ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
"НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
"ВЫСШАЯ ШКОЛА ЭКОНОМИКИ"
ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ НАУК

Смородинов Александр Андреевич

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Георазпределённые транзакции в YTsaurus

Geodistributed Transactions in YTsaurus

по направлению подготовки 01.04.02 Прикладная математика и информатика

образовательная программа «Современные компьютерные науки»

Научный руководитель
к.ф-м.н, доцент департамента больших данных и
информационного поиска ФКН НИУ ВШЭ

А.Э. Хузиева

Соруководитель
Руководитель службы разработки динамических таблиц
ООО "Яндекс.Технологии"

_____

Р.А. Савченко

Студент

_____

А.А. Смородинов

Москва 2025

Аннотация

В современных приложениях часто используются базы данных (БД), но бывает, что пользовательские данные размещаются в разных локациях (городах, странах или континентах) - для лучшей масштабируемости, отказоустойчивости, или приватности. В таком случае, чем реже система обращается к удалённым данным, тем эффективнее она работает, так как сетевые запросы выполняются с задержкой.

База данных YTsaurus для реализации транзакций использует единый источник времени (для генерации временных меток), из-за которого снижается производительность в георасределённом сценарии использования. Для того, чтобы избавиться от единого источника времени, можно использовать гибридные логические часы (HLC) и генерировать временные метки локально, такой подход реализован например в CockroachDB.

В рамках данной работы были реализованы оба подхода (с единым источником времени и с HLC), написаны тесты для проверки их корректности и производительности в различных конфигурациях. Исходный код написан на C++ и находится в открытом доступе. Помимо кода, были написаны docker и docker compose файлы конфигурации, которые позволяют с помощью одной команды запускать локально кластер из серверов и настраивать сетевую задержку между ними (для симуляции георасределённой БД).

В тексте работы подробно описываются различные подходы для реализации транзакций, их преимущества и недостатки, результаты тестирования производительности, приводится доказательство их корректности.

Тестирование производилось на независимым от YTsaurus (но приближенным к нему) прототипе, в будущем будет необходимо перенести код из прототипа в YTsaurus.

В итоге, был проведён подробный обзор существующих решений и продемонстрирована возможность поддержки в YTsaurus георасределённых транзакций, масштабирующихся лучше, чем использующийся сейчас подход.

Abstract

Modern applications often use databases, but sometimes user data is stored in different locations (cities, countries or continents) - for better scalability, fault tolerance, or privacy. In this scenario, the less often the system accesses remote data, the more efficiently it operates, since network requests are executed with a delay.

The YTsaurus database uses a global time source to implement transactions (for timestamp generation), which reduces performance in a geodistributed use case. In order to get rid of a global time source, hybrid logical clock (HLC) can be used instead, this approach is implemented, for example, in CockroachDB.

In this paper, both approaches were implemented, tests were written to check their correctness and performance in various configurations. The source code is written in C++ and is publicly available. In addition to the code, docker and docker compose configuration files were written, which allow to launch a cluster of servers locally with a single command and configure the network delay between them (to simulate a geodistributed database).

This paper describes various approaches for implementing transactions, their advantages and disadvantages, compares their performance and provides proof of their correctness.

For now, testing was performed on an independently written (but similar in the implementation) prototype. In the future, the code from this prototype could be transferred to the YTsaurus.

As a result, the survey of existing approaches was provided, and the possibility of supporting geodistributed transactions that scale better than currently used approach in YTsaurus was demonstrated.

Содержание

1 Введение 6

1.1 Предметная область 6

1.1.1 Транзакции 6

1.1.2 Шардирование и георасперделённые базы данных 7

1.1.3 Контроль параллельности (concurrency control) 8

1.1.4 Атомарный коммит 10

1.1.5 YTsaurus 10

1.1.6 CockroachDB 10

1.2 Постановка задачи 10

1.2.1 Цели 10

1.2.2 Задачи 11

1.3 Актуальность 11

1.4 Значимость 12

1.5 Полученные результаты 12

1.5.1 Репозиторий 13

1.6 Структура работы 13

2 Существующие работы и решения 14

2.1 Google Spanner 14

2.2 YTsaurus 15

2.3 CockroachDB 15

2.4 YugabyteDB 16

2.5 Отличия от существующих решений 16

2.5.1 Практические отличия 16

2.5.2 Теоретические отличия 17

3 Описание реализованных алгоритмов 18

3.1 Общая теория про concurrency control 18

3.2 Формальное определение сериализуемости 19

3.3 Двухфазная блокировка (2PL) 20

3.3.1 Предотвращение дедлоков 22
3.4 Двухфазный коммит (2PC) 24
3.5 MVCC и Snapshot Isolation 25
3.5.1 MVCC 25
3.5.2 Snapshot Isolation (SI) 26
3.6 Гибридные логические часы (HLC) 28
3.7 Неблокирующее чтение по timestamp 28

4 Детали реализации 30
4.1 Приложение 30
4.2 Репозиторий 30
4.3 Реализованные алгоритмы 30

5 Тестирование корректности и производительности 32
5.1 Описание тестов 32
5.2 Описание тестового стенда 33
5.2.1 Параметры запуска 33
5.2.2 Конфигурация сети 33
5.3 Синхронизация часов 34
5.4 Результаты тестирования 35
5.4.1 1 сценарий 35
5.4.2 2 сценарий 35
5.4.3 3 сценарий 36

6 Заключение 37

7 Библиографический список 38

1 Введение

1.1 Предметная область

1.1.1 Транзакции

Транзакции - это механизм, позволяющий в базе данных (БД) выполнять несколько операций (чтений и записей) атомарно, то есть либо выполнить все операции, либо не выполнить ни одну. Пока транзакция не завершилась, частичные результаты также не должны быть видимы для других транзакций.

У транзакций принято выделять 4 свойства: Atomicity, Consistency, Isolation, Durability (сокращённо ACID) [1].

Про атомарность (atomicity) уже было написано выше, это означает, что не может быть такого, что транзакция выполнена частично (какие-то операции выполнились, а какие-то нет). Транзакцию можно явно отменить, если вызвать abort, либо попытаться закоммитить (commit), но есть возможность, что в системе возникнет конфликт (или ещё какие-то проблемы), и после коммита всё равно транзакция отменится.

В случае успешного выполненного коммита все операции должны быть применены, а в случае неудачного коммита, либо abort - все операции должны быть отменены.

Консистентность (consistency) - это свойство системы, которое должно сохранятся при выполнении транзакций. Например, при переводе денег, общая сумма денег в системе не должна измениться.

Изоляция (isolation) - насколько параллельные транзакции могут друг на друга влиять. Уровень изоляции можно формально определить через гарантии, которые предоставляет система, либо через допустимость “артефактов“ при выполнении транзакций в системе.

Устойчивость (durability) - если транзакция успешно завершилась, то её изменения будут доступны при всех следующих чтениях. Из этого в частности следует, что данные должны храниться на диске, так как оперативная память не переживает перезагрузку системы в случае сбоя. И для хранения данных

нужно использовать несколько реплик, так как одна реплика может отказать.

В данной работе будут рассмотрены только изоляция, атомарность и консистентность. Устойчивость нас не будет интересовать.

