Заглушка для нумерации страниц

Содержание

1 Введение

При разработке современного веб-сервиса необходимо уделять особое внимание его быстродействию и надежности, особенно когда речь идет о веб-сервисах, помогающих в ведении бизнеса (взаимодействие с банками, электронная отчетность и т.д.). Отказ подобных систем может повлечь финансовые потери клиента, а порой и поставить под угрозу весь его бизнес. Например, не вовремя или неверно сданная отчетность может повлечь серьезные штрафы со стороны государства, вплоть до отзыва лицензии.

Отказ системы может произойти по разным причинам, в частности, из-за сетевых неполадок или аварийного завершения работы одной из компонент. Например, если сервис, отвечающий за поиск по пользовательским данным, перестанет отвечать, то многие клиенты потеряют возможность полноценной работы в системе, поскольку лишатся возможности быстро находить нужную информацию. Эту проблему можно решить с помощью запуска сервиса на нескольких серверах: в таком случае если одна из его реплик выйдет из строя, остальные смогут продолжить обработку запросов. Также данное решение может избавить нас от проблем с недоступностью из-за сетевых неполадок: недоступность одной из реплик не остановит работу системы.

По тем же причинам необходимо использовать базу данных, поддерживающую запуск на нескольких серверах и толерантную к периодической недоступности одного или нескольких серверов. В зависимости от типов запросов, которые база данных должна уметь обрабатывать, и нагрузок, которые она должна выдерживать, можно выбрать одно из множества доступных решений: любое NoSQL хранилище, например *MongoDB*[1] или *Berkeley DB*[2], или же MySQL с master-slave репликацией.

При разработке многопоточных приложений часто встает задача предоставления исключительного доступа к какому-либо разделяемому ресурсу. Рассмотрим простой пример: снятие денег со счета с двух банкоматов.

[1] https://www.mongodb.org

[2] http://www.oracle.com/us/products/database/berkeley-db/

Предположим, что на счете лежит 200 условных единиц и два человека хотят снять с этого счета по 150 единиц в один и тот же момент времени. Алгоритм снятия денег в первом приближении достаточно прост: узнать сколько денег на счете, и, если средств достаточно, выдать требуемую сумму и списать ее со счета. Если этот алгоритм будет выполняться одновременно двумя потоками без каких-либо блокировок, то может возникнуть следующая ситуация: оба потока одновременно запросят остаток на счете, поймут, что средств достаточно и каждый снимет со счета по 150 условных единиц, оставив на счете отрицательное значение, что в действительности неприемлемо. Если же перед началом выполнения поток может сказать, что сейчас он будет работать со счетом и никакой другой поток этого делать не должен, то такой ситуации получиться не может.

Большинство современных языков программирования содержит механизмы, позволяющие решить эту задачу, но только в рамках одного процесса, в некоторых языках реализованы механизмы для решения этой задачи для разных процессов, запущенных на одном сервере. Однако когда речь идет о процессах, запущенных на разных серверах, возникает задача распределенной блокировки — необходим механизм, позволяющий решать проблему исключительного доступа к ресурсу с разных серверов.

Существует множество готовых решений данной задачи. Существует *Chubby*[3] от Google, но оно является проприетарным и его исходного кода нет в открытом доступе. Есть *Apache ZooKeeper*[4], это решение лежит в свободном доступе и активно развивается, но оно обладает достаточно высоким порогом входа: более или менее удобный и надежный клиент для ZooKeeper есть только для Java, а реализация своей обертки — очень трудоемкая задача. Можно решить задачу с использованием транзакционной базы данных (MySQL, MSSQL), но в некоторых случаях это решение будет иметь недостаточно высокую производительность.

В работе [1] задача о распределенной блокировке появляется как под-

[3]http://research.google.com/archive/chubby.html

[4]https://zookeeper.apache.org

задача при построении распределенной очереди на базе данных *Apache Cassandra*[5]. Однако в ходе использования предложенного алгоритма была отмечена достаточно важная проблема — падение производительности в случае попытки взятия одной и той же блокировки достаточно большим количеством потоков.

В рамках работы была разработана новая версия алгоритма, не имеющая проблемы со снижением производительности при большом количестве потоков.

[5]http://cassandra.apache.org

2 Apache Cassandra

Apache Cassandra — распределенная система управления базами данных, относящаяся к классу NoSQL. За счет отказа от реляционности и транзакционности в NoSQL системах обеспечиваются возможности хорошей горизонтальной масштабируемости и репликации. Это направление в компьютерных науках сейчас находится в стадии активного развития, и очень многие компании стали использовать такие базы для решения большого числа задач. Cassandra изначально была разработкой Facebook, однако в 2009 году было решено отдать проект фонду Apache Software.

