

Изоляция транзакций и Многоверсионность

Yan Khonski, Wriker

<https://cz.linkedin.com/in/yan-khonski>

Я работаю в команде Back End Reliability (надежность)

- Масштабируем Wrike, чтобы удвоить число пользователей
- Инциденты с продакшеном
- Мониторинг, оповещения
- Улучшение производительности
- Начала карьеры в 2012
- В outsourcing компаниях, в start-up продукт для магазинов и сотрудников, корпорациях
- В основном Java бэкенды
- Также работал с другими инструментами (python, C++) и front end



Wrike – Лидер в управлении проектами и построению совместной работы

Wrike позволяет командам видеть проект во всех стадиях, зависимости и блокировки, а также способствует более продуктивной совместной работе.



Основаны в
2006



9 офисов
по всему
миру



20,000+
Клиентов



1100+
Сотрудников



5 лет в
Fast 500

Масштаб Enterprise

- **2M+** пользователей в **140+** странах на **9** языках
- **1B+** задач, папок, проектов уже созданы
- **58M+** созданы цифровых файлов
- Дата центры в **США** и **ЕС**

Лидирующее решение для
совместной работы
предприятий

Forrester Research **2016, 2018**

Выбор клиента для для
управления портфелем
проектов по всему миру

Gartner Peer Insights **2018**

Инноватор года

Microsoft Windows
Developer Award **2018**

Customer Experience
Награда за выдающиеся
достижения в ПО и услугах

Software Reviews **2018**

20,000+ Организации выбрали Wrike

Google

airbnb

SIEMENS

DeLonghi

ESTÉE
LAUDER
COMPANIES

DELL

WesternUnion WU

Kelly

Hootsuite

f5

TIFFANY & CO.

Ogilvy

okta

SWAROVSKI

snowflake

Aerotek

OSF
HEALTHCARE

Magellan
HEALTH.

Children's Hope
INTERNATIONAL

Davita
Kidney Care

Premier | Sotheby's
INTERNATIONAL REALTY

MTD
For A Growing World

TOPGOLF

TD Bank

STITCH FIX

ATM
TEXAS A&M
UNIVERSITY

flashbay

SPRINT
ONE CUP AT A TIME

JABIL

gwynnie bee

Tenet
Health

esurance

Содержание

Уровни изоляции транзакций,
Их плюсы и минусы,
Как они реализованы

Часть 1: Вспомним уровни изоляции транзакций

- Без конкурентного доступа
- Уровни изоляции транзакций
- Формулировка задачи

Часть: Конкурентный доступ (concurrency control)

- Оптимистичный и пессимистичный уровни конкурентного (параллельного) доступа
- Механизмы блокировки для доступа к данным
- Многоверсионность
- VACUUM

База данных

Набор данных, и описаний данных которые хранится постоянно

```
1      {
2        "Actors": [
3          {
4            "name": "Tom Cruise",
5            "age": 56,
6            "Born At": "Syracuse, NY",
7            "Birthdate": "July 3, 1962",
8            "photo": "https://jsonformatter.org/img/tom-cruise.jpg"
9          },
10         {
11           "name": "Robert Downey Jr.",
12           "age": 53,
13           "Born At": "New York City, NY",
14           "Birthdate": "April 4, 1965",
15           "photo": "https://jsonformatter.org/img/Robert-Downey-Jr.jpg"
16         }
17       ]
18     }
```

Name	E-mail	Phone Ext	Home Phone	Cell Phone	Dept
Aaron Orozco	aaron@timeips.com	x3015			Administration
Ashley Fahey	ashley@timeips.com	x3024			Office Support
Camille Culbertson	camille@timeips.com	x3112			Marketing
Carrie Bradford	carrie@timeips.com	x3058			Accounting
Dennis Fahey	dennis@timeips.com	x3045			Sales
Fred Aguilera	fred@timeips.com	x3103			Accounting
Jamie Edmonds	jamie@timeips.com	x3005			Marketing
Jason Miller	jason@timeips.com	x3097			Distribution
Kirk Jones	kirk@timeips.com	x3022			IT
Kristy Jones	kristy@timeips.com	x3184			Human Resources
Mark Edwards	marc@timeips.com	x3104			Sales
Martine Taylor	martine@timeips.com	x3066			Office Support
Matthew Jacobs	matt@timeips.com	x3101			Marketing
Michael O'Connor	michael@timeips.com	x3064			Distribution
Rob Heaston	rob@timeips.com	x3061			Administration
Shea Sylvia	shea@timeips.com	x3000			Administration

Системы управления базами данных

Системы ПО, позволяющие пользователям (и приложениям) создавать, управлять и использовать базы данных. Также позволяют определить уровни доступа к данным

- Сохранение данных, доступ и модификация данных
- Поддерживают транзакции и параллельный доступ
- Позволяет восстанавливать данные, в случаи их повреждения или сбоев системы
- Позволяет контролировать уровни доступа чтения и модификации данных
- Обеспечивает согласованность данных, чтобы они соответствовали правилам валидации, непротиворечивости

ACID - свойства

Требования к надёжным транзакционным системам

- А - атомарность.
- С - согласованность.
- I - изоляция.
- D - устойчивость.



Не все базы данных подчиняются строгим требованиям ACID

Amazon DynamoDB (based on Dynamo) до 2018

Dynamo - согласованность в конечном счете

- https://www.allthingsdistributed.com/2007/10/amazons_dynamo.html
- <https://aws.amazon.com/blogs/aws/new-amazon-dynamodb-transactions/>
- <https://www.infoq.com/news/2018/12/dynamodb-transactions-aws/>

Запись нескольких документов не гарантировала атомарности их вместе, только по одному документу ([db.collection.updateMany\(\)](#))

Транзакции поддерживающие множество документов с версии 4+

<https://www.mongodb.com/blog/post/mongodb-multi-document-acid-transactions-general-availability>



Что такое конкурентный доступ (concurrency control)

