Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 24.М41-мм

Планирование исполнения подзапросов в PosDB

Щека Дмитрий Вадимович

Отчёт по учебной практике
в форме «Теоретическое исследование»

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

Санкт-Петербург
2025

Оглавление

Введение

Существует несколько классификаций систем управления базами данных, одна из них разделяет их на два класса: колоночные и строчные. Разница заключается в формате данных, хранимых на диске, в случае колоночных — это файлы отдельных колонок, строчные хранят кортежи целиком. Выбор конкретного вида зависит от структуры хранимых данных. Так, например, для аналитических запросов больше подходят колоночные системы, потому что позволяют избежать излишнего сканирования больших объемов данных [1]. Запрос, требующий вывода лишь небольшого подмножества атрибутов таблицы будет выполняться быстрее в колоночной базе данных, однако не во всех сценариях она эффективна.

PosDB [2] — это распределенная колоночная СУБД, которая использует механизм материализации. Это позволяет переходить от позиционного представления данных к кортежному практически в любой момент исполнения запроса. Это дает необходимую гибкость для того, чтобы подобрать план эффективный для каждого конкретного запроса, таким образом перенимая сильные стороны как колоночных, так и строчных систем.

Подавляющее большинство хоть сколько-нибудь сложных запросов будут включать себя использование подзапросов. Это мощный инструмент, который позволяет пользователю представить запрос в более ясной форме. Основные операторы для поддержки подзапросов уже реализованы в PosDB, однако их взаимодействие с механизмом материализации на данный момент недостаточно исследовано. Это затрудняет построение планов для запросов, которые содержат подзапросы. Эта работа фокусируется как раз на этом взаимодействии, чтобы позволить лучше понять, каким образом можно применять механизм материализации вместе с новыми видами операторов.

1 Постановка задачи

Целью настоящей работы является обзор основных операторов для поддержки подзапросов, а также анализ их применения при различных стратегиях материализации. Для ее достижения были поставлены следующие задачи:

1. Провести теоретический обзор операторов, необходимых для поддержки подзапросов.
2. Описать механизмы взаимодействия операторов для поддержки подзапросов в условиях различных стратегий материализации.
3. Составить планы различных типов запросов с содержанием подзапросов, выполнение которых возможно в системе PosDB.

2 Обзор

Подход к реализации поддержки подзапросов бывает различным. Есть большое количество научных статей, которые рассматривают возможность преобразования запросов, чтобы убрать из них подзапрос в принципе [3, 4]. Однако такой подход не получается применить в общем случае. В PostgreSQL используется внутренняя рекурсия, которая проста в реализации, однако не позволяет эффективно выполнять запросы. Этот подход сложен в оптимизации, а время выполнения таких запросов оценивается как ($O(n^2)$).

Существует еще один подход, который используется и в PosDB: расширение множества операторов реляционной алгебры. План с использованием таких операторов значительно проще в оптимизации, а также этот подход позволяет проводить оптимизацию запросов с содержанием подзапросов в общем случае, а не только для некоторых их типов. Этот метод был предложен разработчиками HyperDB в обзорной статье [6].

Далее рассмотрим теоретические определения операторов, реализованных в PosDB для поддержки подзапросов.

2.1 Зависимые соединения

Сложность в реализации подзапросов в первую очередь представляют коррелированные подзапросы. Коррелированный подзапрос — это подзапрос, который ссылается на один или несколько столбцов основного запроса. Такой подзапрос выполняется не один раз перед выполнением запроса, в который он вложен, а для каждой строки, которая может быть включена в окончательный результат. То есть обработка должна повторяться для каждого значения, извлекаемого из внешнего запроса. Выполнение такого запроса занимает много времени, но использование оператора зависимого соединения дает простор для оптимизаций. Подход к таким оптимизациям обсуждается в еще одной статье команды HyperDB [5], однако это не является фокусом данной работы.

