Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 24.М41-мм

Оптимизация логических выражений внутри подзапросов в PosDB

Кузин Яков Сергеевич

Отчёт по производственной практике
в форме «Решение»

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

Санкт-Петербург
2025

Оглавление

Введение

В современных системах управления базами данных (СУБД) сложные запросы часто включают в себя логические выражения и подзапросы [8]. Эффективность выполнения таких запросов напрямую зависит от выбранного плана выполнения, что делает оптимизацию критически важной для обеспечения производительности системы [2]. Без специальных механизмов оптимизации СУБД может оказаться непригодной для использования в промышленных условиях, где требования к скорости обработки данных и ресурсам являются особо высокими.

Одним из перспективных направлений оптимизации логических выражений в подзапросах является разбиение сложного подзапроса на несколько более простых, сохраняя при этом логическую структуру, заложенную в исходном запросе. Данная методика может привести к значительному увеличению производительности, что будет продемонстрировано в рамках данной работы.

PosDB [6] представляет собой распределённую колоночную СУБД, ориентированную на эффективное выполнение аналитических запросов. На момент начала учебной практики в PosDB отсутствуют инструменты для оптимизации как логических выражений, так и подзапросов в целом, что и служит отправной точкой для разработки предложенных в данной работе методов оптимизации.

Данная работа направлена на решение актуальной проблемы оптимизации логических выражений в подзапросах, что, в свою очередь, может значительно повысить производительность системы PosDB и расширить ее возможности.

1. Постановка задачи

Целью настоящей работы является разработка прототипа, направленного на оптимизацию логических выражений в подзапросах системы PosDB. Для ее достижения были поставлены следующие задачи:

1. Провести обзор внутренней архитектуры PosDB.
2. Предложить метод оптимизации подзапросов с логическими выражениями. Разработать и внедрить прототип, реализующий этот метод.
3. Рассмотреть практические примеры использования разработанного прототипа в различных сценариях работы с базами данных.

2. Обзор

Подзапросы в реляционных системах управления базами данных (СУБД) могут включать логические связки, такие как AND и OR. Пример подобного подзапроса представлен в листинге 1. Наличие этих логических связок может значительно усложнить выполнение запроса, особенно в сценариях с большими объемами данных [5]. Это особенно актуально для коррелированных подзапросов, где для каждой записи одной таблицы требуется просмотр всей другой таблицы, что может привести к значительным затратам по времени и ресурсам.

Листинг 1: Подзапрос с логическими связками

```
SELECT S.name
    FROM students AS S
    WHERE S.name IN (
        SELECT T.name
            FROM teachers AS T
            WHERE T.subject = 'Math' OR
                  T.salary > 50000 AND
                  T.age < 30
    );
```

Одним из возможных подходов к оптимизации логических выражений в подзапросах является разбиение сложного подзапроса на несколько более простых, сохраняя при этом логическую структуру, заложенную в исходном запросе. Одним из вариантов такого разбиения является разделение на коррелированную и некоррелированную части. Данная методика может существенно повысить производительность, поскольку позволяет избежать ненужных вычислений, когда результат уже известен на определенном этапе обработки данных в некоррелированной части подзапроса.

Похожая идея используется компиляторами во многих языках программирования [4]. Например, в логическом выражении типа “X AND Y AND Z” вычисление прекращается, как только одно из выражений принимает значение False. Аналогично, в логическом выражении типа “X OR Y OR Z” вычисление останавливается, как только одно из выра-

жений становится равным True. В описываемой методике оптимизации логических выражений в подзапросах вычисления также прекращаются, не дойдя до конца, но при достижении определенного результата в некоррелированной части.

В данном разделе будут рассмотрены сценарии, в которых может быть применена описанная оптимизация, а также приведены практические примеры использования разработанного прототипа в различных условиях работы с базами данных.

