ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ   
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ   
ВЫСШЕГО ОБРАЗОВАНИЯ  
МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Отчёт по лабораторной работе №1

по курсу «Базы данных»

Студент: Андрюшин Л.Д.

Группа: М8О-312Б-21

Преподаватель: Сеченых П.А.

Москва 2025

**Задание**

Предметной областью для лабораторной работы берётся из курсовой работы из прошлого сема

В задачах необходимо пользоваться таблицами, где количество строк будет не менее 5 млн

Запрещено использовать одинаковые предметные области в рамках учебной группы

Можно использовать датасеты с kaggle, либо генерировать синтетические (не рекомендуется)

**1.1: Типы индексов и их использование на практике**

Цель работы: Изучить различные типы индексов в PostgreSQL и их применение для ускорения выполнения запросов в различных предметных областях.

Задачи:

1. Внедрить B-tree, GIN, BRIN индексы для ускорения выполнения запросов.

2. Сравнить производительность до и после добавления индексов.

3. Проанализировать, в каких случаях каждый тип индекса наиболее эффективен

**1.2: Транзакции в PostgreSQL: виды и использование на практике**

Цель работы: Изучить работу с транзакциями в PostgreSQL, их применение для обеспечения целостности данных и управления параллельными операциями.

Задачи:

1. Внедрить 2-3 транзакции для своей предметной области.

2. Протестировать транзакции с разными уровнями изоляции.

3. Показать возможные аномалии.

**1.3: Использование расширений PostgreSQL для полнотекстового поиска и криптографических операций**

Цель работы: Изучить использование расширений PostgreSQL: pg\_bigm, pg\_trgm и pgcrypto для решения реальных задач по поиску и безопасности данных.

Задачи:

1. Внедрить эти расширения в свою предметную область.

2. Показать плюсы и минусы использования pg\_bigm, pg\_trgm.

3. Оценить, как использование pgcrypto влияет на безопасность данных.

**1.1: Типы индексов и их использование на практике**

B-tree (от англ. **Balanced Tree**) — это сбалансированная структура данных, которая используется по умолчанию в PostgreSQL при создании индексов. Она предназначена для эффективного выполнения операций поиска, вставки, удаления и обновления, обеспечивая при этом логарифмическую сложность O(logn).

Структурно B-tree представляет собой иерархическое дерево, в каждом узле которого содержится несколько ключей и указатели на дочерние узлы. Листовые узлы находятся на одном уровне, что гарантирует сбалансированность дерева. В этих узлах хранятся пары "ключ-указатель" на физическое расположение строки в таблице. Такой подход позволяет быстро находить нужную запись без необходимости последовательного просмотра всей таблицы.

Алгоритм поиска по B-tree начинается с корневого узла, где значение сравнивается с ключами узла. Далее выбирается соответствующее поддерево, и процесс повторяется до тех пор, пока не будет достигнут лист, содержащий ссылку на нужную запись.

B-tree индексы наиболее эффективны при использовании стандартных операций сравнения: =, <, <=, >, >=, BETWEEN, а также при выполнении сортировок и фильтраций. Например, они хорошо подходят для индексации числовых значений, дат, идентификаторов и строк, по которым часто выполняется поиск или сортировка. Кроме того, такие индексы полезны при работе с диапазонами и при использовании предложений ORDER BY.

Тем не менее, B-tree индексы менее эффективны в случаях, когда выполняются операции поиска по подстроке или по шаблону, включающему символы % в начале (например, LIKE '%abc'). В таких ситуациях применимы специализированные индексы, такие как pg\_trgm или pg\_bigm, которые поддерживают полнотекстовый и приближённый поиск.

Следует также учитывать, что при высоком уровне дублирования значений в столбце (низкой селективности) эффективность B-tree индекса снижается, так как PostgreSQL может предпочесть последовательное сканирование таблицы.

Одним из преимуществ B-tree индексов является их способность поддерживать упорядоченный доступ к данным, что делает их особенно полезными для операций сортировки. Кроме того, можно создавать составные индексы (по нескольким столбцам) или покрывающие индексы (которые содержат все данные, необходимые для выполнения запроса, позволяя избежать обращения к основной таблице).

Вот пример использования B-tree:

***EXPLAIN ANALYZE SELECT \* FROM clients WHERE email = 'user123@example.com';*** - запрос, которые показывает за какое время мы находим определенный имейл без B-tree индекса

Вот что он вывел:

***"Seq Scan on clients (cost=0.00..27539.40 rows=398 width=2076) (actual time=125.916..125.917 rows=0 loops=1)"***

***" Filter: ((email)::text = 'user123@example.com'::text)"***

***" Rows Removed by Filter: 749990"***

***"Planning Time: 0.175 ms"***

***"Execution Time: 125.933 ms"***

Как видно, он нашел нужную почту за 125.933 ms

***CREATE INDEX idx\_clients\_email ON clients(email);*** - создаем индексы

***EXPLAIN ANALYZE SELECT \* FROM clients WHERE email = 'user123@example.com';*** - теперь анализируем время выполнения того же запроса, но уже после добавления B-tree индекса

Вот что вывел этот же запрос, но уже после добавления B-tree индекса:

***"Bitmap Heap Scan on clients (cost=85.49..10326.69 rows=3750 width=2076) (actual time=0.031..0.031 rows=0 loops=1)"***

***" Recheck Cond: ((email)::text = 'user123@example.com'::text)"***

***" -> Bitmap Index Scan on idx\_clients\_email (cost=0.00..84.55 rows=3750 width=0) (actual time=0.029..0.029 rows=0 loops=1)"***

***" Index Cond: ((email)::text = 'user123@example.com'::text)"***

***"Planning Time: 0.159 ms"***

***"Execution Time: 0.047 ms"***

Как мы видим время значительно уменьшилось. На несколько порядков.