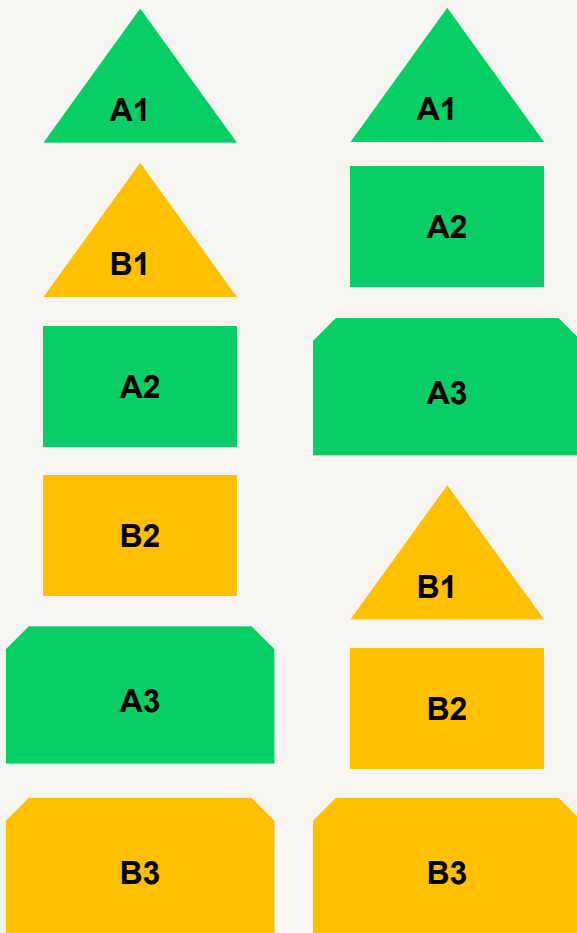
Гарантирует, что мы получим правильный результат, при этом как можно быстрее.

Мы можем исполнить части программы быть в любом другом порядке, но при этом результат должен остаться прежним.

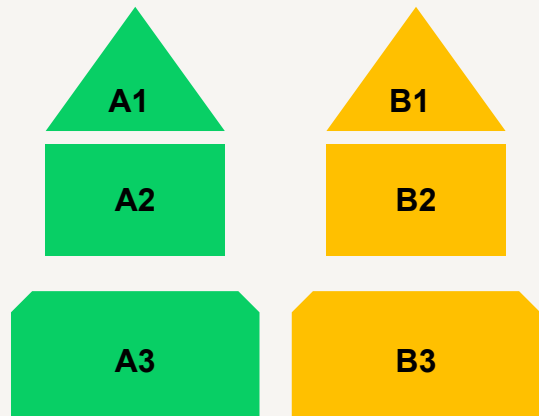
Необходимое условие для параллелизма:

Позволяет нам выделить части программы, которые могут быть выполнены независимо друг от друга (в том числе параллельно / одновременно)

Пример
конкурентного
выполнения
Давайте
поменяем
порядок
инструкций



А теперь распараллелим
инструкции



Формулировка задачи

Множество клиентов читают и пишут одни и те же данные одновременно!

T1

```
SELECT id, title FROM task WHERE id > 1;
```

T2

```
UPDATE task SET title = 'task-123' WHERE id = 6;
```

T3

```
UPDATE task SET title = 'abc' WHERE id = 6;
```



Решение: Без конкурентного доступа

Транзакция - логическая единица работы с данными.

Может быть выполнена либо целиком и успешно, соблюдая целостность данных и независимо от параллельно идущих других транзакций, либо не выполнена вообще, и тогда она не должна произвести никакого эффекта.

Jim Gray 1981 “В то время, крупнейшие авиакомпании и банки имели где-то 10,000 терминалов и 100 активных транзакций в любой момент времени”.

- <http://jimgray.azurewebsites.net/papers/thetransactionconcept.pdf>
- <https://vladmihalcea.com/a-beginners-guide-to-acid-and-database-transactions/>

Redis

Нет проблемы конкурентного доступа. Один worker потом обрабатывает из очереди запросы по одному.

- <https://redis.io/docs/reference/internals/internals-rediseventlib/>
- <https://redis.io/docs/reference/clients/>



Можно применить секционирование (partitioning). Всё равно для каждого индивидуального редиса, будет один поток, который в момент времени либо читает либо пишет данные.

<https://redis.io/docs/getting-started/faq/#how-can-redis-use-multiple-cpus-or-cores>

Изоляция транзакций

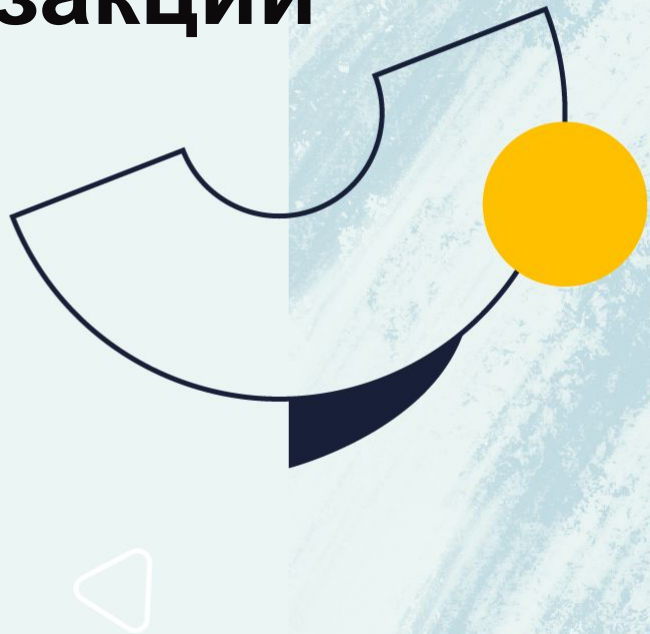
Уровень изоляции транзакции определяется степенью, насколько данная транзакция изолирована от других транзакций, выполняющихся одновременно; побочными явлениями (аномалиями), которые возможны

при одновременном выполнении транзакций, но не допускаются на определённом уровне.

Нам придётся заплатить цену за более высокий уровень изоляции.

Стандарт SQL определяет четыре уровня изоляции транзакций

Наиболее строгий из них — **сериализуемый**, определяется так, что при параллельном выполнении несколько сериализуемых транзакций должны гарантированно выдавать такой же результат, как если бы они запускались по очереди в некотором порядке.