2.1. Сценарии применения

Следует отметить, что предложенная оптимизация логических выражений в подзапросах применима исключительно к сложным запросам, содержащим подзапросы, образованные операторами IN, SOME, ALL или EXISTS. Эти подзапросы должны включать как коррелированную, так и некоррелированную части. Пример подобного подзапроса представлен в листинге 2. В случаях, когда запрос является достаточно простым, оптимизация может оказаться нецелесообразной: в таких ситуациях либо не будет возможности для оптимизации, либо могут быть применены более простые методы.

Листинг 2: Подзапрос с двумя частями

```
SELECT S.name
    FROM students AS S
    WHERE S.name IN (
        SELECT T.name
            FROM teachers AS T
            WHERE T.subject = 'Math' OR
                  T.age < S.age
    );
```

При наличии сложного запроса всегда возможно выделение некоррелированной части. Однако процесс выделения будет варьироваться в зависимости от логических связок, таких как AND или OR, а также от типа оператора, формирующего подзапрос. В данном разделе будет подробно рассмотрена основная часть этой вариативности. Это позво-

лит лучше понять, как различные логические конструкции влияют на процесс оптимизации и какие подходы могут быть использованы для повышения эффективности выполнения запросов.

Когда подзапрос находится после оператора IN, выделить из него некоррелированную часть не представляет никакой сложности. Для этого достаточно из исходного подзапроса “вырезать” некоррелированный блок условий и переместить его в аналогичный второй подзапрос. При этом сами подзапросы будут связаны той же логической связкой, какой были соединены блоки коррелированных и некоррелированных условий.

Например, если мы ищем всех студентов, чьи имена совпадают с именами сотрудников, которые либо работают в отделе “Dep1”, либо моложе данного студента, мы можем написать запрос, отраженный в листинге 3. После выделения некоррелированной части он будет преобразован в эквивалентный ему запрос, представленный в листинге 4.

Листинг 3: Подзапрос после IN: исходный

```
SELECT S.name, S.age
    FROM students AS S
    WHERE S.name IN (
        SELECT E.name
            FROM employees AS E
            WHERE E.department = 'Dep1' OR
                  E.age < S.age
    );
```

Если случилось так, что почти все работники работают в отделе “Dep1”, то данное разделение даст значительный прирост в производительности за счет того, что исполнителю запроса не нужно будет проходить таблицу employees для каждой записи из таблицы students. Вместо этого большинство логических выражений вернут True на этапе проверки первой их части.

Листинг 4: Подзапрос после IN: преобразованный

```
SELECT S.name, S.age
    FROM students AS S
    WHERE S.name IN (
        SELECT E.name
            FROM employees AS E
            WHERE E.department = 'Dep1'
    ) OR S.name IN (
        SELECT E.name
            FROM employees
            WHERE E.age < S.age
    );
```

Например, если у нас есть таблицы 1 и 2, то для всех $n_1, ..., n_{m+1}$ проверка фильтра потребует лишь валидации принадлежности множеству (SELECT E.name FROM employees AS E WHERE department = 'Dep1'), которое будет закешировано. И лишь для n_{m+2} и n_{m+3} будет выполнена проверка второй части фильтра, которая потребует два прохода по всей таблице employees.

Таблица 1: students

name	age	university
n_1	20	spbu
n_2	18	spbu
n_3	19	spbu
...	...	...
n_{k+1}	20	spbu
n_{k+2}	21	spbu
n_{k+3}	24	spbu

Таблица 2: employees

name	age	department
n_1	20	Dep1
n_2	18	Dep1
n_3	19	Dep1
...	...	...
n_{m+1}	20	Dep1
n_{m+2}	21	Dep2
n_{m+3}	24	Dep2

В случае, если подзапрос находится после оператора SOME и связь некоррелированной и коррелированной частей в сложном подзапросе осуществляется через OR, правила выделения некоррелированной части такие же, что и для подзапроса после оператора IN.