***DROP INDEX idx\_clients\_email;*** - с помощью этой команды мы удаляем индексы из нашей БД

GIN (от англ. **Generalized Inverted Index**) — это обобщённый инвертированный индекс, используемый в PostgreSQL для эффективного поиска по множеству значений, хранящихся в одном поле. В отличие от B-tree, индекс GIN предназначен для таких случаев, когда элемент данных сам по себе содержит коллекцию значений, например массив, JSON, текст с разбивкой на слова, hstore и т.д.

Принцип работы GIN заключается в хранении для каждого отдельного значения (например, слова или элемента массива) списка идентификаторов строк, в которых это значение встречается. То есть, GIN индекс создаёт **инвертированное отображение**: не от строки к значению, как в обычных индексах, а от значения к строкам. Благодаря этому возможно быстро определить, какие строки содержат конкретный элемент.

Такая структура идеально подходит для задач, связанных с полнотекстовым поиском, когда требуется находить документы, содержащие определённые слова, фразы или даже набор слов. PostgreSQL применяет GIN-индексы, например, при использовании tsvector/tsquery в полнотекстовом поиске.

Ключевые особенности GIN-индексов:

* **Высокая производительность при запросах на включение**: GIN особенно эффективен, когда нужно проверить, содержится ли элемент в массиве или документе.
* **Медленное обновление**: вставка и обновление строк, использующих GIN-индексы, происходят медленнее, чем у B-tree, поскольку необходимо обновить множество ссылок.
* **Высокая стоимость создания**: построение GIN-индекса занимает больше времени и ресурсов, чем обычных индексов.
* **Требуют дополнительной настройки**: для оптимизации GIN можно использовать параметры fastupdate, gin\_pending\_list\_limit и др.

Вот пример использования GIN:

Для начала подготовим данные для использования GIN индекса:  
***ALTER TABLE tags ADD COLUMN search\_vector tsvector;  
  
UPDATE tags SET search\_vector = to\_tsvector('english', description);***

Теперь проверим время работы программы до подключения индексов:

***EXPLAIN ANALYZE SELECT \* FROM tags***

***WHERE to\_tsvector('english', description) @@ plainto\_tsquery('english', 'building');***

Вот вывод данного запроса:

***"Seq Scan on tags (cost=0.00..30.55 rows=1 width=552) (actual time=0.136..1.634 rows=135 loops=1)"***

***" Filter: (description ~~\* '%building%'::text)"***

***" Rows Removed by Filter: 865"***

***"Planning Time: 0.065 ms"***

***"Execution Time: 1.656 ms"***

Теперь добавим GIN индексы:  
***CREATE INDEX idx\_tags\_search\_vector ON tags USING GIN(search\_vector);***  
И проверим как быстро будет работать поиск уже с GIN индексом:

***EXPLAIN ANALYZE SELECT \* FROM tags***

***WHERE search\_vector @@ plainto\_tsquery('english', 'building');***

Вот вывод данного запроса:

***"Bitmap Heap Scan on tags (cost=8.04..23.21 rows=5 width=584) (actual time=0.035..0.080 rows=135 loops=1)"***

***" Recheck Cond: (search\_vector @@ '''build'''::tsquery)"***

***" Heap Blocks: exact=22"***

***" -> Bitmap Index Scan on idx\_tags\_search\_vector (cost=0.00..8.04 rows=5 width=0) (actual time=0.022..0.022 rows=135 loops=1)"***

***" Index Cond: (search\_vector @@ '''build'''::tsquery)"***

***"Planning Time: 0.165 ms"***

***"Execution Time: 0.102 ms"***

Как можно заметить время выполнения запроса значительно уменьшилось

Ну и теперь удалим и индексы и специальную колонку, которую мы ради них добавляли:

***DROP INDEX idx\_tags\_search\_vector;***

***ALTER TABLE tags DROP COLUMN search\_vector;***

1. Сравнение производительности

* Без GIN-индекса: каждый запрос будет медленно сканировать всю таблицу, проверяя, есть ли текст в колонке description.
* С GIN-индексом: PostgreSQL использует индекс для быстрого нахождения строк, которые содержат ключевые слова, и выполнение такого запроса в разы быстрее, особенно если таблица большая.

2. Когда использовать GIN?

* GIN-индекс хорошо подходит для поиска по полнотекстовым данным, где в одном поле может быть много информации (например, описание механик).
* Этот индекс эффективен, когда ты часто ищешь по частям строк (например, слова или фразы) и хочешь ускорить поисковые запросы.

3. Важное отличие

* Без индекса мы выполняем линейный поиск, где PostgreSQL проверяет каждое поле.
* С GIN-индексом PostgreSQL использует структуру данных, которая позволяет искать быстрее, избегая полного сканирования.