В первом приближении на Cassandra можно смотреть как на следующие сущности (в порядке вложенности):

- кластер — множество серверов, на которых хранится множество баз данных;
- пространство ключей — база данных, множество таблиц;
- семейство колонок — таблица, множество элементов;
- колонка — ячейка, хранящая в себе конкретную запись.

Ячейка содержит в себе следующую информацию:

- имя строки, в которой лежит ячейка;
- имя колонки, в которой лежит ячейка;
- набор байтов с хранимой информацией;
- временная отметка ячейки;
- время жизни ячейки.

Фактически семейство колонок является разреженной таблицей, в которой каждая строка содержит множество ячеек, упорядоченных по имени

колонки в лексикографическом порядке. Порядок ячеек в строке — важная особенность хранения данных, являющаяся ключевой для предлагаемого алгоритма.

Cassandra предоставляет множество возможностей для записи и чтения данных, среди них стоит отметить следующие:

- записать одну ячейку;
- удалить ячейку с заданными координатами;
- записать множество ячеек в одно семейство колонок;
- вычитать последовательно из строки заданное количество ячеек, начиная с ячейки с заданным именем колонки.

При удалении ячейки из строки на ее место записывается специальная ячейка, называемая маркером удаления, которая означает, что в данной строке и данной колонке ячейка когда-то существовала, но была удалена.

При синхронизации данных между узлами возникает потребность в выборе лучшей ячейки из нескольких с одинаковыми координатами. В этом случае ячейки сравниваются с помощью следующего алгоритма:

Листинг 1: Алгоритм GetBestCell(cellA, cellB)

```
1.  Если ячейки cellA и cellB имеют разные временные отметки:
2.    Вернуть ячейку с большей временной отметкой
3.  Иначе:
4.    Если одна из ячеек является маркером удаления:
5.      Вернуть ячейку, являющуюся маркером удаления
6.    Иначе:
7.      Посчитать MD5 от набора байтов с хранимой информацией
8.      Вернуть ячейку, MD5 которой меньше
```

Кластер Cassandra состоит из множества узлов. Каждая строка из семейства колонок хранится не на всех узлах, а только на определенном числе из них. Это число называется $ReplicationFactor$ и указывается при конфигурировании базы. Выполнение запроса на чтение или запись данных в строке возможно на любом узле, при этом он будет выполнять роль координатора запроса и взаимодействовать с узлами, отвечающими за хранение этой строки.

Cassandra поддерживает разные стратегии чтения и записи, отличающиеся друг от друга количеством узлов, от которых будет ожидаться ответ. Мы используем стратегии $WriteQuorum$ для записи и $ReadQuorum$ для чтения. При использовании стратегии $WriteQuorum$ в момент записи данных в конкретную строку координатор пошлет запрос на все узлы, отвечающие за хранение данной строки, и дождется ответа хотя бы от $\lfloor \frac{ReplicationFactor}{2} + 1 \rfloor$ из них. Каждый узел, получивший запрос на запись, попытается обновить данные в ячейке с использованием алгоритма $GetBestCell$. Аналогично в момент чтения координатор запроса дождется ответа хотя бы от $\lfloor \frac{ReplicationFactor}{2} + 1 \rfloor$ узла и выберет лучшую ячейку с использованием алгоритма $GetBestCell$.

Существует утверждение, известное как *теорема CAP*, которое заключается в следующем: при реализации распределенных вычислений возможно обеспечить не более двух из трех следующих свойств:

- согласованность (consistency) — в любой момент времени один и тот же запрос к любому узлу в случае успеха даст один и тот же ответ;
- доступность (availability) — любой запрос к распределенной системе завершается корректным откликом;
- устойчивость к разделению (partition tolerance) — расщепление распределенной системы на несколько изолированных секций не приводит к некорректности отклика от каждой из секций.

Cassandra гарантирует, что использование стратегий $ReadQuorum$ и $WriteQuorum$ дает согласованность и устойчивость к разделению. В частности, это означает, что после записи ячейки в некоторую строку результаты запросов на чтение этой строки обязательно будут содержать эту ячейку. Этот факт будет использован далее при доказательстве корректности алгоритмов.

3 Требования к интерфейсу сервиса распределенной блокировки

Сформулируем требования к интерфейсу сервиса распределенной блокировки. Фактически необходимо реализовать две стратегии блокировки:

- обязательная блокировка — подразумевает, что блокировка будет взята в любом случае: поток в любом случае дождется освобождения нужного ресурса и обязательно возьмет блокировку;
- мягкая блокировка — подразумевает, что блокировка может быть и не взята: поток попытается взять блокировку, но, если ресурс уже занят другим потоком, во взятии блокировки будет отказано, после чего поток сможет сам решить, как ему обработать эту ситуацию.

