Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 24.М41-мм

# Оптимизация выполнения подзапросов в PosDB путем выделения из них некоррелированных частей

***Кузин Яков Сергеевич***

Отчёт по производственной практике
в форме «Решение»

Научный руководитель:
ассистент кафедры информационно-аналитических систем Чернышев Г. А.

Санкт-Петербург
2026

# Оглавление

# Введение

В данной работе рассмотрен новый метод обработки запросов, содержащих коррелированные подзапросы. Он позволяет уменьшить объем вычислений в коррелированной части, что в некоторых случаях позволяет добиться значительного повышения производительности. Для исследования этих случаев было выделено несколько классов запросов, для которых применение рассматриваемого метода может оказаться полезным.

Кроме того, проанализирована зависимость эффективности рассматриваемого метода в зависимости от селективности некоррелированных предикатов, а также преимущество использования данного метода в PosDB [7] — позиционной колоночной СУБД, поддерживающей позднюю материализацию. На момент начала учебной практики в PosDB отсутсвуют методы оптимизации сложных подзапросов, что и служит отправной точкой для внедрения в нее новых механизмов.

Для подтверждения эффективности предложенного подхода были проведены эксперименты с использованием PosDB и PostgreSQL. Результаты показали, что при определенных условиях рассматриваемые оптимизации могут обеспечить улучшение производительности до пяти раз.

# 1. Постановка задачи

Целью настоящей работы является разработка метода оптимизации выполнения запросов в системе PosDB, основанного на выделении из подзапросов некоррелированных частей. Для ее достижения были поставлены следующие задачи:

1. Провести обзор существующих методов оптимизации выполнения подзапросов.

2. Разработать метод оптимизации выполнения подзапросов, основанный на выделении из них некоррелированных частей, и внедрить его в архитектуру PosDB.

3. Осуществить сравнительный анализ производительности системы до и после внедрения предложенного метода оптимизации.

# 2. Обзор

Подзапросы являются важным инструментом в языке SQL для анализа данных. Они представляют из себя запрос, находящийся внутри другого запроса, и широко распространены в современных системах управления базами данных для реализации сложных операций выборки. Например, в бенчмарке TPC-H [8] больше половины запросов содержат один или несколько подзапросов.

Тем не менее, использование подзапросов может приводить к снижению производительности. Это особенно касается кореллированных подзапросов, так как при их выполнении для каждой строки внешнего запроса требуется повторное выполнение внутреннего. В связи с этим оптимизация подзапросов является актуальной задачей, а ее успешное решение может принести значительную пользу отрасли.

Предложенный метод выполнения запросов для составных предикатов дополняет предыдущие исследования по оптимизации подзапросов, такие как работы [4] и [3], которые сосредоточены на декорреляции, объединении и удалении подзапросов. В отличие от этих подходов рассматриваемая методика фокусируется на изоляции некоррелированной части предикатов для снижения накладных расходов, что представляет собой вариант оптимизации, когда полная декорреляция нецелесообразна.

Также данный метод дополняет ряд предыдущих стратегий оптимизации. Но в отличие от метода QuerySplit, работающего на уровне логического плана [9], фреймворка для полной декорреляции в Microsoft Fabric [2] и ресурсоемкого выполнения запросов из статьи [6], он работает на уровне предикатов.

Таким образом, изоляция некоррелированных частей составных предикатов, являющаяся основной идеей рассматриваемой оптимизации, позволяет снизить накладные расходы на выполнение запросов без необходимости значительных изменений в механизме их обработки или оптимизаторе.

# 3. Реализация

В данной работе предлагается новый метод обработки подзапросов, направленный на случаи, когда блок WHERE содержит составные предикаты, состоящие из коррелированных и некоррелированных частей. В некоторых случаях можно изолировать некоррелированную часть предиката и использовать ее для сокращения количества вычислений коррелированной. Этот подход рассматривается в контексте колоночной СУБД с поддержкой поздней материализации [1, 5], что позволяет эффективно использовать позиции во время выполнения запросов и открывает возможности для различных оптимизаций.