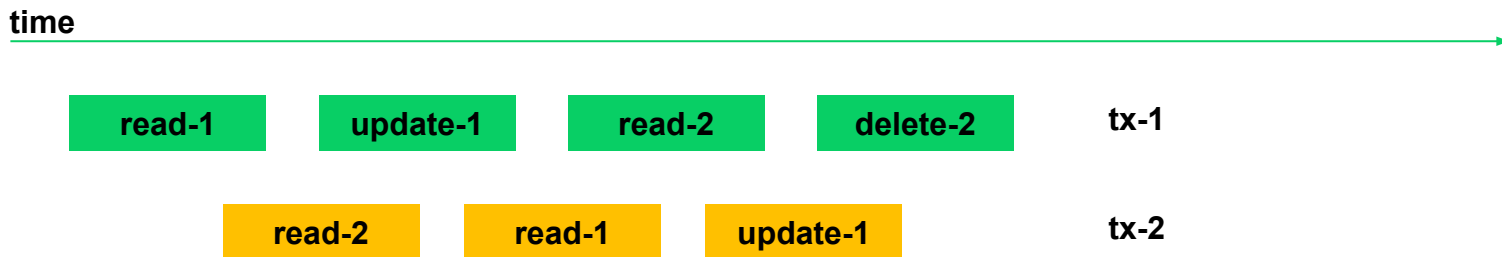
Граф предшествования (Serializability)

Расписание (Schedule) - список всех инструкций со всех транзакций, которые мы выполняем на базой данных

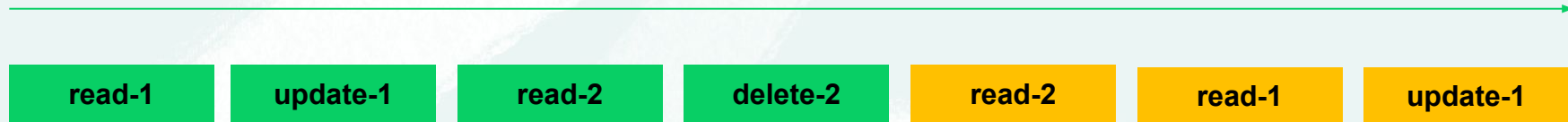
Полное расписание (Complete schedule) - если содержит все операции со всех транзакций

Последовательное расписание (Serial schedule) - транзакции выполняются независимо, следующая транзакция начинается после окончания предыдущей. Ограничивает производительность, но такое расписание легко изучить / понять.

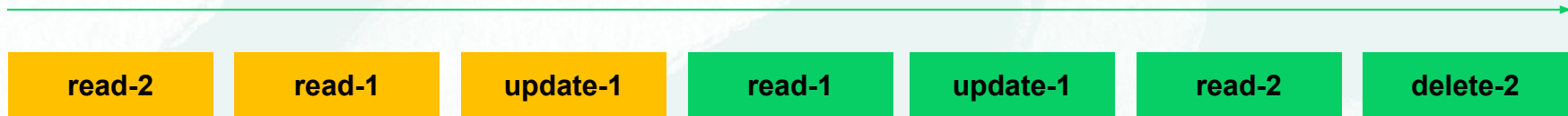
Сериализуемое расписание (Serializable schedule) - эквивалентно полному и последовательному расписанию над данным множеством транзакций



time



time



Грязное чтение



Dirty reads

Транзакция читает данные, записанные параллельной незавершённой транзакцией.



Transaction-1

BEGIN;

**UPDATE task SET title = 'new-text'
WHERE id = 6;**

ROLLBACK;

Transaction-2

SELECT id, title FROM task WHERE id = 6;

May return (6, 'new-text')

Actual data

(6, 'task-6')

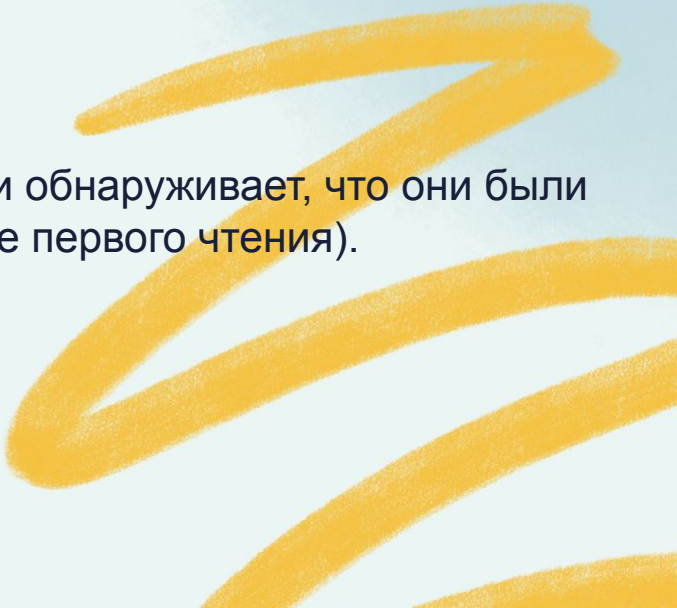
(6, 'new-text')

(6, 'task-6')

Неповторяемое чтение

Nonrepeatable reads

Транзакция повторно читает те же данные, что и раньше, и обнаруживает, что они были изменены другой транзакцией (которая завершилась после первого чтения).

A decorative graphic consisting of several thick, yellow, hand-drawn brushstrokes that curve upwards from the bottom right towards the center of the slide.

Transaction-1

BEGIN;

**UPDATE task SET title = 'new-text'
WHERE id = 6;**

COMMIT;

Transaction-2

SELECT id, title FROM task WHERE id = 6;

Will return (6, 'task-6')

SELECT id, title FROM task WHERE id = 6;

Will return (6, 'new-text')

Actual data

(6, 'task-6')

(6, 'task-6')

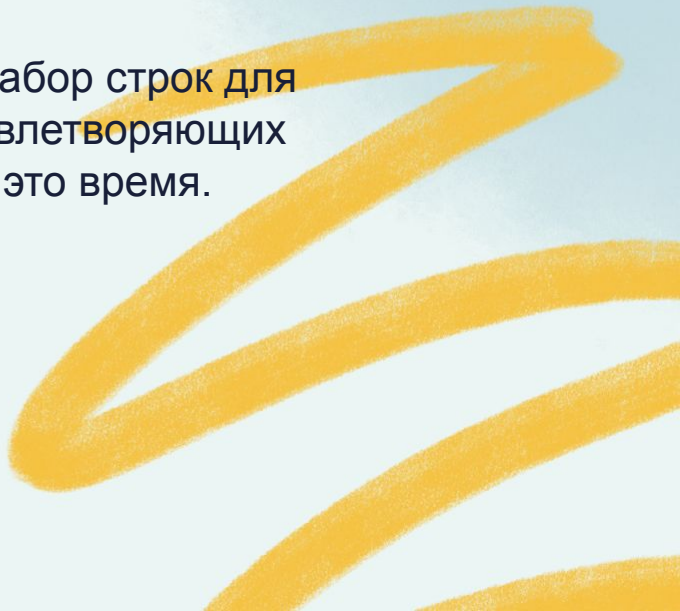
(6, 'new-text')

Фантомное чтение



Phantom reads

Транзакция повторно выполняет запрос, возвращающий набор строк для некоторого условия, и обнаруживает, что набор строк, удовлетворяющих условию, изменился из-за транзакции, завершившейся за это время.