Разница достаточно проста и прозрачна — первая стратегия применяется в случае, когда оба потока обязательно должны совершить свое действие (например, снять деньги со счета пользователя), вторая же применима, если нам достаточно, чтобы хотя бы один из потоков совершил свое действие (например, отправить SMS-уведомление пользователю).

Опишем данные требования на языке C#.

Листинг 2: Описание интерфейса

```
public interface IRemoteLocker
{
  IRemoteLock GetLock(string lockId); // обязательная блокировка
  bool TryGetLock(string lockId, out IRemoteLock remoteLock); // мягкая блокировка
}

public interface IRemoteLock : IDisposable
{
  public string ThreadId { get; } // идентификатор потока
  public string LockId { get; } // идентификатор ресурса
}
```

Здесь *lockId* — строковый идентификатор ресурса, доступ к которому необходимо получить. Метод *Dispose* в реализациях интерфейса *IRemoteLock* должен освобождать блокировку, в таком случае пользоваться сервисом блокировок будет достаточно удобно с помощью конструкции *using*.

Листинг 3: Использование конструкции using

```
...
using(remoteLocker.GetLock(lockId))
{
  // действия в блокировке
}
```

4 Обзор существующего решения

Рассмотрим алгоритм, описанный в [1]. В качестве основы для реализаций стратегий блокировок предлагается алгоритм, который пытается взять блокировку и возвращает в качестве результата одно из трех состояний:

- Success — поток успешно взял блокировку;
- AnotherThreadIsOwner — другой поток уже владеет блокировкой;
- ConcurrentAttempt — поток не смог взять блокировку, так как другой поток попытался сделать это одновременно с ним.

Заведем в Cassandra две таблицы — основную и теневую. В теневой таблице будет происходить борьба потоков за право взятия блокировки: поток будет записывать туда свой идентификатор и проверять, один ли он в таблице путем полного ее вычитывания, и если он действительно один, то блокировка будет считаться взятой. В основной таблице будет фиксироваться факт взятия блокировки тем или иным потоком путем записывания в таблицу идентификатора потока-владельца. Алгоритм выглядит следующим образом:

Листинг 4: Алгоритм Cassandra.TryLock(lockId, threadId)

```
 1. Взять ячейки из основной таблицы из строки lockId
 2. Если ячейка одна:
 3.   Если columnKey = threadId:
 4.     Вернуть Success
 5.   Иначе:
 6.     Вернуть AnotherThreadIsOwner
 7. Добавить ячейку в теневую таблицу
 8. Взять ячейки из теневой таблицы
 9. Если ячейка в теневой таблице одна
10.   Если нет ячеек в основной таблице
11.     Добавить ячейку в основную таблицу
12.     Удалить свою ячейку из теневой таблицы
13.     Вернуть Success
14. Удалить свою ячейку из теневой таблицы
15. Вернуть ConcurrentAttempt
```

С использованием этого алгоритма достаточно просто реализовать алгоритм обязательной блокировки — просто будем запускать алгоритм $Cassandra.TryLock$ до тех пор, пока блокировка не будет захвачена.

Листинг 5: Алгоритм Cassandra.GetLock(lockId, threadId)

```
 1. Присвоить attempt = 1
 2. Вызвать Cassanrda.TryLock(lockId, threadId)
 3. Если Success:
 4.    Закончить
 5. Если AnotherThreadIsOwner
 6.    Подождать случайный промежуток времени от 0 до 1000 мс
 7.    Перейти к 2
 8. Если ConcurrentAttempt:
 9.    Подождать случайный промежуток времени от 0 до 50*attempt мс
10.    Присвоить attempt = attempt + 1
11.    Перейти к 2
```

По факту потоки будут бороться друг с другом за право захвата блокировки, не пытаясь договориться друг с другом. Таким образом, с увеличением количества потоков увеличивается время, необходимое для взятия блокировки хотя бы одним потоком. Еще один недостаток этого алгоритма менее очевидный. Представим себе ситуацию, когда два потока последовательно много раз пытаются взять одну и ту же блокировку. Как только один из них сможет ее захватить, второй поток не сможет взять блокировку после первого же ее освобождения: возможно в момент отпускания блокировки первым потоком второй поток будет находиться в состоянии ожидания, в таком случае первый поток тут же этим воспользуется и захватит блокировку повторно. Таким образом второй поток не сможет захватить блокировку пока не выйдет из состояния ожидания в нужный момент. Это приводит к тому, что потоки будут выполнять свои действия неравномерно — сначала большую пачку действий выполнит первый поток, потом второй, затем первый вернет себе лидерство и так далее. В идеале хотелось бы несколько более справедливый алгоритм, который не будет неявным образом отдавать предпочтение тому или иному потоку на протяжении долгого времени.

