neq j$ и $c_j \in m$, то $c_i \in m$ и $c_i <_m c_j$.

В данной работе мы будем пользоваться следующим определением SI, приведенным в работе [2].

Опр. 4. Мультиверсионное расписание S удовлетворяет критерию SI, если оно удовлетворяет:

(SI-V) версионная функция отображает каждую операцию чтения $r_i(x)$ транзакции T_i в последнюю зафиксированную на момент старта T_i операцию записи $w_j(x)$,

(SI-W) множества записи двух конкурирующих транзакции не пересекаются.

Под конкурирующими транзакциями подразумеваются транзакции, временные интервалы исполнения которых перекрываются.

Временную метку начала транзакции $t \in T$ будем обозначать start-ts(t), фиксации или обрыва — end-ts(t). Тогда временной интервал исполнения транзакции $t \in T$ есть [start-ts(t), end-ts(t)). Введем отношение порядка на множестве T:

$$t_1, t_2 \in T,$$

$$t_1 <_g t_2 <=> \text{end-ts}(t_1) \le \text{start-ts}(t_2),$$

т.е. $t_1 <_g t_2$ обозначает, что t_2 началась после фиксации t_1.

Утв. Пусть s — история, $T_{w(x)} = \{ t \in T \mid w(x) \in t \}$ - множество всех транзакций, записывавших x в истории s, тогда (SI-W) равносилен тому, что для любого объекта x временные интервалы любых двух транзакций, в множество записи которых входит x, не перекрываются, т.е.

$$\forall x \in D, \forall t_1, t_2 \in T_{w(x)}$$

$$t_1 <_g t_2 \vee t_2 <_g t_1.$$

Следствие. Для того чтобы история s удовлетворяла (SI-W) необходимо и достаточно, чтобы $\forall x \in D$ отношение $<_g$ определяло линейный порядок на множестве $T_{w(x)}$.

Иначе говоря, s удовлетворяет (SI-W) <=> $P = <_g \wedge T_{w(x)} \times T_{w(x)}$ — линейный порядок на $T_{w(x)}$. Последнее равносильно сериализуемости по записи для истории s.

2.2. Предпосылки

Предположим что во время выполнения для каждой транзакции определены:

1. Ее состояние (активна, зафиксирована, оборвана);
2. Временная метка старта транзакции start-ts(t), а также временная метка фиксации или обрыва end-ts(t), если транзакция была зафиксирована или оборвана.

При реализации протокола изоляции естественным является желание для каждой версии объекта уметь определить транзакцию, которая её записала. В HBase каждый объект имеет набор версий, каждая из которых имеет временную метку. В предположении, что вся необходимая информация о состоянии транзакции доступна по идентификатору транзакции, имеет смысл в поле временной метки версии хранить идентификатор транзакции, записавшей её.

2.3. Описание механизма чтения

Пусть T — конечный набор транзакций, транзакция $t_j \in T$, t_j содержит шаг $r_j(x)$, определим функцию h_0 на операции $r_j(x)$ следующим образом:

1. Производится выборка всех версий объекта x;
2. Из полученного списка версий удаляются значения, записанные активными транзакциями и транзакциями, зафиксированными после старта транзакции t_j; единственное исключение составляют версии, записанные данной транзакцией, такая версия, будучи найденной, сразу возвращается как значение $h_0(r_j(x))$, т.о.

$$h_0(r_j(x)) = w_j(x), \text{ если } w_j(x) <_j r_j(x);$$

3. Из списка выбирается значение, записанное транзакцией с наиболее поздним временем фиксации. Пусть $\text{argmax} \{ \text{end-ts}(t_i) \mid t_i \in T_{w(x)}, c_i \in s, t_i <_g t_j \} = t_k$, тогда $h_0(r_j(x)) = w_k(x)$.

На операциях записи h_0 — идентичное отображение.

Эти правила для получения значения функции h_0 определяют механизм реализации операции чтения.

2.4. Описание механизма записи

Для того чтобы протокол удовлетворял свойству (SI-W), протокол может быть реализован, например, одним из двух способов. Первый из этих двух подходов, заключается в использовании блокирующих протоколов для записи. Второй, названный в [1] прямолинейным, состоит в следующем: при записи значения x транзакция производит запись версии в виртуальную память транзакции без каких-либо дополнительных действий, после, во время фиксации транзакции, проверяется (SI-W), и, если история, дополненная операциями транзакции, удовлетворяет этому критерию, происходит запись значений в таблицу.

В предлагаемом в данной работе протоколе перед каждой записью транзакцией T объекта x, список всех версий x проверяется на наличие версий не зафиксированных транзакций, или версий, добавленных транзакциями, зафиксированными после старта транзакции T. Если в списке оказывается подобная версия, то транзакция T должна быть оборвана, если нет, то производится запись обновления.

Последний подход является более эффективным, чем первые два, но при его использовании могут иметь место дополнительные обрывы по протокольным причинам. «Лишние» обрывы могут проявляться, например, в случае, когда из-за записей долгоживущей транзакции, обрывается некоторое количество коротких, после чего обрывается и сама долгоживущая транзакция. С другой стороны, такой подход является более честным, в том смысле, что проблема обрыва долгоживущих транзакций из-за записей коротких не должна так проявлять себя как в первом варианте реализации.

Критерий (SI-W) в литературе обычно называют фразой «первый фиксирующийся преуспевает» («first-commiter-wins»). Подход, который реализуется в данной работе является более ограничительным, чем требует критерий «первый фиксирующийся преуспевает», более точно его можно было бы описать именем «первый записывающий преуспевает».

2.5. Доказательство корректности протокола

Для того чтобы множество операций набора транзакций T можно было рассматривать как историю, необходимо определить на нем порядок. В определении истории задается необходимый минимум требований к отношению порядка на множестве её операций. Включим в отношение порядка $<_s$ на op(s) все пары из op(s)×op(s), необходимые для того чтобы $<_s$ удовлетворяло требованиям к истории (c) и (d). Пусть p,q ∈ op(s) — операции, производимые над одним и тем же объектом x, пусть p ∈ t_i, q ∈ t_j, тогда p $<_s$ q, если end-ts(t_i) ≤ start-ts(t_j) и t_i была зафиксирована. Таким образом определенная пара (op(s), $<_s$) является историей.

Утв. 1. Функция h_0 является версионной функцией.

Доказательство:

Необходимо доказать

$$h_0(r_i(x)) = w_j(x) \text{ для некоторого } w_j(x) <_s r_i(x)$$

В случае, когда транзакция читает собственные записи $h_0(r_j(x)) = w_j(x)$. При этом $w_j(x) <_t r_j(x)$. Поскольку $<_t \subset <_s$, имеем $w_j(x) <_s r_j(x)$.

В случае чтения значения записанного другой транзакцией

$h_0(r_j(x)) = w_k(x)$, где $t_k : t_k = \text{argmax} \{ \text{end-ts}(t_i) \mid t_i \in T_{w(x)}, c_i \in s, t_i <_g t \}$

Поскольку $t_k < t_j$, end-ts(t_k) < start-ts(t_j) и $c_k \in s$, имеем, что ∀p ∈ t_k, ∀q ∈ t_j p $<_s$ q, следовательно $w_k(x) <_s r_j(x)$.

ЧТД.

Утв. 2. История m = (h(op(s)), $<_s$) является мультиверсионной.

Доказательство:

Доказательства требуют два свойства:

(b) ∀t ∈ T, ∀ p,q ∈ op(t) выполняется следующее: p $<_t$ q => h(p) $<_s$ h(q) и

(c) если $h(r_j(x)) = w_i(x_i)$, i ≠ j и $c_j \in m$, то $c_i \in m$ и $c_i <_m c_j$.