Определение зависимого соединения выглядит следующим образом (зависимость соединения будем обозначать горизонтальной чертой над

знаком оператора):

$$T_1 \overline{\bowtie}_p T_2 = \{\ t_1 \circ t_2 \mid t_1 \in T_1 \wedge t_2 \in T_2(t_1) \wedge p(t_1, t_2)\ \}$$

Аналогичные определения можно ввести для зависимых полусоединений и антиполусоединений:

$$T_1 \overline{\ltimes}_p T_2 = \{\ t_1 \mid t_1 \in T_1 \wedge \exists t_2 \in T_2(t_1) : p(t_1, t_2)\ \}$$

$$T_1 \overline{\rhd}_p T_2 = \{\ t_1 \mid t_1 \in T_1 \wedge \nexists t_2 \in T_2(t_1) : p(t_1, t_2)\ \}$$

В данной работе не рассматриваются внешние соединения, поскольку в используемой подсистеме запросов отсутствует поддержка NULL-значений.

В статье [6] указывается, что применение зависимого соединения является лишь первым шагом в разрешении проблемы производительности коррелированного подзапроса. Оптимизатор как можно скорее должен преобразовать такое соединение в другие более эффективные операторы, опустив его ниже по дереву плана запроса. В данный момент в PosDB отсутствует оптимизатор, однако наличие такого оператора открывает больше возможностей при его реализации.

2.2 Одиночные соединения

Одиночные соединения необходимы для поддержки скалярных подзапросов. Скалярные подзапросы должны возвращать максимум один кортеж, который должен состоять из одного значения атрибута. Он работает почти так же, как внешнее соединение, но выдаёт ошибку, если найдено более одного партнёра по соединению:

$$T_1 \overline{\bowtie}^1_p T_2 = \begin{cases} \text{ошибка}, & \text{если } \exists t_1 \in T_1 : |\{t_1\} \bowtie_p T_2(t_1)| > 1. \\ T_1 \overline{\bowtie}_p T_2, & \text{иначе.} \end{cases} \quad (1)$$

Чтобы продемонстрировать необходимость этого оператора, рас-

смотрим такой запрос из [6]:

```
select p.PersId, p.Name, (select a.Name
                          from Assistants a
                          where a.Boss = p.PersId
                          and JobTitle = 'personal␣assistant')
from Professors p
```

Листинг 1: Пример запроса с одиночным соединением

Этот запрос находит имя личного помощника для каждого профессора. Мы не хотим прибегать к рекурсии, то есть не хотим вычислять подзапрос для каждого профессора, так как это приведёт к времени выполнения ($O(n^2)$). Вместо этого мы хотим соединить отношение «Professors» с подзапросом, но нам нужно учитывать семантику SQL: если подзапрос возвращает один результат, мы используем его как скалярное значение, если результатов несколько, мы должны сообщить об ошибке.

2.3 Соединения с маркировкой

Для объяснения необходимости оператора соединения с маркировкой приведем следующий пример, также из работы [6]:

```
SELECT Name
FROM Professors
WHERE EXISTS (SELECT *
              FROM Assistants
              WHERE Lecturer = PersId)
      OR Sabbatical = true
```

Листинг 2: Запрос с проверкой на пустоту

Большинство систем преобразовали этот подзапрос в полусоединение и сделали бы это без дизъюнкции, но здесь это невозможно. Мы должны выводить профессора, даже если не существует курса, который он читает, и поэтому не можем использовать полусоединение.

Введение нового типа соединений с маркировкой позволяет поддерживать подзапросы, использующие операторы **EXISTS**, **NOT EXISTS**, **UNIQUE**, **NOT UNIQUE** и сравнения с множеством (например, $\leq$ **SOME**, $=$ **ALL**). Определение соединения с маркировкой выглядит следующим образом:

$$T_1 \overline{\bowtie}_p^{M:m} T_2 = \{\ t_1 \circ (m : (\exists t_2 \in T_2(t_1) : p(t_1, t_2)))) \mid t_1 \in T_1\}$$

Если маркер используется только для конъюнктивных предикатов, то обычно можно преобразовать соединение с маркировкой в полусоединение или антиполусоединение. Однако в общем случае это невозможно (например, в случае с дизъюнкцией). Но даже так соединение с маркировкой можно эффективно вычислить, обычно за $(O(n))$ при использовании хеширования.

