Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 24.М41-мм

Интеграция механизма гибридности в колоночную СУБД

Щека Дмитрий Вадимович

Отчёт по учебной практике
в форме «Производственное задание»

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

Санкт-Петербург
2025

Оглавление

Введение

Колоночные системы управления базами данных (СУБД) [2] зарекомендовали себя как эффективное решение для обработки аналитических нагрузок, в том числе благодаря использованию легковесных позиций. Одним из ключевых механизмов таких систем является материализация [1], в процессе которой позиции преобразуются в кортежи на различных этапах выполнения запроса. Классические подходы: ранняя, поздняя и ультра-поздняя материализация — имеют фундаментальные ограничения, связанные с жёстким выбором между работой только с позициями или только со значениями на каждом этапе выполнения запроса.

В данной учебной практике исследуется и дорабатывается архитектура колоночной СУБД, расширенной для поддержки гибридной материализации — стратегии, позволяющей одновременно оперировать как позициями, так и значениями атрибутов в рамках одного оператора. Такой подход сочетает преимущества всех существующих стратегий: минимизацию объемов передаваемых данных, характерную для ультра-поздней материализации, и снижение количества дорогостоящих повторных чтений данных, как в ранней и поздней материализации.

PosDB [3] — это распределенная колоночная СУБД, которая использует механизм материализации. В начале 2024 года для нее были реализованы механизмы гибридности, но так и не были интегрированы в основную ветку проекта. В данной работе будет производиться включение этих механизмов в основную ветку текущей версии PosDB, а также рассматриваться архитектурные изменения, необходимые для полноценной поддержки гибридности.

1. Постановка задачи

Целью настоящей работы является обзор механизма гибридной материализации, а также интеграция существующей реализации в систему PosDB и анализ необходимых архитектурных изменений. Для ее достижения были поставлены следующие задачи:

1. Провести теоретический обзор механизма гибридной материализации.
2. Описать архитектуру PosDB с поддержкой гибридного представления.
3. Интегрировать гибридность в PosDB и описать необходимые для этого изменения.

2. Обзор

Материализация [4] в контексте колоночных СУБД — это процесс преобразования легковесных ссылок на данные (позиций) в фактические значения атрибутов и сборки из этих значений полноценных кортежей (строк), пригодных для возврата пользователю.

Пусть имеется таблица, где каждый столбец хранится отдельно. Чтобы быстро найти нужные данные, система сначала работает только с номерами строк (позициями), которые удовлетворяют условиям запроса. Эти позиции — это компактные указатели, а не сами данные. Таким образом, материализация — это момент, когда система идет по данным указателям и получает конкретные значения строк таблицы.

Выбор стратегии материализации напрямую влияет на время исполнения конкретного запроса.

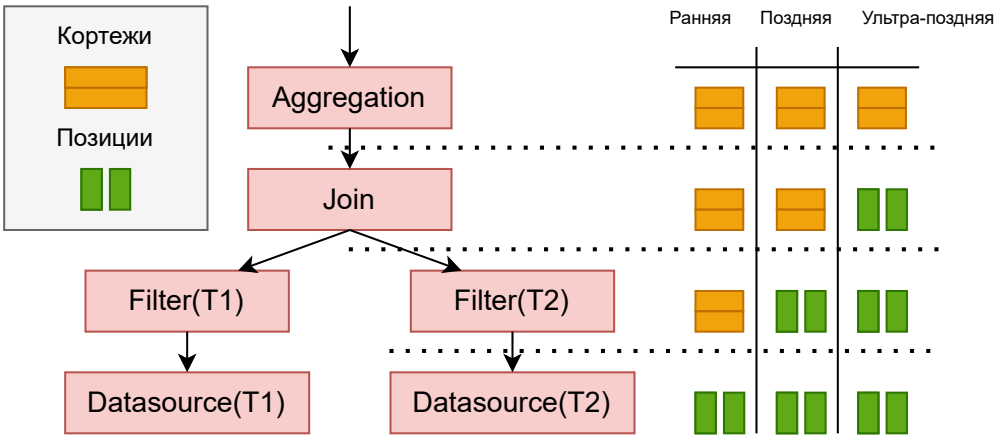

Рис. 1: Стратегии материализации. Адаптировано из [3]

Ранняя материализация

В случае стратегии ранней материализации система сразу после чтения данных с диска объединяет все атрибуты кортежа. Это позволяет читать каждый атрибут только один раз. Также, поскольку все данные читаются последовательно, не возникает проблем с производительностью по причине чтения из произвольных частей диска. Однако в таком случае требуется читать все атрибуты кортежа (даже те, которые не требуется выводить и использовать

в вычислениях), что повышает объем данных, передаваемых между операторами.