В предположении, что p,q ∈ op(t) — операции над одним и тем же объектом x, свойство (b) безусловно выполняется, в случае же, когда операции производятся над разными объектами, в рамках требований SI это условие не может быть выполнено. Это требование является слишком строгим в его изначальном звучании. В данной работе мы будем считать, что достаточным является выполнение этого свойства для операций над одним и тем же объектом.

Пункт (c) требует чтобы происходило чтение только зафиксированных версий, версионная функция h_0 определена так, что других чтений произойти не может.

ЧТД.

Утв. 3. История s удовлетворяет критерию (SI).

Доказательство:

Критерий (SI-V) удовлетворяется, поскольку h_0 определялась именно таким образом.

Проверим (SI-W). Пусть транзакция $t \in T$ началась в момент времени u_0, записала объект x в момент времени u, и была зафиксирована в момент времени u_1.

Поскольку транзакция записала объект x в момент u_0, то в промежутке времени $[u_0,u]$ объект x не имел не зафиксированной версии, иначе либо к моменту u эта версия оставалась бы не зафиксированной, и следовательно транзакция t была бы оборвана, либо эта версия могла быть зафиксирована в промежутке $[u_0,u)$ и тогда транзакция t аналогично была бы оборвана. Отсутствие не зафиксированной версии x на промежутке времени $[u_0,u]$ означает, что в $T_{w(x)}$ нет транзакции, временной интервал которой в пересечении с $[u_0,u]$ был бы не пуст кроме t. Аналогично, любая другая транзакция t_1 с start-ts(t_1) ≥ u при попытке записи x была бы оборвана.

Обозначим множество всех транзакций из истории, интервал активности которых пересекается с [a,b], $T_{[a,b]} = \{ t_i \in T \mid [start(t_i), end(t_i)) \wedge [a,b] \}$. Мы получили следующее: $T_{[u_0,u]} \wedge T_{w(x)} = \{T\}$ и $T_{[u,u_1]} \wedge T_{w(x)} = \{T\}$. Дальнейшее доказательство того, что $T_{[u_0,u_1]} \wedge T_{w(x)} = \{T\}$ сводится к доказательству того, что не существует отрезка, не пустого в пересечении с $[u_0,u_1]$ и не пересекающегося с отрезками $[u_0,u]$ и $[u,u_1]$.

Из этого непосредственно следует, что $\forall x \in D, \forall t_1,t_2 \in T_{w(x)} (t_1 <_g t_2 \vee t_2 <_g t_1)$.

ЧТД.

Глава 3. Описание хранилища данных

3.1. Модель данных

HBase является распределенным хранилищем данных с колоночной организацией информации.

Идеи модели данных, высказанные в статье [4] «Bigtable: A Distributed Storage System for Structured Data» оказались очень востребованными, так что результатом выхода статьи стала целая серия проектов-клонов Bigtable, среди них HBase, Hypertable и Cassandra, разрабатываемая Facebook и др.

Модель данных HBase во многом копирует модель данных Bigtable, поэтому мы воспользуемся определением, данным Ченом и остальными в статье [4]: «Bigtable является разреженным распределенным постоянным много размерным сортированным словарем данных»(«Bigtable is a sparse distributed persistent multidimensional sorted map»). Это определение, будучи тяжелым, является кратким и исчерпывающим для модели данных как Bigtable, так и HBase.

Словарь индексируется по строке, ключу колонки и временной метке.

(row:string, column:string, time:int64) → string

Ключи строк являются произвольными строками в HBase. Под строками здесь подразумеваются произвольные наборы байт, т.к. HBase не интерпретирует данные. Каждая операция чтения или записи над строкой является атомарной вне зависимости от количества затронутых колонок.

Данные поддерживаются в лексикографическом порядке. Диапазон строк таблицы динамически разбивается на набор регионов. Регион HR отвечает подмножеству строк в таблице [startkey(HR), endkey(HR)).

Ключ колонки выглядит следующим образом: key = family_id:column_qualifier, он включает в себя идентификатор семейства колонок family_id и идентификатор колонки внутри семейства column_qualifier. Семейство колонок описывается во время создания или изменения таблицы, тогда как квалификатор колонки внутри семейства может быть произвольным набором байт, таким образом разные строки могут обладать разным набором колонок. Данные хранятся в файловой системе по семействам колонок. При описании колонки для неё доступно множество опций, таких как максимальное количество версий, есть ли необходимость хранить семейство колонок всегда в оперативной памяти или оно может быть записано на диск, а также необходимо ли сжатие при записи на диск.

В статье [4] набор данных, доступных по паре из ключа строки и ключа колонки называется ячейкой, в данной работе в дальнейшем под термином объект данных мы будем подразумевать именно такую ячейку. Версии объекта доступны по временной метке. При записи объекта в хранилище данных приложение может указать его временную метку.

В статье [4] говорится, что приложения, которым необходимо избежать коллизий, должны самостоятельно генерировать уникальные временные метки.

3.2. Хранение данных

Версии объектов в HBase представляются объектами класса KeyValue. KeyValue является неизменяемым объектом (immutable). KeyValue содержит ссылку на массив из байтов, сдвиг в этом массиве, начиная с которого начинаются байты, принадлежащие данному KeyValue, и длину, которую KeyValue занимает в массиве.

HBase написана на Java, где отсутствуют средства для непосредственной работы с указателями, при этом чтение с диска или запись на диск происходит блоками фиксированного размера, который задается в файле конфигурации HBase. Поскольку время задержки при выполнении пользователем чтения из хранилища данных или записи в него должно быть минимально, необходимо минимизировать количество операций копирования памяти на стороне сервера. Создатели HBase исходя из подобных соображений реализовали считывание версий объектов с диска следующим образом: с диска считывается блок данных и при помощи специального индекса создаются объекты KeyValue для всех версий, которые хранятся в этом блоке. При таком подходе к считыванию KeyValue с диска не происходит «лишних» операций копирования данных в памяти, т.е. HBase предоставляет zero-copy доступ к данным.

KeyValue является объектом-оберткой и предоставляет набор утилитарных функций, в том числе функции для получения частей составного ключа объекта и его значения. Вся необходимая для этого мета-информация находится в самом массиве.

Точно так же как Google Bigtable использует для хранения данных Google File System (GFS), HBase является хранилищем данных, построенным над Hadoop Distributed File System (HDFS). HBase использует HDFS для хранения файлов данных и лога.

Для рутин чтения блока из файловой системы и записи в неё используется класс HFile, содержащий классы HFile.Reader и HFile.Writer, предоставляющие функциональность чтения и записи соответственно. В свою очередь каждый из этих объектов использует класс TFile, входящий в API HDFS.

В [4] описан формат файла SSTable, используемый Bigtable для хранения данных в файловой системе GFS. SSTable представляет собой постоянный упорядоченный словарь данных, в котором ключами и значениями являются произвольные наборы байтов. Интерфейс SSTable предоставляет функции поиска значения по ключу и функции работы с итератором. SSTable на диске представляется в виде набора блоков фиксированной длины, последний из которых является блоком индекса. Когда открывается файл SSTable, блок индекса загружается в память, таким образом поиск на диске происходит за одно считывание с диска.

TFile является отражением SSTable для HDFS.

3.3. Описание API

Для администрирования HBase, создания и удаления таблиц, управления их схемами используется HBaseAdmin. После того как таблица была однажды создана клиент может обращаться к ней при помощи класса HTable.

Добавление данных происходит построчно, для этого клиент создает экземпляр класса Put, который содержит идентификатор строки, временную метку, а также набор квалификаторов колонок, в соответствие каждому из которых ставится

объект KeyValue. Иначе говоря, Put оборачивает набор KeyValue, относящийся к одной строке. Для того чтобы добавить этот набор KeyValue в хранилище данных, клиент передает созданный экземпляр Put как аргумент в метод HTable put().