Для демонстрации подобного преобразования рассмотрим следующую ситуацию. Пусть нам необходимо найти всех студентов, снисходительность которых больше или равна хотя бы одного сотрудника, ко-

торый либо работает в отделе “Dep1”, либо моложе данного студента. В этом случае мы можем воспользоваться запросом, отраженном в листинге 5. После всех преобразований он превратится в запрос, представленный в листинге 6.

Листинг 5: Подзапрос после SOME: исходный

```
SELECT students.name, students.age
    FROM students
    WHERE students.tolerance >= SOME (
        SELECT employees.tolerance
            FROM employees
            WHERE employees.department = 'Dep1' OR
                  students.age > employees.age
    );
```

Листинг 6: Подзапрос после SOME: преобразованный

```
SELECT students.name, students.age
    FROM students
    WHERE students.tolerance >= SOME (
        SELECT employees.tolerance
            FROM employees
            WHERE employees.department = 'Dep1'
    ) OR students.tolerance >= SOME (
        SELECT employees.tolerance
            FROM employees
            WHERE students.age > employees.age
    );
```

Прирост производительности при использовании данного преобразования объясняется по аналогии с выкладками, приведенными для схожего запроса с оператором IN.

В случае, когда коррелированная и некоррелированная части в сложном подзапросе связаны через AND, необходимо применять метод дублирования некоррелированной части, который будет рассмотрен далее на примере подзапроса после оператора EXISTS.

Выделение некоррелированной части из подзапроса после оператора ALL будет зависеть от структуры подзапроса. Для примера рассмотрим подзапрос, отраженный в листинге 7. С его помощью можно найти все

книги, которые имеют рейтинг, значение которого выше рейтингов всех книг, которые либо не являются любимыми, либо имеют больше 400 страниц.

Листинг 7: Подзапрос после ALL: исходный

```
SELECT B.title
    FROM books AS B
    WHERE B.rating >= ALL (
        SELECT L.rating
            FROM liked_books AS L
            WHERE L.pages > 400 OR
                  L.book_id != B.id
    );
```

В данном подзапросе коррелированная и некоррелированная части связаны через OR. При этом выделить вторую по аналогии с тем, как это было сделано для операторов IN и SOME, нельзя. Более того, несмотря на кажущуюся корректность, нельзя также использовать UNION. Вместо этого две части исходного подзапроса необходимо связать через AND. Неформально это можно описать при помощи рисунка 1: ALL по объединению есть ALL на двух частях одновременно. Формально же это вытекает из свойств минимума и максимума по целочисленному рейтингу. В результате будет получен запрос, отраженный в листинге 8.

Данное преобразование позволит сразу отсекать книги, рейтинг которых меньше рейтинга хотя бы одной любимой книги, у которой больше 400 страниц. Это окажет существенное влияние, если имеется множество любимых романов с высоким рейтингом. Например, если мы имеем таблицы 3 и 4, то для всех книг, не имеющих идентификатор 4, будет выполнена только первая часть комплексного логического выражения, которая вернет False. Для книги с идентификатором 4 будет выполнена и коррелированная часть, однако затраты на один такой проход незначительны.

