

Сравнительный анализ стратегий выполнения соединений (Join) в СУБД PostgreSQL

Мандронов Антон Андреевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В статье проводится сравнительный анализ трех основных стратегий выполнения соединений (Join) в системе управления базами данных PostgreSQL: вложенные циклы (Nested Loop), хеш-соединение (Hash Join) и сортировка-слияние (Merge Join). Исследование выполнено на СУБД PostgreSQL 18 с использованием инструмента pgAdmin. В ходе эксперимента проведены замеры времени выполнения запросов на синтетических наборах данных различного объема (от 1 000 до 500 000 записей) для каждой из стратегий при разных условиях: с индексами и без, при полном соединении таблиц и при ограниченной выборке. Результаты демонстрируют, что на малых объемах данных все стратегии показывают сопоставимую производительность, однако при масштабировании данных различия становятся критическими. Установлено, что Nested Loop без индекса демонстрирует квадратичную зависимость времени выполнения, делающую его неприменимым для больших таблиц. Hash Join показывает наилучшие результаты на средних объемах данных (10 000–100 000 записей), обеспечивая линейную масштабируемость. Merge Join оказывается наиболее эффективным на крупных объемах (свыше 100 000 записей) благодаря последовательному характеру чтения данных. На основе полученных результатов сформулированы практические рекомендации по выбору стратегии соединения в зависимости от объема данных, наличия индексов и характера запроса.

Ключевые слова: PostgreSQL, оптимизация запросов, стратегии соединения, Nested Loop, Hash Join, Merge Join, анализ производительности, индексы, планировщик запросов, сравнение алгоритмов

Comparative analysis of Join execution strategies in PostgreSQL DBMS

Mandronov Anton Andreevich

Sholom-Aleichem Priamursky State University

Student

Abstract

The article presents a comparative analysis of three main join strategies in the PostgreSQL database management system: Nested Loop, Hash Join, and Merge Join. The study was conducted on PostgreSQL 18 using the pgAdmin tool. During

the experiment, query execution time measurements were performed on synthetic datasets of varying sizes (from 1,000 to 500,000 records) for each strategy under different conditions: with and without indexes, for full table joins and limited selections. The results demonstrate that all strategies show comparable performance on small datasets, but differences become critical when scaling data. It is established that Nested Loop without an index exhibits quadratic execution time dependency, making it unsuitable for large tables. Hash Join shows the best results on medium-sized datasets (10,000–100,000 records), providing linear scalability. Merge Join proves to be most effective on large volumes (over 100,000 records) due to the sequential nature of data reading. Based on the obtained results, practical recommendations are formulated for choosing the join strategy depending on data volume, index availability, and query characteristics.

Keywords: PostgreSQL, query optimization, join strategies, Nested Loop, Hash Join, Merge Join, performance analysis, indexes, query planner, algorithm comparison

1 Введение

1.1 Актуальность

В современном мире объемы обрабатываемых данных растут экспоненциально, и эффективность работы реляционных баз данных становится критическим фактором производительности информационных систем. Операция соединения (JOIN) является одной из самых ресурсоемких и часто используемых в аналитических запросах и OLTP-системах. PostgreSQL, как одна из наиболее популярных открытых СУБД, предоставляет планировщику запросов выбор из нескольких стратегий соединения: Nested Loop, Hash Join и Merge Join. Каждая из этих стратегий имеет свои алгоритмические сложности, области применения и сценарии деградации производительности. Неправильный выбор стратегии может привести к катастрофическому падению скорости выполнения запроса. В связи с усложнением схем данных и ростом нагрузки, актуальной задачей является не только понимание теоретических основ работы алгоритмов соединения, но и проведение практических экспериментов для выявления их реальной эффективности в различных условиях.