Transaction-1

BEGIN;

**INSERT INTO task (id, title)
VALUES (7, 'task-7');**

COMMIT;

Transaction-2

SELECT id, title FROM task WHERE id >= 5;

Will return (5, 'task-5'), (6, 'task-6')

SELECT id, title FROM task WHERE id >= 5;

Will return (5, 'task-5'), (6, 'task-6'), (7, 'task-7')

Actual data

**(5, 'task-5'),
(6, 'task-6')**

**(5, 'task-5'),
(6, 'task-6')**

**(5, 'task-5'),
(6, 'task-6'),
(7, 'task-7')**

Аномалия сериализации

Serialization anomaly

Результат успешной фиксации группы транзакций оказывается несогласованным при всевозможных вариантах исполнения этих транзакций по очереди.

Например, две транзакции A and B, но в результате состояние базы данных, не было бы таким, если бы мы выполнили эти транзакции в каком-нибудь порядке: (A, B) или (B, A).

Каждая индивидуальная транзакция будет соблюдать согласованность данных и правила, но комбинация этих транзакций даст результат, который может нарушить эти правила.

Несогласованная запись (write skew) пример аномалии сериализации.

Transaction-1

BEGIN;

**SELECT SUM (value) FROM mytab
WHERE class = 1;**

Will return 30

**INSERT INTO mytab (class, value)
VALUES (2, 30);**

COMMIT;

class	value
1	10
1	20
2	100
2	200

Transaction-2

BEGIN;

**SELECT SUM (VALUE) FROM mytab
WHERE class = 2;**

Will return 300

**INSERT INTO mytab (class, value)
VALUES (1, 300);**

COMMIT;

Actual data

(1, 10), (1, 20),
(2, 100), (2, 200)

(1, 10), (1, 20),
(2, 100), (2, 200),
(2, 30)

(1, 10), (1, 20),
(2, 100), (2, 200),
(2, 30), (1, 300)

Выполним эти транзакции последовательно

Tx-1:

```
SELECT SUM (value) FROM mytab WHERE class = 1;  
INSERT INTO mytab (class, value) VALUES (2, 30);
```

Tx-2:

```
SELECT SUM (VALUE) FROM mytab WHERE class = 2;  
INSERT INTO mytab (class, value) VALUES (1, 300);
```

(transaction-1, transaction-2):

(1, 10), (1, 20), (2, 100), (2, 200), (2, 30), (1, 330)

(transaction-2, transaction-1):

(1, 10), (1, 20), (2, 100), (2, 200), (1, 300), (2, 330)

However, we had

(1, 10), (1, 20), (2, 100), (2, 200), (2, 30), (1, 300)

class	value
1	10
1	20
2	100
2	200

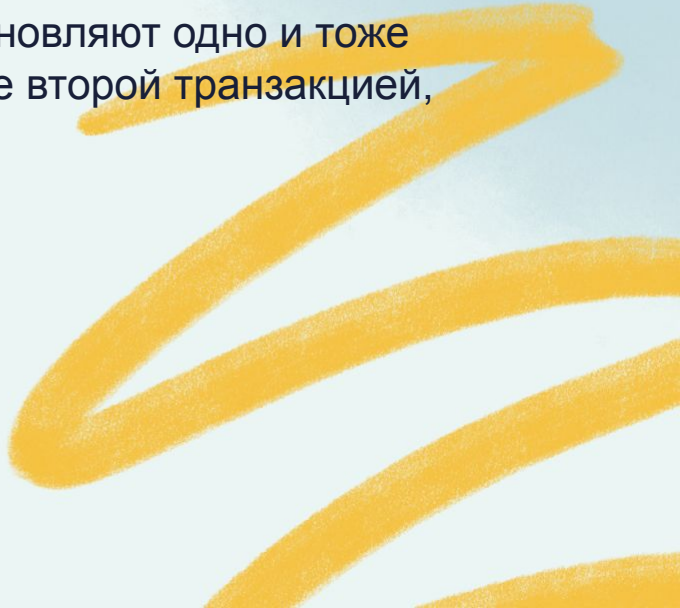
Грязная запись



Dirty write

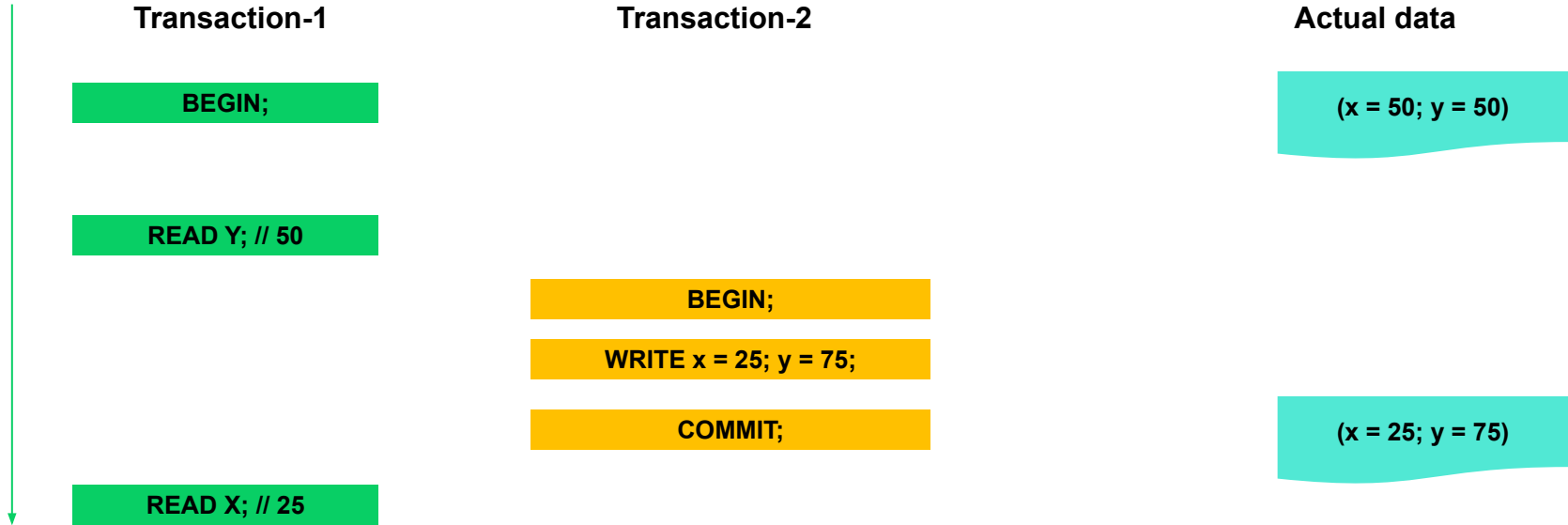
Не случается даже на самом низком уровне изоляции.