Аналогично операции вставки, для того чтобы прочесть данные, соответствующие строке, клиент определяет идентификатор строки, набор колонок и необязательную временную метку в объекте Get, который передает в метод HTable get(). Результатом выполнения операции чтения get() является объект класса Result, который содержит список прочтенных KeyValue.

Клиент также имеет функциональность для работы со сканерами, которые предоставляют доступ к данным наподобие курсора. Для получение сканера ResultScanner клиент создает экземпляр класса Scan и передает его методу getScanner(). Scan позволяет задать интервал идентификаторов строк [startKey, endKey), набор колонок, из которых будет производится выборка данных, интересующий временной интервал для фильтрации KeyValue по временным меткам, а также фильтр, который является наследником класса Filter, и может использоваться для предикатной фильтрации строк. Объект ResultScanner, который является результатом вызова getScanner(), предоставляет функции next() для получения экземпляра Result, соответствующего следующей строке или количеству строк, указанному в аргументе, и close() для закрытия сканера.

3.4. Организация кластера

Проект HBase включает в себя ПО для серверной стороны и клиентской стороны.

Кластер HBase состоит из одного сервера-мастера и множества серверов регионов. Основное назначение мастера - присвоение регионов серверам регионов во время запуска кластера, а также балансирование нагрузки при расщеплении или слиянии регионов. Кроме того мастер следит за тем, все ли сервера находятся в состоянии онлайн, и в случае, если произошло падение одного из серверов регионов, распределяет регионы, за которые отвечал упавший сервер, по другим серверам. Сервера регионов могут динамически добавляться к кластеру, мастер несет ответственность за их обнаружение и перераспределение нагрузки в кластере. Кроме этого мастер управляет изменениями схемы данных.

Данные хранятся на серверах регионов, сервера регионов ответственны за непосредственное выполнение операций над данными, управление расщеплением и слиянием регионов и т.д.

Из сказанного выше ясно, что мастер не занимается адресацией запросов клиентов, но отвечает за управление распределением регионов по серверам данных. Адресация происходит при помощи Zookeeper.

Zookeeper — высоко доступный координационный сервис для распределенных приложений. Он предоставляет набор примитивов, на которых могут строится более высокоуровневые сервисы синхронизации распределенных приложений.

Кластер серверов, на которых запущен Zookeeper, называется кворумом.

Клиент подключается к кворуму Zookeeper, в Zookeeper хранится адрес сервера данных, на котором находится специальный регион -ROOT-. Этот регион в свою очередь содержит адреса всех регионов специальной таблицы .META., в

которой хранится метаинформация о регионах таблицы, в операциях над данными которой заинтересован клиент, и их адресах. Таким образом, в HBase как и в Bigtable для хранения адресов регионов используется схема, похожая на трехуровневое B+ дерево.

Получив из таблицы .META. адрес нужного сервера данных, клиент работает непосредственно с ним.

3.5. О репликации в HBase

В распределенным системах часто применяется механизм репликации — использования множественных копий сервера (называемых репликами) для увеличения доступности и производительности. В HBase репликации подвергаются регионы, при этом используется механизм основной копии (Primary Copy). Один из серверов, имеющих реплику региона, выбирается активным для данного региона — основной копией (publisher), все операции чтения и записи происходят только в основной копии, изменения периодически отсылаются вторичным копия.

Эти изменения распространяются по вторичным копиям в случае когда файлы, отвечающие за хранение региона на диске основной копии обновляются. Иначе говоря, в момент записи на диск нового снимка данных региона (snapshot), снимок распространяется по вторичным копиям. При падении основной копии, одна из вторичных копий, используя журнал (Write Ahead Log) и свой снимок, идентичный снимку, хранившемуся на диске основной копии, производит восстановление.

Такой подход является, наверное, самым простым из подходов к репликации, в свое время он был реализован в таких СУБД как Oracle8i и DB2. Он не позволяет получить хоть какого-то выигрыша в производительности, но направлен на увеличение доступности данных.

Этот подход позволяет при реализации API операций над данными вовсе не задумываться о репликации. Таким образом репликация не будет нас интересовать при описании деталей реализации протокола.

3.6. Описание реализации сервера регионов

HRegionServer — класс, описывающий поведение сервера данных. HRegionServer содержит множество регионов сервера (HRegion) и отвечает за их жизненный цикл. Этот класс также несет ответственность за взаимодействие с другими серверами внутри кластера (преимущественно, с мастером), за сбор статистики, управляет суммарным размером оперативной памяти, используемой регионами. Кроме этого HRegionServer обеспечивает функциональность по отслеживанию активности клиентов. Для внешних клиентов на сервере могут выделяться ресурсы, которые должны быть освобождены, если по прошествии некоторого фиксированного интервала времени клиент не проявлял активности.

Класс HRegionServer не имеет непосредственного отношения к хранению данных и операциям над ними: при получении запроса на чтение или запись объектов строки он получает замок на строку и передает его в соответствующую функцию региона, содержащего эту строку.

HRegion хранит данные региона таблицы, он содержит все колонки для каждой строки региона. Объект HRegion имеет уникальный идентификатор и пространство ключей [startkey, endkey). HRegion агрегирует множество объектов

Store, каждый из которых содержит данные определенного семейства колонок таблицы для строк региона. Аналогично тому как HRegionServer управляет жизненным циклом множества HRegion и суммарным использованием оперативной памяти для хранения данных всеми регионами, так HRegion управляет слиянием и расщеплением Store, а также их суммарным потреблением оперативной памяти.

При получении запроса на чтение или запись объекта строки HRegion проверяет запрос на предмет согласованности ключа строки с диапазоном строк региона, а также набора колонок запроса с набором семейств колонок таблицы и направляет его соответствующим Store. HRegion также отвечает за поддержание лога предварительной записи (Write Ahead Log) и восстановление данных в случае сбоя.

HRegion гарантирует получение замка перед выполнением запроса на чтение или запись даже в случае когда замок не был взят на более высоком уровне (HRegionServer).

Как было сказано выше, Store хранит данные семейства колонок внутри региона. Как регион агрегирует множество Store по именам семейств колонок, так и Store является абстракцией, объединяющей множество объектов StoreFile и один объект Memstore в логическую единицу. На этом уровне абстракции не производится логирования или работы с замками, основным назначением Store является предоставление функциональности по управлению набором StoreFile, процессом их уплотнения и механизмом сливания Memstore на диск.

Memstore хранит недавно записанные версии объектов в оперативной памяти: при добавлении новой версии она попадает в Memstore, затем, когда объем памяти, занимаемый Memstore, превышает некоторое заданное значение, записывается в файловую систему. При чтении версии сначала происходит поиск версий объекта в Memstore, затем, в случае, если в Memstore нет версий объекта, производится поиск на диске.

StoreFile представляет собой абстракцию над файлом на диске и по сути является оберткой над двумя классами: HFile.Reader, предоставляющим функциональность чтения значений из файла HDFS, и HFile.Writer, предоставляющим функциональность записи.

3.7. Минорное и мажорное уплотнение

Как уже было сказано Memstore не растет неограниченно, но по достижении определенного объема записывается в отдельный StoreFile. Этот процесс называется минорным уплотнением[4].

Исполнение входящих операций чтения и записи продолжается во время минорного уплотнения.

Memstore содержит два словаря данных: основной словарь, открытый для чтения и записи и дополнительный, замороженный, доступный только для чтения, этот словарь также носит имя снимка (snapshot). Когда Store запрашивает минорное уплотнение Memstore, Memstore переводит основной словарь данных в замороженный (это производится простой заменой ссылок), а в качестве основного словаря после этого использует созданный пустой словарь. Memstore блокируется на время этой «подмены», но поскольку она происходит очень быстро, это почти не

4 Здесь используется терминология работы [5], где подобный процесс назывался минорным уплотнением.