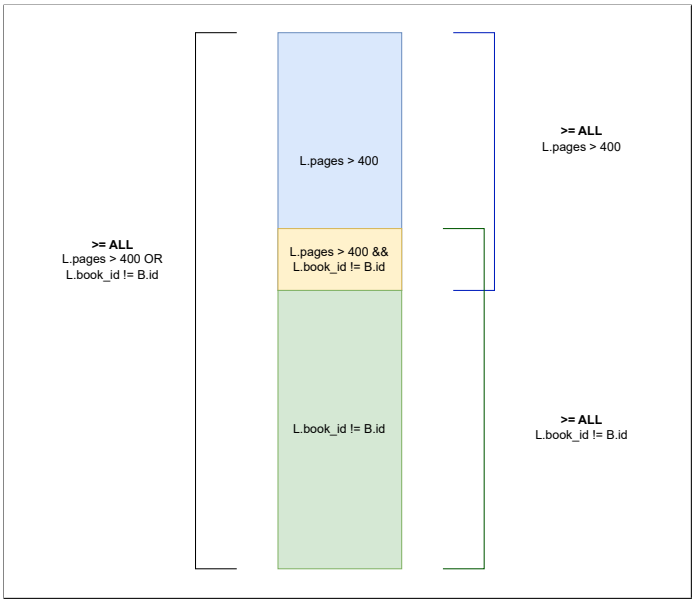

Рис. 1: Неформальная иллюстрация перехода от OR к AND

Листинг 8: Подзапрос после ALL: преобразованный

```
SELECT B.title
    FROM books AS B
    WHERE B.rating >= ALL (
        SELECT L.rating
            FROM liked_books AS L
            WHERE L.pages > 400
    ) AND B.rating >= ALL (
        SELECT L.rating
            FROM liked_books AS L
            WHERE L.book_id != B.id
    );
```

Таблица 3: books

id	rating	pages	title
1	7	100	title_1
2	3	300	title_2
3	4	100	title_3
4	9	900	title_4
5	1	250	title_5
...	...	...	...
n	1	250	title_n

Таблица 4: liked_books

id	rating	pages	title
4	9	900	title_4
$n+1$	10	950	title_n1
$n+2$	10	950	title_n2
...	...	...	...
$n+k$	10	950	title_nk

Когда подзапрос находится после оператора EXISTS, выделить из него некоррелированную часть тоже непросто: выделение будет зависеть от структуры подзапроса. Для примера рассмотрим запрос, отраженный в листинге 9. С его помощью можно найти все книги, для которых существуют писатели с таким же именем, что и имя их автора, а также которые моложе 30 лет и имеют рейтинг выше 8.

Листинг 9: Подзапрос после ALL: исходный

```
SELECT B.title
    FROM books AS B
    WHERE EXISTS (
        SELECT 1
            FROM writers AS W
            WHERE W.name = B.author_name AND
                  W.age < 30 AND
                  W.rating > 8
    );
```

В данном подзапросе коррелированная и некоррелированная части связаны через AND. При этом выделить вторую по аналогии с тем, как это было сделано для всех предыдущих операторов, нельзя, поскольку запрос, отраженный в листинге 10, не эквивалентен исходному запросу. Вместо этого мы можем разбить его так, как это показано в листинге 11.

Листинг 10: Подзапрос после EXISTS: неэквивалентный

```
SELECT B.title
    FROM books AS B
    WHERE EXISTS (
        SELECT *
            FROM writers AS W
            WHERE W.age < 30 AND
                  W.rating > 8
    ) AND EXISTS (
        SELECT *
            FROM writers AS W
            WHERE W.name = B.author_name
    );
```

Листинг 11: Подзапрос после EXISTS: преобразованный

```
SELECT T.title
    FROM books AS B
    WHERE EXISTS (
        SELECT *
            FROM writers AS W
            WHERE W.age < 30 AND
                  W.rating > 8
    ) AND EXISTS (
        SELECT *
            FROM writers AS W
            WHERE W.name = B.author_name AND
                  W.age < 30 AND
                  W.rating > 8
    );
```

Как можно заметить, некоррелированная часть продублирована во втором подзапросе. Из-за этого дублирования подобное преобразование менее эффективно, чем предыдущие, и может привести к избыточным расходам, если некоррелированная часть не сможет дать однозначный ответ False. Однако оно все равно может дать хороший прирост производительности, когда структура данных такова, что этот однозначный ответ будет получен для большинства записей. В рассматриваемом примере подобная структура запроса поможет быстро вернуть False, если не существует писателя моложе 30 лет с рейтингом выше 8. А значит исполнителю запроса не придется для каждой записи из таблицы books итерироваться по всей таблице writers.