Если бы такое случилось, представим. Две транзакции обновляют одно и тоже значение, первая транзакция увидит значение, записанное второй транзакцией, а потом мы откатим первую транзакцию.



Read skew

Constraint: $X + Y = 100$



Потерянное обновление

Lost update

Транзакция Тх-1 читает запись, а другая транзакция Тх-2 обновляет эту же запись перед тем, как первая транзакция обновит ту же запись.

В результате, мы потеряли запись значения записанным Тх-2.

A decorative graphic consisting of several thick, yellow, hand-drawn brushstrokes that curve upwards from the bottom right towards the center of the slide.

Transaction-1

BEGIN;

**SELECT id, title FROM task WHERE id = 1;
(1, 'original')**

**UPDATE task SET title = 'tx-1-value'
WHERE id = 1;**

COMMIT;

Transaction-2

BEGIN;

UPDATE task SET title = 'tx-2' WHERE id = 1;

COMMIT;

Actual data

(1, 'original')

(1, 'tx-2')

(1, 'tx-1-value')

Уровни изоляции транзакции в PostgreSQL

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly	Read skew	Lost update
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible	Not possible	Not possible
Serializable	Not possible	Not possible	Not possible	Not possible	Not possible	Not possible

Какой уровень изоляции использовать?

READ_COMMITTED: транзакция будет откатена только в случае ошибки
Ошибки сериализации не могут произойти, так что тебе не нужно думать о перезапуске транзакций.

REPEATABLE_READ: решает некоторые аномалии, но не все. Разработчик должен обработать ошибки сериализации. Транзакции такой изоляции могут использоваться для построения отчётов, когда нужно выполнить несколько запросов, но при этом они будут видеть данные "как бы зафиксированные" в конкретный момент начала транзакции.

SERIALIZABLE: не нужно беспокоиться об ошибках согласованности. Нужно перезапустить транзакции, которые были отменены из-за ошибки сериализации. Более того, увеличение количества перезапущенных транзакций и связанные с ними блокировки негативно скажутся на производительности.

Как PostgreSQL достиг параллельности

Блокировки (locks)
Многоверсионность



Конкурентный (параллельный) доступ

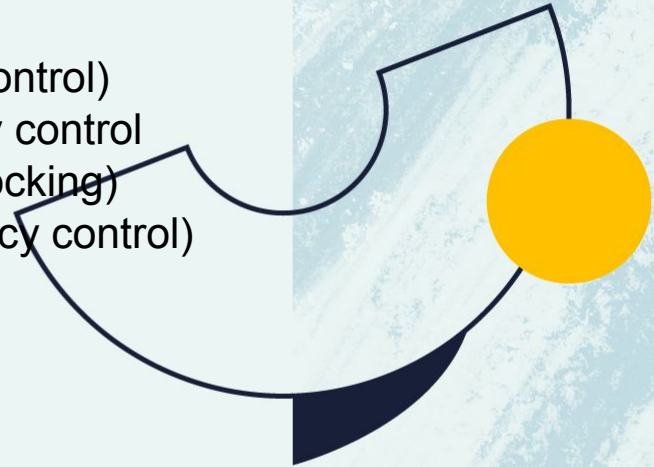
Concurrency control

Механизм который гарантирует корректность данных, и гарантирует атомарность транзакций, несмотря на то, что транзакции могут быть выполнены конкурентно (параллельно).



- Оптимистичный доступ (Optimistic concurrency control)
- Пессемистичный доступ (Pessimistic concurrency control)
Например, двухфазная блокировка (two-phase locking)
- Многоверсионный доступ (Multiversion concurrency control)

У каждого из них свои вариации.



Optimistic concurrency control

Позволяет транзакциям читать и писать. Если нет конфликта, то обе транзакции продолжают выполнение. Иначе, одна из конфликтующих транзакций будет откатена.

Здесь мы предполагаем, что такие конфликты случаются редко, иначе перезапуски транзакций ударят по производительности.

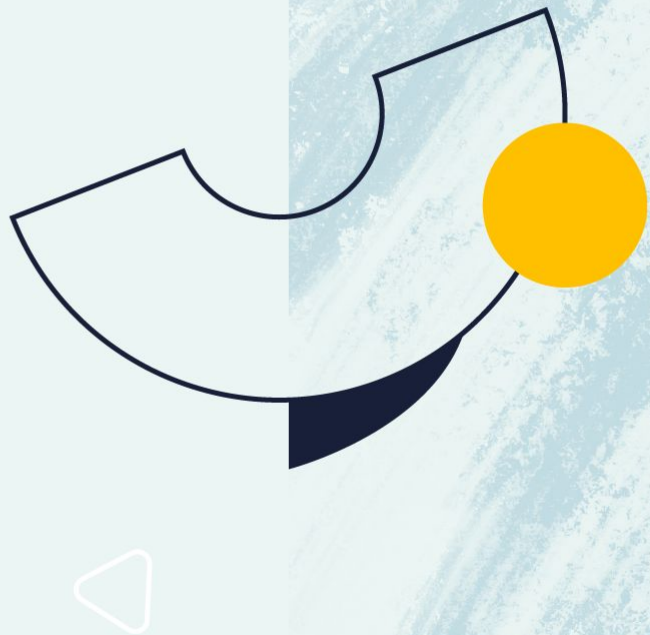


Pessimistic concurrency control

Методы бывают с блокировками и без блокировок.

Блокировочные подходы (lock-based), нам нужно держать блокировки на записях, чтобы другие транзакции их не читали и не писали.

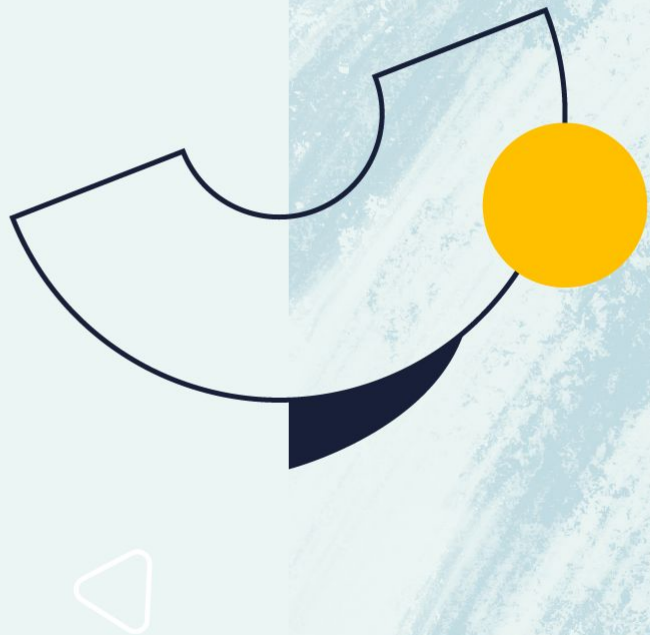
Есть подходы, не использующие блокировки.
Они поддерживают список всех операций чтения и записи, ограничивают их выполнение в зависимости от расписания незаконченных транзакций.