3 Подзапросы в PosDB

В PosDB существует несколько стратегий материализации: ранняя, поздняя и ультра-поздняя. Для перехода от колоночного (позиционного) представления к строчному (кортежному) используется оператор материализации, который может находиться после любого позиционного оператора. Из этого также следует, что почти все операторы должны быть в двух экземплярах: позиционном и кортежном (в планах кортежные операторы имеют префикс Tuple, а колоночные не имеют префикса). То же относится и к новым операторам, однако возможность зависимости еще больше усложняет их использование в некоторых контекстах. Рассмотрим примеры планов с использованием описанных операторов.

3.1 Зависимые соединения

Коррелированные подзапросы должны ссылаться на столбцы основного запроса. Из-за этого возникла необходимость добавления узла зависимого выражения (**DepNode**). Он позволяет операторам из правого поддерева зависимого соединения получать значение выражения из левого поддерева. Можно подумать, что может возникнуть проблема в случае, когда левое поддерево имеет представление, отличное от правого. Однако зависимый узел указывает на единственное значение для каждого исполнения правого поддерева, из-за чего вид этого значения неважен.

Рассмотрим план для следующего запроса:

```
select Name, Total from Professors,
    lateral (select sum(ECTS) as Total from Courses
             where Professors.PersId = Courses.Lecturer)
```

Листинг 3: Запрос с Lateral

На Рис. 1 изображен план с использованием зависимого соединения. Оператор **TupleDepJoin** отвечает в том числе за то, чтобы зависимый узел обращался к левому поддереву единожды для каждого исполнения

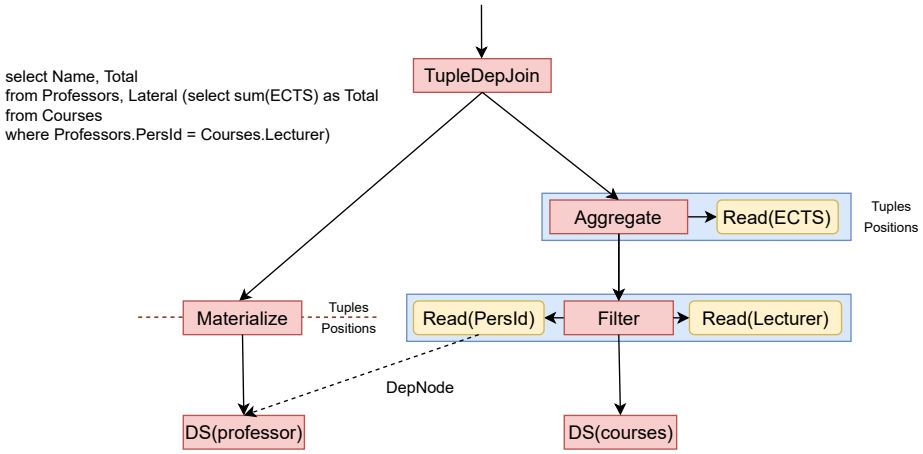

Рис. 1: План для запроса с Lateral

правого поддерева. При наличии корреляции всегда требуется использовать зависимые версии операторов, а все ссылки зависимых узлов должны располагаться в правом поддереве этого зависимого оператора. В связи с семантикой оператора агрегации, который в качестве результата может выдавать только кортежное представление, приходится использовать кортежный вариант зависимого соединения. Фильтр в данном плане отражает предикат самого соединения, из-за чего сам **TupleDepJoin** имеет в качестве предиката значение true. Это было сделано для демонстрации наличия зависимого узла в правом поддереве (нотация отображения зависимого узла еще не утверждена, так что в будущих работах она может измениться). Данный план можно переписать без необходимости материализации в процессе его выполнения, как можно увидеть на Рис. 2.