1.1.2 Шардирование и георасределённые базы данных

Во многих современных базах данных поддерживается шардирование таблиц, то есть разбиение данных на несколько непересекающихся частей (шардов, или таблетов).

Это бывает необходимо по нескольким причинам:

1. Горизонтальное масштабирование - один сервер может хранить ограниченное количество ключей, поэтому с ростом числа записей необходимо шардирование

2. Локальность данных - данные можно хранить как можно ближе к пользователям, и засчёт этого минимизировать сетевую задержку при обращении к БД

3. Защита конфиденциальной информации - существуют различные законы, ограничивающие обработку пользовательских данных (GDPR в Европе, закон № 152-ФЗ «О персональных данных» в России, и т.д.). Например, согласно GDPR, данные пользователей можно хранить только на территории ЕС, то есть их необходимо хранить в отдельном шарде (таблете) от остальных пользовательских данных.

Транзакции могут быть локальные (затрагивающие только данные в одном таблете), либо распределённые (затрагивающие данные в нескольких таблетах, возможно даже в разных таблицах).

Распределённые транзакции сложнее в реализации, поэтому некоторые БД (например BigTable [2]) поддерживают только локальные транзакции.

В георасределённой базе данных таблеты размещаются в разных локациях. Сетевая задержка (round trip time, RTT) между разными таблетами зависит от

расстояния между серверами и может занимать от единиц до сотен миллисекунд [3].

Для того, чтобы эффективно реализовать георасперделённые транзакции, необходимо минимизировать количество запросов к удалённым серверам и по возможности работать только с локальными данными, так как как походы по сети могут занимать значительное время.

На рисунке 1.1 приведён пример георасперделённой базы данных, с отдельными шардами для каждого континента.

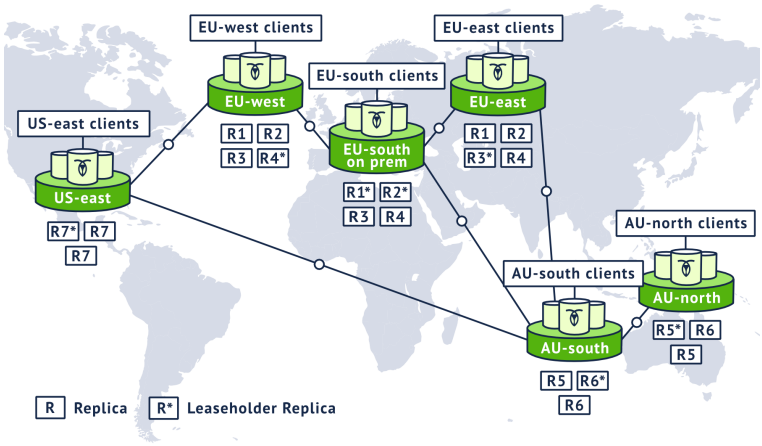


Рис. 1.1: Пример георасперделённой базы данных [4]

1.1.3 Контроль параллельности (concurrency control)

Когда в базе данных выполняется параллельно несколько транзакций, необходим механизм, изолирующий транзакции друг от друга. Если этого не делать, то в системе будут возможны различные артефакты, которые как правило являются нежелательными.

Пример возможного артефакта (dirty read), если не использовать concurrency control:

1. Изначально $x = 0$
2. T1 записывает $x = 1$
3. T2 читает $x = 1$

4. T1 отменяется (вызван abort)

В этом случае, так как T1 отменилась, то $x = 1$ должно тоже отмениться, а значит T2 должна была прочитать $x = 0$. Но это невозможно, так как T2 уже прочитала неправильное значение.

Этот пример показывает, что записи, которые делает транзакция, не должны быть видны до коммита, так как в произвольный момент может произойти abort, и записи нужно будет отменить, хотя их уже прочитали.

Существует целая иерархия уровней изоляции, в зависимости от гарантий, которые они предоставляют, и какие артефакты допускаются (см. рисунок 1.2). Подробнее про это можно почитать на сайте [5] и в книге [6] (2 глава).

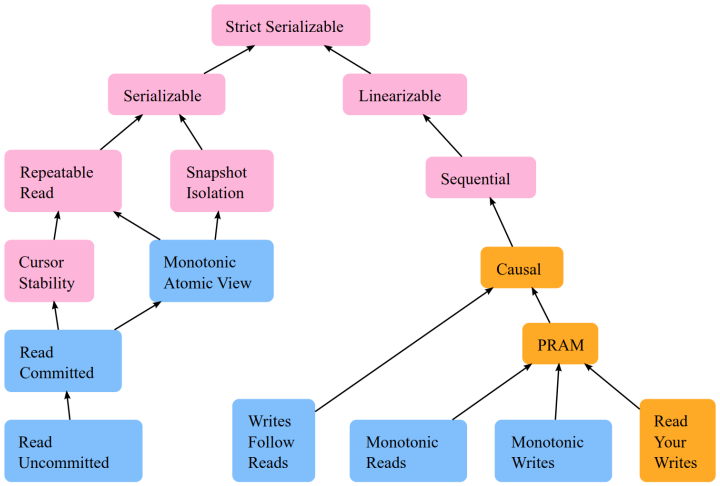


Рис. 1.2: Уровни изоляции [5]

Нас будут в основном интересовать serializable (и немного strict serializable) и snapshot isolation уровни.

Также, существует множество реализаций concurrency control [7], но в этой работе нас будут интересовать только MVCC (multiversion concurrency control) + 2PL (two-phase locking).

При MVCC подходе, каждой записи присваивается временная метка, и это позволяет читать из таблицы по timestamp, не блокируя записи. Временные метрики нужно как-то генерировать, это можно делать разными способами.

1.1.4 Атомарный коммит

Для того чтобы транзакции были атомарными, необходим протокол атомарного коммита. Один из самых популярных - это протокол двухфазного коммита (2PC). Подробнее прочитать про двухфазный коммит и другие атомарные коммиты можно в [8].

1.1.5 YTsaurus

В YTsaurus [9] для генерации временных меток (для MVCC) используется выделенный сервер (timestamp_provider), он является общим на весь кластер. Для выполнения транзакций, приходится ходить по сети до timestamp_provider, что накладывает дополнительные расходы в случае георасперделённой системы (так как timestamp_provider может находиться далеко от текущего таблета или клиента).

1.1.6 CockroachDB

В CockroachDB [4] для генерации временных меток используется HLC - гибридные логические часы [10]. Такой подход позволяет не делать лишних обращений к timestamp_provider, и засчёт этого является более эффективным в георасперделённом сценарии.

1.2 Постановка задачи

1.2.1 Цели

Цель работы - изучить, реализовать и протестировать георасперделённые транзакции в YTsaurus, то есть отказаться от timestamp_provider и использовать HLC (аналогично CockroachDB).

В качестве результата, должен получиться прототип базы данных, в которой георасперделённые транзакции работают более эффективно, по сравнению с исходным протоколом.