Блокировки (locks)

Гранулярность блокировки (multi granularity locking), когда количество низкоуровневых, мелкогранулярных блокировок превышает определенный предел, они заменяются на одну блокировку более высокого уровня.

- Разделяемый (Shared lock)
- Исключительный режим блокировки (Exclusive lock)



Двухфазная блокировка

Two phase locking

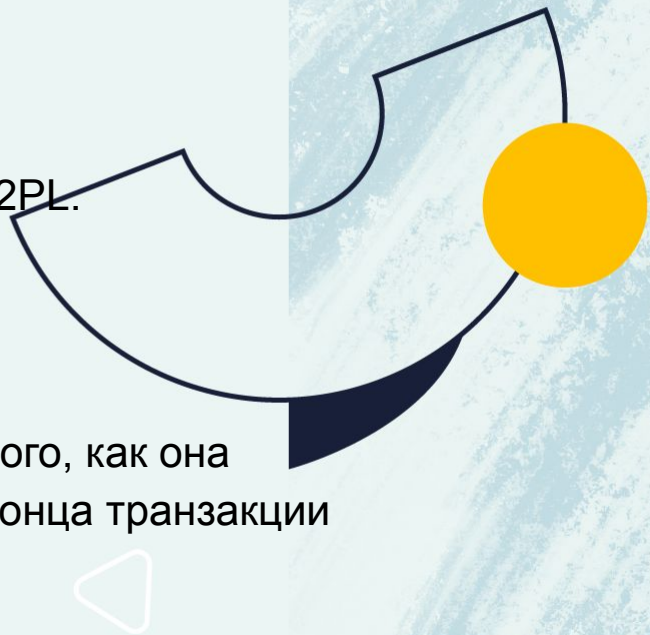
Сериализуемость может быть достигнута с помощью 2PL.

- Фаза расширения (Expanding phase)
- Фаза сокращения (Shrinking phase)

Транзакция не может взять новую блокировку, после того, как она освободила хотя бы одну. Блокировка держалась до конца транзакции обычно.

[The Notions of Consistency and Predicate Locks in a Database System](#)

by Kapali.P. Eswaran, Jim N. Gray, R.A. Lorie, and I.L. Traiger



Transaction-1

BEGIN;

**SELECT id, title FROM task WHERE
status = 'in-progress'; (1, 'original')
FOR UPDATE;**

**// after that the application applies
additional logic and filtering**

**UPDATE task SET status = 'auto-closed'
WHERE id IN (1);**

COMMIT;

Transaction-2

BEGIN;

**UPDATE task SET status = 'reopen'
WHERE id = 1;**

Tx-2 is waiting until the lock is released

COMMIT;

Actual data

(1, 'in-progress'),
(2, 'new'),
(3, 'in-progress')

(1, 'auto-closed'),
(2, 'new'),
(3, 'in-progress')

Случай с MS SQL

READ COMMITTED in SQL server

SQL server использует **shared locks** для того, чтобы другие транзакции не модифицировали прочитанные записи в первой транзакции.

Caused by: java.sql.SQLTransientConnectionException:
HikariPool-0 - Connection is not available, request timed
out after 30000ms.



```

@Path("/item")
@Transactional
public class ItemEndpoint {

    private ItemService itemService; // Calls itemMapper to execute SQL UPDATE
    private Cache<Integer, Item> itemCache; // caffeine cache

    @PUT("/")
    public Response updateCustomItem(Request request) {
        validateRequest(request);
        ItemRequestDto requestItem = retrieveItemFromRequest(request);
        Item item = convertFromDto(itemDto);

        Item updatedItem = itemService.update(item); // SQL UPDATE BY id
        itemCache.put(updatedItem.getId(), updatedItem); // ConcurrentHashMap inside
        ItemUpdatedEvent itemUpdatedEvent = createEvent(updatedItem);

        // let's send the message to kafka, so other services know about the updatedItem.
        sendKafkaMessage(itemUpdatedEvent);

        ItemResponseDto itemResponseDto = buildResponseDto(updatedItem);
        return Response.Updated(itemResponseDto);
    }
}

```

- **ALTER DATABASE item_supplier SET READ_COMMITTED_SNAPSHOT ON;**
- **ALTER DATABASE item_supplier SET ALLOW_SNAPSHOT_ISOLATION ON;**

- Транзакционный код должен содержать только изменения / чтения в базе (никаких кэшей, очередей, S3 и т.д).
- Инвалидируй кэш после транзакции (либо перед).
- "Outbox pattern", "Saga" если нужно сделать атомарность записи в базу и очередь.
- **Нужно понимание, как твоя технология работает изнутри, перед тем, как её использовать.**

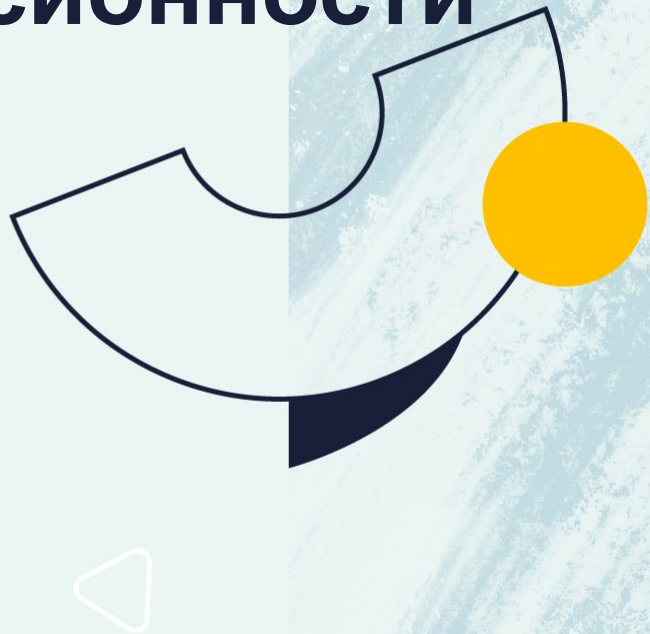


Параллельный доступ посредством многоверсионности

Multiversion concurrency control

Блокировки сильно ограничивают время ответа и масштабируемость. Транзакции должны ждать, пока получат нужную им блокировку. Долго-выполняющиеся транзакции сильно замедляют параллельные транзакции.