**BRIN (Block Range Index)** — это тип индексной структуры, предназначенный для эффективной работы с очень большими таблицами, особенно в случаях, когда данные в таблице имеют естественную упорядоченность. Вместо хранения ссылок на конкретные строки, как это делают традиционные индексы (например, B-tree), BRIN-индекс хранит агрегированную информацию о диапазонах смежных блоков таблицы, таких как минимальные и максимальные значения индексируемого столбца в каждом диапазоне.

Главная идея BRIN заключается в том, чтобы не индексировать каждую запись отдельно, а охватывать определённое количество последовательных страниц (блоков) таблицы и описывать их содержимое с помощью метаинформации. Это позволяет значительно уменьшить объём самого индекса и ускорить его построение и обновление.

BRIN-индексы особенно эффективны в тех случаях, когда данные в таблице имеют физическую корреляцию с логическими значениями — например, когда строки добавляются в порядке возрастания времени или чисел. При таких условиях BRIN позволяет быстро исключить целые группы блоков, не содержащих нужных данных, что делает поиск гораздо быстрее по сравнению с полным перебором.

Одним из ключевых преимуществ BRIN-индексов является их компактность — они занимают существенно меньше места на диске, чем B-tree или GIN, и требуют меньше ресурсов на сопровождение. Это делает их особенно полезными при работе с таблицами, содержащими миллионы или миллиарды записей. Также их создание и сопровождение в случае обновлений происходит гораздо быстрее.

Однако у BRIN есть и ограничения. Они не подходят для точечного поиска и не дают высокоточной фильтрации, особенно если значения в таблице распределены случайным образом. Если порядок значений нарушен или данные часто обновляются вразброс, эффективность BRIN падает. Кроме того, при низкой селективности запроса (то есть если извлекается значительная часть таблицы) такие индексы могут не дать прироста производительности.

BRIN-индексы находят своё применение, прежде всего, в таблицах журналов, логов и временных рядов, где данные добавляются последовательно. Они особенно полезны в аналитических задачах, когда необходимо сканировать большие объёмы информации с минимальными затратами ресурсов.

Таким образом, BRIN представляет собой лёгкий и быстрый индекс, оптимальный для задач, где важна экономия пространства и разумная производительность при обработке больших объёмов упорядоченных данных.

Вот пример использования BRIN индексов:

Для начала замерием время без индексов:

***EXPLAIN ANALYZE***

***SELECT \****

***FROM rentals***

***WHERE start\_date >= '2024-03-13' AND start\_date <= '2024-05-21';***

Вот вывод данного запроса:  
***"Gather (cost=1000.00..47508.36 rows=18408 width=20) (actual time=19.009..95.512 rows=325497 loops=1)"***

***" Workers Planned: 2"***

***" Workers Launched: 2"***

***" -> Parallel Seq Scan on rentals (cost=0.00..44667.56 rows=7670 width=20) (actual time=7.291..65.259 rows=108499 loops=3)"***

***" Filter: ((start\_date >= '2024-03-13'::date) AND (start\_date <= '2024-05-21'::date))"***

***" Rows Removed by Filter: 1024834"***

***"Planning Time: 0.049 ms"***

***"Execution Time: 104.258 ms"***

Теперь добавим индексы

***CREATE INDEX idx\_rentals\_start\_date\_brin ON rentals USING BRIN (start\_date);***

***SET max\_parallel\_workers\_per\_gather = 0;***

***SET enable\_seqscan = false;***

***EXPLAIN ANALYZE***

***SELECT \****

***FROM rentals***

***WHERE start\_date >= '2024-03-01' AND start\_date <= '2024-03-04';***

Вот такой вывод мы получили:

***"Bitmap Heap Scan on rentals (cost=21.69..72678.69 rows=17000 width=20) (actual time=2.507..5.550 rows=18369 loops=1)"***

***" Recheck Cond: ((start\_date >= '2024-03-01'::date) AND (start\_date <= '2024-03-04'::date))"***

***" Rows Removed by Index Recheck: 21823"***

***" Heap Blocks: lossy=256"***

***" -> Bitmap Index Scan on idx\_rentals\_start\_date\_brin (cost=0.00..17.44 rows=3400000 width=0) (actual time=0.077..0.077 rows=2560 loops=1)"***

***" Index Cond: ((start\_date >= '2024-03-01'::date) AND (start\_date <= '2024-03-04'::date))"***

***"Planning Time: 0.058 ms"***

***"Execution Time: 6.283 ms"***

Как заметно, время с индексом запрос работает значительно быстрее

Теперь отключим индексы с помощью данного запроса:

***DROP INDEX idx\_rentals\_start\_date\_brin;***

**1.2: Транзакции в PostgreSQL: виды и использование на практике**

Транзакция — это логическая единица работы с базой данных, которая представляет собой последовательность операций, выполняемых как единое целое. Цель транзакции — обеспечить согласованность и надёжность данных при выполнении множества операций, особенно в условиях многопользовательского доступа и возможных сбоев.