1.2.2 Задачи

1. Изучить материалы: [4, 7, 8, 9, 10, 11, 12, 13, 14, 15]
2. Поддержать HLC в протоколе 2PC и (2PL+MVCC)-модели YTsaurus
3. Добавить возможность поднимать кластер с нодами в разных контейнерах, между которыми пакеты ходили бы с большой задержкой
4. Реализовать бенчмарки, показывающие время работы транзакций локальных для каждой локации и георасределённых
5. Добавить тесты, демонстрирующие корректность семантики транзакций
6. Написать подробное описание реализованных алгоритмов, устройства стенда, бенчмарков и тестов
7. Привести результаты экспериментов
8. [Опционально] защититься от clock-skew - если часы в разных локациях сильно разъедутся и проверить эту защиту

1.3 Актуальность

Проблема актуальна, так как в настоящее время всё больше и больше приложений становятся георасределёнными, и для их реализации необходимы БД с эффективными распределёнными транзакциями.

Для разработчиков, транзакции являются очень полезным инструментом, так как позволяют не беспокоиться об отказах, которые могут произойти в процессе выполнения запроса, или о конкурентных исполнениях запросов.

В рамках транзакций, достаточно проверить, что каждая транзакция оставляет систему в консистентном состоянии, тогда по итогу всех транзакций система тоже будет в консистентном состоянии.

Например, в банковских приложениях, переводы денег должны производиться с помощью транзакций (если при переводе с одного счёта списались деньги,

то они должны зачислиться на другой счёт, и наоборот, иначе пользователь будет недоволен тем, что деньги не дошли до получателя, а пропали).

Помимо банковской сферы, практически в любом stateful приложении будут нужны транзакции в том или ином виде.

Для YTsaurus как открытого ПО, отсутствие в ней эффективных геораспределённых транзакций снижает её привлекательность для пользователей, по сравнению с другими базами данных (например CockroachDB).

В рамках Яндекса, несмотря на то, что датацентры находятся только в России и на сравнительно небольшом расстоянии, всё равно было бы полезно по возможности избавиться от лишних походов между датацентрами, так как YTsaurus используется во многих высоконагруженных сервисах Яндекса.

1.4 Значимость

По результатам экспериментов, была показана лучшая эффективность реализации с HLC, при этом не нарушая корректность транзакций. Эти результаты позволят в будущем реализовать в YTsaurus более эффективный подход, сэкономить ресурсы и улучшить SLA сервисов (по времени обработки запросов).

С теоретической точки зрения, работа не является новой, но в ней на практике были реализованы и протестированы два подхода для транзакций и проведено их сравнение. Помимо этого, подробно описываются существующие алгоритмы, обосновывается их корректность, то есть работа имеет ценность как обзорная статья.

1.5 Полученные результаты

1. Код серверной и клиентской части БД, в котором поддержаны:
   (a) Алгоритм двухфазного коммита (2PC), двухфазной блокировки (2PL), мультиверсионное хранилище (MVCC)
   (b) Алгоритм для предотвращения дедлоков между транзакциями (Wait-Die)

(c) Выбор источника временных меток: timestamp_provider, либо HLC

(d) Неблокирующее чтения

2. Конфигурационные файлы для docker и docker compose, позволяющие удобно поднимать кластер из серверов и настраивать сетевую задержку между контейнерами (для симуляции георасределённой системы)

3. Стресс тесты для транзакций, проверяющие производительность и корректность протоколов транзакций. Реализовано два вида тестов с разными типами запросов.

4. Документация кода

5. Описание реализованных алгоритмов

6. Результаты тестирования производительности

1.5.1 Репозиторий

Исходный код проекта находится открытом доступе в репозитории [16].

1.6 Структура работы

В сеции 2 будут описаны существующие работы и решения. В секции 3 - описание реализованных алгоритмов. В секции 4 - детали реализации, в секции 5 - результаты тестирования, в секции 6 - заключение.

2 Существующие работы и решения

2.1 Google Spanner

Google Spanner [17] - это одна из первых систем, добившаяся строгой консистентности и эффективно реализующая геораспределённые транзакции.

Google Spanner использует 2PL (two-phase locking) алгоритм для read-write транзакций, MVCC для чтений.

В качестве протокола двухфазного коммита используется 2PC (двухфазный коммит).

Для генерации временных меток Spanner полагается на специализированное железо. Эта технология называется TrueTime, она состоит из часов, синхронизируемых по GPS, и атомных часов. Протокол позволяет синхронизировать часы с максимальным интервалом неопределённости примерно 7 мс (по данным статьи [17]), что довольно точно, по сравнению с 100-250 мс в случае NTP.

В Spanner поддерживается самый сильный уровень изоляции - external consistency. Также Spanner поддерживает lock-free read-only и snapshot read транзакции.

Чтобы гарантировать изоляцию, в Spanner используется ожидание при коммите (commit wait) и при чтении, которое ограничено интервалом неопределённости.

Spanner использует TrueTime, для которого нужно специализированное железо и проприетарный протокол. На текущий момент аналог (насколько я знаю) есть только у Amazon - Amazon Time Sync [18]. Возможно, в будущем у других облачных провайдеров (в том числе и в Яндекс Облаке) также появятся аналоги, так как атомные часы не являются очень дорогой технологией.

Но на данный момент, в базах данных с открытым исходным кодом обычно используют какие-то другие подходы, помимо TrueTime, чтобы не зависеть от конкретного облачного провайдера.

2.2 YTsaurus

Динамические таблицы в YTsaurus [9] и реализация транзакций похожи на Google Spanner, но вместо TrueTime YTsaurus использует выделенный сервер для генерации временных меток (timestamp_provider). За счёт этого гарантируется монотонность и уникальность временных меток, необходимая для корректности алгоритма.

По умолчанию уровень изоляции в YTsaurus - SI (snapshot isolation), что слабее чем external consistency в Spanner.

Но если использовать двухфазную блокировку (2PL) для чтений и записей (то есть брать shared lock для чтений), то YTsaurus может предоставлять изоляцию serializable snapshot.

Минусом текущего подхода является наличие глобального timestamp_provider, потому что обращение по сети (из другого датацентра) к нему может занимать много времени. В случае Яндекса это не очень критично, так как датацентры находятся недалеко друг от друга, а в каких-то приложениях кросс-датацентр таблицы и транзакции не используются.

Но всё равно, какая-то задержка между датацентрами есть[1], и кросс-датацентр запросы являются менее эффективными, чем локальные.

2.3 CockroachDB

CockroachDB [4, 11] не использует глобальный timestamp_provider и не использует TrueTime.

Вместо этого, он использует HLC (Hybrid Logical Clock), которые являются локальными для сервера. Это позволяет избавиться от лишних походов по сети, не используя специализированное железо.

CockroachDB предоставляет уровень изоляции serializable и сохраняет причинный порядок с помощью HLC.

В случае, если чтение встречает запись с временной меткой в интервале неопределённости, то CockroachDB перезапускает транзакцию или обновляет

[1] Около 6-7 мс [19]

временную метку для чтения.

Подробнее про консистентность CockroachDB можно почитать в [20, 21, 22]

2.4 YugabyteDB

YugabyteDB и CockroachDB предоставляют схожую функциональность. По производительности также нет однозначного победителя, в каких-то сценариях лучше показывает себя одна БД, в каких-то другая [23]. У YugabyteDB есть хорошая вводная статья про HLC [15].

2.5 Отличия от существующих решений

2.5.1 Практические отличия

В данной работе воспроизводятся алгоритмы атомарного коммита и concurrency control из двух существующих решений (CockroachDB, YTsaurus).