## 3.1. Классы запросов

Чтобы оценить область применимости рассматриваемого метода были выделены следующие классы запросов:

1. `SELECT ... WHERE X IN (SELECT ... WHERE NC AND C)`
2. `SELECT ... WHERE X IN (SELECT ... WHERE NC OR C)`
3. `SELECT ... WHERE X <SOME (SELECT ... WHERE NC AND C)`
4. `SELECT ... WHERE X <SOME (SELECT ... WHERE NC OR C)`

Под NC подразумевался некоррелированный предикат, а под C — коррелированный. В этих запросах варьировались предикаты подзапроса и логический оператор, их соединяющий. Выбранные предикаты демонстрируют преобразования, обобщающие другие случаи, и позволят выделить важные детали реализации в дальнейшем.

Классы 1 и 3 могут быть преобразованы в "`OP NC AND OP (NC AND C)`", где `OP` может быть либо `IN`, либо `<SOME`. Важно отметить, что включение предиката `NC` как в коррелированную, так и в некоррелированную части запроса является необходимым для обеспечения корректности преобразования.

Выполнение преобразованного запроса не требует изменений в коде операторов исполнителя запросов (движка). Достаточно лишь изменять порядок операторов внутри плана запроса так, чтобы его выполнение шло следующим образом. Сначала из внутренней таблицы исключаются строки, которые не удовлетворяют предикату `NC`, после чего для каждой строки внешней таблицы выполняется проверка предиката `C` для каждой строки отфильтрованной таблицы. Подобная фильтрация может значительно сократить объем промежуточных результатов. Однако это приводит к дополнительным затратам: предикат `NC` будет проверяться дважды.

Классы 2 и 4 можно преобразовать в "`OP NC OR OP C`", где `OP` может быть либо `IN`, либо `<SOME`. Однако в данном случае выполнение преобразованного запроса уже имеет свои особенности и требует переработки ядра движка: простого изменения плана запроса будет недостаточно.

## 3.2. Преобразование запроса в эквивалентный

Рассмотрение предложенной оптимизации начнем с анализа запроса в листинге 1. Простейший метод выполнения данного запроса заключается в итерации по строкам таблицы lineitem для каждой строки таблицы part с проверкой составного предиката.

**Листинг 1: Исходный запрос**

```
SELECT P.partkey
    FROM part AS P
    WHERE P.retailprice < SOME (
        SELECT 3 * L.extendedprice * L.discount * L.tax
            FROM lineitem AS L
            WHERE L.suppkey = X or P.size = L.quantity
    );
```

Идея рассматриваемого метода заключается в изоляции некоррелированной части и использовании ее в качестве фильтра, позволяющего сократить количество итераций по внутренней таблице. Для этой цели запрос следует переписать, как показано в листинге 2.

**Листинг 2: Преобразованный запрос**

```
SELECT P.partkey
    FROM part AS P
    WHERE P.retailprice < SOME (
        SELECT 3 * L.extendedprice * L.discount * L.tax
            FROM lineitem AS L
            WHERE L.suppkey = X
    ) OR P.retailprice < SOME (
        SELECT 3 * L.extendedprice * L.discount * L.tax
            FROM lineitem AS L
            WHERE P.size = L.quantity
    );
```

Полученный запрос содержит два подзапроса в блоке WHERE, причем один из них — некоррелированный, результат которого не меняется с каждой строкой таблицы part. В свзязи с этим его можно вычислить только один раз, используя один проход по таблице lineitem. В результате для каждой записи из таблицы part сначала проверяется “дешевый” некоррелированный подзапрос. Затем, если результат ложный, выполняется дорогостоящий коррелированный подзапрос, который выполняет полный проход по таблице lineitem.

## 3.3. LP оператор

Для реализации рассматриваемого метода оптимизации выполнения подзапросов был создан оператор LP (Logical Predicate). Его схема представлена на рисунке 1. Этот оператор координирует обработку всего запроса, вызывая различные части плана, и состоит из следующих частей:

1. Предиката, образующего внешний блок WHERE.
2. Оператора DS, предоставляющего позиционные данные для внешней таблицы.
3. Двух дочерних узлов, представляющих части подзапроса.