Основу концепции транзакций составляют **четыре свойства**, объединённые под акронимом **ACID**:

1. **Atomicity (атомарность)** — транзакция выполняется полностью или не выполняется вовсе. Если в процессе её выполнения происходит ошибка, все изменения откатываются до исходного состояния.
2. **Consistency (согласованность)** — после завершения транзакции база данных остаётся в согласованном состоянии, соблюдаются все ограничения и правила целостности.
3. **Isolation (изолированность)** — параллельные транзакции не должны мешать друг другу; каждая из них должна выполняться как бы в одиночку, даже если фактически они происходят одновременно.
4. **Durability (надёжность)** — после фиксации транзакции её изменения сохраняются в базе данных и не теряются даже при сбое системы.

Транзакции позволяют обеспечить корректное поведение в таких ситуациях, как перевод денег между счетами, бронирование билетов, регистрация заказов и т. д., где критически важно, чтобы изменения не были частичными.

#### Работа транзакций

Транзакция начинается явным или неявным образом. Все последующие операции (вставка, обновление, удаление и т.д.) накапливаются до тех пор, пока транзакция не будет либо зафиксирована (commit), либо отменена (rollback). Фиксация делает все изменения постоянными. Откат полностью аннулирует все действия, выполненные в рамках данной транзакции.

#### Типы изоляции транзакций

В системах управления базами данных транзакции могут исполняться с разным уровнем изоляции, влияющим на поведение параллельных операций. В стандарте SQL определены следующие уровни:

1. **Read Uncommitted** — самый низкий уровень, допускает чтение незавершённых (нефиксированных) изменений других транзакций. Возможны "грязные чтения".
2. **Read Committed** — предотвращает грязные чтения, но допускает неповторяющиеся чтения и фантомы.
3. **Repeatable Read** — обеспечивает повторяемость чтений: одна и та же строка не изменится в ходе транзакции. Однако фантомы (новые строки, подходящие под условие) всё ещё возможны.
4. **Serializable** — самый высокий уровень, полностью исключает все виды аномалий. Параллельные транзакции исполняются так, как если бы выполнялись последовательно.

Выбор уровня изоляции всегда представляет собой компромисс между производительностью и безопасностью данных. Чем выше уровень, тем надёжнее защита от аномалий, но тем ниже параллелизм и выше накладные расходы.

#### Назначение транзакций

Транзакции необходимы для обеспечения целостности данных в условиях:

* **Сбоев оборудования или программного обеспечения** — например, при отключении питания во время обновления таблицы;
* **Ошибок в бизнес-логике** — откат позволяет исправить некорректные действия;
* **Параллельного доступа** — несколько пользователей могут одновременно обращаться к одним и тем же данным.

Таким образом, транзакции являются основой надёжности и согласованности в реляционных базах данных, позволяя системам гарантировать корректную работу даже в самых сложных и нагруженных сценариях.

### Аномалии параллельного выполнения транзакций

При параллельном выполнении нескольких транзакций, если не обеспечивается достаточный уровень изоляции, в базе данных могут возникать **аномалии согласованности данных**. Эти аномалии нарушают логическую целостность и могут приводить к ошибочным результатам, даже несмотря на корректность отдельных транзакций.

Существует три основных типа аномалий:

#### 1. **Грязное чтение (Dirty Read)**

Это ситуация, когда одна транзакция читает данные, изменённые другой транзакцией, которая ещё не зафиксирована. Если вторая транзакция затем откатится, первая окажется использовавшей недопустимую (несуществующую) информацию.

**Пример:**  
Транзакция A обновляет значение, но не фиксирует изменения. Транзакция B читает это значение. Затем A откатывается, но B уже использовала "грязные" данные.

**Возникает на уровне изоляции:** Read Uncommitted.

#### 2. **Неповторяющееся чтение (Non-Repeatable Read)**

Происходит, когда одна транзакция дважды читает одну и ту же строку, и между этими чтениями другая транзакция изменяет или удаляет эту строку. В результате значения на первом и втором чтении оказываются разными.

**Пример:**  
Транзакция A читает строку X. Транзакция B изменяет X и фиксирует изменения. Транзакция A снова читает X и получает уже другое значение.

**Возникает на уровне изоляции:** Read Committed.

#### 3. **Фантомное чтение (Phantom Read)**

Аномалия возникает, когда транзакция повторно выполняет запрос с одним и тем же условием, а между этими вызовами другая транзакция вставляет новые строки, которые теперь соответствуют этому условию. Таким образом, результат запроса неожиданно "расширяется" фантомными строками.

**Пример:**  
Транзакция A делает выборку всех заказов со стоимостью > 1000. Транзакция B добавляет такой заказ и фиксирует. При повторной выборке A увидит новую строку, которую "не было" при первом вызове.

**Возникает на уровне изоляции:** Repeatable Read.

Пример  **Non-Repeatable Read:**

**В таблице rentals у нас есть такая строка:**

***rental\_id = 1, copy\_id = 123, end\_date = '2024-01-01'***

**Сессия 1:**

***BEGIN;***

***SET TRANSACTION ISOLATION LEVEL READ COMMITTED;***

***SELECT end\_date FROM rentals WHERE rental\_id = 1;***

**-- Результат: '2024-01-01'**

**Сессия 2:**

***BEGIN;***

***UPDATE rentals SET end\_date = '2024-02-01' WHERE rental\_id = 1;***

***COMMIT;***

**Сессия 1:**

***SELECT end\_date FROM rentals WHERE rental\_id = 1;***

**-- Результат: '2024-02-01'**

**Хотя мы еще не сделали коммита!!!**

**Это** неповторяющееся чтение **— данные изменились между двумя SELECT в одной транзакции.**

**👻 Пример фантомного чтения**

Сессия 1:

BEGIN;

***SET TRANSACTION ISOLATION LEVEL READ COMMITTED;***

***SELECT COUNT(\*) FROM copies WHERE status = 'available';***

-- Результат: 2

Сессия 2:

***BEGIN;***

***INSERT INTO copies (copy\_id, game\_id, status, location\_id)***

***VALUES (3, 10, 'available', 1);***

COMMIT;

Сессия 1:

***SELECT COUNT(\*) FROM copies WHERE status = 'available';***

-- Результат: 3

Это и есть фантомное чтение. Между двумя SELECT появилась новая строка, которая «привиделась».