5 Описание алгоритмов

Желание получить алгоритм, справедливо распределяющий доступ к разделяемому ресурсу между потоками, подводит к достаточно простой идее использования некоторого подобия очереди потоков. Для каждого идентификатора разделяемого ресурса будем хранить в Cassandra две строки: первую будем использовать в качестве подобия очереди (далее «очередь»), во второй будет храниться идентификатор потока, владеющего ресурсом (далее «основная строка»). При добавлении потока в очередь мы будем добавлять новую ячейку в строчку с очередью, при этом имя колонки этой ячейки будет подсчитано следующим образом:

Листинг 6: Определение имени колонки для ячейки в очереди

```
1. Положить в переменную currentTime строковое значение текущего времени в микросекундах
2. Добавлять к значению currentTime ведущие нули до тех пор, пока его длина не станет равной 20
3. Назначить именем колонки конкатенацию значений currentTime и threadId
```

В силу того, что ячейки в строке хранятся в лексикографическом порядке имен колонок, раньше всех в очереди окажется тот поток, который будет добавлен в очередь раньше, в случае равенства времен ближе к вершине очереди окажется поток с лексикографически меньшим идентификатором. При добавлении потока в основную строку именем колонки будем считать идентификатор потока.

Стоит отметить, что очередь получится не очень честной. На практике бывает сложно добиться того, чтобы время на всех серверах было одинаковым. Предположим, что у потока A время отстает от времени потока B на одну секунду. Если поток A поставит себя в очередь, а через полсекунды поток B поставит себя в очередь, то B может оказаться в ней раньше, так как время, записанное в префиксе имени колонки ячейки B, окажется меньше чем у ячейки, записанной потоком A. Частично эта проблема решается реализацией сервиса времени, который по запросу будет отдавать неубывающее время. Его использование избавит нас от проблемы с несинхронным временем, однако останется еще одна проблема: операция взятия времени и записи ячейки с соответствующим именем колонки не

является атомарной, и, следовательно, между этими двумя действиями может произойти задержка (например, запуск сборки мусора или передача управление другому потоку процесса). Соответственно до сих пор возможна ситуация, когда в очереди появляется поток со временем меньшим? чем у всех остальных в очереди. Но, во-первых, это будет происходить гораздо реже, во-вторых, если у потока возникла задержка и он встанет в очередь перед всеми другими потоками, то он быстро захватит блокировку и отработает, после чего будет вставать в очередь вместе со всеми, и вечного превосходства над другими потоками он не получит.

5.1 Алгоритм обязательной блокировки

Алгоритм обязательной блокировки устроен следующим образом: поток ставит себя в очередь на блокировку, ждет пока он окажется первым в очереди и записывает себя в основную строку. Если после этого в основной строке оказалась одна ячейка, то блокировка считается взятой, в противном случае поток удалит записанную ячейку из основной строки и повторит итерацию еще раз.

Листинг 7: Алгоритм Cassandra.GetLock(lockId, threadId)

```
1.  Добавить поток в очередь
2.  Если наш поток - первый в очереди:
3.    Добавить ячейку в основную строку
4.    Если в основной строке есть только одна запись:
5.      Закончить
6.    Иначе:
7.      Удалить свою ячейку из основной строки
8.      Перейти к шагу 2
9.  Иначе:
10.   Перейти к шагу 2
```

Лемма 1. *Если два потока A и B одновременно запишут по одной ячейке в одну строку в Cassandra и после этого прочитают ее, то хотя бы один из них увидит в строке обе записи.*

Доказательство: Предположим, что это не так, и каждый из потоков прочитал строчку, в которой смог найти лишь свою запись. Из консистентности Cassandra следует, что если поток A не смог увидеть запись потока B,

то поток B еще не успел записать свою ячейку, то есть A сделал запись строго раньше B. Но поток B также не увидел запись потока A, следовательно, B сделал запись строго раньше A. Получено противоречие, следовательно, хотя бы один из потоков увидит обе записи. □

Теорема 1. *В ходе выполнения алгоритма Cassandra.GetLock несколькими потоками блокировка не будет взята более чем одним потоком одновременно.*

Доказательство: Предположим, что два потока одновременно смогли взять блокировку. Это означает, что они оба записались в одну и ту же строку, и после ее прочтения каждый из них увидел лишь одну запись, что противоречит лемме 1. Следовательно, блокировку может быть взята лишь одним из них. □

Теорема 2. *Если каждый поток будет завершать действие в блокировке за конечное время, то в ходе выполнения алгоритма Cassandra.GetLock несколькими потоками каждый поток получит блокировку за конечное время.*

Доказательство: При попытке захватить блокировку $Cassandra.GetLock$ поток в самом начале запишется в очередь и не удалится из нее до тех пор, пока блокировка не будет захвачена. Если поток в некоторый момент времени окажется первым в очереди и в очередь никто не сможет записаться перед этим потоком, то блокировка будет захвачена, как только текущий владелец ее освободит, а остальные потоки увидят, что они не первые в очереди. Освобождение блокировки текущим владельцем произойдет сразу после завершения его действий в блокировке, то есть за конечное время. Остальные потоки поймут, что они не первые в очереди, в момент очередного исполнения строки 2, то есть тоже за конечное время.