Поздняя материализация

Здесь материализация обычно происходит после фильтрации и перед соединением. Это позволяет быстро производить фильтрацию и хотя бы на ранних этапах передавать между операторами меньше данных. Это промежуточный вариант, поэтому недостатки ранней материализации присутствуют и здесь, но становятся уже не столь значительными.

Ультра-поздняя материализация

При данной стратегии материализация происходит в самом конце плана запроса. Поэтому с такой стратегией достигается наибольшая экономия памяти и наименьшее количество данных, передаваемых между операторами. В данном случае также отсутствует необходимость тратить ресурсы системы на материализацию данных, которые позднее будут отброшены. Однако теперь может быть необходимо считывать атрибут с диска многократно, если его значения необходимы на разных этапах. Вдобавок, многократное чтение усугубляется неупорядоченным доступом к данным (порядок позиций может нарушиться после соединения). Чтение значений атрибутов по этим хаотичным позициям приводит к медленному случайному доступу к диску, что снижает производительность.

Гибридная материализация

В этом случае система может одновременно работать и с позициями, и с материализованными значениями для разных атрибутов внутри одного оператора, позволяя гибко выбирать момент материализации для каждого атрибута. Гибкость данной стратегии позволяет избежать большинства недостатков остальных подходов при правильном составлении плана запроса. Имеется возможность бороться с неупорядоченным доступом, заранее материализуя проблемные атрибуты, пока позиции еще упорядочены. Также можно откладывать материализацию атрибутов, если для конкретного запроса это более выгодно. К минусам данной стратегии можно отнести сложность в ее реализации и накладные расходы на использование гибридных блоков.

Таблица 1: Сравнительные аспекты стратегий материализации из [3]

Стратегия	**Быстрые предикаты**	**Re-read в предикатах**	**Re-read в соединениях**	**Неупорядоченный доступ**
Ранняя	Нет	Нет	Нет	Нет
Поздняя	Да	Да	Нет	Нет
Ультра-поздняя	Да	Да	Да	Да
Гибридная	Да	Да	Да (гибко)	Нет (избегается)

Рассмотрим на примере план запроса с использованием гибридных операторов с Рис. 2.