влияет на производительность сервера регионов в смысле задержки при выполнении операций над данными. Создание StoreFile и запись «замороженного» словаря в него производится в одном из параллельных потоков без каких-либо блокировок, в них нет необходимости, так как этот словарь открыт только для чтения. После этого созданный StoreFile регистрируется в Store, а замороженный словарь данных в Memstore очищается, это требует блокировки Store на запись и чтение, но эта блокировка как и первая является кратковременной.

В случае когда в Store находится слишком много файлов StoreFile, операции чтения производятся медленнее, поскольку для их исполнения необходимо произвести поиск и чтение данных во множестве StoreFile, а затем слияние полученных результатов. Кроме того, при выполнении операции удаления объекта из Store вместо физического удаления данных производится добавление в Memstore специальной записи удаления, которая будет подавлять все ранее записанные версии объекта. Такие записи удаления могут вытесняться вместе со всем содержимым Memstore в файловую систему, и значит, StoreFile, созданные в результате минорного уплотнения могут хранить значительное количество излишней информации: как удаленные версии, так и подавляющие версии для них.

Максимальный размер Memstore определяется в настройках HBase и обычно в несколько раз меньше максимального размера StoreFile. Например, в настройках HBase по умолчанию первый составляет приблизительно 64Мб, тогда как второй 256Мб. Мажорное уплотнение предназначено для слияние несколько таких StoreFile — продуктов минорного уплотнения в один, не содержащий избыточных данных.

Глава 4. Реализация

4.1. Информация о состоянии транзакций

Протокол, истории генерируемые которым удовлетворяют критерию (SI), был описан в предположении, что для каждой версии объекта x по идентификатору транзакции, записавшей его, доступна информация о её состоянии и временных метках начала и конца. Информация о состоянии транзакции может быть инкапсулирована в несложном объекте, который в нашей реализации имеет имя TransactionStateDesc.

В локальном случае (случае единственного сервера хранилища данных) для быстрого доступа к TransactionStateDesc по id транзакции может быть использован любой словарь данных, отображающий ключ-идентификатор в значение — объект класса TransactionStateDesc, например, хеш-таблица. Подобный словарь в нашей реализации имеет имя TransactionStateKeeper.

Для реализации глобального протокола для случая распределенного хранилища данных нам потребуется распределенный вариант TransactionStateKeeper.

Интерфейс TransactionStateKeeper может быть разделен на две части: методы читающие состояние транзакции и методы, записывающие состояние транзакции. Минимальный необходимый набор методов записи:

beginTransaction(long id, long start_ts),

abortTransaction(long id, long abort_ts) и

commitTransaction(long id, long commit_ts).

Каждый из этих методов в распределенном случае не может быть локальным и требует реализации через RPC (Remote Procedure Call). Для этого аналогичным набором методов должен обладать каждый сервер регионов. Реализация этих методов для сервера должна заключаться в их перенаправлении локальному экземпляру TransactionStateKeeper.

Пусть клиент начинает транзакцию t, для этого ему необходимо зафиксировать время старта транзакции u_0, и при помощи вызова метода beginTransaction(id(t), u_0) сервера S_i известить о начале транзакции каждый из серверов $S_1,...,S_n$, которые участвуют в исполнении транзакции. При этом он должен известить S_i о начале транзакции до начала выполнения первой операции над данными на этом сервере.

Перед фиксацией транзакции клиент рассчитывает время глобальной фиксации транзакции u_1 с некоторым запасом, так чтобы произвести commitTransaction(id(T), u_1) на серверах $S_1,...,S_n$ до наступления момента времени u_1.

Таким образом, всегда, когда на сервере S_i необходима информация о состоянии транзакции T, она доступна непосредственно в памяти этого сервера.

Когда мы говорили о доступности информации о состоянии транзакций, мы работали с этой информацией как с локальной и неявно подразумевали также и атомарность операций над состоянием транзакций. Поскольку чтение состояния

транзакции всегда происходит локально, его атомарность не будет нас интересовать. Атомарность метода beginTransaction() не является критичной, т.к. в случае если транзакция не зарегистрирована на сервере, при попытке выполнения операции в рамках этой транзакции транзакция будет оборвана. Атомарность abortTransaction() также не является необходимой. Необходимой является только атомарность commitTransaction(), т.е. необходим атомарный протокол фиксации.

4.2. Фиксация транзакций

Для случая распределенной системы одним из самых известных и наиболее надежных протоколов фиксации транзакций является двухфазный протокол 2PC (Two Phase Commit [3]). Фазы этого протокола:

1. Фаза запроса коммита. Сервер-координатор, ответственный за проведение транзакции в распределенной системе, рассылает всем серверам, принимавшим участие в транзакции запрос о возможности зафиксировать транзакцию. В случае если каждый из серверов ответил утвердительно, координатор принимает решение о фиксации, если нет — об обрыве.
2. Фаза коммита. Сервер-координатор посылает каждому из серверов сообщение о фиксации.

Поскольку система RPC в HBase устроена так, что клиент вынужден ждать окончания каждой распределенной операции, будь-то чтение или запись, клиент может принять решение о фиксации транзакции только после того, как вся работа на серверах данных завершена.

Таким образом, если провести аналогию с алгоритмом 2PC, к моменту начала фиксации транзакции клиентом фаза запроса коммита уже пройдена.

Начав фиксацию транзакции, клиент должен вызвать метод commitTransaction() на каждом из серверов, и в случае успеха может считать транзакцию зафиксированной. Случай неудачи также возможен, например, к неудаче может привести падение одного из серверов данных. В случае неудачи клиент должен вызвать abortTransaction() на всех серверах, которые зафиксировали транзакцию до наступления момента времени t1 (таким образом, при расчете времени глобальной фиксации необходимо учитывать время доставки сообщения по сети). Вызов метода abortTransaction() на остальных серверах может быть отложен на произвольное время. Тем не менее, его стоит вызвать как можно скорее в целях экономии ресурсов.

Протокол фиксации сходен с 2PC по логике действий, хотя и имеет от него отличия. Этот протокол неизбежно наследует и один из главных недостатков 2PC: для успешного коммита необходимо большое количество сообщений синхронизации, что может сделать процесс фиксации непозволительно долгим.

Этот протокол имеет дополнительный недостаток: в случае падения клиента во время фиксации часть серверов может считать транзакцию зафиксированной, а часть — нет. Решением этой проблемы могло бы стать выделение дополнительного сервера, который взял бы на себя обязанности по фиксации транзакций. Мы не будем учитывать возможность падения клиента во время фиксации транзакции.

4.3. Хранение информации о транзакциях

Выше была высказана идея, что для того, чтобы иметь возможность определить транзакцию, которая произвела обновление объекта, в его поле временной метки можно хранить идентификатор этой транзакции. Для хранения обновлений, которые были зафиксированы довольно давно, удобнее хранить в поле временной метки объекта непосредственно время фиксации, поскольку после фиксации транзакции это единственная необходимая информация о ней. Это дает незначительный выигрыш во времени, поскольку во время выполнения операций записи или чтения нет необходимости выполнять поиск записи (TransactionStateDesc) в контейнере (TransactionStateKeeper). Что более важно, мы получаем возможность безопасно «забыть» о транзакции: нет необходимости хранить информацию о состояниях транзакций «вечно».

Таким образом, в системе одновременно могут присутствовать два типа объектов KeyValue: KeyValue, имеющие в качестве временной метки время фиксации транзакции, и KeyValue, которые вместо этого хранят идентификатор транзакции. Такое разделение между объектами KeyValue естественно было организовать следующим образом: на диске (в Store файлах) хранятся KeyValue с временной меткой момента фиксации транзакции, в памяти (Memstore) — с идентификатором транзакции.