Осталось показать, что поток окажется первым в очереди за конечное время. Предположим, что это неправда, и поток никогда не окажется в очереди первым. Это означает, что перед этим потоком в любой момент

времени найдется хотя бы один поток, следовательно, потоки будут постоянно записываться с меньшей или равной временной отметкой, чем у данного потока. Однако это противоречит неубыванию времени в системе. Следовательно, поток окажется первым в очереди за конечное время. □

5.2 Алгоритм мягкой блокировки

Алгоритм мягкой блокировки действует аналогично алгоритму обязательной блокировки, но в случае если наш поток не является первым в очереди, блокировка считается не взятой и возвращается соответствующий вердикт.

Листинг 8: Алгоритм Cassandra.TryGetLock(lockId, threadId)

```
1.  Если в основной строке есть хотя бы одна запись
2.    Вернуть false
3.  Добавить поток в очередь
4.  Если наш поток - первый в очереди:
5.    Добавить ячейку в основную строку
6.    Если в основной строке есть только одна запись:
7.      Вернуть true
8.    Иначе:
9.      Удалить свою ячейку из основной строки
10.     Перейти к шагу 4
11. Иначе:
12.   Удалить свой поток из очереди
13.   Вернуть false
```

Теорема 3. *В ходе выполнения алгоритма Cassandra.TryGetLock несколькими потоками блокировку сможет взять не более одного потока.*

Доказательство: Доказательство этой теоремы полностью повторяет доказательство теоремы 1, так как условие успешного взятия блокировки то же самое, что и в алгоритме обязательной блокировки: поток записался в основную строку и оказался в ней один. □

Теорема 4. *В ходе выполнения алгоритма Cassandra.TryGetLock несколькими потоками хотя бы один поток захватит блокировку.*

Доказательство: Заметим, что за конечное время система придет в состояние, в котором первый поток в очереди не изменится: в силу неубывания времени перед всеми в очередь смогут встать либо потоки, имеющие

ту же отметку времени, что и текущий первый в очереди, либо потоки, у которых произошла задержка между взятием времени и постановкой в очередь. Очевидно, что этих потоков конечное число, следовательно, время их постановки в очередь тоже конечно. Предположим, что блокировку не удалось захватить ни одному из потоков. В частности, это означает, что блокировку не смог взять поток, стоящий первый в очереди. По построению алгоритма этот поток будет выполнять строки 4 —10 до тех пор, пока в основной строке будет находиться записи других потоков. Так как поток является первым в очереди, остальные потоки не будут пытаться записаться в основную таблицу, как только увидят этот поток в очереди. Следовательно, рано или поздно основная строка будет пуста, отсюда следует что первый в очереди поток успешно возьмет блокировку. □

5.3 Аренда и освобождение блокировки

Алгоритмы *Cassandra.GetLock* и *Cassandra.TryGetLock* позволяют захватить блокировку, при этом в результате захвата блокировки в очереди и в основной строке остаются колонки, порожденные потоком. Для освобождения блокировки достаточно удалить эти колонки из строчек. Эту логику достаточно поместить в метод *Dispose* у реализации интерфейса *IRemoteLock*, в таком случае при использовании конструкции *using* в конце исполнения кода внутри блокировки разделяемый ресурс освободится.

Давайте представим себе следующую ситуацию: поток успешно взял блокировку, после чего завершил свою работу таким образом, что метод *Dispose* не отработал. В итоге система находится в таком состоянии, что ни один поток не сможет взять блокировку на этот ресурс, и освободить его тоже никто не сможет, то есть блокировка захвачена навечно.

Можно решить эту проблему достаточно просто с использованием Cassandra: при записи ячеек в очередь и в основную таблицу будем выставлять для этих ячеек время жизни в размере N секунд, а в случае успешного взятия блокировки создадим дополнительный фоновый поток, который раз в $N/2$ секунд будет перезаписывать ячейки в строчках. Таким образом в

случае аварийного завершения процесса, поток которого захватил блокировку и не успел ее освободить, ячейки исчезнут из таблицы, так как вместе с потоком, удерживающим блокировку, прекратит свою работу и поток, отвечающий за продление аренды.

Таким образом при успешном взятии блокировки нам необходимо создать фоновый поток, который будет время от времени перезаписывать существующие ячейки в строчках, а при освобождении блокировки нужно будет остановить этот фоновый поток и удалить записанные ячейки из очереди и из основной строки.