**1.3: Использование расширений PostgreSQL для полнотекстового поиска и криптографических операций**

**pg\_bigm**

Расширение pg\_bigm в PostgreSQL предоставляет возможность эффективного полнотекстового поиска по подстроке за счёт построения индексов на основе биграмм. Биграммы — это пары подряд идущих символов в строке. Например, строка hello разбивается на биграммы he, el, ll, lo. Это позволяет ускорить выполнение запросов с операторами LIKE '%substring%', которые в стандартной реализации не используют индекс и выполняются крайне медленно при работе с большими объёмами текста. После установки расширения и создания GIN-индекса с использованием оператора bigm\_ops, PostgreSQL может использовать данный индекс при выполнении таких запросов, значительно уменьшая время ответа.

Преимущество pg\_bigm заключается в его простоте и высокой производительности при поиске по подстроке. Он особенно полезен в тех случаях, когда необходимо искать в больших текстовых колонках (например, названия продуктов, описания или сообщения), и когда поиск по шаблону должен быть максимально быстрым. Однако у pg\_bigm есть и ограничения. Во-первых, индексы на основе биграмм могут занимать значительно больше места по сравнению с классическими B-tree или даже pg\_trgm. Во-вторых, в отличие от pg\_trgm, pg\_bigm не предоставляет средств для измерения степени схожести между строками. Это делает его менее подходящим для задач, где важно не просто нахождение совпадения, но и ранжирование результатов по релевантности.

В целом, pg\_bigm — это мощный инструмент, когда необходимо реализовать быстрый подстрочный поиск без необходимости в сложных механизмах оценки сходства. Его стоит использовать в тех проектах, где важна скорость ответа на простые текстовые запросы и где объёмы данных не позволяют выполнять полные переборы.

**pg\_trgm**

Расширение pg\_trgm — одно из самых широко используемых в PostgreSQL расширений для реализации полнотекстового поиска и приближённого сравнения строк. Оно основано на концепции триграмм, то есть последовательностей из трёх символов. При добавлении строки в индекс она разбивается на триграммы (например, hello → hel, hel, ell, llo, lo ), и именно они индексируются. Это позволяет PostgreSQL использовать GIN или GiST индекс для ускорения поиска по шаблонам с LIKE, ILIKE, SIMILAR TO, а также выполнять сравнение строк по степени схожести.

Одним из главных достоинств pg\_trgm является возможность измерять степень похожести строк через встроенные функции similarity, word\_similarity, strict\_word\_similarity и оператор %, которые позволяют выполнять fuzzy-поиск. Это особенно важно в случаях, когда пользователь может вводить запрос с ошибками, либо требуется найти строки, содержащие похожие фразы. Например, pg\_trgm позволяет эффективно находить строки, содержащие опечатки, что делает его незаменимым в проектах, связанных с пользовательским вводом, поиском по именам, товарам или адресам.

Среди недостатков pg\_trgm можно отметить то, что он может давать более слабую производительность при банальных LIKE '%...%' по сравнению с pg\_bigm, особенно на очень больших объёмах данных. Также его настройка требует выбора правильного типа индекса (GIN или GiST) в зависимости от задачи, что может быть неочевидным для начинающих. Индексы на pg\_trgm тоже занимают немало дискового пространства, хотя обычно меньше, чем у pg\_bigm.

Таким образом, pg\_trgm идеально подходит для реализации пользовательского поиска по вводу с ошибками, приближённого поиска по строкам и сравнения текстов по степени сходства. Это расширение обеспечивает гибкий и мощный механизм работы с неточными строками, которого нет ни в одном стандартном индексе PostgreSQL.

**Pgcrypto**

Расширение pgcrypto в PostgreSQL предоставляет средства для выполнения криптографических операций прямо внутри базы данных. Оно включает функции для хэширования, симметричного и асимметричного шифрования, генерации случайных чисел и работы с цифровыми подписями. Основное назначение pgcrypto — обеспечить безопасность хранения и обработки чувствительных данных, таких как пароли, личные данные, токены и т. д.

Одним из наиболее часто используемых сценариев применения pgcrypto является безопасное хэширование паролей с использованием функции crypt() в сочетании с gen\_salt(), позволяющей генерировать соль для хэша. Например, при сохранении пароля можно выполнить crypt('password123', gen\_salt('bf')), где 'bf' означает использование алгоритма Blowfish. Это обеспечивает защиту от прямого восстановления пароля при компрометации базы данных. Проверка пароля затем осуществляется с помощью crypt(input\_password, stored\_hash) = stored\_hash, что не требует знания исходного пароля.