Такое решение также позволяет считать, что количество записей в объектах TransactionStateKeeper относительно не велико, поскольку такой объект хранит только информацию о транзакциях, касавшихся региона, которые активны или были зафиксированы относительно недавно. Значит, экземпляры класса могут полностью хранится в оперативной памяти и нет необходимости в записи на диск информации о транзакциях (за исключением логирования).

Выше также обсуждались механизмы минорного и мажорного уплотнения в HBase. В нашей реализации во время минорного уплотнения, когда содержимое Memsore выталкивается на диск, необходимо выталкивать на диск только обновления зафиксированных транзакций, попутно подменяя временные метки в соответствующих объектах KeyValue на время фиксации транзакций.

Преимуществом такого подхода является то, что информация о состоянии транзакций в принципе не попадает на диск: на диске не могут оказаться обновления оборванных или активных транзакций, только зафиксированных. Получаемые файлы не отличаются от создаваемых HBase, благодаря этому не приходится заботиться о специальной реализации мажорного уплотнения.

4.4. *Описание реализации сервера регионов*

В секции 3.6 был описан набор классов, которые реализуют основную функциональность сервера регионов. В данной работе реализованы наследники этих классов, все они имеют префикс SI, который будет отличать класс-наследник от класса-родителя.

Как было сказано выше HRegionServer не занимается непосредственным исполнением операций над данными, но является менеджером регионов. Информация о состоянии транзакций, которую содержит объект TransactionStateKeeper и функциональность, которую он предоставляет, по своему смыслу более присуща региону, чем серверу регионов, поэтому TransactionStateKeeper находится на уровне региона. SiRegionServer несет в себе

очень небольшую добавочную функциональность отношению к HRegionServer: SiRegionServer направляет вызовы функции старта, фиксации и обрыва транзакций соответствующему региону, кроме того, он отвечает за проверку того, чтобы транзакция была зарегистрирована в регионе перед вызовом функции региона get() или put() от её имени.

В документации файла Store сказано, «Блокировки и транзакции[5] обрабатываются на более высоком уровне», тем не менее, львиную долю функциональности по реализации нашего протокола вопреки логике создателей HBase содержат два класса: SiStore и SiMemstore. SiRegion содержит только логику реализации протокола верхнего уровня, а также собственный экземпляр TransactionStateKeeper, к методам которого он перенаправляет все запросы старта, обрыва и фиксации транзакции. Ссылкой на этот экземпляр также будут обладать все объекты более низкого уровня, которые обслуживают SiRegion: его набор SiStore и SiMemstore.

SiRegion после предварительного оформления запроса Get направляет его нужным регионам и затем оборачивает результат объектом Result. При вызове put() SiRegion вызывает у нужного SiStore `checkFirstCommiterWins( long transactionId, Put put )`, и в случае, если объект Put успешно прошел проверку свойства «первый фиксирующийся преуспевает» перманентно записывает его в хранилище данных. Все обновления сперва попадают в SiMemstore (вместо Memstore), поэтому перед записью SiRegion изменяет временные метки всех KeyValue, содержащихся в Put, на id транзакции, от имени которой записываются обновления.

4.5. Выполнение операции get() в SiStore.

В функции get() создается объект специального класса SiQueryMatcher, именно этот объект отвечает за фильтрацию KeyValue при их чтении из файла в SiStoreFile или из оперативной памяти в SiMemstore.

Родителем этого класса в HBase является QueryMatcher. Ключевым методом QueryMatcher является match(KeyValue), который для данной версии объекта позволяет определить удовлетворяет ли она запросу. SiQueryMatcher обладает тремя режимами работы, которые определяют способ обработки поля временной метки в KeyValue:

- IGNORE — версии объекта не фильтруются по временной метке, необходим для того, чтобы получить все версии объекта, например, при проверке «первый фиксирующийся выигрывает»;
- COMMIT_TIME — временная метка версии обрабатывается как время коммита транзакции, записавшей KeyValue, этот режим используется во время обработки KeyValue, полученных из файла;
- TRANSACTION_ID — временная метка KeyValue воспринимается SiQueryMatcher как идентификатор создавшей его транзакции, этот режим используется при обработке KeyValue, полученных из SiMemstore.

При работе SiQueryMatcher в режиме COMMIT_TIME метод match(KeyValue) выдает положительный результат только в случае, если время фиксации транзакции, записавшей KeyValue, меньше времени старта текущей транзакции. В режиме

5 В HBase на время выполнения каждой операции над объектом (или объектами) строки блокируется вся строка. Под транзакциями подразумеваются вероятно «транзакции на уровне строки».

работы TRANSACTION_ID в дополнение к этому проверяется была ли транзакция, записавшая KeyValue зафиксирована. Кроме того match(KeyValue) в этом режиме выдаст положительный результат в случае, если KeyValue записан текущей транзакцией.

При помощи объекта SiQueryMatcher затем производится выборка значений из SiMemstore. Для этого в SiMemstore производится обход KeyValue, содержащихся в основном и «замороженном» словарях данных, в случае если объект KeyValue проходит фильтрацию SiQueryMatcher он добавляется в результирующий набор.

Для каждого полученного KeyValue создается копия, временная метка которой содержит время старта текущей транзакции, если KeyValue было создано текущей транзакцией, или, в противном случае, время фиксации транзакции. После, к этому набору добавляются KeyValue, полученные при выборке в объектах StoreFile.

Полученный набор версий подлежит дальнейшей обработке: в этом наборе для каждого объекта может присутствовать множество версий, из которых нужно выбрать созданную последней. На данный момент в нашей реализации выбор позднейшей версии производится хешированием KeyValue по имени столбца и выборе из подмножества KeyValue с одинаковым значением хеш-функции версии с наибольшим значением временной метки.

Полученный таким образом набор отвечает требованиям версионной функции h_0, описанным в 2.3.

4.6. Проверка «первый фиксирующийся выигрывает» в SiStore

Возможен случай, когда транзакция T1 записала версию объекта после старта текущей транзакции T0 и была зафиксирована до момента проверки FCW («first-commiter-wins»). При этом также нельзя исключать возможности, что после фиксации T1 содержимое SiMemstore и вместе с ним обновления T1, были вытеснены в файл.

В итоге необходима проверка FCW для KeyValue, которые находятся как в SiMemstore, так и в файловой системе. Обе эти проверки производятся в SiStore.

Получение версий из SiMemstore производится при помощи объекта SiQueryMatcher в режиме IGNORE. С диска версии должны быть получены в режиме COMMIT_TIME, это дает возможность на раннем этапе отсеивать версии, транзакции, записавшие которые были зафиксированы до старта T0, и которые, следовательно не могут привести к нарушению FCW.

При проверке версий, полученных из файла, алгоритм проверяет не созданы ли они позже времени начала транзакции. Успешный результат проверки KeyValue, находящихся в SiMemstore кроме этого гарантирует отсутствие активных версий каждого объекта.

4.7. Вытеснение содержимого SiMemstore на диск

В пункте 3 описывается решение о хранении KeyValue в StoreFile и в SiMemstore в различном виде: KeyValue, находящиеся в SiMemstore хранят в поле временной метки идентификатор создавшей транзакции, тогда как в StoreFile хранятся только зафиксированные обновления, во временной метке которых записано время фиксации транзакции.

Для того чтобы воплотить такое решение в жизнь, необходимо было изменить алгоритм минорного уплотнения.

При описании минорного уплотнения в HBase упоминалась фаза, во время которой Store производит обход замороженного словаря данных Memstore, и последовательно добавляет в созданный StoreFile KeyValue. Потребовалось следующее изменение: добавление KeyValue в StoreFile происходит только в случае, если транзакция, создавшая KeyValue, была зафиксирована. При этом на диск добавляются не непосредственно KeyValue, находившиеся в «замороженном» словаре, а их копии с скорректированными временными метками.