5.4 Сервис времени

Для решения проблемы с рассинхронизированным временем на серверах был реализован сервис с несколькими репликами, который в ответ на последовательные запросы отдает гарантированно неубывающие значения времени.

Заведем в Cassandra ячейку $GlobalTime$, в которой будем хранить максимум из локальных времен запущенных реплик сервиса. Каждая реплика с некоторой периодичностью (например, раз в 500 миллисекунд) будет пытаться обновить значение времени в Cassandra, используя алгоритм $Cassandra.TryUpdateGlobalTime$

Листинг 9: Алгоритм Cassandra.TryUpdateGlobalTime

```
1.  Взять локальное значение времени, положить в переменную localTime
2.  Прочитать из Cassandra ячейку GlobalTime, положить в переменную globalTime
3.  Если localTime > globalTime
4.    Перезаписать ячейку GlobalTime со значением localTime с временной отметкой localTime
```

Теорема 5. *Значение в ячейке GlobalTime не будет убывать со временем.*

Доказательство: Предположим, что найдутся два последовательных момента времени $A < B$ таких, что в момент времени A значение $Time_A$ в ячейке $GlobalTime$ будет больше значения $Time_B$ в момент времени B. Это означает, что в промежуток времени между и было произведено обновление ячейки $GlobalTime$ со значением меньшим, чем $Time_A$. По построению

алгоритма значение времени в ячейке $GlobalTime$ в любой момент времени совпадает с ее временной отметкой, следовательно, при изменении значения ячейки на меньшее нарушилось бы правило неуменьшения временной отметки ячейки. Следовательно, $Time_A$ не может быть больше $Time_B$. □

Для ответа на запрос текущего времени достаточно вернуть текущее значение ячейки $GlobalTime$.

6 Реализация решения

Описанные алгоритмы были реализованы на языке C#. Код покрыт тестами, проверяющими корректность реализации по основным сценариям:

- предоставление блокировки по одному ключу лишь одному потоку одновременно;
- предоставление блокировки по нескольким ключам нескольким потокам одновременно;
- продление аренды блокировки;
- автоматическое снятие блокировки в случае аварийного завершения работы потока.

Также был реализован инструмент для измерения эффективности алгоритмов блокировок, в частности было проведено сравнение нового алгоритма со старым.

Для измерения был развернут кластер Cassandra из трех узлов, каждый узел был запущен на сервере с процессором Intel Xeon CPU E5-2620 2.00 GHz и 32 гигабайтами оперативной памяти. Также были запущены три реплики сервиса времени, каждая на сервере с процессором Intel Xeon CPU E5-2620 2.00 GHz и 19,5 гигабайтами оперативной памяти.

В каждом эксперименте принимали участие несколько процессов, которые брали блокировку по одинаковому для всех процессов ключу фиксированное количество раз одновременно. Измерялось время ожидания осво-

бождения блокировки и время взятия каждой блокировки с момента начала эксперимента.

Всего было проведено 6 экспериментов:

- 1 процесс берет блокировку 5000 раз с помощью старого алгоритма обязательной блокировки;
- 1 процесс берет блокировку 5000 раз с помощью нового алгоритма обязательной блокировки;
- 3 процесса берут блокировку 5000 раз с помощью старого алгоритма обязательной блокировки;
- 3 процесса берут блокировку 5000 раз с помощью нового алгоритма обязательной блокировки;
- 5 процессов берут блокировку 5000 раз с помощью старого алгоритма обязательной блокировки;
- 5 процессов берут блокировку 5000 раз с помощью нового алгоритма обязательной блокировки;

Ниже приведены результаты измерений времен ожидания освобождения блокировки.

Таблица 1: Среднее время взятия блокировки

Количество процессов	Новый алгоритм	Старый алгоритм
1	25 мс	15 мс
3	53 мс	56 мс
5	107 мс	106 мс

Таблица 2: Максимальное время взятия блокировки

Количество процессов	Новый алгоритм	Старый алгоритм
1	146 мс	39 мс
3	537 мс	155817 мс
5	605 мс	502441 мс

Таблица 1 показывает, что оба алгоритма при наличии конкурентных запросов в среднем показывают практически одинаковые результаты. Из таблицы 2 видно, что максимальное время ожидания освобождения блокировки при наличии конкурентных запросов в случае использовании старого алгоритма может достигать нескольких минут, в случае же использования нового алгоритма оно не превышает секунды.

Графики, приведенные ниже, показывают время взятия процессами очередной блокировки при использовании старого и нового алгоритма. Вдоль оси X указан номер попытки взять блокировку, вдоль оси Y — количество миллисекунд, прошедших с начала эксперимента. Каждая линия соответствует одному процессу, прохождение линии через точку с координатами (x, y) означает, что попытка взять блокировку с номером x успешно завершилась через y миллисекунд с момента начала эксперимента.