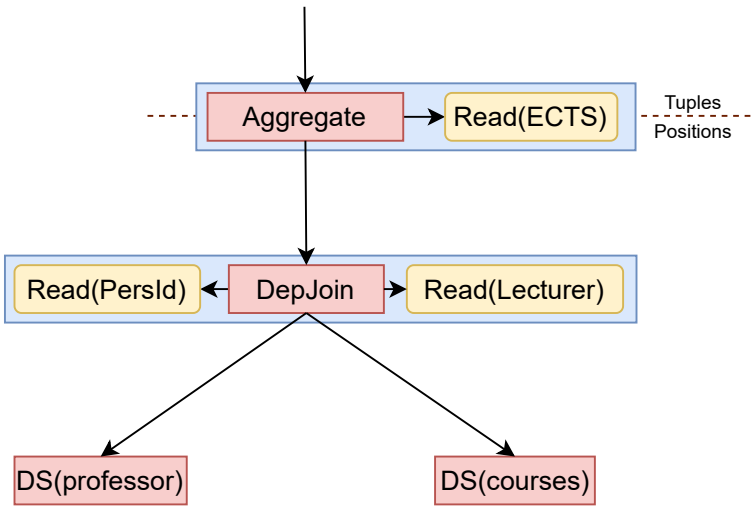

Рис. 2: Альтернативный план для запроса с Lateral

3.2 Одиночные соединения

В качестве примера запроса для одиночного соединения возьмем запрос из Листинга 1.

Этот запрос также является коррелированным, так что использовать будет оператор зависимого одиночного соединения. Рассмотрим план для данного запроса со стратегией ультра-поздней материализации на Рис. 3.

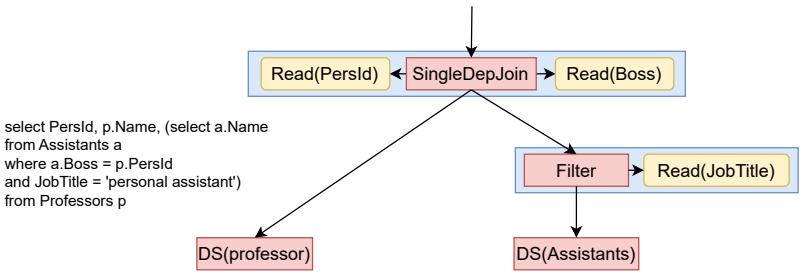

Рис. 3: План запроса с одиночным соединением

В этом запросе также используется зависимый узел, который содержится непосредственно в операторе одиночного соединения. Как и любой другой оператор зависимого соединения, этот оператор будет выполнять операторы правого поддерева для каждого значения *PersId*

из левого поддерева. Все операторы в данном плане являются позиционными, то есть происходит чтение только нужных для выполнения запроса колонок. Результат будет получен с помощью оператора материализации, который станет новым корнем этого дерева операторов.

3.3 Соединения с маркировкой

Из всех новых операторов соединение с маркировкой наиболее сильно вмешивается в стратегию материализации. Этот оператор всегда кортежный, потому что в данном операторе добавляется новая колонка из значений, а не из позиций. Поэтому нам необходимо взаимодействовать в данном случае именно с кортежами, а из правого поддерева данные должны приходить уже в кортежном представлении. Рассмотрим план для запроса из Листинга 2 на Рис. 4.