3. Реализация

В данном разделе будет проанализирована архитектура PosDB, а также представлен прототип, который иллюстрирует практическое применение ранее описанных теоретических концепций. Он служит наглядным примером реализации методов оптимизации, что позволяет оценить их эффективность и выявить потенциальные направления для дальнейшего совершенствования.

3.1. Архитектура PosDB

В основе архитектуры PosDB лежит итераторная модель Volcano [3], которая представляет собой мощный и гибкий подход к обработке запросов в системах управления базами данных. Эта модель обеспечивает эффективное управление потоками данных, позволяя передавать данные поблочно, что значительно улучшает производительность системы. Для чтения данных в PosDB используются отдельные сущности, известные как считыватели, которые играют ключевую роль в решении проблемы ромбовидных шаблонов в потоках данных [7]. Данная проблема возникает, когда данные обрабатываются в сложных иерархиях, что может привести к избыточным вычислениям и снижению общей производительности.

PosDB представляет собой распределенную систему управления базами данных, что способствует достижению высокой эффективности работы системы. Подобная архитектура также обеспечивает масштабируемость, что является важным аспектом для современных приложений, требующих обработки больших объемов данных в реальном времени.

В PosDB существуют два типа операторов: позиционные и кортежные. Вторые подразумевают передачу данных между операторами в виде кортежей, которые представляют собой простые наборы значений. Позиционные операторы, в свою очередь, передают данные в виде позиций, описывающих местоположение значений в таблице. Это различие позволяет добиться более эффективного использования ресурсов, так как позиционные операторы могут минимизировать объем передавае-

мых данных, работая лишь с информацией о местоположении значений, а не самими значениями.

Сперва в системе используется передача данных с использованием позиций, но в определенный момент происходит переход на кортежи. Данный подход позволяет системе откладывать материализацию данных до тех пор, пока это не станет абсолютно необходимым, что, в свою очередь, способствует повышению общей производительности системы. Для реализации указанного подхода необходимо поддерживать сразу два типа операторов на уровне плана запроса. При этом подзапросы, будучи неотъемлемым элементом структуры SQL, не являются исключением.

Для выполнения запросов в PosDB строятся планы запросов, которые представляют собой последовательность шагов, необходимых для получения результата [1]. Сначала формируется логический план, в котором описываются узлы, необходимые для выполнения запроса. Этот план затем преобразуется в физический, где представлены конкретные операторы, которые будут использоваться для обработки данных. Такой подход позволяет более точно управлять ресурсами и оптимизировать выполнение запросов.

Добавление новых операторов в систему подразумевает создание специализированных узлов для логического плана, а также разработку правил преобразования этих узлов в физические операторы. Важно учитывать внутренние контракты, которые описывают абстрактные классы и интерфейсы, а также возможность взаимодействия нового оператора с другими операторами. Это взаимодействие критически важно для обеспечения совместимости и эффективного функционирования системы в целом. Кроме того, добавление новых операторов требует обязательного написания юнит-тестов, которые позволяют проверить корректность работы новых компонентов и их интеграцию в существующую архитектуру, что, в свою очередь, способствует повышению надежности и устойчивости системы.

Таким образом, PosDB представляет собой высокоэффективную и гибкую распределённую СУБД, которая использует итераторную мо-

дель Volcano для оптимизации обработки данных. Кроме того, она является масштабируемой системой, что позволяет легко добавлять новые компоненты и операторы. Это упрощает процесс расширения функциональности и адаптации к изменяющимся требованиям пользователей.