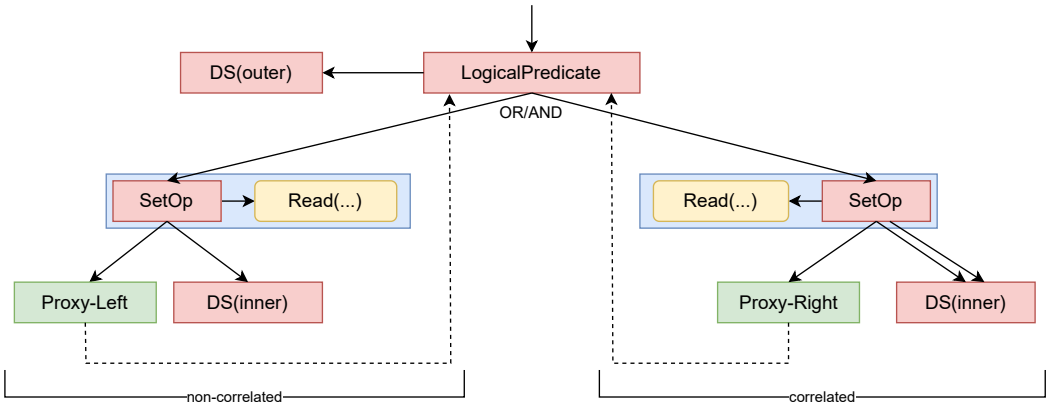


Рис. 1: Архитектура оператора LP

В PosDB для поддержки обработки подзапросов реализован оператор SetOp. Он выражает логику любого предиката подзапроса на основе множеств. LP взаимодействует сразу с двумя такими операторами, соответствующими кореллированной и некореллированной частям.

Реализация оператора LP основана на блочной модели Volcano. Основной оператор разделен на три основных компонента (Modulator, Inverter и Merger) со вспомогательным Proxy оператором, пересылающим блоки между ними. Эти операторы обмениваются блоками позиций и синхронизируют внешние и внутренние потоки данных. Вместе они координируют кэширование, а также исключение и слияние блоков.

Поток данных оператора LP показан на рисунке 2 и состоит из следующих этапов:

1. Modulator выбирает блоки данных из внешнего оператора и передает их через левый Proxy в левый (некоррелированный) подплан, осуществляя кэширование для будущего использования.

2. Inverter извлекает блоки из левого подплана до тех пор, пока не накопит достаточное количество входных данных, после чего запрашивает кэшированные блоки у Modulator. Затем он выполняет исключение позиций из этих блоков для подготовки необходимых данных для правого (коррелированного) подплана.

3. Merger извлекает данные как из Inverter, так и из правого подпла-

на, который получает внешние блоки с исключенными позициями через правый Proxy. Наконец, Merger объединяет некоррелированные результаты Inverter-а с коррелированными результатами из правого подплана и возвращает вывод оператора в позиционной форме.

Таким образом, был предложен новый оператор для обработки подзапросов внутри позиционной колоночной СУБД и, тем самым, расширен набор позиционных операторов в системе. Преимущество подобных систем заключается в возможности составлять длинные цепочки позиционных операторов, которые распространяют позиции на более высокие уровни плана запроса. Чем длиннее цепочки, тем позже происходит материализация и тем больше потенциальные преимущества такого подхода.