Версии, записанные активными транзакциями должны быть возвращены в «замороженный» словарь данных по окончании этой фазы, оборванными транзакциями — удалены. Поскольку запись «замороженного» словаря в файл происходит без каких-либо блокировок и одновременно с этим может происходить чтение из «замороженного» словаря в SiMemstore, непосредственное удаление KeyValue из «замороженного» словаря на этом этапе может привести к некорректному чтению. Для того чтобы избежать этого, обновления активных транзакции сохраняются в специальный список. По окончании этой фазы, SiStore будет заблокирован, после этого также под блокировкой на чтение и запись в SiMemstore происходит замена старого «замороженного» словаря на новый (пустой), в который сразу же добавляются обновления активных транзакций из списка. Далее созданный файл добавляется в набор файлов SiStore и SiStore снимает собственную блокировку.

Такие изменения в алгоритме минорного уплотнения влекут за собой увеличение интервала времени, когда SiStore заблокирован на чтение и запись. Временной промежуток увеличивается на время добавления обновлений активных транзакций в новый «замороженный» словарь. Физически происходит добавление некоторого количества ссылок из списка в пустой словарь, вероятно, устроенный как дерево. Временной интервал блокировки из-за этого может увеличиться во много раз, но тем не менее должен остаться приемлемо коротким.

Глава 5. Эксперименты

5.1. Окружение

Эксперименты проводились на ноутбуке с процессором Intel Pentium Dual Core T4200 2.0 ГГц и размером оперативной памяти 2 Гб.

HBase является распределенным хранилищем данных. Кластер серверов, может обнаруживать аномалии невозможные для одиночного сервера. По этой причине эксперименты, проводимые на единственном сервере, могут не отражать результатов аналогичных экспериментов для распределенного случая. Тем не менее, в предположении, что распределенная система свободна от отказов всех видов (как отказов серверов, так и соединений), некоторые результаты тестов можно обобщить и на случай использования протокола в полноценном кластере.

Тестом, результаты которого нельзя обобщить на распределенный случай, является тест производительности системы. Нельзя сказать, что тест будет показывать схожие результаты в распределенном случае, даже если сделать дополнительное предположение, что сообщения в сети доходят за ограниченное время. Время доставки сообщений в сети может превышать время выполнения локальной транзакции на несколько порядков. В реализации протокола используются дополнительные сообщения для оповещения о изменении состояния транзакции, это может повлечь заметное увеличение времени исполнения транзакций относительно уровня событийной согласованности. Не смотря на это, в данной работе приводится такое сравнение производительности. Оно может быть полезно, для того чтобы понять насколько «болезненным» в смысле увеличения времени отклика является реализация нашего протокола для сервера регионов.

5.2. Описание базового эксперимента

Вывод о применимости протокола в программных проектах стоит делать, рассмотрев, какие преимущества и недостатки несет реализация протокола. Необходимо выяснить в какой степени приходиться поступиться временем задержки при использовании предложенного протокола, и насколько сильные гарантии мы получаем. Нельзя забывать, что протокол, отдавая приоритет скорости исполнения, может показывать большое количество обрывов. Для того чтобы охарактеризовать нашу реализацию SI с точки зрения каждой из этих позиций, был проведен ряд тестов, каждый из которых построен на следующем базовом эксперименте.

БД для каждого из экспериментов состояла из N строк, в каждой из которых находился один элемент данных типа double, изначальное которого было равно 1.

Базовый эксперимент состоит в запуске M одинаковых транзакций на N элементах данных в 30 потоков.

Действия каждой транзакции состоят в следующем:

- транзакция произвольным образом выбирает из диапазона 0,...,N три индекса i_1, i_2, i_3;
- значение элемента данных в строке с номером i_1 транзакция уменьшает вдвое, а элементы данных, находящиеся в строках с индексами i_2, i_3 увеличивает на четверть значения этого элемента.

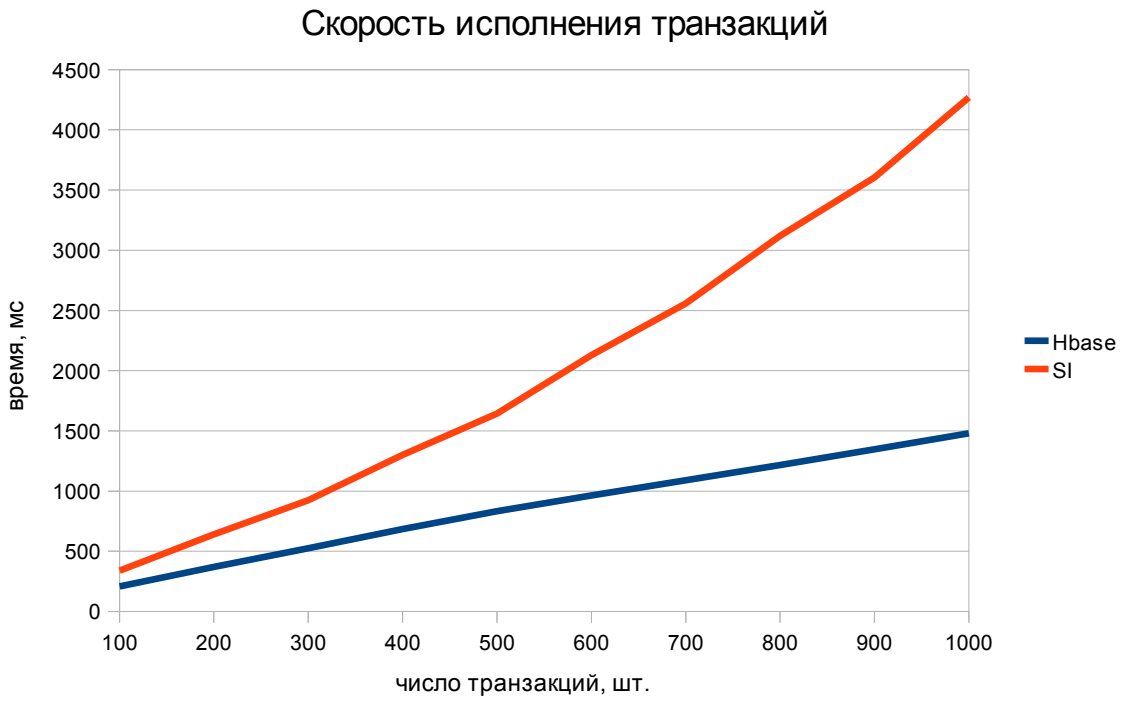

Рис. 1: Сравнение производительность HBase и реализации SI

5.3. Сравнение производительности

Первый эксперимент состоял в запуске M транзакций в БД, описанной выше, и измерении времени исполнения этого набора. При этом число элементов данных в БД N было фиксировано и равно 1000.

В HBase отсутствуют транзакции, для тестирования были написаны специальные задания, производящие те же действия, что и вышеописанная транзакция. График на Рис. 1 показывает зависимость времени исполнения транзакций (заданий для HBase), от их количества M.

Для краткости мы будем далее называть нашу реализацию протокола в HBase HBase SI.

HBase SI демонстрирует время выполнения набора транзакций, большее времени исполнения такого же количества заданий в HBase в 2-3 раза. Такое падение производительности не является катастрофическим1. Стоит отметить, что в HBase SI имели место обрывы транзакций, на графике отражается зависимость времени от числа зафиксированных транзакций.

5.4. Процент зафиксированных транзакций

Для измерения этого показателя был проведен тот же тест, в котором варьировалось число элементов данных N, при этом измерялся процент зафиксированных транзакций для фиксированного общего числа транзакций M, равного 1000. График этой зависимости можно увидеть на Рис. 2.

На графике мы наблюдаем быстрый рост процента зафиксированных транзакций для N < 900, так при запуске 1000 транзакций в 30 потоках на БД содержащей всего 100 ячеек процент зафиксированных транзакций совсем невелик — он составляет 5,37%. Для 1000 элементов данных этот процент составляет 87,97%, после этого, когда число транзакций становится меньшим числа элементов

данных, темп роста значительно снижается и для N = 2000 процент составляет 95,77%.