3.2. Complex Subquery Predicate

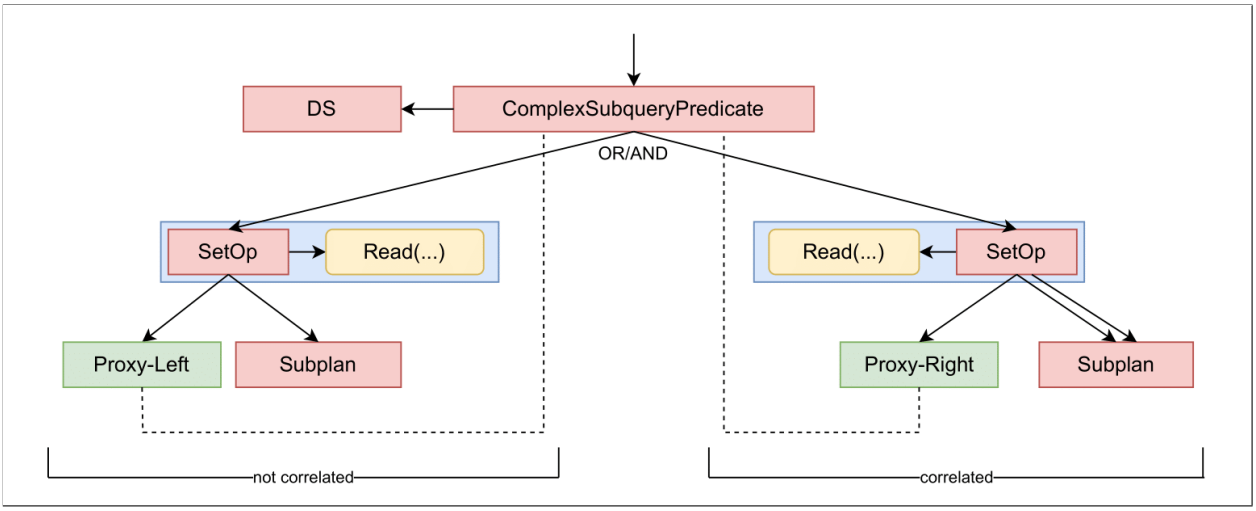

Рис. 2: Архитектура CSP

Для иллюстрации практического применения ранее описанных оптимизаций был создан прототип: специальный оператор, названный Complex Subquery Predicate (CSP). Он представляет собой мощный механизм, который позволяет эффективно вычислять логические выражения, что, в свою очередь, значительно улучшает производительность обработки подзапросов. Его архитектура представлена на рисунке 2.

Complex Subquery Predicate имеет три дочерних узла: поток входных данных, а также два подплана, соответствующих подзапросам, которые содержат некоррелированную (левый ребенок) и коррелированную (правый ребенок) части соответственно. Кроме того, он оперирует двумя прокси-узлами, необходимых для проброса отдельных частей блока из исходного потока данных в узлы подпланов.

Данные в CSP двигаются следующим образом. Из потока входных данных принимается очередной блок позиций, им инициализируется

прокси-оператор для левого ребенка. Далее идет вызов метода получения следующего блока из этого ребенка. Следующие шаги зависят от того, какая логическая связка находится между подзапросами. Если это операция AND, прокси-оператор для правого ребенка инициализируется полученным блоком, после чего из этого ребенка осуществляется получение блока, который передается следующему оператору. Если же подзапросы были связаны через OR, возможны два сценария дальнейшей работы. В случае, если позиции исходного блока совпадают с позициями, полученными из левого ребенка, вверх по структуре операторов предается соответствующий блок. В противном случае перед этим из исходного блока исключаются позиции, полученные из левого ребенка, а также выставляется флаг, говорящий CSP о том, что на следующей итерации оставшиеся позиции необходимо передать через прокси-оператор правому ребенку. Полученный из правого подплана блок передается следующему оператору, а следующая итерация CSP происходит уже с участием левого ребенка.

Таким образом, использование Complex Subquery Predicate обеспечивает более оптимизированное выполнение запросов, позволяя системе быстрее и эффективнее обрабатывать сложные логические условия. Это особенно важно в контексте современных приложений, где объемы данных постоянно растут, а требования к скорости обработки становятся все более жесткими. Его внедрение в архитектуру PosDB не только улучшило производительность существующих подзапросов, но и создало основу для дальнейшего расширения функциональности системы.