Рисунок 1 показывает, что новый алгоритм позволяет потокам брать блокировки равномерно, не отдавая последовательно множество блокировок одному потоку, тем самым максимальное время взятия блокировки минимизируется. Рисунок 2 объясняет, почему старый алгоритм дает такие большие значения максимального времени взятия блокировки: фактически у каждого потока есть несколько больших промежутков времени, в течение которых он непрерывно захватывает множество блокировок, не отдавая ее другим потокам.

Рис. 1: Моменты взятия блокировок новым алгоритмом

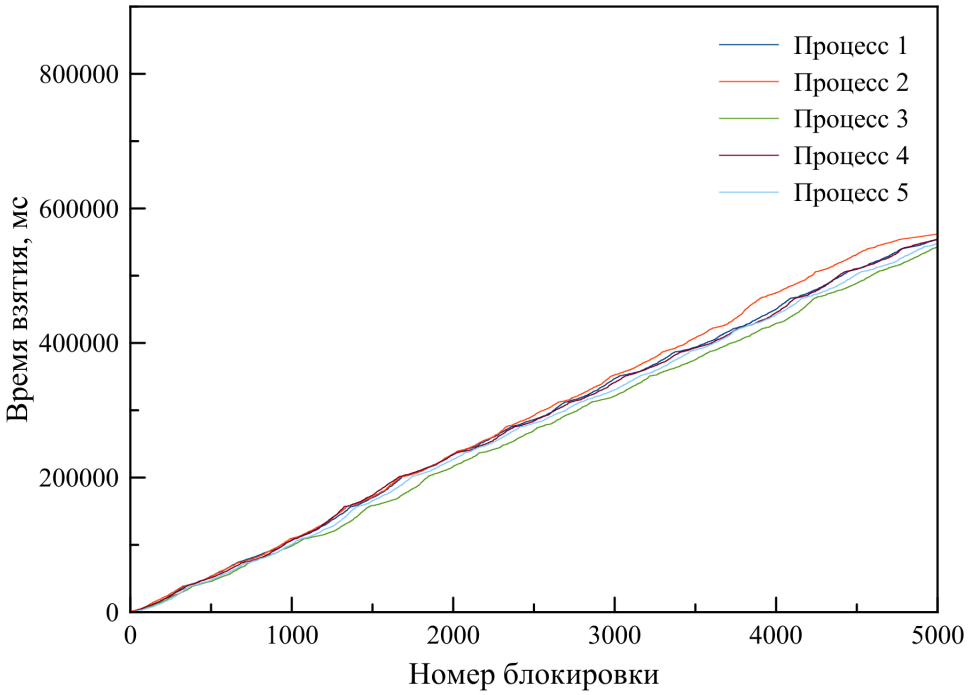
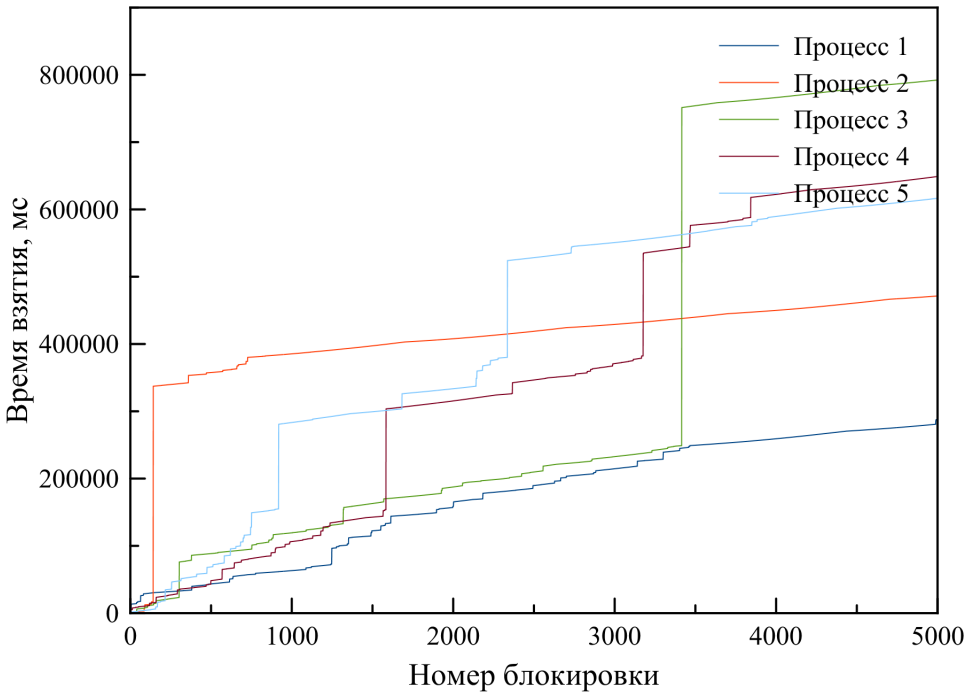


Рис. 2: Моменты взятия блокировок старым алгоритмом



7 Заключение

7.1 Проблемы подхода

В ходе проектирования и реализации алгоритма был отмечен ряд проблем:

- рассинхронизация времени на серверах;
- долгое отпускание блокировки в случае аварийного завершения работы потока-владельца;
- потеря связи с Cassandra в момент продления аренды.