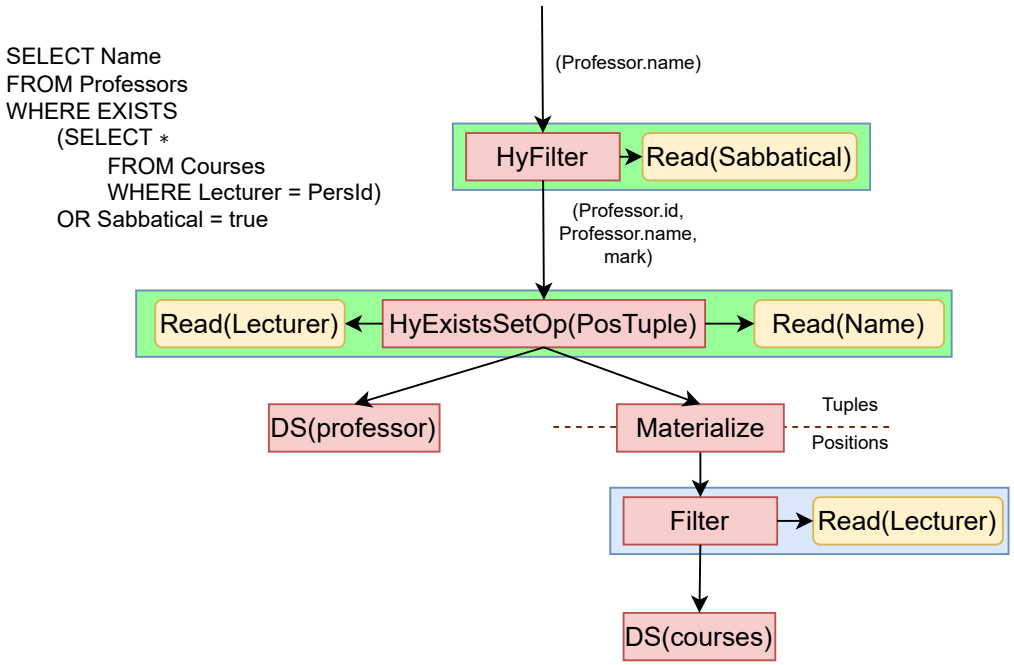

Рис. 2: План гибридного запроса

Здесь из запроса мы хотим получить имена профессоров, которые ведут какие-либо курсы, либо же имеют перерыв от работы. В правом поддереве мы видим фильтрацию, принимающую позиции, где для получения конкретных значений используются считыватели (здесь обозначены желтыми прямоугольниками). После фильтра мы применяем материализацию, преобразовывая уже отфильтрованные позиции в кортежи. Таким образом, гибридный оператор получает из левого поддерева позиции, а из правого кортежи. Из полученных данных он формирует гибридные блоки, которые можно будет

передавать следующим гибридным операторам. Тут же видно, как для позиционных данных внутри гибридного оператора используются считыватели, в то время как для кортежных данных это не требуется. Демонстрация возможности улучшения производительности не является фокусом данной учебной практики, однако эксперименты на эту тему также присутствуют в [3].

Учитывая гибкость, которую можно получить путем использования гибридности, для PosDB был реализован этот механизм. Однако по прошествию двух лет с момента реализации он так и не был интегрирован в основную ветку проекта. Далее опишем, какие изменения были сделаны при его реализации, чтобы лучше понять, как следует интегрировать механизм гибридности в текущую версию PosDB.

3. Гибридность в PosDB

Когда в системе отсутствует гибридность, любой оператор можно разделить на позиционный и кортежный, в зависимости от данных, которые он обрабатывает. Таким же образом можно разделить и другие сущности на эти два вида. В том числе предикаты и другие выражения (например, арифметические операции). Однако для гибридного оператора удобнее работать с единственным классом для обоих случаев, чтобы избежать чрезмерно сложной логики.

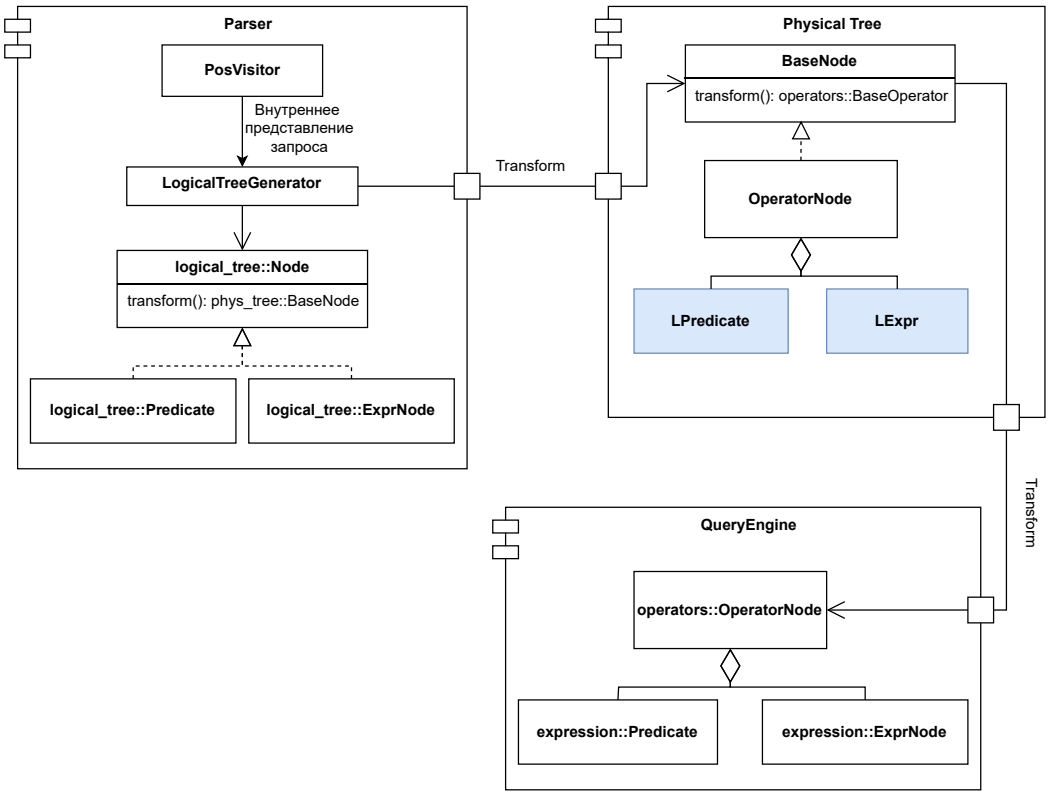

Рис. 3: Компоненты PosDB

Рассмотрим Рис. 3, где изображены некоторые компоненты PosDB. Здесь можно обнаружить три основных дерева: логическое (используется в **Parser**), физическое (**PhysicalTree**) и исполняемое (в **QueryEngine**). Каждое из них содержит набор операторов, предикатов и выражений. Перед реализацией гибридности, в физическом дереве операторы содержали в себе предикаты и выражения, которые уже имели строго зафиксированную принадлежность к позиционному или