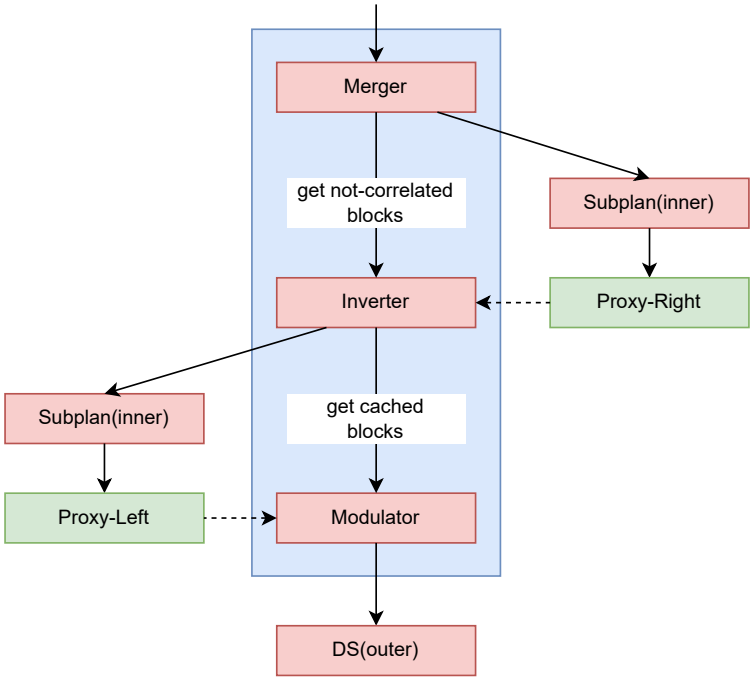


Рис. 2: Поток данных оператора LP

# 4. Эксперименты

Для подтверждения эффективности предложенных оптимизаций Logical Predicate был интегрирован в PosDB. Разработка осуществлялась на языке C++ 17, что обеспечило высокую производительность и гибкость.

## 4.1. Условия экспериментов

Оценка производительности проводилась на ПК со следующим аппаратным и программным обеспечением: Archlinux 6.15.9-arch1-1, AMD Ryzen 5900X CPU, G Skill F4-3600C17-16GTZR RAM, Samsung SSD 990 PRO 2TB, gcc 15.1.1, PostgreSQL 17.5.

## 4.2. Проведенные эксперименты

Эксперименты показали, что оптимизация оператора `IN`, применяемая в PostgreSQL, демонстрирует уровень производительности, сопоставимый с предложенной оптимизацией для второго класса запроса. В связи с этим было принято решение сосредоточить усилия на оценке запросов, относящихся к четвертому классу. Для проведения экспериментов использовался запрос, представленный в листинге 4, а его вид после преобразования можно найти в листинге 3.

**Листинг 3: Преобразованный запрос**

```
SELECT P.partkey
    FROM part AS P
    WHERE P.retailprice < SOME (
        SELECT 3 * L.extendedprice * L.discount * L.tax
            FROM lineitem AS L
            WHERE L.suppkey = X
    ) OR P.retailprice < SOME (
        SELECT 3 * L.extendedprice * L.discount * L.tax
            FROM lineitem AS L
            WHERE P.size = L.quantity
    );
```

**Листинг 4: Исходный запрос**

```
SELECT P.partkey
    FROM part AS P
    WHERE P.retailprice < SOME (
        SELECT 3 * L.extendedprice * L.discount * L.tax
            FROM lineitem AS L
            WHERE L.suppkey = X or P.size = L.quantity
    );
```

Было произведено исследование прироста производительности в условиях изменяющейся селективности некоррелированного подзапроса. Для удобства в запрос была введена переменная X. Значения X, равные 39, 1198, 734, 210 и 451, обеспечивают селективность подзапроса в 20%, 40%, 60%, 80% и 99.9% соответственно. При этом селективность коррелированного подзапроса составляет 40%. Оценка включает в себя измерение времени выполнения запросов для каждого из значений селективности в пяти различных условиях:

1. Исходный запрос в PostgreSQL.
2. Преобразованный запрос в PostgreSQL.
3. Исходный запрос в PosDB.
4. Преобразованный запрос в PosDB с кэшированием NC подзапроса без LP.
5. Преобразованный запрос в PosDB с кэшированием NC подзапроса с LP.

Результаты экспериментов представлены на рисунке 3. Для преобразованного запроса видно, что для PosDB с LP и PostgreSQL время выполнения уменьшается линейно с увеличением селективности некоррелированного подзапроса, стремясь к нулю при достижении селективности, близкой к 100%. В случае PosDB без LP наблюдается более-менее постоянное время выполнения для всех значений селективности. Это явление объясняется тем, что PosDB использует жестко заданный план выполнения запроса без оптимизации во время выполнения.