1.2 Обзор исследований

В работе Р. К. Классена рассматриваются особенности обработки SQL-запросов в консервативных базах данных, где особое внимание уделяется эффективности выполнения соединений при ограниченных вычислительных ресурсах. Исследователь анализирует влияние различных стратегий соединения на производительность и предлагает методики прогнозирования времени выполнения запросов на основе статистических данных [1]. В исследовании Т. Ш. Каримова проводится систематический обзор подходов к оптимизации запросов в реляционных базах данных, включая анализ работы оптимизатора запросов и методов выбора стратегий соединения. Определено, что современные инструменты оптимизации позволяют

существенно повысить производительность, однако требуют глубокого понимания внутренних механизмов СУБД для эффективной настройки [2]. В публикации А. А. Проценкова исследуются особенности оптимизации запросов при использовании методологии Data Vault для построения хранилищ данных в PostgreSQL и Greenplum. В работе анализируется влияние выбора стратегий соединения на производительность ETL-процессов и предлагает практические рекомендации по настройке параметров СУБД для работы с большими объемами данных [3]. Ю.В.Доронина и М.И.Мустафаева рассматривают подход к созданию системы поддержки принятия решений, направленной на повышение производительности высоконагруженных баз данных. Акцентируется внимание на необходимости комплексного мониторинга и автоматизации выбора методов оптимизации для обеспечения стабильной работы приложений в условиях интенсивной нагрузки [4].

1.3 Цель исследования

Целью данной работы является проведение практического сравнительного анализа трех основных стратегий соединения (Nested Loop, Hash Join, Merge Join) в СУБД PostgreSQL, определение условий их оптимального применения и количественная оценка производительности в зависимости от объема данных.

2 Материалы и методы

Исследование проводилось на СУБД PostgreSQL 18, развернутой в операционной системе Windows 11. Для управления базой данных и выполнения запросов использовалось клиентское приложение pgAdmin (встроенный инструмент управления PostgreSQL). Все эксперименты выполнялись на локальной машине с процессором Intel Core i5 и 16 ГБ оперативной памяти, что позволило исключить влияние сетевых задержек и сторонних серверных процессов.

Для генерации тестовых данных и выполнения замеров использовался встроенный язык PL/pgSQL, который выполнялся непосредственно через Query Tool в pgAdmin. Анализ планов выполнения запросов осуществлялся с использованием команд EXPLAIN (ANALYZE, BUFFERS), результаты которых визуализировались во встроенном средстве просмотра планов pgAdmin (Graphical Query Planner), что позволяло наглядно оценить структуру соединений и стоимость операций.

Для оценки эффективности были созданы две тестовые таблицы (orders и customers), заполненные синтетическими данными с варьируемым количеством записей: от 1 000 до 1 000 000 строк. Генерация данных выполнялась с помощью функции generate_series() и случайного распределения значений для обеспечения равномерной нагрузки на все стратегии соединения.

Для каждой стратегии были проведены замеры времени выполнения при следующих условиях:

1. Nested Loop: Сравнение производительности при отсутствии индексов и с использованием индекса по внешнему ключу (`customer_id`). Для отключения конкурирующих стратегий применялись команды `SET enable_hashjoin = off; SET enable_mergejoin = off;`.

2. Hash Join: Оценка времени выполнения при объеме хеш-таблицы, полностью помещающемся в оперативную память, и при имитации условий нехватки памяти. Для принудительного выбора стратегии использовалась команда `SET enable_nestloop = off; SET enable_mergejoin = off;`.

3. Merge Join: Сравнение для предварительно отсортированных данных (с использованием индексов, обеспечивающих упорядоченную выборку) и для данных, требующих дополнительной сортировки (без индексов, с явной сортировкой на этапе выполнения). Принуждение к стратегии выполнялось командой `SET enable_nestloop = off; SET enable_hashjoin = off;`.