кортежному представлению. Как упоминалось выше, это затрудняет реализацию гибридных операторов на физическом уровне. Поэтому были написаны новые классы: **LExpr** и **LPredicate**, которые отмечены голубым цветом. Они никак не закладывают свою реализацию на то, с чем будут в дальнейшем работать узлы, полученные из них на исполняемом уровне.

Прежде, на месте **LExpr** и **LPredicate** использовались соответствующие классы из **QueryEngine**: **expression::Predicate** и **expression::ExprNode**, которые вместе с этим являлись исполняемыми, из-за чего были более склонны зависеть от непосредственного типа данных. Часть функциональности из них была перенесена в новые классы, что позволило им оставить себе лишь функциональность, необходимую именно для выполнения соответствующих операций. Таким образом деревья предикатов и выражений из **QueryEngine** были разбиты на две пары: исполняемое и физическое (условно).

На физическом и исполняемом уровне были добавлены гибридные операторы. На логическом это не требуется, поскольку на этот момент определяется только набор операторов, а не их конкретная реализация. То же самое относится к выражениям и предикатам: на этот момент еще неважно, с какими данными они будут работать. Это ключевой момент, который будет использован в следующей секции.

4. Интеграция гибридности

При интеграции гибридности, как было описано в предыдущей секции, появляется новое дерево выражений и предикатов. Рассмотрим приблизительную структуру классов на Рис. 4.

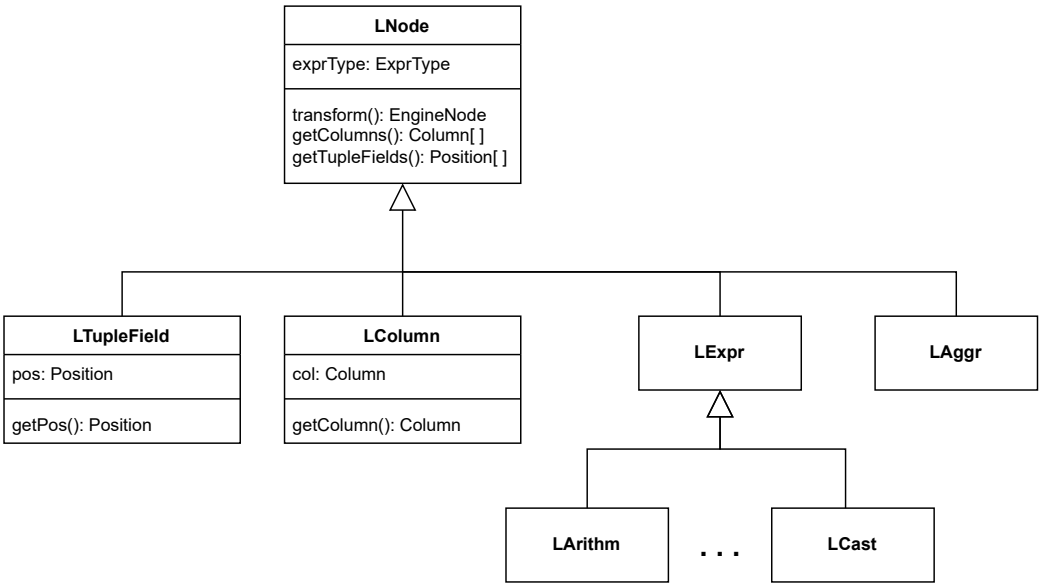

Рис. 4: Дерево выражений

Особый интерес здесь для нас представляют классы **LColumn** и **LTupleField**. Экземпляры данных классов создаются при трансформации логического дерева, полученного после этапа парсинга. Изначально, в дереве выражений, которое используется в логическом дереве операторов, и позиции, и кортежи находятся в одном классе: **UniversalColumn**. При трансформации из выражения извлекается информация о том, что там хранится на самом деле, чтобы создать соответствующий класс нового дерева выражений. Однако это в некоторой степени противоречит изначальной задумке нового дерева выражений: отказ от привязки к конкретному формату данных. Вдобавок, если пристально рассмотреть деревья выражений и предикатов в реализации гибридности и логическом дереве, становится ясно, что присутствует большое количество дублированного кода. Обе эти проблемы можно решить объединением этих деревьев.