В этом тесте смоделирована крайняя степень конкурентности: в реальных приложениях трудно встретить одновременную работу 30 транзакций над 100 ячейками. Для приложений даже такая ситуация, которую мы моделируем использованием 2000 элементов данных, достаточно редка, так что частота обрывов в реальных приложениях должна быть очень не велика.

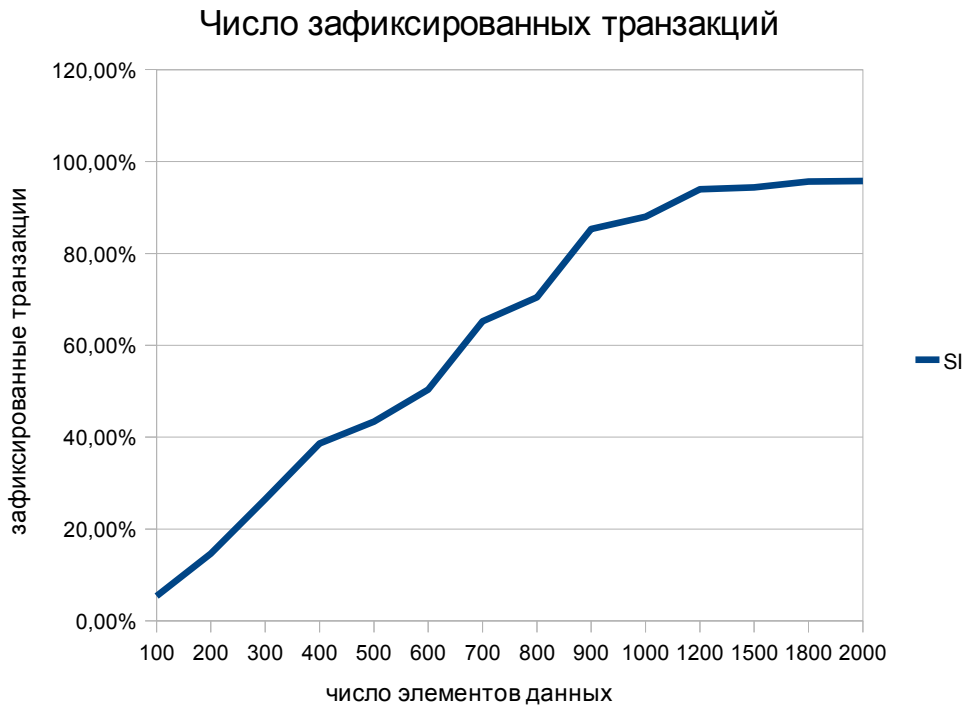

Рис. 2: Процент зафиксированных транзакций в зависимости от числа элементов данных в БД

5.5. Измерение ошибки

Аналогом действий транзакции может быть перевод денег с одного счета на два других. Транзакция сохраняет среднее арифметическое трех упомянутых элементов данных, а значит, если история, состоящая из таких транзакций, сериализуема по конфликтам, то среднее арифметическое N элементов данных составит 1.0. Представим теперь историю, состоящую из подобных транзакций, которая не сериализуема по конфликтам. По выполнении такой истории, вычислив отклонение среднего арифметическое N элементов от 1.0 получим некоторую характеристику «порчи» данных.

Рассмотрим историю, состоящую из таких транзакций, удовлетворяющую критерию SI. Множество записи транзакции совпадает с множеством чтения. Поскольку FCW гарантирует сериализуемость по записи, мы получаем сериализуемость по конфликтам этой истории. Более того, в общем случае транзакция, исполняемая на уровне изоляции SI, не может наблюдать никаких феноменов, происходящих над теми элементами данных, которые находятся в её множестве записи. Поскольку в приложениях очень естественным является обновление элемента данных после чтения его значения, это делает уровень SI

«неожиданно сильным».[6]

HBase SI действительно не демонстрировал никакой ошибки в среднем арифметическом.

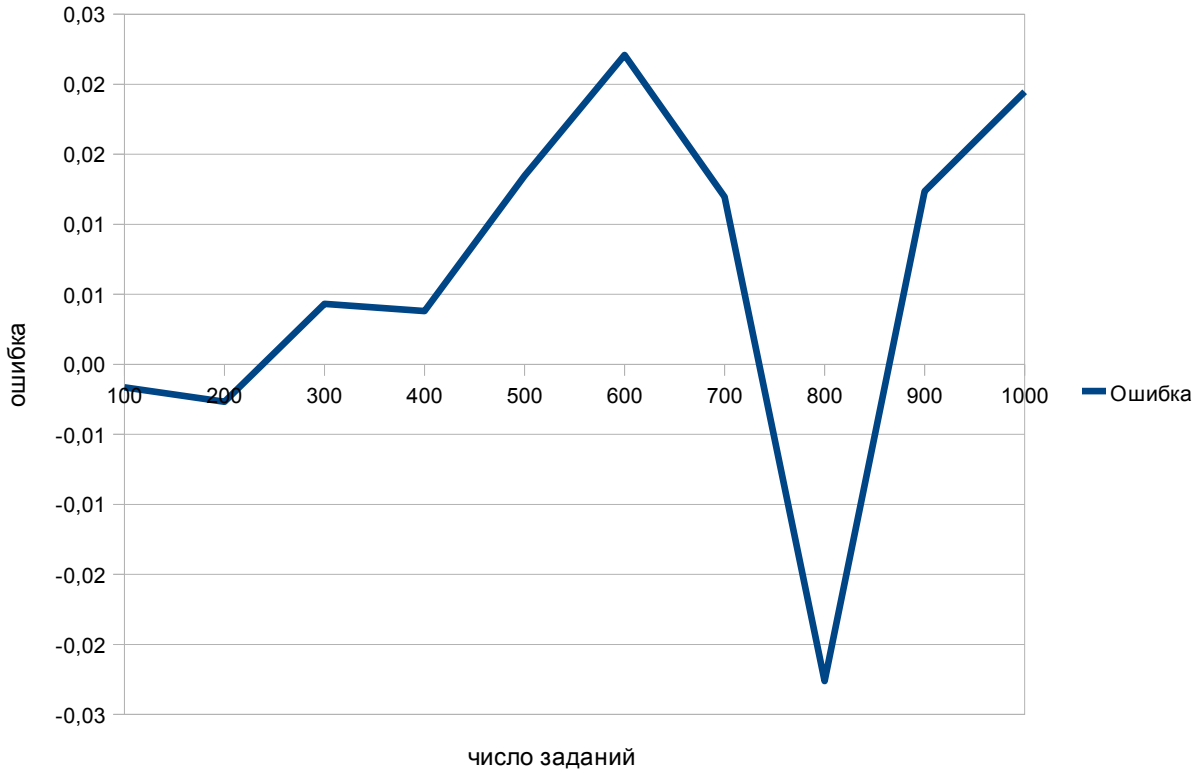

Рис. 3: Ошибка в среднем арифметическом, HBase (событийная согласованность)

На графике Рис. 3 изображен один из вариантов измерений ошибки в среднем арифметическом для различного числа заданий (ось X). Измерялась случайная величина, поэтому на графике представлена не усредненная величина, измеренная после проведения нескольких тестов, а измерения одного теста.

5.6. Аномалия Write Skew

В предыдущем эксперименте HBase SI показала очень хороший результат. Уровень SI очень распространен в индустрии как хороший компромисс между силой предлагаемых гарантий и пропускной способностью протокола. В работе [5] авторы показали, что уровень изоляции SERIALIZABLE стандарта ANSI SQL 92, определенный как уровень изоляции, который не может испытывать ни одного из феноменов описанных в стандарте, в действительности, если понимать определения феноменов буквально, является более слабым, чем уровень сериализуемости по конфликтам.