Алгоритмы для устойчивости (Durability) и репликации (Raft / Paxos) не были реализованы для упрощения. В коде предполагается, что сервера не отказывают, не перезагружаются, диски не выходят из строя (и вообще все данные хранятся только в оперативной памяти).

Конфигурация серверов статичная, по каждому ключу детерминированно определяется шард (таблет), куда он попадёт (через $\mathrm{hash}(key) \mod N$, где $\mathrm{hash}(x)$ - хеш функция, key - ключ, N - количество таблетов).

Таблицы состоят из набора пар $key, value$, где $key, value$ - строки[2]. При желании, можно любой тип сериализовать в строку, и потом десериализовать.

В MVCC старые значения никогда не удаляются и остаются в оперативной памяти. После остановки кластера, значения никуда не сохраняются.

Так как сервера никогда не отказывают, то и протокол восстановления с диска тоже не нужно писать.

[2]На самом деле внутри базы данных более сложная структура, т.к. каждому ключу в MVCC модели соответствует список из (t_i, v_i), где v_i - i-е значение в истории ключа, t_i - временная метка его записи в таблицу

В реальной распределённой системе, очевидно, все эти предположения недопустимы, но фокус данной работы именно на алгоритме атомарного коммита и concurrency control, целью было изучить только их.

Для тестов, все эти допущения приемлемы, так как даже при бенчмарках реальных баз данных часто отключают Write Ahead Log, чтобы измерить эффективность самих транзакций, а не throughput / latency дисков (и тем самым жертвуют устойчивостью).

2.5.2 Теоретические отличия

В данной работе приводится сравнение двух подходов, то есть это больше проект про обзор, воспроизводимость и тестирование существующих подходов, а не самостоятельная статья.

3 Описание реализованных алгоритмов

3.1 Общая теория про concurrency control

Транзакционную базу данных в общем виде можно представить в виде двух частей:

1. DataStore (или Data Manager, в терминах [7]) - key-value хранилище данных, поддерживающее линеаризуемые операции записи $\mathbf{write}(k, v)$ и чтения $\mathbf{read}(k)$. Линеаризуемость означает, что для любого конкурентного исполнения существует последовательная история операций H, такая, что если в физическом времени $\mathbf{end}(A) < \mathbf{start}(B)$ (то есть операция B начала выполняться строго после операции A), то тогда $A \prec_H B$ (то есть история сохраняет порядок операций в реальном времени). При этом в H чтение возвращает последнюю запись по ключу, чтения в H возвращают такие же ответы, как в исходном конкурентном исполнении и после всех записей, хранилище данных находится в одинаковых состояниях.

2. Планировщик (или Transaction Manager, в терминах [7]) - взаимодействует с клиентами, реализует изоляцию транзакций, перенаправляет запросы в DataStore.

DataStore-ов и планировщиков может быть произвольное количество, не обязательно по одному (см. рисунок 3.1). На одном физическом сервере могут быть одновременно запущены и DataStore и планировщик.

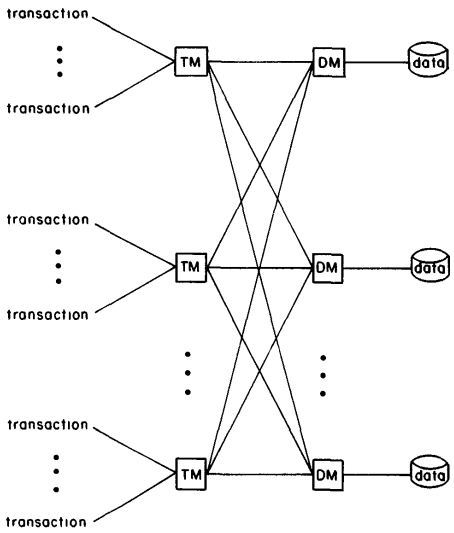


Рис. 3.1: Архитектура СУБД, поддерживающей транзакции [7]

Планировщик перенаправляет запросы в DataStore, и в итоге получается некоторое расписание операций. Пользуясь линеаризуемостью, можно считать, что расписание - это порядок операций на каждом DataStore.

3.2 Формальное определение сериализуемости

Определение 1. Серийное расписание - это расписание, в котором есть порядок на транзакциях $S = T_1 T_2 \dots T_n$, такой что операции транзакции T_{i+1} начинают выполняться только после того, как закончит выполняться T_i (все операции в S выполняются последовательно, в данном случае можно считать, что DataStore всего один)

Определение 2. Расписания $S_1 \sim S_2$ - view-эквивалентны, если все чтения в S_1 и S_2 возвращают совпадающие значения и после примения всех записей DataStore остаются в одинаковом итоговом состоянии. То есть, с точки зрения пользователя, эти расписания неотличимы друг от друга.

Определение 3. Расписание S - view-сериализуемо, если существует S^* - серийное расписание, такое что $S \sim S^*$ - view-эквивалентны

Определение 4. Уровень изоляции serializable означает, что любой набор входных запросов планировщики преобразуют в view-сериализуемое расписание.

Серийное расписание поддерживает систему в консистентном состоянии, при условии что каждая отдельная транзакция сохраняет консистентность. Тогда view-сериализуемые расписания тоже поддерживают консистентность, так как для них можно найти view-эквивалентное серийное (а значит консистентное) расписание.

Поэтому из уровня изоляции serializable следует консистентность системы (при условии, что транзакции по отдельности её не нарушают).

3.3 Двухфазная блокировка (2PL)

Двухфазная блокировка - это один из самых простых алгоритмов для обеспечения concurrency control. Существуют различные вариации 2PL (C2PL, S2PL, SS2PL), но нас будет интересовать только стандартный 2PL.

Протокол 2PL для исполнения транзакции состоит из двух фаз (см. рисунок 3.2):

1. Фаза роста: транзакция захватывает локи для всех своих строк. Для чтения берётся shared lock, для записи эксклюзивный. После того, как был взят лок, транзакция может читать или обновлять строку.

2. После того, как все нужные строчки были прочитаны или записаны, начинается фаза сжатия: после коммита, либо аборта, транзакция возвращает захваченные локи.

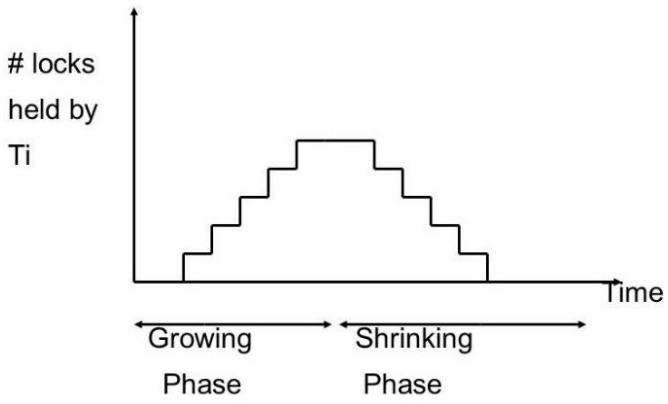


Рис. 3.2: Количество взятых транзакцией локов в 2PL с течением времени [24]

Утверждение 5. Использование 2PL обеспечивает уровень изоляции serializable.

Это известное утверждение, доказательство есть в статье [7], поэтому приводить здесь его полностью не буду. Также про concurrency control есть лекция на курсе Липовского Р.Г. “Теория отказоустойчивых распределённых систем”.