Было решено встроить новые деревья вместо тех, которые

используются при построении логического дерева операторов. Для этого было необходимо в первую очередь объединить классы **LColumn** и **LTupleField** в новый класс **LUniversalColumn**, который в состоянии хранить оба типа данных. Поскольку остальные классы выражений не закладываются на реализацию замененных классов, достаточно просто правильно выбрать интерфейс для нового. Внутри вместо указателя на колонку или кортежного поля можно хранить экземпляр **UniversalColumn**, который использовался в выражениях логического дерева. При правильном извлечении данных из **UniversalColumn** остальные классы дерева выражений не должны заметить подмены. В предикатах тоже требуется незначительно изменить интерфейс и добавить функциональность под нужды парсера.

Интерфейс классов нового дерева выражений незначительно отличается от используемого при построении логического дерева, поэтому тут тоже было необходимо его несколько переработать. После этого все использования выражений и предикатов в парсере можно подменить на новые. На этапе трансформации логического дерева в физическое теперь нет необходимости трансформировать выражения и предикаты. Теперь их можно скопировать с помощью метода **clone**, либо передать напрямую в физическое дерево, что могло бы потенциально сократить время работы движка.

Остальные компоненты системы практически не потребовали изменений: в основном переименования вызовов методов. Сами гибридные операторы также без труда интегрировались в систему.

Объединение деревьев выражений и предикатов может показаться сильным расширением зоны ответственности этих частей системы. Однако они выполняют идентичную роль с аналогичными классами из логического дерева. В будущем вряд ли предполагается необходимость изменений, которые затронули бы только что-то одно. Поэтому их объединение может предотвратить дальнейшее дублирование кода при расширении возможностей системы.

PosDB продолжает развиваться, а предыдущие работы получили признание, благодаря «пионерству» [5] в подходах к гибридной

материализации. С механизмом гибридности в основной ветке проекта можно двигаться дальше, пробуя использовать ее в различных сценариях. Например, одной из интересных для исследования тем может оказаться использование гибридности в запросах, которые содержат подзапросы.

5. Апробация

В ходе работы были затронуты многочисленные компоненты системы:

- Изменено свыше 3000 строк кода в различных модулях системы, не считая удаленных.
- Удалено 12 классов, которые заменили классы из реализации гибридности.
- Модифицировано свыше 25 классов для поддержки механизмов гибридности.
- Добавлен один новый класс, позволяющий встроить новые деревья выражений и предикатов в парсер.

Для проверки отсутствия деградации производительности было запущено 156 различных тестовых сценариев по 10 раз каждый. Тестирование проводилось на идентичных конфигурациях до и после внедрения изменений. Расхождение в производительности не превысило 5 процентов, поэтому можно заключить, что внедрение гибридной материализации не привело к падению производительности системы.

Заключение

В процессе учебной практики был проведен обзор механизма гибридной материализации и интеграция существующей реализации в систему PosDB. Как результат, были выполнены следующие задачи:

1. Проведен теоретический обзор механизма гибридной материализации.
2. Описана архитектура PosDB с поддержкой гибридного представления.
3. Гибридность интегрирована в PosDB, описаны необходимые для этого изменения.

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] The Design and Implementation of Modern Column-Oriented Database Systems / Daniel Abadi, Peter Boncz, Stavros Harizopoulos et al. — 2013.

[3] Hybrid Materialization in a Disk-Based Column-Store / Evgeniy Klyuchikov, Michael Polyntsov, Anton Chizhov et al. // Proceedings of the 7th Joint International Conference on Data Science & Management of Data (11th ACM IKDD CODS and 29th COMAD). — CODS-COMAD '24. — New York, NY, USA : Association for Computing Machinery, 2024. — P. 164–172. URL: https://doi.org/10.1145/3632410.3632422.

[4] Materialization Strategies in a Column-Oriented DBMS / Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, Samuel R. Madden // 2007 IEEE 23rd International Conference on Data Engineering. — 2007. — P. 466–475.

[5] Selective Late Materialization in Modern Analytical Databases / Yihao Liu, Shaoxuan Tang, Yulong Hui et al. // Proc. VLDB Endow. — 2025. Vol. 18. — P. 4616–4628. — URL: https://doi.org/10.14778/3749646.3749717.