В работе [5] этот уровень называется ANOMALY SERIALIZABLE из-за интерпретации феноменов как произошедших аномалий. Авторы также заметили,

6 Цитата из [5]

что уровень ANOMALY SERIALIZABLE слабее SI, при этом SI слабее SERIALIZABLE[7]. Кроме этого, авторы представили дополнительную аномалию, которой нет в стандарте ANSI SQL 92 и которую может испытывать SI. Эта аномалия носит имя Write Skew и заключается в следующем: транзакция T1 читает x и y, значения которых согласуются с ограничением C(), затем транзакция T2 читает x и y, обновляет x и фиксируется; после этого T1 записывает y. В этом случае ограничение C() может быть нарушено. Это не единственная аномалия, которая может возникать в SI, но мы обратимся именно к ней в следующем эксперименте.

В предыдущем эксперименте множество записи транзакции совпадало с множеством чтения, поэтому она не могла претерпеть Write Skew. В данном эксперименте, мы будем использовать следующий вид транзакции:

- транзакция T читает значения всех N элементов данных в БД и вычисляет их сумму S;
- затем произвольным образом выбирается элемент данных и его значение увеличивается на половину среднего арифметического всех элементов данных S / 2N.

Заметим, что в случае, когда M таких транзакций выполняются последовательно, каждая из них увеличивает значение S на величину S / 2N, иначе говоря

$$S_{i+1} = S_i + S_i / 2N, \text{ или}$$

$$S_{i+1} = p * S_i, \text{ где } p = 1 + 1 / 2N.$$

После выполнения M транзакций $S_M = p^M S_0$ или $S_M / S_0 = p^M$.

Аналогичное соотношение мы получим в случае уровня изоляции сериализуемости по конфликтам. В случае SI в следствии многократного проявления аномалии Write Skew в истории из M транзакций мы получим $S_M / S_0 = p^M$, но в случае, когда уровень изоляции слабее, мы получим, что $S_M / S_0 \leq p^M$. Поскольку S_M / S_0 экспоненциально зависит от числа транзакций в случае последовательного их исполнения, интуитивным кажется, что ошибка, порожденная проявлением Write Skew также должна распространяться экспоненциально быстро от числа произошедших аномалий.

Введем величину $\varphi(M) = \log_p (S_M / S_0) - M$. В случае последовательного выполнения транзакций $\forall$ M $\varphi(M) = 0$. В случае использования SI этот логарифмический показатель характеризует число произошедших в истории аномалий Write Skew.

На Рис. 4 представлены график φ в HBase и HBase SI. Заметим, φ для HBase SI близок к линейному, это говорит о линейном росте числа аномалий в зависимости от числа транзакций и экспоненциальном росте ошибки.

По графику также видно, что показатель φ(M) для HBase SI в несколько раз больше соответствующего значения для HBase. Это необходимо происходит поскольку транзакция на уровне изоляции SI читает данные, зафиксированные на момент её начала, в результате этого она может «упускать» обновления, которые в аналогичной ситуации обнаруживает выполняющееся задание в HBase, где используются только кратковременные замки на чтение и запись.

7 Разница между ANOMALY SERIALIZABLE и SERIALIZABLE в терминологии работы [5] заключается именно в способе интерпретации феноменов: ANOMALY SERIALIZABLE соответствует буквальному понимаю их определений, SERIALIZABLE — расширенному (см. секцию 2.3).

Как видно, и в случае HBase и в случае HBase SI имеет место экспоненциальный рост ошибки. Следует отметить, что разница между ошибками HBase и HBase SI также имеет экспоненциальный рост в зависимости от числа транзакций (заданий).

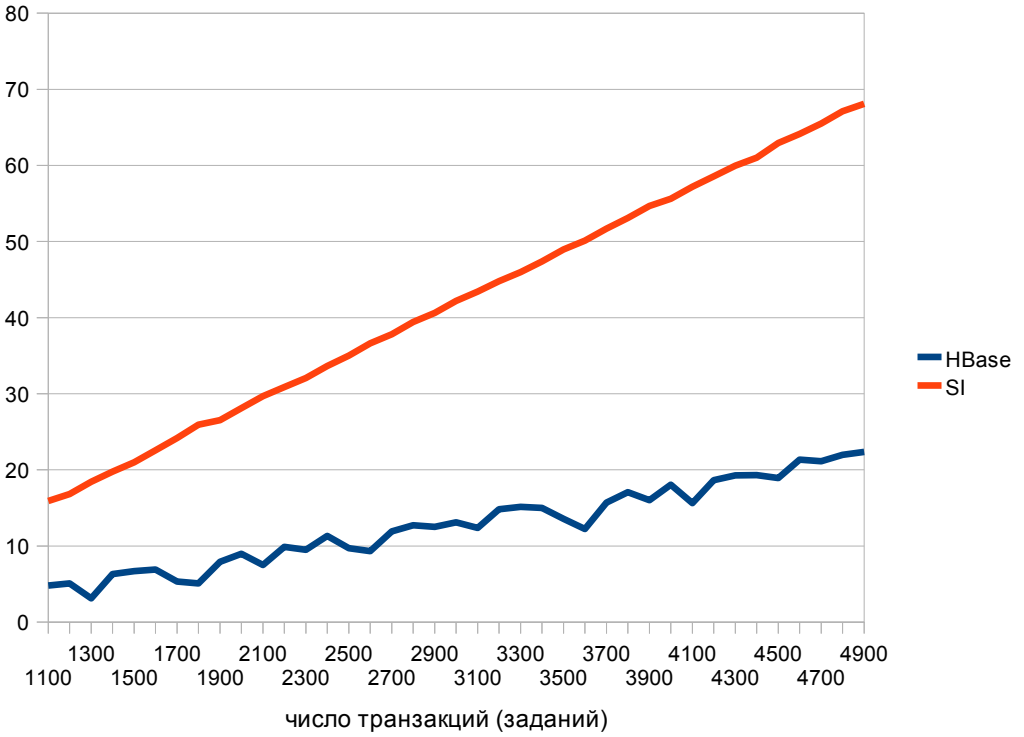

Рис. 4: График зависимости φ.

Заключение

В данной работе были достигнуты следующие результаты:

- предложен неблокирующий протокол, реализующий уровень SI;
- доказана корректность разработанного протокола;
- протокол реализован в распределенном хранилище данных HBase.

Проведенные тесты недостаточны, для того чтобы сделать положительный вывод о применимости протокола в распределенном случае, поскольку были проведены на единственном сервере. Тем не менее, для случая единственного сервера они показали вполне приемлемые результаты.

Уровень SI очень популярен в коммерческих СУБД. Он предоставляет достаточно сильные гарантии согласованности, так в Oracle этот уровень изоляции скрывается за названием SERIALIZABLE. Проведенный эксперимент демонстрирует ситуацию, в которой определение критерия SI необходимо влечет появление дополнительных аномалий при работе с данными относительно уровня событийной согласованности[8].

Ссылки

[1] G. Weikum, G. Vossen, *Transactional Information Systems*

[2] R. Schenkel, G. Weikum, N. Weißenberg, X. Wu, *Federated Transaction Management with Snapshot Isolation*

[3] C. Mohan, B. Lindsay, and R. Obermarck, *Transaction Management in the R* Distributed Database Management System*

[4] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, *Bigtable: A Distributed Storage System for Structured Data*

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil Patrick, O'Neil, *A Critique of ANSI SQL Isolation Levels*

[6] S. Harizopoulos, D.J. Abadi, S. Madden, M. Stonebraker, *OLTP Through the Looking Glass, and What We Found There*

8 Вне зависимости от протокола, реализующего SI, SI показывает большее число аномалий, чем система, использующая только кратковременные замки.