План доказательства утверждения 5 состоит в следующем:

1. Вводится понятие конфликта - это 2 операции $o_1 \in T_1$, $o_2 \in T_2$, пишущие или читающие по одному ключу, при этом одна из них пишущая (то есть бывают rw, wr, ww конфликты).

2. Давайте разрешим в расписании переставлять только соседние элементы, также чтобы была view-эквивалентность, разрешим переставлять только не конфликтующие операции (то есть 2 чтения одного ключа, или операции над разными ключами), так как операции конфликтуют $\Leftrightarrow$ при изменении их порядка теряется view-эквивалентность. Если в S_1 можно переставить операции по правилу выше (то есть переставляя только соседние неконфликтующие операции) и получить S_2, то $S_1 \sim S_2$ - конфликт-эквивалентные.

3. Расписание S - конфликт-сериализуемое, если $\exists\ S^*$, такое что $S \sim S^*$ - конфликт-эквивалентные.

4. Из конфликт-эквивалентности следует view-эквивалентность $\Rightarrow$ из конфликт-сериализуемости следует view-сериализуемость $\Rightarrow$ достаточно доказать, что 2PL порождает только конфликт-сериализуемые расписания.

5. Пусть $o_1 \in T_1, o_2 \in T_2$, $o_1 \prec_S o_2$ и o_1, o_2 конфликтуют. Тогда если $S \sim S^*$ - конфликт-эквивалентны, где S^* - серийное расписание, то $T_1 \prec_{S^*} T_2$ (так как переставлять o_1 и o_2 запрещено)

6. Пусть $CG(S)$ - граф конфликтов, $V = \text{Txns}(S)$, $E = \{(t,t') : \exists o_1 \in t, o_2 \in t' : o_1 \prec_S o_2 \ \wedge\ o_1$ конфликтует с $o_2\}$

7. Тогда несложно проверить, что $CG(S)$ - ациклический граф $\Leftrightarrow S$ - конфликт-сериализуемое [7].

8. Осталось доказать, что 2PL не допускает циклов в графе конфликтов. Это следует из того, что $\forall L(x), U(y) : L(x) < U(y)$, где x, y - произвольные ключи, $L(x)$ - время взятия блокировки для ключа x, $U(y)$ - время снятия блокировки для ключа y.

Подробнее про двухфазную блокировку можно прочитать в [6, 7]

3.3.1 Предотвращение дедлоков

При взятии локов, возможны дедлоки между разными транзакциями, если не контролировать порядок захвата локов.

Пусть возник дедлок, то есть цикл, в котором например T_1 пытается захватить лок для M_1 и уже захватила лок для M_2, T_2 пытается захватить лок для M_2 и уже владеет локом для M_1 (см. рисунок 3.3).

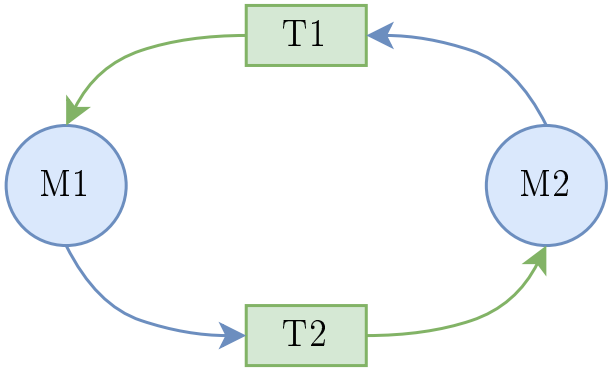


Рис. 3.3: Пример deadlock. Стрелка помечает зависимость между объектами, например стрелка T1→M1 означает, что T1 пытается захватить M1, стрелка M1→T2 означает, что лок для M1 захвачен транзакцией T2.

Тогда транзакции T_1 и T_2 никогда не смогут завершится, т.к. они обе заблокированы (такая ситуация и называется deadlock).

Необходимо прервать одну из транзакций в цикле, чтобы разрушить его и дедлок смог разрешится.

Для этого, в момент блокировки на локе можно делать дополнительную проверку, и прерывать одну из транзакций, чтобы дедлоков не возникало.

Есть два подхода, Wound-Wait и Wait-Die [6, 7].

Wound-Wait

В Wound-Wait проверяется, что если $ts(T_{curr}) < ts(T_{owner})$ (где T_{curr} - текущая транзакция, T_{owner} - транзакция, владеющая локом, $ts(T)$ - время начала транзакции T), то T_{owner} отменяется (то есть отменяется более новая транзакция).

Если $ts(T_{curr}) > ts(T_{owner})$, то транзакция ожидает, пока завершится предыдущая.

Так как ожидание происходит только при $ts(T_{curr}) > ts(T_{owner})$ и $ts(T)$ - монотонные, то циклов в графе ожидания получится не может, и дедлоки будут предотвращены.

Ожидание есть только от более новой транзакции к более старой.

Отменяются более новые транзакции, поэтому каждая транзакция когда-нибудь завершится.

Wait-Die

В Wait-Die, если $ts(T_{curr}) > ts(T_{owner})$ (то есть транзакция младше, чем захватившая лок), то отменяется T_{curr}.

Иначе, если $ts(T_{curr}) < ts(T_{owner})$, то T_{curr} ожидает освобождения лока.

В этом случае наборот, более старые транзакции ожидают завершения новых, но циклов в графе ожидания тоже получится не может.

Здесь также отменяется более новая транзакция.

В работе был реализован метод Wait-Die, так как он non-preemptive - не отменяет уже работающие транзакции, а только текущую транзакцию.

Дедлоки предотвращаются, но появляются отмены транзакций по инициативе системы, и такие транзакции приходится перезапускать.

3.4 Двухфазный коммит (2PC)

Протокол двухфазного коммита используется, чтобы обеспечить атомарность - свойство, при котором либо все операции транзакции будут применены, либо ни одна не будет применена.

Протокол также является относительно простым, он состоит из следующих шагов:

1. Старт транзакции - пользователь (случайно) выбирает координатора и отправляет сообщение StartTransaction координатору, который регистрирует транзакцию у себя и возвращает TransactionID.

2. Далее транзакция выполняет чтения и записи. Вместо изменения данных inplace, используется вызов WriteIntent - сообщение, что транзакция хочет записать какие-то значения. WriteIntent-ы и чтения отправляются напрямую соответствующим таблетам.

3. Пользователь либо зовёт abort - и транзакция отменяется, либо делает вызов commit. Оба вызова пользователь отправляет координатору.

4. Координатор отправляет себе и остальным участникам сообщение Prepare.

5. Когда участник транзакции получает сообщение Prepare, он пытается взять локи на соответствующие строки и проверяет отсутствие конфликтов. Если локи были успешно взяты и конфликтов не было, то участник гарантирует, что в будущем, когда придёт коммит, он применит его. Например, если сервер перезапускается, то после рестарта он должен восстановить данные с диска, а если он полностью отказывает, то его данные должны быть реплицированы на другой сервер (например через paxos или raft).

6. Если все участники успешно выполнили Prepare и ответили координатору, то координатор отправляет всем участникам сообщение Commit. Если хотя бы один участник не ответил, или ответил с ошибкой, то транзакция отменяется (и всем участникам рассылается abort).

7. Когда участнику приходит сообщение Commit, он записывает значения (из WriteIntent-ов) в таблицу, и затем снимает блокировки.