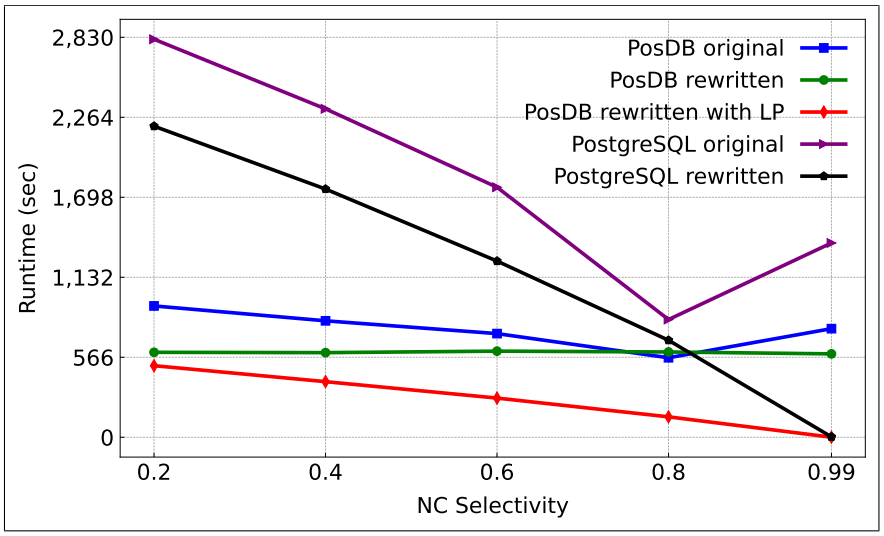


Рис. 3: Время выполнения в зависимости от селективности NC

Касательно оригинального запроса следует отметить несколько факторов. Мы полагаем, PostgreSQL не применяет предложенное преобразование запроса, поскольку его производительность на оригинальном запросе отличается от производительности на преобразованном. Это особенно заметно в случае 99.9% селективности некоррелированного подзапроса. Данный случай интересен тем, что, несмотря на выгодную селективность, время выполнения значительно превышает время при 80% селективности и сопоставимо с запросом при 60% селективности. Это происходит из-за того, что как PosDB, так и PostgreSQL реализуют оптимизацию выхода на ранней стадии при оценке условной конструкции SOME.

Рисунок 4 показывает максимум вычисленного значения выражения “$3 * \text{extendedprice} * \text{discount} * \text{tax}$” на определенном этапе для всех исследованных некоррелированных подзапросов. Следует отметить, что столбец retailprice таблицы PART имеет равномерное распределение в диапазоне от 900 до 1900. Можно заметить, что, несмотря на высокое значение селективности предиката, максимальное значение выражения достигается только на середине сканирования. Это означает, что для значительной части совпадающих строк выполнение не сможет произвести выход на ранней стадии. Таким образом, 99.9% селективность некоррелированного подзапроса можно считать неудачной для оптими-

зации выхода на ранней стадии. В то же время предложенный подход с применением LP не имеет такой уязвимости.