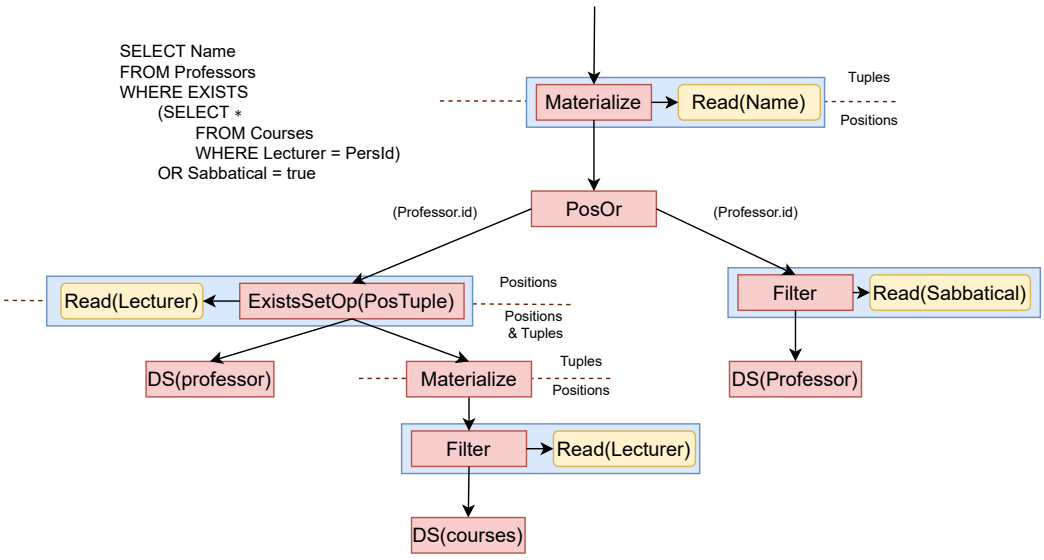

Рис. 4: План запроса с соединением с маркировкой

Соединение с маркировкой в данном случае представлено в виде оператора **ExistSetOp**. Видно, что в правом поддереве мы были вынуждены провести материализацию, чтобы предоставить данные в нужном виде. Результат же мы возвращаем в виде позиций, так как операторы над множествами возвращают представление, соответствующее представлению, полученному из левого поддерева. Это позволит применить

позиционное логическое «или», которое эффективнее строчного. Однако теперь материализацию приходится производить несколько раз в разных частях плана, что затрудняет классифицировать данный план по стратегии материализации. Этот пример заставляет задуматься о возможности добавления гибридных операторов соединения с маркировкой в PosDB.

Заключение

Был проведен обзор основных операторов для поддержки подзапросов, а также анализ их применения при различных стратегиях материализации. Как результат, были выполнены следующие задачи:

1. Проведен теоретический обзор операторов, необходимых для поддержки подзапросов.

2. Описаны механизмы взаимодействия операторов для поддержки подзапросов в условиях различных стратегий материализации.

3. Составлены планы различных типов запросов с содержанием подзапросов, выполнение которых возможно в системе PosDB.

Список литературы

[1] Abadi Daniel J., Boncz Peter A., Harizopoulos Stavros. Column-oriented database systems // Proc. VLDB Endow. — 2009. — Aug.. — Vol. 2, no. 2. — P. 1664–1665. — URL: https://doi.org/10.14778/1687553.1687625.

[2] 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.

[3] 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.

[4] Kim Won. On Optimizing an SQL-like Nested Query // ACM Trans. Database Syst. — 1982. — sep. — Vol. 7, no. 3. — P. 443–469. — URL: https://doi.org/10.1145/319732.319745.

[5] Neumann Thomas, Kemper Alfons. Unnesting arbitrary queries // Datenbanksysteme fur Business, Technologie und Web, BTW 2015 - Proceedings / Ed. by Thomas Seidl, Norbert Ritter, Harald Schoning et al. — Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI). — Gesellschaft fur Informatik (GI), 2015. — P. 383–402. — 16. Fachtagung "Datenbanksysteme fur Business,

Technologie und Web", BTW 2015 - 16th Conference on Database Systems for Business, Technology and Web, BTW 2015 ; Conference date: 04-03-2015 Through 06-03-2015.

[6] Neumann Thomas, Leis Viktor, Kemper Alfons. The Complete Story of Joins (in HyPer) // Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings / Ed. by Bernhard Mitschang, Daniela Nicklas, Frank Leymann et al. — Vol. P-265 of LNI. — GI, 2017. — P. 31–50. — URL: https://dl.gi.de/handle/20.500.12116/657.