8. В конце координатор сообщает пользователю результат операции - был ли успешно выполнен коммит (но это сообщение может быть потерянно при доставке).

Корректность протокола основывается на том, что после того, как таблет сделал Prepare, он блокирует соответствующие строки, и гарантирует, что после получения Commit он закоммитит данные.

Протокол относительно простой, но в случае, когда координатор отказывает, транзакция зависнет.

Подробнее про 2PC и другие протоколы атомарного коммита можно почитать в статье [8].

3.5 MVCC и Snapshot Isolation

3.5.1 MVCC

Multiversion concurrency control (MVCC) предполагает, что в системе хранится для каждого ключа история его значений, то есть список пар $(timestamp_i, value_i)$.

Когда необходимо обновить значение, то в список добавляется запись с новой версией, а не перезаписывается старое значение. Это позволяет эффективно реализовать неблокирующие чтения по таймстемпу и снапшоты, засчёт того, что старые данные не меняются, а только добавляются новые в конец истории.

MVCC хранилище предоставляет следующие операции:

1. Read(key,$timestamp$) - прочитать последнее значение ключа с временем записи $\leq timestamp$.

2. Write(key,$value$,$timestamp$) - записать по ключу key значение $value$ с временем записи $timestamp$.

Подробнее про MVCC можно почитать в [6, 7].

3.5.2 Snapshot Isolation (SI)

Snapshot Isolation (SI) - это более слабый уровень изоляции, по сравнению с serializable, но тем не менее, он всё ещё часто бывает полезен, так как он не допускает большинство артефактов и предоставляет достаточно сильные гарантии [5]. Но при этом, SI как правило можно реализовать проще и эффективнее, чем serializable.

В случае serializable, можно было думать про исполнение транзакции, как последовательное, в случае SI это уже не так.

Описание протокола

В SI при старте транзакции выбирается временная метка (read timestamp), и все чтения в рамках транзакции, читают значения по данной временной метке, как будто из снапшота базы данных, взятого в этот момент.

В момент коммита, генерируется ещё одна временная метка (commit timestamp), и данные записываются в MVCC хранилище с этой меткой.

Может получиться, что пока выполнялась одна транзакция, в таблицу закоммитили новые значения (для прочитанных, или записываемых ключей). Тогда, при коммите возникает конфликт, и транзакция, которая попыталась закоммитить значения позднее, отменяется (такой подход называется First Commit Wins).

Это очень похоже на то, как работают системы контроля версий, когда при попытке merge может возникнуть merge conflict.

Write Skew

Один из артефактов, который может возникнуть при использовании SI - это Write Skew [25]. Пример:

1. Изначально $x = y = 0$
2. Транзакция T1: if (read(x) == 0) { write (y, 1); }

3. Транзакция T2: if (read(y) == 0) { write (x, 1); }

4. В итоге может получится, что обе транзакции читают $x = y = 0$, и потом успешно коммитят $x = 1$, $y = 1$. В serializable уровне изоляции такой сценарий невозможен.

Существуют различные способы бороться с этим, например добавить искусственные вызовы write(x, read(x)), write(y, read(y)), тогда в SI в сценарии выше возникнет конфликт, и одна транзакция будет перезапущена с новым read timestamp.

Read-Only Transaction Anomaly

Ещё один артефакт - это Read-Only Transaction Anomaly [26], например возможна следующая ситуация:

У пользователя есть 2 счёта в банке, изначально, $X = Y = 0$. Банк допускает, что при снятии денег с счёта может получиться отрицательный баланс, но требует за это коммисию в 1 у.е., если $X + Y < 0$ (то есть если суммарно на обоих счетах не хватает денег). Если $X + Y \geq 0$, то банк не берёт комиссию.

1. Клиент кладёт на счёт Y 20 у.е (T1).

2. Клиент снимает с счёта X 10 у.е. (T2).

3. Между первым и вторым пунктом клиент запрашивает баланс (T3).

Возможная история:

$$R_2(X_0, 0), R_2(Y_0, 0)$$

$$R_1(Y_0, 0), W_1(Y_1, 20), C_1$$

$$R_3(X_0, 0), R_3(Y_1, 20), C_3$$

$$W_2(X_2, -11), C_2$$

(здесь $R_i(X_t, V)$ - чтение транзакцией T_i $X_i = V$ в момент времени t, $W_i(X_t, V)$ - запись транзакцией T_i $X_i = V$ в момент времени t, C_i - коммит транзакции T_i)

Аномалия состоит в том, что баланс вернулся $X = 0, Y = 20$, но при этом банк взял комиссию 1 у.е., т.к. T2 прочитала $X = Y = 0$ и не увидела запись T1.

Генерация временных меток

Для SI необходим механизм, который будет генерировать монотонные и уникальные временные метки.

В Spanner [17] используется TrueTime, в CockroachDB [11] и YugabyteDB [27] используется HLC (гибридные логические часы), в YTsaurus [9] используется timestamp_provider (выделенный сервер).

3.6 Гибридные логические часы (HLC)

Гибридные логические часы состоят из двух частей: $h = (pt, lc)$, $h.pt$ - физическая компонента, $h.lc$ - логическая часть.

Гибридные логические часы работают примерно так же, как часы Лэмпорта (и так же позволяют отслеживать причинный порядок и находить консистентные снапшоты), но за счёт разделения на физическую и логическую компоненту, они приближены к реальному времени.

Наличие физической составляющей удобно тем, что пользователь может сам сгенерировать метку, чтобы прочитать данные. В случае с часами Лэмпорта непонятно, как это делать, а в случае физических часов он может просто передать своё локальное время, или время в прошлом (но нужно учитывать, что показания часов на клиенте не синхронизированы с часами на сервере).

Подробнее про HLC можно почитать в оригинальной статье [10].

3.7 Неблокирующее чтение по timestamp

Для того, чтобы читать по timestamp, необходимо найти консистентный снапшот в системе.

По умолчанию, HLC предоставляет такие снашоты, но есть нюанс. Если просто попытаться прочитать данные по timestamp, то в процессе чтения, может

появится новая запись в одном из таблетов, тогда чтение будет неконсистентно.

Для того, чтобы этого избежать, сейчас в коде алгоритм неблокирующего чтения устроен следующим образом:

1. Для каждого таблета узнаём timestamp последнего закоммиченного значения

2. Берём из этих timestamp минимум и запрашиваем данные с этим timstamp.

4 Детали реализации

4.1 Приложение

1. Приложение написано на C++ с зависимостями от многопоточныз примитивов из yt/yt/core.
2. В качестве системы сборки используется yatool (ya make).
3. При запуске таблета можно выбирать, какую реализацию использовать - с HLC, или без.
4. Помимо самого приложения, написаны стресс тесты, проверяющие корректность и тестирующие производительность.
5. Есть также docker и docker compose конфиги, которые позволяют собирать и запускать приложение локально.
6. Суммарно получается примерно 2 тысячи SLOC.

4.2 Репозиторий

Исходный код проекта находится открытом доступе в репозитории [16].