MVCC обещает: читатели не будут блокировать писателей, и писатели не блокируют читателей. Только операция записи может заблокировать другую операцию записи, так что остальные конкурирующие транзакции будут откочены.



Структура кортежа (Tuple Structure)

Снимок ряда создаётся для каждого его изменения

<https://www.postgresql.org/docs/15/ddl-system-columns.html>

cmax The command identifier within the deleting transaction, or zero.

cmin The command identifier (starting at zero) within the inserting transaction.

ctid The physical location of the row version within its table.

Row

xmin	xmax	cmin	cmax	ctid	User data
------	------	------	------	------	-----------

xmin holds the id of the transaction that inserted this tuple.

xmax holds the id of the transaction that deleted or updated this tuple.

Снимок (Snapshot)



- Каждая таблица и индекс сохранены как массив страниц фиксированного размера.
- Страница может содержать несколько версий одного и того же ряда.
- Каждая транзакция может видеть максимум одну версию ряда.
- Видимые версии всех рядов - и составляют снимок.
- В снимке будут данные, которые были закомичены перед тем, как снимок был создан.



Структура снимка

xmin is the snapshot's lower boundary, which is represented by the ID of the oldest active transaction.

xmax is the snapshot's upper boundary, which is represented by the value that exceeds the ID of the latest committed transaction by one.

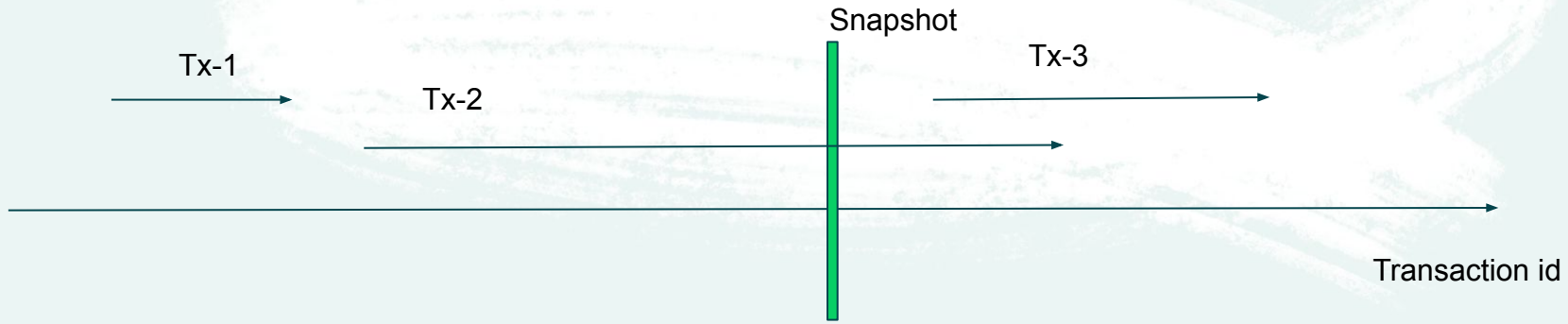
The upper boundary defines the moment when the snapshot was taken.

xip_list is the list of IDs of all the active transactions (except for virtual ones)



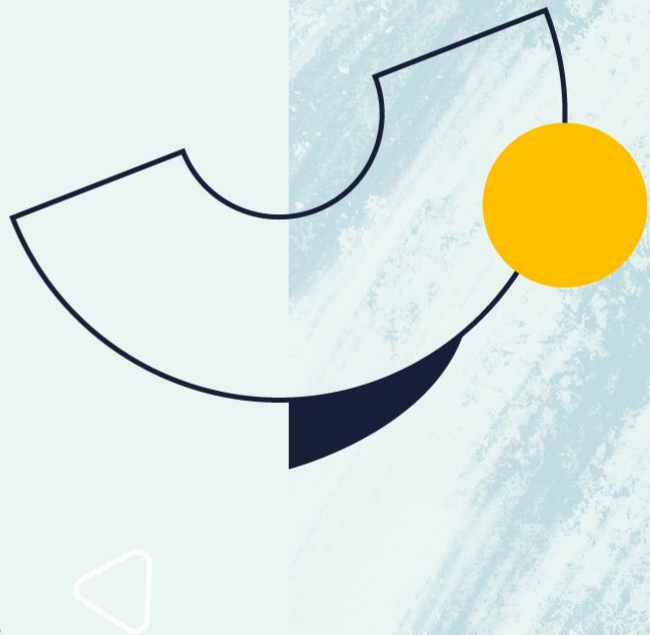
Видимость версии кортежа

https://github.com/postgres/postgres/blob/master/src/backend/access/heap/heapam_visibility.c#L169