Проблема с рассинхронизацией времени интересна в разрезе разных наборов серверов.

В случае рассинхронизации на серверах с репликами сервиса времени может возникнуть следующая ситуация: если реплика, время на которой сильно убежало вперед относительно других, выйдет из строя, то значение времени в Cassandra не будет обновляться до тех пор, пока реплика не возобновит свою работу или пока время на других репликах не станет больше значения, записанного в Cassandra. В такой ситуации сервис времени будет отдавать одно и то же значение времени на протяжении нескольких секунд или даже минут, что повлияет на работу алгоритмов блокировки: потоки будут вставать в случайное место в очереди, так как имена колонок ячеек, которые мы будем записывать в очередь, будут отличаться лишь идентификатором потока.

В случае рассинхронизации времени на серверах с репликами Cassandra ситуация более плачевна. В силу внутреннего устройства Cassandra, если время на репликах разъедется больше чем на время жизни ячейки, то ячейка будет тут же удаляться из строки, и корректность работы алгоритма блокировки будет нарушена. Поэтому время жизни ячейки приходится указывать достаточно большим, около 15 минут.

Для избавления от проблем с рассинхронизацией времени на серверах требуется детально разобраться в том, как не давать времени сильно убегать вперед относительно других серверов.

Проблема с долгим отпусканием блокировки заключается в следующем: если поток-владелец блокировки аварийно завершит работу, то освобождение блокировки произойдет только по истечении времени жизни ячеек в очереди и в основной строке, а это время может быть достаточно большим.

Проблема с потерей связи с Cassandra в момент продления аренды заключается в том, что разработанный алгоритм блокировки рассчитывает на стабильную и безотказную работу потока-арендатора блокировки. Однако возможна такая ситуация, что во время выполнения действий в блокировке связь с Cassandra может прерваться, из-за чего продление аренды может не сработать и блокировка освободится. Требуется придумать, как поток, отвечающий за продление аренды, сможет просигнализировать потоку, выполняющему действия в блокировке, что блокировку не удалось продлить и, возможно, она уже захвачена другим потоком.

7.2 Внедрение в проект

Описанные выше проблемы не являются критичными для проекта, в который решение внедряется в данный момент, по следующим причинам:

- Использование Cassandra в проекте обязывает нас следить за тем, чтобы время сильно не рассинхронизировалось. В нашем проекте проблемами рассинхронизации времени на серверах занимается отдел СПС[6]. В данный момент есть гарантия того, разница времен на серверах не превысит 5 минут. Ведутся работы для уменьшения этого числа.

- Аварийное завершение работы потока-владельца происходит достаточно редко, и в этих случаях мы готовы к тому, что остальным потокам придется подождать некоторое время.

[6]Служба поддержки серверов

- Возможная потеря связи с Cassandra в момент очередной попытки продлить аренду также не является критичной проблемой, так как все общие для процессов данные хранятся в том же кластере Cassandra, который используется алгоритмом блокировки. Поэтому если связь с Cassandra будет прервана, то во время выполнения действий в блокировке мы не сможем поменять состояние данных в Cassandra.

Однако внедрение нового решения осложняется тем, что необходимо провести плавный переход со старого алгоритма на новый без остановки сервисов. Для этого на время перехода будет работать комбинированный алгоритм, который будет брать блокировку с использованием нового алгоритма, при этом учитывая, что часть блокировок могла быть захвачена с помощью старого.

Список литературы

[1] *Фоминых Ф.М.* Построение распределенной очереди в условиях БД Cassandra : магистерская работа — Екатеринбург, 2012

[2] *Grinev M. A* Quick Introduction to the Cassandra Data Model [Электронный ресурс]. Режим доступа: http://goo.gl/2tCZ9E

[3] *Burrows M.* The Chubby lock service for loosely-coupled distributed systems [Электронный ресурс]. Режим доступа: http://goo.gl/motBYy

[4] General information on Apache ZooKeeper [Электронный ресурс]. Режим доступа: https://goo.gl/VOMHLm

[5] *DataStax Corporation* What's New in Apache Cassandra^{TM} 1.1? [Электронный ресурс]. Режим доступа: http://goo.gl/0TVLUW

[6] *deBruijn N.G.* Additional comments on a problem in concurrent programming control //Communications of the ACM 10, 3 – 1967, march – P. 137-138