4.3 Реализованные алгоритмы

В итоге, в приложении были реализованы следующие алгоримты:

1. Двухфазная блокировка (2PL)
2. Двухфазный коммит (2PC)
3. Версионированное хранилище (MVCC)
4. Генерация временных меток в timestamp_provider

5. Генерация временных меток локально, с помощью гибридных логических часов (HLC)

6. Чтение по timestamp

5 Тестирование корректности и производительности

5.1 Описание тестов

Было реализовано два вида тестов.

1. Тест, в котором внутри транзакции выполняется операция

$$increment(X_1, x), increment(X_2, x), \ldots, increment(X_n, x)$$

   То есть, для n ключей к ним прибавляется одинаковое значение x, выбираемое случайно.

   Читающие запросы проверяют консистентность, они считывают все ключи и проверяют, что

$$X_1 = X_2 = \ldots = X_n$$

2. Тест, в котором внутри транзакции выполняется операция

$$increment(X_i, x), decrement(X_j, x)$$

   То есть, выбираются случайные два ключа X_i, X_j, $i \neq j$ и из одного ключа вычитается значение x, а к другому прибавляется.

   Читающие запросы проверяют консистентность, они считывают все ключи и проверяют, что

$$X_1 + X_2 + \ldots + X_n = 0$$

В каждом тесте запускается (параллельно, в пуле потоков) настраиваемое количество транзакций, и можно задавать частоту, с которой будут отправляться читающие запросы, проверяющие корретность.

По результатам тестов, выводится информация о TPS (количество успешно выполненных транзакций в секунду) и о среднем времени исполнения (задержке) транзакций.

5.2 Описание тестового стенда

В тестах с помощью docker compose поднимается 3 таблета (это количество можно настраивать), контейнеры, из которых будут запускаться тесты и (при необходимости) timestamp_provider.

Далее, запускаются тесты и в конце выводятся результаты тестирования.

5.2.1 Параметры запуска

Далее представлены параметры запуска для второго типа тестов, для тестирования производительности. Корректность проверялась для большого количества транзакций и читающих запросов (10000), и для двух типов тестов.

1. Количество тредов: 4 (для таблетов и тестов), timestamp_provider однопоточный
2. Количество ключей: 30
3. Частота читающих запросов: 20% (после каждой пятой транзакции запускается читающий запрос)
4. Количество транзакций: 100 (и 20 чтений)

5.2.2 Конфигурация сети

Сеть создаётся виртуальная, через docker compose. Внутри одного контейнера можно запустить команду 5.1

Листинг 5.1: Добавление задержки

```
1 qdisc add dev eth0 root netem $DELAY
```

Команда была взята из статьи [28], она добавляет задержку к исходящим пакетам.

С помощью команды выше, были протестированы три сценария, симулирующие георасределённую систему.

1. Между каждой парой контейнеров задержка одинаковая (0-5мс)

2. Между всеми контейнерами нулевая задержка, кроме timestamp_provider, до которого RTT (round trip time) в пределах 0 - 25 мс

3. Между всеми контейнерами нулевая задержка, кроме клиента. От клиента до таблета RTT составляет от 0 до 25 мс.

Первый сценарий тестирует георасприделённую систему, в которой все таблеты (и клиент и timestamp_privider) расположены далеко друг от друга.

Второй сценарий тестирует случай, когда есть локальный кластер, и клиент находится рядом с этим кластером, но timestamp_provider находится в другой локации. Это худший сценарий для текущей реализации YTsaurus.

Третий сценарий тестирует, насколько задержка до клиента влияет на итоговую задержку транзакций. Тут результаты довольно ожидаемые, задержка растёт прямо пропорционально.

5.3 Синхронизация часов

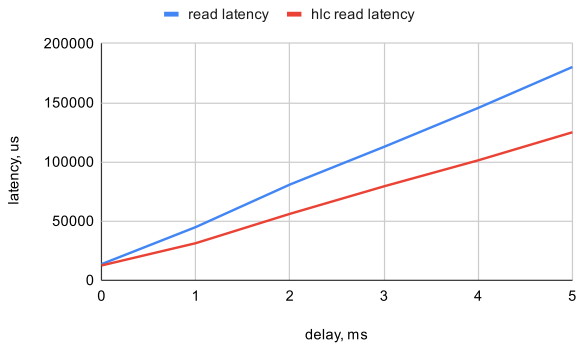
Для HLC реализации у таблетов при старте настраивается максимальная задержка для локальных часов. К каждому чтению часов прибавляется фиксированная задержка, которая генерируется как random(0, max_delay) (равномерное случайное распределение) при старте программы.

Это необходимо для тестов корректности, чтобы проверить, что система корректно работает при несинхронизированных часах.

5.4 Результаты тестирования

5.4.1 1 сценарий

(a) Задержка записей



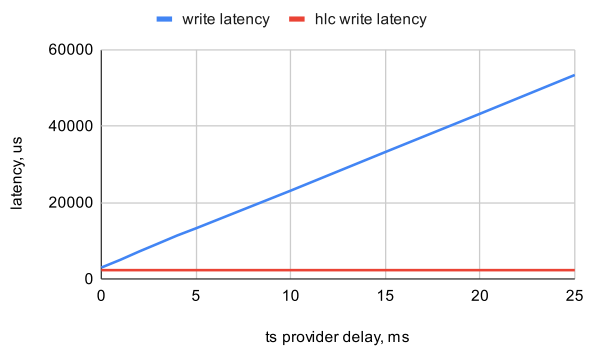
(b) Задержка чтений

Рис. 5.1: Зависимость задержки записей и чтений от задержки между контейнерами

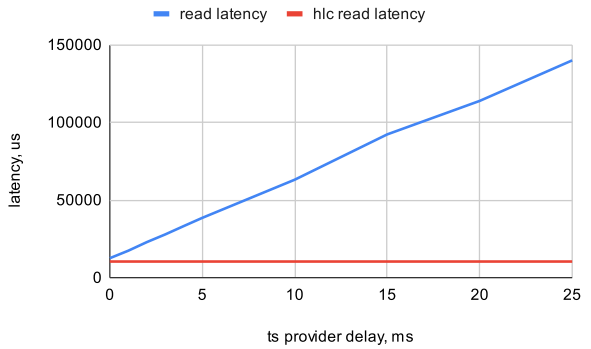
По результатом тестов видно, что задержка транзакций растёт линейно с ростом сетевой задержки, при этом реализация с HLC работает более эффективно, так как в ней не используется timestamp_provider.

5.4.2 2 сценарий

В реализации с HLC задержка постоянная, так как в ней не используется timestamp provider. В исходной реализации задержка транзакций прямо пропорциональна расстоянию до timestamp provider.