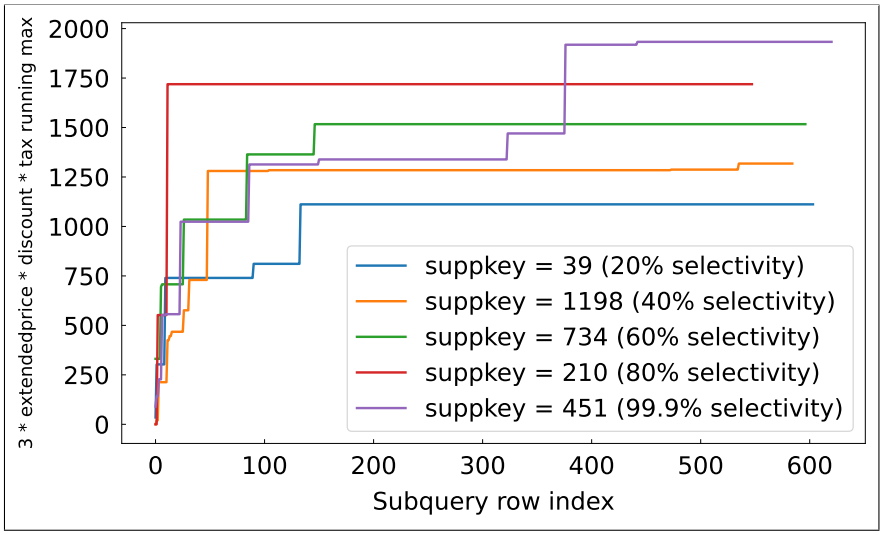


Рис. 4: Максимум значения выражения на разных этапах выполнения запроса

# Заключение

Был разработан метод оптимизации выполнения запросов в системе PosDB, основанный на выделении из подзапросов некоррелированных частей. Как результат, были выполнены следующие задачи:

1. Проведен обзор существующих методов оптимизации выполнения подзапросов.

2. Разработан метод оптимизации выполнения подзапросов, основанный на выделении из них некоррелированных частей. Данный метод успешно внедрен в архитектуру PosDB.

3. Осуществлен сравнительный анализ производительности системы до и после внедрения предложенного метода оптимизации.

# Список литературы


[1] Abadi Daniel, Boncz Peter, Harizopoulos Stavros. The Design and Implementation of Modern Column-Oriented Database Systems. — Hanover, MA, USA : Now Publishers Inc., 2013. — ISBN: 1601987544, 9781601987549.

[2] Bruno Nicolas, Galindo-Legaria César, Joshi Milind. Query Decorrelation in the Fabric Data Warehouse // Companion of the 2025 International Conference on Management of Data. — SIGMOD/PODS '25. — New York, NY, USA : Association for Computing Machinery, 2025. — P. 297–309. — URL: https://doi.org/10.1145/3722212.3724448.

[3] Enhanced subquery optimizations in Oracle / Srikanth Bellamkonda, Rafi Ahmed, Andrew Witkowski et al. // Proc. VLDB Endow. — 2009. — Aug.. — Vol. 2, no. 2. — P. 1366–1377. — URL: https://doi.org/10.14778/1687553.1687563.

[4] Execution strategies for SQL subqueries / Mostafa Elhemali, César A. Galindo-Legaria, Torsten Grabs, Milind M. Joshi // Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data. — SIGMOD '07. — New York, NY, USA : Association for Computing Machinery, 2007. — P. 993–1004. — URL: https://doi.org/10.1145/1247480.1247598.

[5] Harizopoulos Stavros, Abadi Daniel, Boncz Peter. Column-Oriented Database Systems, VLDB 2009 Tutorial. — 2009. — URL: nms.csail.mit.edu/~stavros/pubs/tutorial2009-column_stores.pdf.

[6] Kim Albert, Madden Samuel. Optimizing Disjunctive Queries with Tagged Execution // Proc. ACM Manag. Data. — 2024. — May. — Vol. 2, no. 3. — 25 p. — URL: https://doi.org/10.1145/3654961.

[7] PosDB: A Distributed Column-Store Engine / George A. Chernishev, Viacheslav Galaktionov, Valentin D. Grigorev et al. // Perspectives

of System Informatics - 11th International Andrei P. Ershov Informatics Conference, PSI 2017, Moscow, Russia, June 27-29, 2017, Revised Selected Papers / Ed. by Alexander K. Petrenko, Andrei Voronkov. — Vol. 10742 of Lecture Notes in Computer Science. — Springer, 2017. — P. 88–94. — URL: https://doi.org/10.1007/978-3-319-74313-4_7.

[8] TPC BENCHMARK(TM) H (Decision Support) Standard Specification Revision 3.0.1. — https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf.

[9] Zhao Junyi, Zhang Huanchen, Gao Yihan. Efficient Query Re-optimization with Judicious Subquery Selections // Proc. ACM Manag. Data. — 2023. — Jun.. — Vol. 1, no. 2. — 26 p. — URL: https://doi.org/10.1145/3589330.