4. Эксперименты

Для подтверждения эффективности предложенных оптимизаций был разработан прототип Complex Subquery Predicate, который был интегрирован в проект PosDB, в рамках которого проводились эксперименты. 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. Аппартаное обеспечение: Intel® Core™ i7-11800H CPU @ 2.30GHz (8 cores), 16GB DDR4 3200MHz RAM, 512GB SSD SAMSUNG MZVL2512HCJQ-00BL2. Программное обеспечение: Kubuntu 23.10, Kernel 6.5.0-14-generic (64-bit), gcc 13.2.0.

4.2. Исходные данные

В ходе экспериментов использовались сгенерированные исходные данные, целью которых было продемонстрировать потенциал прототипа. Их описание приведено ниже:

- books1 содержит 4 столбца, а также 100000 строк, 10 из которых соответствуют книгам с рейтингом 10.
- liked_books1, liked_books2 и liked_books3 содержат только те книги, число страниц которых не менее 700. При этом одна из этих книг имеет рейтинг 10. Кроме того, они содержат 3 столбца, а также 1000, 2000 и 3000 строк соответственно.
- books2 содержит 500 строк и 4 столбца.
- writers1, writers2 и writers3 содержат 3 писателя моложе 30 лет с рейтингом не менее 9. Кроме того, они содержат 4 столбца, а также 500, 550 и 600 строк соответственно.

Таблицы составлены так, чтобы при определенных обстоятельствах полный проход по таблице, задействованной в коррелированном подзапросе, был избыточным.

4.3. Проведенные эксперименты

Были проведены эксперименты, демонстрирующие значительный прирост производительности при выполнении запросов, содержащих подзапросы после операторов ALL и EXISTS. Анализ эффективности использования Complex Subquery Predicate в связке с операторами IN и SOME оставлен для дальнейших исследований.

В каждом сравнении применялись два запроса: основной и преобразованный в соответствии с правилами CSP. Для преобразованного запроса разрабатывались два плана выполнения: один с использованием Complex Subquery Predicate, другой — без него. В запросах использовались таблицы с различным количеством строк, что позволило продемонстрировать зависимость времени выполнения от объема входных данных. Более сложные эксперименты и углубленный анализ были оставлены для дальнейших исследований.

4.3.1. Подзапрос после оператора ALL

Эксперименты были проведены с использованием запросов, отраженных в листингах 12 и 13. Их результаты представлены на рисунке 5. Можно наблюдать, что запросы Q1 (без CSP) и Q1' выполняются практически за одно и то же время. При этом даже на самых маленьких из рассматриваемых таблицах (books1 и liked_books1) использование Complex Subquery Predicate дало двухсоткратное увеличение производительности. Причем этот показатель стремительно растет при увеличении размера таблиц. Кроме того, при увеличении объема входных данных его использование все также позволяет получить результат практически мгновенно.

Листинг 12: Q1

```
SELECT books1.title
    FROM books1
    WHERE books1.rating >= ALL (
        SELECT liked_books1.rating
            FROM liked_books1
            WHERE liked_books1.pages > 400 AND
                  liked_books1.book_id != books1.id
    );
```

Листинг 13: Q1'

```
SELECT books1.title
    FROM books1
    WHERE books1.rating >= ALL (
        SELECT liked_books1.rating
            FROM liked_books1
            WHERE liked_books1.pages > 400
    ) AND books1.rating >= ALL (
        SELECT liked_books1.rating
            FROM liked_books1
            WHERE liked_books1.book_id != books1.id
    );
```

Таблица 5: Результаты тестирования CSP + ALL

# Любимых книг	Q1' + CSP	Q1'	Q1	Ускорение
1000	12	3484	4076	287x, 340x
2000	13	6695	8089	515x, 622x
3000	13	9938	11975	764x, 921x

4.3.2. Подзапрос после оператора EXISTS

Эксперименты были проведены с использованием запросов, отраженных в листингах 14 и 15. Их результаты представлены на рисунке 6. Как можно видеть, время выполнения запросов Q2 (без CSP) и Q2' практически не отличается. При этом даже на небольших таблицах (books2 и workers2) использование Complex Subquery Predicate дало шестидесятикратное увеличение производительности. Кроме того, при

увеличении объема входных данных его использование все также позволяет получить результат практически мгновенно.