Кроме хэширования, pgcrypto поддерживает симметричное шифрование (через pgp\_sym\_encrypt) и асимметричное (через pgp\_pub\_encrypt, pgp\_priv\_decrypt), совместимое с PGP (Pretty Good Privacy). Это позволяет шифровать данные как с ключами пользователя, так и с открытыми/закрытыми ключами, обеспечивая гибкую модель безопасности. Однако важно понимать, что использование шифрования требует аккуратного управления ключами: в случае их потери расшифровать данные будет невозможно.

К преимуществам pgcrypto можно отнести встроенность в PostgreSQL, отсутствие необходимости передавать данные во внешние приложения и достаточно простую реализацию базовых сценариев безопасности. При этом к недостаткам относится нагрузка на процессор при массовом шифровании и потенциальное снижение производительности при активном использовании. Также важно учитывать, что pgcrypto не защищает данные на уровне физических файлов — оно предназначено именно для защиты на уровне содержимого полей.

Таким образом, pgcrypto является мощным и гибким инструментом для реализации шифрования и хэширования данных в PostgreSQL, особенно при построении безопасных систем, где данные должны быть защищены не только на уровне доступа к таблице, но и на уровне их содержимого. Его стоит использовать для защиты паролей, персональных данных, ключей доступа и других чувствительных сведений.

Пример использования pg\_trgm:

***CREATE EXTENSION IF NOT EXISTS pg\_trgm;***

***CREATE INDEX trgm\_idx ON games USING gin (game\_name gin\_trgm\_ops);***

***SET pg\_trgm.similarity\_threshold = 0.1;*** - можем задать любое значение, данное число является нашим «процентом совпадения»

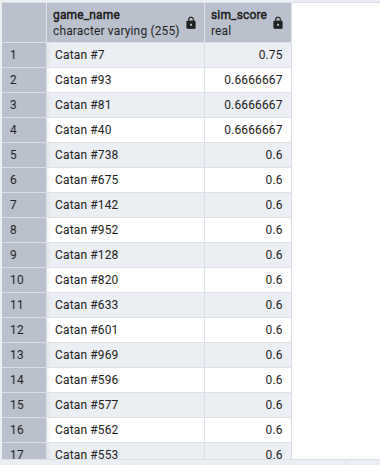
***SELECT game\_name, similarity(game\_name, 'Catan') AS sim\_score***

***FROM games***

***WHERE game\_name % 'Catan'***

***ORDER BY sim\_score DESC;***

Вывод:



Где числа справа от game\_name — проценты совпадения.

Удаляем индексы:

***DROP INDEX IF EXISTS trgm\_idx;***

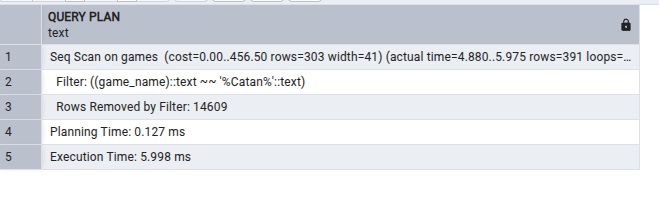
***DROP EXTENSION IF EXISTS pg\_trgm;***

Пример использования pg\_bigm:

Замерим скорость поиска без индексов:

EXPLAIN ANALYZE

SELECT \* FROM games WHERE game\_name LIKE '%Catan%';



Теперь добавим индексы:

***CREATE EXTENSION pg\_bigm;***

***CREATE INDEX game\_name\_bigm\_idx***

***ON games***

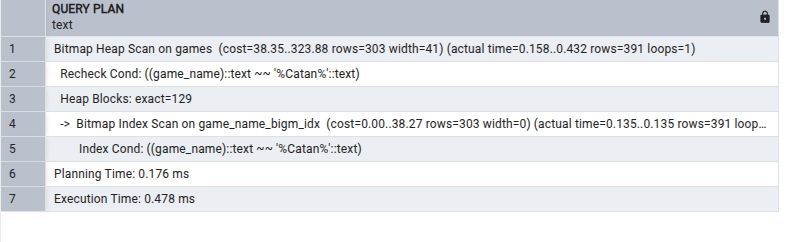
***USING gin (game\_name gin\_bigm\_ops);***

***EXPLAIN ANALYZE***

***SELECT \****

***FROM games***

***WHERE game\_name LIKE '%Catan%';***



Как можно заметить скорость возросла в 10 раз.

Удалим индексы

***DROP INDEX IF EXISTS game\_name\_bigm\_idx;***

***DROP EXTENSION IF EXISTS pg\_bigm;***

Пример использования pgcrypto:

***CREATE EXTENSION IF NOT EXISTS pgcrypto;***

***INSERT INTO users (user\_id, username, password, user\_type)***

***VALUES (***

***750001,***

***'vika',***

***crypt('supersecure123', gen\_salt('bf')), 'client'***

***);***



***UPDATE clients***

***SET email = pgp\_sym\_encrypt(email, 'mysecretkey')***

***WHERE client\_id = 1;***



Данные для данной лабораторной работы были созданы с помощью следующего кода:

***import csv***

***import random***

***from faker import Faker***

***from tqdm import tqdm***

***import os***

***from datetime import timedelta***

***fake = Faker()***

***os.makedirs("generated\_data", exist\_ok=True)***

***ROW\_COUNTS = {***

***"users": 750\_000,***

***"clients": 749\_990,***

***"games": 15\_000,***

***"tags": 1\_000,***

***"games\_to\_tags": 90\_000,***

***"locations": 500,***

***"copies": 1\_000\_000,***

***"rentals": 3\_400\_000***

***}***

***games\_list = [***

***"Catan", "Ticket to Ride", "Carcassonne", "Dominion", "7 Wonders", "Pandemic", "Azul", "Splendor",***

***"Terraforming Mars", "Scythe", "Root", "Wingspan", "Gloomhaven", "Clank!", "Codenames",***

***"Everdell", "Brass: Birmingham", "The Crew", "Ark Nova", "Lost Ruins of Arnak", "Patchwork",***

***"Tzolk'in", "Agricola", "Twilight Struggle", "Blood Rage", "Viticulture", "The Quacks of Quedlinburg",***

***"Isle of Skye", "Tiny Towns", "Decrypto", "The Mind", "Paladins of the West Kingdom", "Cascadia",***

***"Project L", "Obsession", "Sleeping Gods", "Dune: Imperium", "PARKS", "Calico", "The Isle of Cats"***

***]***

***tags\_list = [***

***"Strategy", "Deck Building", "Card Drafting", "Cooperative", "Worker Placement",***

***"Dice Rolling", "Tile Placement", "Bluffing", "Resource Management", "Area Control",***

***"Auction", "Set Collection", "Hand Management", "Engine Building", "Storytelling",***

***"Hidden Roles", "Modular Board", "Variable Player Powers", "Trading", "Pattern Building"***

***]***

***def generate\_users():***

***with open("generated\_data/users.csv", "w", newline="", encoding="utf-8") as f:***

***writer = csv.writer(f)***

***writer.writerow(["user\_id", "password", "username", "user\_type"])***

***for i in tqdm(range(1, ROW\_COUNTS["users"] + 1), desc="users"):***

***if i <= 10:***

***user\_type = "admin"***

***else:***

***user\_type = "client"***

***name = fake.first\_name()***

***surname = fake.last\_name()***

***username = f"{name}\_{surname}\_{i}"***

***writer.writerow([i, "Pass1234", username, user\_type])***

***def generate\_clients():***

***with open("generated\_data/clients.csv", "w", newline="", encoding="utf-8") as f:***

***writer = csv.writer(f)***

***writer.writerow(["client\_id", "user\_id", "name", "phone\_number", "client\_address", "email", "birth\_date"])***

***for i in tqdm(range(1, ROW\_COUNTS["clients"] + 1), desc="clients"):***

***full\_name = fake.name()***

***writer.writerow([***

***i, i + 10, full\_name, fake.phone\_number(),***

***fake.address().replace('\n', ', '), fake.email(),***

***fake.date\_of\_birth(minimum\_age=18, maximum\_age=90)***

***])***

***def generate\_games():***

***with open("generated\_data/games.csv", "w", newline="", encoding="utf-8") as f:***

***writer = csv.writer(f)***

***writer.writerow(["game\_id", "game\_name", "min\_players", "max\_players", "release\_year", "play\_time", "age\_rating"])***