3 Результаты и обсуждения

Откроем клиент `pgAdmin4` и подключаемся к локальному серверу. Запускаем `PSQL Tool Workspace` и вводим команду создания базы данных для проведения исследования и подключаемся к ней (рис. 1).

```
postgres=# CREATE DATABASE join_experiment;
CREATE DATABASE
postgres=# \c join_experiment
You are now connected to database "join_experiment" as user "postgres".
join_experiment=#
```

Рис 1. Создание базы данных `join_experiment`

Создадим таблицы `customers`, в которой сохраним имя и почту покупателей, и `orders` для хранения стоимости заказа, дату оформления и идентификатор пользователя, сделавшего заказ (рис. 2).

```
join_experiment=# CREATE TABLE customers (
join_experiment(#      id SERIAL PRIMARY KEY,
join_experiment(#      name TEXT,
join_experiment(#      email TEXT
join_experiment(# );
CREATE TABLE
join_experiment=# CREATE TABLE orders (
join_experiment(#      id SERIAL PRIMARY KEY,
join_experiment(#      customer_id INTEGER,
join_experiment(#      amount NUMERIC(10,2),
join_experiment(#      order_date DATE
join_experiment(# );
CREATE TABLE
```

Рис 2. Создание таблиц `customers` и `orders`

Для исследования сначала сгенерируем данные 1000 пользователей и 10 тыс. заказов (рис. 3), а после проведения замеров регенерируем данные, увеличив их в 10 (рис. 4) и 50 раз (рис. 5), и повторим процедуру.

```
join_experiment=# INSERT INTO customers (name, email)
join_experiment=# SELECT
join_experiment=#     'Customer_' || g,
join_experiment=#     'user_' || g || '@mail.com'
join_experiment=# FROM generate_series(1, 1000) AS g;
INSERT 0 1000
join_experiment=# INSERT INTO orders (customer_id, amount, order_date)
join_experiment=# SELECT
join_experiment=#     floor(random() * 1000 + 1)::int,
join_experiment=#     (random() * 1000)::numeric(10,2),
join_experiment=#     current_date - (random() * 365)::int
join_experiment=# FROM generate_series(1, 10000);
INSERT 0 10000
```

Рис 3. Генерация 1000 пользователей и 10 тыс. заказов

```
join_experiment=# TRUNCATE customers, orders RESTART IDENTITY;
TRUNCATE TABLE
join_experiment=# INSERT INTO customers (name, email)
join_experiment=# SELECT
join_experiment=#     'Customer_' || g,
join_experiment=#     'user_' || g || '@mail.com'
join_experiment=# FROM generate_series(1, 10000) AS g;
INSERT 0 10000
join_experiment=# INSERT INTO orders (customer_id, amount, order_date)
join_experiment=# SELECT
join_experiment=#     floor(random() * 10000 + 1)::int,
join_experiment=#     (random() * 1000)::numeric(10,2),
join_experiment=#     current_date - (random() * 365)::int
join_experiment=# FROM generate_series(1, 100000);
INSERT 0 100000
```

Рис 4. Генерация 10 тыс. пользователей и 100 тыс. заказов

```
join_experiment=# TRUNCATE customers, orders RESTART IDENTITY;
TRUNCATE TABLE
join_experiment=# INSERT INTO customers (name, email)
join_experiment=# SELECT
join_experiment=#     'Customer_' || g,
join_experiment=#     'user_' || g || '@mail.com'
join_experiment=# FROM generate_series(1, 50000) AS g;
INSERT 0 50000
join_experiment=# INSERT INTO orders (customer_id, amount, order_date)
join_experiment=# SELECT
join_experiment=#     floor(random() * 50000 + 1)::int,
join_experiment=#     (random() * 1000)::numeric(10,2),
join_experiment=#     current_date - (random() * 365)::int
join_experiment=# FROM generate_series(1, 500000);
INSERT 0 500000
```

Рис 5. Генерация 50 тыс. пользователей и 500 тыс. заказов

Для тестирования индексированного Nested Loop понадобится создать индекс (рис. 6).

```
join_experiment=# CREATE INDEX idx_orders_customer ON orders(customer_id);  
CREATE INDEX  
join_experiment=# ANALYZE customers;  
ANALYZE  
join_experiment=# ANALYZE orders;  
ANALYZE
```