(a) Задержка записей

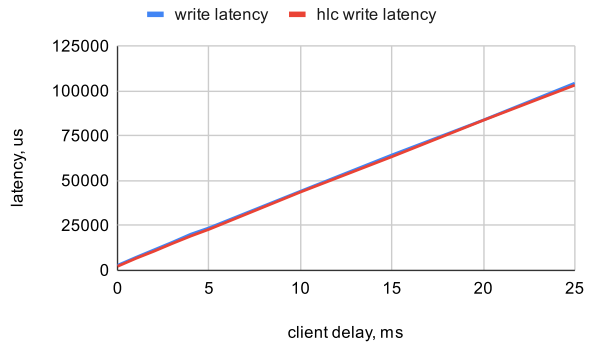


(b) Задержка чтений

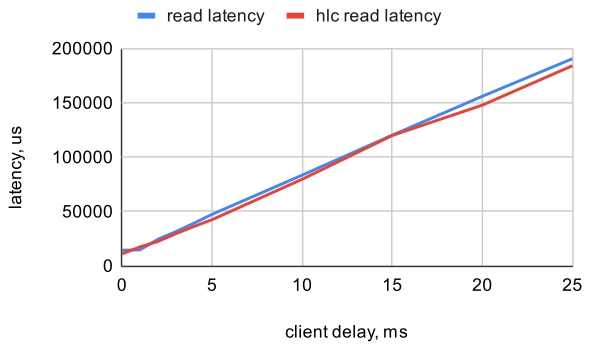
Рис. 5.2: Зависимость задержки записей и чтений от задержки между кластером и timestamp_provider

5.4.3 3 сценарий

В случае, когда меняется только задержка до клиента, обе реализации показывают схожую производительность, но задержка чтений в HLC реализации немного меньше.



(a) Задержка записей



(b) Задержка чтений

Рис. 5.3: Зависимость задержки записей и чтений от задержки между клиентом и кластером

6 Заключение

В данной работе были исследованы, реализованы и протестированы различные алгоритмы для георасперделённых транзакций.

Для поднятия тестового стенда испольуется docker compose, между контейнерами можно настраивать сетевую задержку, симулируя таким образом георасперделённую систему.

По итогам тестов, было показано, что реализация с HLC работает более эффективно, чем текущая реализация в YTsaurus, при этом консистентность базы данных не нарушается. Данный проект - proof of concept, он показывает, что реализация с HLC корректно и эффективно работает, дальше необходимо реализовать этот подход в YTsaurus, но это остаётся для будущих работ.

7 Библиографический список

[1] Gray Jim и др. The transaction concept: Virtues and limitations // VLDB. — Т. 81. — 1981. — С. 144–154.

[2] Bigtable: A distributed storage system for structured data / Fay Chang, Jeffrey Dean, Sanjay Ghemawat и др. // ACM Transactions on Computer Systems (TOCS). — 2008. — Т. 26, № 2. — С. 1–26.

[3] WonderNetwork. Global Ping Statistics. — URL: https://wondernetwork.com/pings (дата обращения: 20.05.2025).

[4] Cockroachdb: The resilient geo-distributed sql database / Rebecca Taft, Irfan Sharif, Andrei Matei и др. // Proceedings of the 2020 ACM SIGMOD international conference on management of data. — 2020. — С. 1493–1509.

[5] Kingsbury Kyle. Consistency Models (left subtree). — URL: https://jepsen.io/consistency/models (дата обращения: 20.05.2025).

[6] Concurrency control and recovery in database systems / Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman и др. — Addison-wesley Reading, 1987. — Т. 370.

[7] Bernstein Philip A, Goodman Nathan. Concurrency control in distributed database systems // ACM Computing Surveys (CSUR). — 1981. — Т. 13, № 2. — С. 185–221.

[8] Gray Jim, Lamport Leslie. Consensus on transaction commit // ACM Trans. Database Syst. — 2006. — март. — Т. 31, № 1. — С. 133–160. — URL: https://doi.org/10.1145/1132863.1132867.

[9] YTsaurus. Исходный код YTsaurus. — URL: https://github.com/ytsaurus/ytsaurus (дата обращения: 20.05.2025).

[10] Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases / Murat Demirbas, Marcelo Leone, Bharadwaj Avva и др. — 2014. — URL: https://api.semanticscholar.org/CorpusID:15965481.

[11] Enabling the next generation of multi-region applications with cockroachdb / Nathan VanBenschoten, Arul Ajmani, Marcus Gartner и др. // Proceedings of the 2022 International Conference on Management of Data. — 2022. — С. 2312–2325.

[12] Cockroach Labs. Документация CockroachDB, Transaction Layer. — URL: https://www.cockroachlabs.com/docs/stable/architecture/transaction-layer (дата обращения: 20.05.2025).

[13] Cockroach Labs. Исходный код CockroachDB. — URL: https://github.com/cockroachdb/cockroach (дата обращения: 20.05.2025).

[14] YTsaurus. Документация YTsaurus, Транзакции. — URL: https://ytsaurus.tech/docs/ru/user-guide/dynamic-tables/transactions (дата обращения: 20.05.2025).

[15] Ranganathan Karthik. A Matter of Time: Evolving Clock Sync for Distributed Databases. — URL: https://www.yugabyte.com/blog/evolving-clock-sync-for-distributed-databases/ (дата обращения: 20.05.2025).

[16] Смородинов Александр. Исходный код проекта. — URL: https://github.com/asmorodinov/distributed_transactions/tree/main/distributed_transactions (дата обращения: 20.05.2025).

[17] Spanner: Google's globally distributed database / James C Corbett, Jeffrey Dean, Michael Epstein и др. // ACM Transactions on Computer Systems (TOCS). — 2013. — Т. 31, № 3. — С. 1–22.

[18] Tiwari Abhishek. Rise of TrueTime: Rationale behind Amazon Time Sync Service. — 2017. — декабрь. — URL: http://dx.doi.org/10.59350/mpztp-jvn26.

[19] Вирилин Александр, Клюев Леонид. MPLS повсюду. Как устроена сетевая инфраструктура Яндекс.Облака. — URL: https://habr.com/ru/companies/yandex/articles/437816/ (дата обращения: 20.05.2025).

[20] Wang Bin. Spanner and Open Source Implementations. — URL: https://www.binwang.me/2018-07-29-A-Review-on-Spanner-and-Open-Source-Implementation.html (дата обращения: 20.05.2025).

[21] Kimball Spencer, Sharif Irfan. Living without atomic clocks: Where CockroachDB and Spanner diverge. — URL: https://www.cockroachlabs.com/blog/living-without-atomic-clocks/ (дата обращения: 20.05.2025).

[22] Matei Andrei. CockroachDB's consistency model. — URL: https://www.cockroachlabs.com/blog/consistency-model/ (дата обращения: 20.05.2025).

[23] Ivanov Evgeniy. YCSB performance series: YDB, CockroachDB, and YugabyteDB. — URL: https://blog.ydb.tech/ycsb-performance-series-ydb-cockroachdb-and-yugabytedb-f25c077a382b (дата обращения: 20.05.2025).

[24] Haroon Muhammad. Challenges of concurrency control in object oriented distributed database systems // Int. J. Modern Comput., Inf. Commun. Technol. — 2019. — Т. 2, № 7. — С. 48–52.

[25] Jaffray Justin. What Does Write Skew Look Like? — URL: https://justinjaffray.com/what-does-write-skew-look-like/ (дата обращения: 20.05.2025).

[26] Fekete Alan, O'Neil Elizabeth, O'Neil Patrick. A read-only transaction anomaly under snapshot isolation // ACM SIGMOD Record. — 2004. — Т. 33, № 3. — С. 12–14.

[27] yugabyte. Исходный код YugabyteDB. — URL: https://github.com/yugabyte/yugabyte-db (дата обращения: 20.05.2025).

[28] Kitaya Kazushi. Simulate high latency network using Docker containers and "tc" commands. — URL: https://medium.com/@kazushi/simulate-high-latency-network-using-docker-containerand-tc-commands-a3e5 (дата обращения: 20.05.2025).