***for i, game in enumerate(games\_list, start=1):***

***writer.writerow([***

***i, game, random.randint(1, 4), random.randint(2, 10),***

***random.randint(1990, 2023), random.randint(15, 180),***

***random.randint(6, 18)***

***])***

***def generate\_tags():***

***with open("generated\_data/tags.csv", "w", newline="", encoding="utf-8") as f:***

***writer = csv.writer(f)***

***writer.writerow(["tag\_id", "name", "description"])***

***for i, tag in enumerate(tags\_list, start=1):***

***description = f"This tag is related to {tag.lower()} mechanics."***

***writer.writerow([i, tag, description])***

***def generate\_games\_to\_tags():***

***max\_game\_id = len(games\_list)***

***max\_tag\_id = len(tags\_list)***

***with open("generated\_data/games\_to\_tags.csv", "w", newline="", encoding="utf-8") as f:***

***writer = csv.writer(f)***

***writer.writerow(["id", "game\_id", "tag\_id"])***

***for i in tqdm(range(1, ROW\_COUNTS["games\_to\_tags"] + 1), desc="games\_to\_tags"):***

***writer.writerow([i, random.randint(1, max\_game\_id), random.randint(1, max\_tag\_id)])***

***def generate\_locations():***

***with open("generated\_data/locations.csv", "w", newline="", encoding="utf-8") as f:***

***writer = csv.writer(f)***

***writer.writerow(["location\_id", "location\_address", "location\_name"])***

***for i in tqdm(range(1, ROW\_COUNTS["locations"] + 1), desc="locations"):***

***writer.writerow([***

***i, fake.address().replace('\n', ', '),***

***f"{fake.city()} Game Center #{i}"***

***])***

***def generate\_copies():***

***max\_game\_id = len(games\_list)***

***max\_location\_id = ROW\_COUNTS["locations"]***

***with open("generated\_data/copies.csv", "w", newline="", encoding="utf-8") as f:***

***writer = csv.writer(f)***

***writer.writerow(["copy\_id", "game\_id", "status", "location\_id"])***

***for i in tqdm(range(1, ROW\_COUNTS["copies"] + 1), desc="copies"):***

***writer.writerow([***

***i,***

***random.randint(1, max\_game\_id),***

***random.choice(["available", "rented", "maintenance"]),***

***random.randint(1, max\_location\_id)***

***])***

***def generate\_rentals():***

***max\_copy\_id = ROW\_COUNTS["copies"]***

***max\_client\_id = ROW\_COUNTS["clients"]***

***with open("generated\_data/rentals.csv", "w", newline="", encoding="utf-8") as f:***

***writer = csv.writer(f)***

***writer.writerow(["rental\_id", "copy\_id", "start\_date", "end\_date", "client\_id"])***

***for i in tqdm(range(1, ROW\_COUNTS["rentals"] + 1), desc="rentals"):***

***start = fake.date\_between(start\_date='-2y', end\_date='today')***

***end = start + timedelta(days=random.randint(1, 30))***

***writer.writerow([***

***i,***

***random.randint(1, max\_copy\_id),***

***start,***

***end,***

***random.randint(1, max\_client\_id)***

***])***

***if \_\_name\_\_ == "\_\_main\_\_":***

***generate\_users()***

***generate\_clients()***

***generate\_games()***

***generate\_tags()***

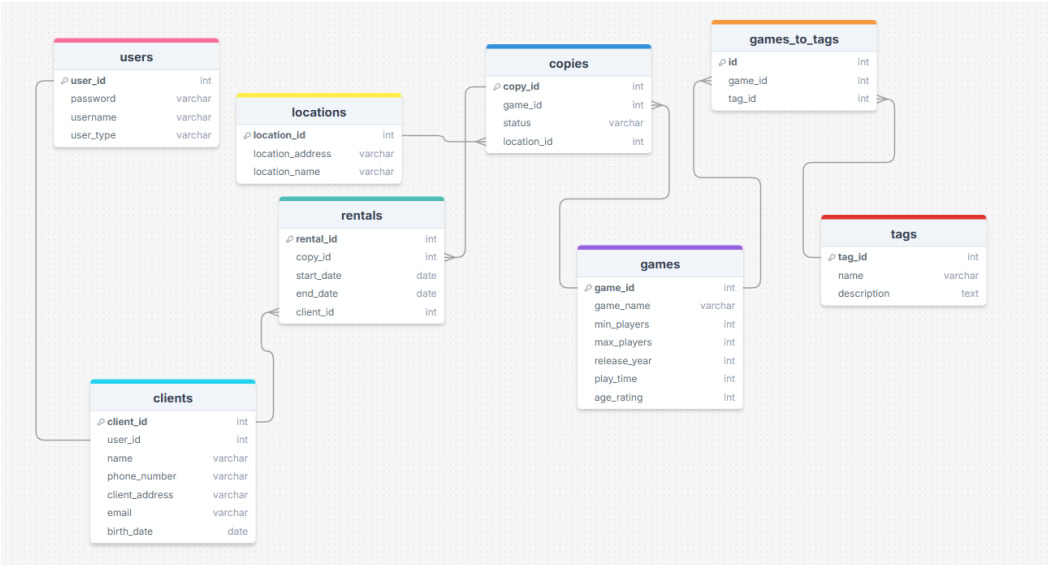
***generate\_games\_to\_tags()***

***generate\_locations()***

***generate\_copies()***

***generate\_rentals()***

***И вот так выглядит сама база данных:***

******

**Вывод:**

В ходе выполнения лабораторной работы я получил практические навыки по оптимизации работы с базами данных PostgreSQL в контексте крупной предметной области, содержащей таблицы с объёмом не менее пяти миллионов строк. Особое внимание было уделено изучению различных типов индексов, транзакционных механизмов и расширений PostgreSQL, направленных на повышение производительности, целостности и безопасности данных.

В рамках первой части работы я реализовал и протестировал несколько типов индексов, включая B-tree, GIN и BRIN, что позволило значительно ускорить выполнение типовых запросов в моей базе данных. Я сравнил производительность системы до и после внедрения индексов, а также проанализировал случаи, в которых каждый тип индекса показывает наибольшую эффективность. Это дало возможность глубже понять внутренние механизмы работы PostgreSQL и научиться выбирать подходящий индекс в зависимости от характера данных и типов запросов.

Во второй части лабораторной я работал с транзакциями: реализовал несколько сценариев транзакционной обработки, применяя различные уровни изоляции. Это позволило изучить поведение системы в условиях параллельного доступа, а также продемонстрировать потенциальные аномалии, такие как неповторяемое чтение и фантомные записи. Я научился использовать транзакции для обеспечения согласованности данных и предотвращения конфликтов между конкурентными запросами.

Наконец, третья часть была посвящена внедрению расширений PostgreSQL — pg\_bigm и pg\_trgm для реализации полнотекстового поиска, а также pgcrypto для криптографической защиты данных. Я оценил эффективность каждого расширения на практике, выявил их сильные и слабые стороны и показал, как их использование может существенно повысить удобство и безопасность хранения и поиска информации в базе данных.

В результате лабораторной работы я освоил важные механизмы PostgreSQL, приобрёл умение выбирать и комбинировать подходящие инструменты для повышения производительности и надёжности информационной системы, а также получил глубокое понимание роли индексов, транзакций и расширений при работе с большими объёмами данных.