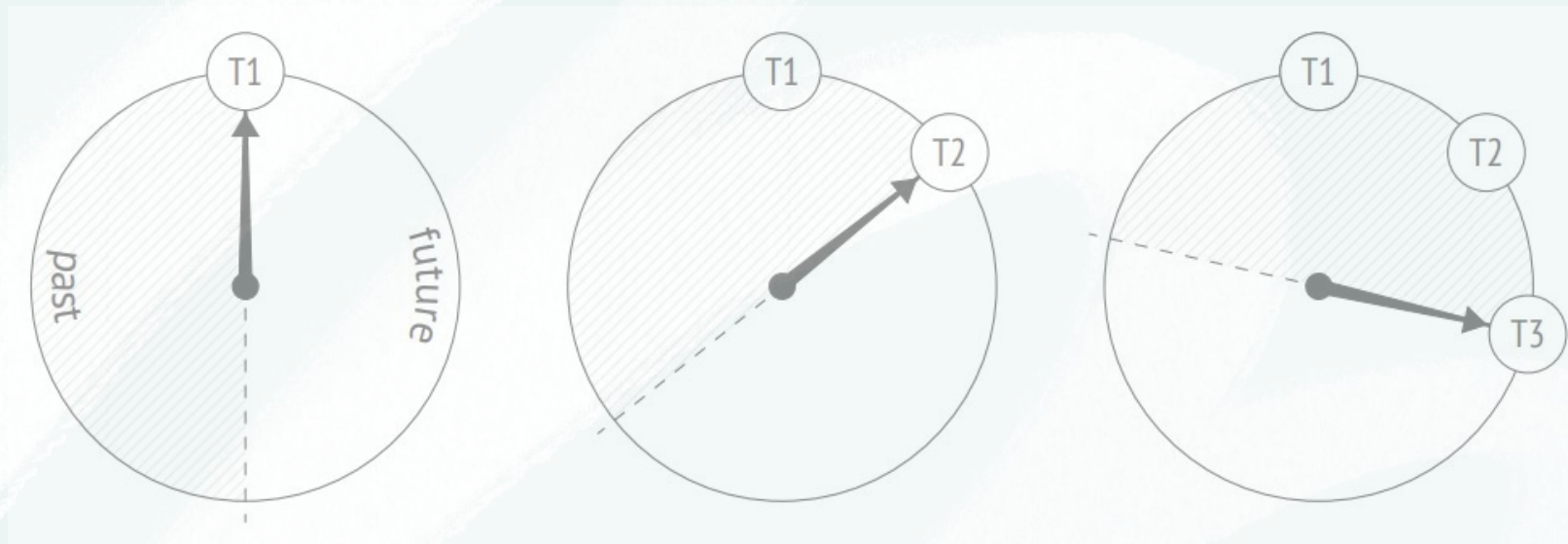


Id транзакции

Когда транзакция начинается, менеджер транзакций (transaction manager) присваивает ей уникальный id (txid).



Переполнение счетчика транзакций (Transaction ID wraparound)



Transaction-1

BEGIN;

**SELECT xmin, xmax, id, title, ctid,
txid_current() FROM task WHERE id = 8;**

105113,0,8,task-8,"(0,18)",105115

**UPDATE task SET title = 'tx-1-title'
WHERE id = 8;**

COMMIT;

Transaction-2

**SELECT xmin, xmax, id, title, ctid, txid_current()
FROM task WHERE id = 8;**

105113,0,8,task-8,"(0,18)",105114

**SELECT xmin, xmax, id, title, ctid, txid_current()
FROM task WHERE id = 8;**

105115,0,8,tx-1-title,"(0,19)",105114

Actual data

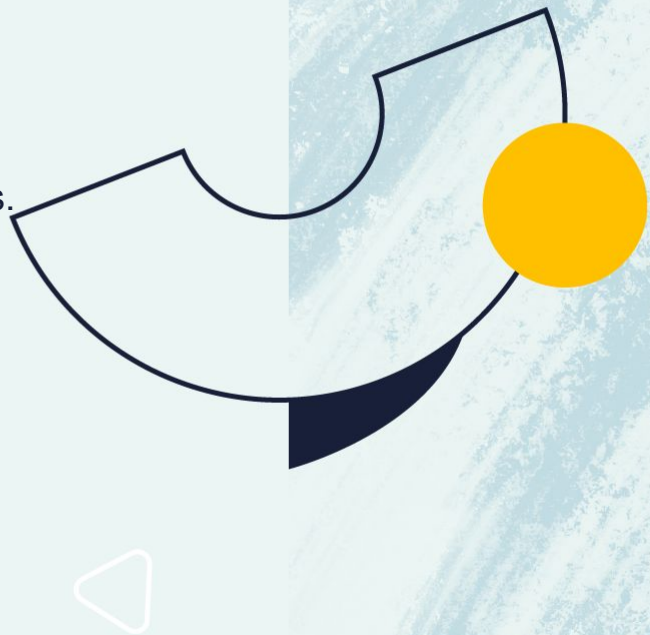
**105113,0,8,task-8,
"(0,15)"**

**105115,0,8,tx-1-title,
"(0,19)"**

VACUUM

Подчищает устаревшие кортежи и индексы, замораживает кортежи (записанные транзакциями в далёком прошлом).

VACUUM can run in parallel with production database operations.



Фазы VACUUM

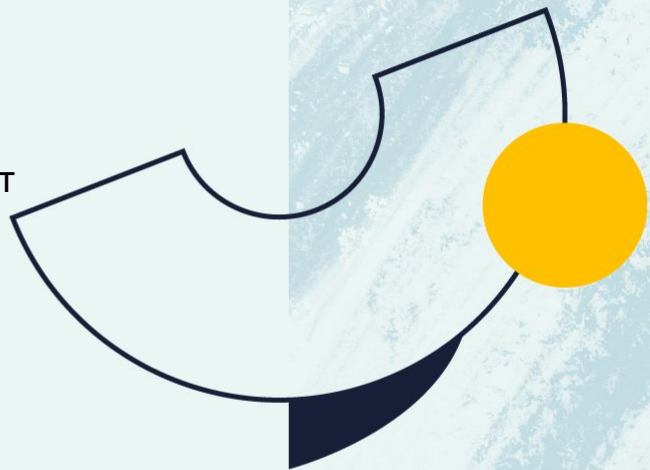
Visibility Map следит, какие страницы содержат свежие (актуальные) кортежи, которые должны быть видны активным транзакциям. Также следит за страницами, содержащие только замороженные кортежи.

Heap scan - пропускает все страницы из visibility map. Заполняет массив **tid** (указатели на кортежи, которые будут удалены).

Index scan - удаляет из индекса указатели на устаревшие кортежи, но сами кортежи ещё остаются в таблице.

Heap vacuuming - удаляет кортежи, которые присутствуют в массиве **tid** и освобождает соответствующие указатели.

Heap truncation - требует короткую исключительную блокировку на всю таблицу. Удаляет пустые страницы в конце файлов.



Выводы

- Нужно понимать, как ваши инструменты и технологии работают изнутри, что позволит правильно выбрать нужный инструмент для данной задачи. Понять какие недостатки и проблемы будут у выбранного решения, пока еще не слишком поздно (цена - миграции, откаты, переделывать заново).
- Хотите получить опыт построения распределенных приложений, научиться их поддерживать, масштабировать на глобальном уровне, мониторить их. Может быть вы просто хотите оказать влияние на глобальный продукт, сделать работу миллионов людей более продуктивной? Тогда Wrike может быть вашим следующим карьерным шагом.
- А теперь идите и смотрите за вашими базами данных :)



Спасибо

Узнайте больше о нас!

[Wrike Engineering](#)
[Read about us on Wrike TechClub](#)



Links, resources

- How does MVCC work from Vlad Michalcea
- Postgres documentation
- PostgreSQL internals (for older versions)
- https://edu.postgrespro.com/postgresql_internals-14_parts1-3_en.pdf
- Bruce Momjian about MVCC
- Architecture Isolation and MVCC https://edu.postgrespro.com/2dintro/03_arch_mvcc.pdf
- Database Internals book