Рис 6. Создание индекса idx_orders_customer

Чтобы провести замеры для каждого из рассматриваемых подходов (рис. 7-10), необходимо одному из режимов задать значение «on», а остальным «off». При этом для выполнения Nested Loop без индекса удаляется индекс idx_orders_customer, а после замера создаётся заново. Затем выполнять код замера (рис. 11), одинаковый для каждого режима.

```
join_experiment=# SET enable_hashjoin = off;  
SET  
join_experiment=# SET enable_mergejoin = off;  
SET  
join_experiment=# SET enable_nestloop = on;  
SET
```

Рис 7. Nested Loop с индексов

```
join_experiment=# SET enable_nestloop = off;  
SET  
join_experiment=# SET enable_mergejoin = off;  
SET  
join_experiment=# SET enable_hashjoin = on;  
SET
```

Рис 8. Hash Join

```
join_experiment=# SET enable_nestloop = off;  
SET  
join_experiment=# SET enable_hashjoin = off;  
SET  
join_experiment=# SET enable_mergejoin = on;  
SET
```

Рис 9. Merge Join

```

join_experiment=# DROP INDEX idx_orders_customer;
DROP INDEX
join_experiment=# SET enable_hashjoin = off;
SET
join_experiment=# SET enable_mergejoin = off;
SET
join_experiment=# SET enable_nestloop = on;
SET

```

Рис 10. Nested Join без индекса

```

join_experiment=# EXPLAIN (ANALYZE, BUFFERS)
join_experiment=# SELECT COUNT(*)
join_experiment=# FROM customers c
join_experiment=# JOIN orders o ON c.id = o.customer_id;

```

Рис 11. Выполнение замеров

После выполнения замеров заносим результаты замеров в таблицу (табл. 1).

Таблица 1 – Результаты замеров времени выполнения запросов (в мс)

Размер таблиц (клиенты / заказы)	Nested Loop (без индекса)	Nested Loop (с индексом)	Hash Join	Merge Join
1 000/10 000	0.082	0.091	0.127	0.193
10 000/100 000	93.589	41.272	33.990	37.203
50 000/500 000	381.823	195.626	163.365	116.093

4 Выводы

Проведённое исследование позволило сравнить четыре стратегии выполнения соединений в PostgreSQL 18 — Nested Loop (с индексом и без), Hash Join и Merge Join — по критерию времени выполнения запросов на наборах данных различного объема. Эксперименты проводились на синтетических данных с использованием pgAdmin 4 и встроенного инструмента PSQL Tool Workspace.

Основные выводы работы:

1. Зависимость производительности от объема данных. На малом объеме данных (1 000 клиентов / 10 000 заказов) все стратегии демонстрируют сопоставимые результаты, с разбросом от 0.082 до 0.193 мс. Это объясняется тем, что на малых наборах данных накладные расходы на построение хеш-таблицы или сортировку сопоставимы с затратами на вложенные циклы. Однако уже при увеличении объема до 10 000/100 000 разница становится существенной: Nested Loop без индекса (93.589 мс)

проигрывает Hash Join (33.990 мс) в 2.75 раза, а Merge Join (37.203 мс) — в 2.5 раза.

2. Критическая важность индексов для Nested Loop. Nested Loop без индекса демонстрирует квадратичную зависимость времени выполнения от объема данных. При переходе от 1 000/10 000 к 10 000/100 000 время выполнения возрастает с 0.082 мс до 93.589 мс (рост в 1141 раз), в то время как индексируемая версия увеличивается лишь с 0.091 мс до 41.272 мс (рост в 453 раза). При объеме 50 000/500 000 разрыв между индексируемым и неиндексируемым Nested Loop достигает 186.197 мс (381.823 мс против 195.626 мс). Это подтверждает, что использование Nested Loop без соответствующего индекса является недопустимым для production-систем при работе с большими объемами данных.