Листинг 14: Q2

```
SELECT books2.title
    FROM books2
    WHERE EXISTS (
        SELECT writers2.name
            FROM writers2
            WHERE writers2.age < 30 AND
                  writers2.rating > 8 AND
                  writers2.name = books2.author_name
    );
```

Листинг 15: Q2'

```
SELECT books2.title
    FROM books2
    WHERE EXISTS (
        SELECT writers2.name
            FROM writers2
            WHERE writers2.age < 30 AND
                  writers2.rating > 8
    ) AND EXISTS (
        SELECT writers2.name
            FROM writers2
            WHERE writers2.age < 30 AND
                  writers2.rating > 8 AND
                  writers2.name = books2.author_name
    );
```

Таблица 6: Результаты тестирования CSP + EXISTS

# Писателей	Q2' + CSP	Q2'	Q2	Ускорение
500	720	47699	46813	66x, 65x
550	857	60316	60255	70x, 70x
600	1026	76285	76180	74x, 74x

4.4. Результаты

Предложенные эксперименты явно продемонстрировали, что использование Complex Subquery Predicate позволяет добиться значительного увеличения производительности при выполнении запросов, затрагивающих таблицы со специальным распределением данных.

Заключение

Был разработан прототип, направленный на оптимизацию логических выражений в подзапросах системы PosDB. Как результат, были выполнены следующие задачи:

1. Проведен обзор внутренней архитектуры PosDB.
2. Предложен метод оптимизации подзапросов с логическими выражениями. Разработан и внедрен прототип, реализующий этот метод.
3. Рассмотрены практические примеры использования разработанного прототипа в различных сценариях работы с базами данных.

Список литературы

[1] A Comprehensive Study of Late Materialization Strategies for a Disk-Based Column-Store / George A. Chernishev, Viacheslav Galaktionov, Valentin V. Grigorev et al. // Proceedings of the 24th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP) co-located with the 25th International Conference on Extending Database Technology and the 25th International Conference on Database Theory (EDBT/ICDT 2022), Edinburgh, UK, March 29, 2022 / Ed. by Kostas Stefanidis, Lukasz Golab. — Vol. 3130 of CEUR Workshop Proceedings. — CEUR-WS.org, 2022. — P. 21–30. — URL: http://ceur-ws.org/Vol-3130/paper3.pdf.

[2] 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.

[3] Graefe G. Volcano — An Extensible and Parallel Query Evaluation System // IEEE Trans. on Knowl. and Data Eng. — 1994. — Feb.. — Vol. 6, no. 1. — P. 120–135. — URL: https://doi.org/10.1109/69.273032.

[4] Kandl Susanne, Chandrashekar Sandeep. Reasonability of MC/DC for safety-relevant software implemented in programming languages with short-circuit evaluation // 16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013). — 2013. — P. 1–6.

[5] Kastrati Fisnik, Moerkotte Guido. Generating Optimal Plans for Boolean Expressions // 2018 IEEE 34th International Conference on Data Engineering (ICDE). — 2018. — P. 1013–1024.

[6] 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.

[7] PosDB: An Architecture Overview / George A. Chernishev, Vyacheslav Galaktionov, Valentin D. Grigorev et al. // Program. Comput. Softw. — 2018. — Vol. 44, no. 1. — P. 62–74. — URL: https://doi.org/10.1134/S0361768818010024.

[8] Silberschatz Abraham, Korth Henry, Sudarshan S. Database Systems Concepts. — 5 edition. — USA : McGraw-Hill, Inc., 2005. — ISBN: 0072958863.