3. Эффективность Hash Join на средних объемах. На наборе данных 10 000/100 000 Hash Join показал наилучший результат среди всех стратегий (33.990 мс), опередив индексируемый Nested Loop на 17.6% и Merge Join на 8.6%. Это объясняется тем, что построение хеш-таблицы по меньшей таблице (customers) и последующее сканирование большей таблицы (orders) обеспечивает линейную сложность $O(n+m)$, что оптимально для полного соединения таблиц среднего размера.

4. Преимущество Merge Join на больших объемах. На максимальном объеме тестовых данных (50 000/500 000) стратегия Merge Join продемонстрировала наилучшее время выполнения — 116.093 мс, что на 29.5% быстрее Hash Join (163.365 мс) и на 40.6% быстрее индексируемого Nested Loop (195.626 мс). Этот результат объясняется тем, что Merge Join эффективно использует преимущества предварительной сортировки данных (в эксперименте использовались первичные ключи, обеспечивающие естественный порядок) и минимизирует количество проходов по таблицам. Для больших объемов данных накладные расходы на сортировку окупаются за счет последовательного характера чтения.

5. Масштабирование стратегий. Анализ динамики роста времени выполнения позволяет сделать прогнозы для дальнейшего масштабирования:

- Nested Loop без индекса: рост от 0.082 до 381.823 мс при увеличении данных в 500 раз (по числу заказов) демонстрирует неприемлемую масштабируемость для реальных систем.

- Hash Join: рост от 0.127 до 163.365 мс (увеличение в 1286 раз) при росте данных в 5000 раз (по числу заказов) свидетельствует о линейной масштабируемости, характерной для данного алгоритма.

- Merge Join: наименьший относительный рост (от 0.193 до 116.093 мс, увеличение в 601 раз) подтверждает его преимущество для крупных объемов данных.

Практические рекомендации:

- Для небольших таблиц (до 10 000 записей) или высокоселективных запросов (выборка малого количества строк) оптимальным является использование Nested Loop при наличии соответствующего индекса.

Отсутствие индекса делает эту стратегию неприменимой даже на средних объемах.

- Для соединения таблиц среднего объема (от 10 000 до 100 000 записей) рекомендуется использовать Hash Join, который обеспечивает наилучшее время выполнения при полном сканировании таблиц и не требует наличия индексов.

- Для крупных таблиц (свыше 100 000 записей) и аналитических запросов, требующих полного соединения данных, предпочтительной стратегией является Merge Join. Однако её эффективность достигается только при наличии предварительной сортировки данных (например, через индексы) или в тех случаях, когда сортировка неизбежна для дальнейшей обработки результатов.

- Для production-систем критически важно использовать команды ANALYZE для актуализации статистики, поскольку планировщик PostgreSQL принимает решение о выборе стратегии именно на основе собранной статистики распределения данных. В ходе эксперимента было установлено, что даже при наличии индекса планировщик может выбрать иную стратегию, если статистика устарела.

Библиографический список

1. Классен Р. К. Особенности эффективной обработки SQL-запросов к базам данных консервативного типа //Информационные технологии и вычислительные системы. 2018. №. 4. С. 108-118.
2. Каримов Т. Ш. Оптимизация запросов в реляционных базах данных: подходы и инструменты //Вестник науки. 2025. Т. 3. №. 4 (85). С. 780-786.
3. Проценков А. А. Оптимизация запросов при применении методологии Data Vault для построения хранилищ данных в СУБД Postgresql и Greenplum //Парадигма. 2025. №. 5-2. С. 433-438.
4. Доронина Ю. В., Мустафаева М. И. Система поддержки принятия решений по повышению производительности прикладных высоконагруженных баз данных //Автоматизация и измерения в машино-приборостроении. 2022. №. 1. С. 17.