Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 23М04-мм

Исследование проблемы out-of-order probing в колоночных СУБД

Полынцов Михаил Александрович

Отчёт по учебной практике
в форме «Решение»

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

Санкт-Петербург
2024

Оглавление

Введение

Реляционные СУБД активно используются для хранения и обработки данных. Важными требованиями к СУБД среди прочих других являются высокая производительность выполнения запросов и общая эффективность. Исполнитель запросов (*query engine*) или движок — центральный компонент СУБД, реализующий всю логику хранения таблиц и выполнения запросов над ними. К хранению таблиц на уровне движка существует [12] два кардинально разных подхода: основанный на строках и основанный на колонках. Соответственно, по этому критерию исполнители запросов делятся на строчные (*row-stores*) и колоночные (*column-stores*).

Колоночные исполнители запросов обладают рядом преимуществ перед основанными на строках, которые позволяют им гораздо эффективнее выполнять OLAP запросы [7, 17]. Колоночные системы можно разделить на наивные и "со свободной поддержкой позиций" (*position-enabled* [5]). Оба типа хранят значения таблиц в колоночной форме, то есть значения каждого атрибута последовательно и отдельно от значений других атрибутов, но первые при выполнения запросов оперируют кортежами (то есть строками таблицы) напрямую, вторые же способны использовать позиции (*positions*, *row indices*) кортежей. В этой работе сосредоточимся только на дисковых движках СУБД с поддержкой позиций.

Одной из наиболее важных операций для OLAP запросов является операция соединения двух таблиц (*join*). В то же время, соединение является дорогой операцией с точки зрения производительности, поэтому в исследовательской литературе и практических реализациях уделяется большое внимание эффективным алгоритмам соединения. Колоночные исполнители запросов с поддержкой позиций позволяют выполнять соединения отношений, заданных как набор индексов строк (позиций). Результатом такого соединения является джоин-индекс (*join-index*), последовательность наборов позиций, где каждый набор устанавливает соответствие между кортежами отношений, участвующих в соедине-

нии. Кортеж задается позицией в отношении. После такой операции соединения последовательность позиций, соответствующих конкретному отношению, часто будет находиться в случайном порядке. Это приведет к большому количеству непоследовательных доступов к диску при вычитывании значений атрибутов по данному набору позиций и соответственно деградации производительности. Такой эффект известен в научной литературе [11] как проблема непоследовательного чтения (*out-of-order probing problem*).

Долгое время жесткий диск (HDD) был самым распространенным запоминающим устройством и соответственно при разработке СУБД принимались решения с учетом особенностей его устройства. Для колоночных дисковых движков, выполняющихся на HDD, единственным хоть сколько-нибудь эффективным решением была сортировка позиций перед чтением, однако с распространением твердотелых накопителей (SSD), начали появляться качественно другие подходы.

Данная работа посвящена исследованию проблемы out-of-order probing в колоночных СУБД, выполняющихся на компьютере с современным SSD.

1 Постановка задачи

Целью данной учебной практики является исследование проблемы out-of-order probing в колоночных СУБД со свободной поддержкой позиций. В частности, изучение особенностей проблемы и потенциального решения при выполнении на современном компьютере с SSD в качестве запоминающего устройства. Для достижения этой цели были поставлены следующие задачи.

1. Выполнить обзор предметной области.
2. Провести экспериментальное исследование, выявляющее особенности проблемы в контексте современного аппаратного обеспечения.
3. Разработать план дальнейшего исследования с целью решения проблемы out-of-order probing.

2 Обзор

2.1 Описание out-of-order probing

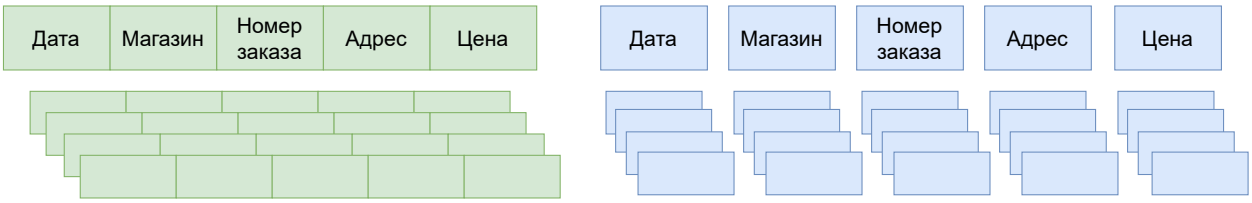

Рис. 1: Модель хранения данных в строчных и колоночных системах.

Колоночные системы от строчных отличаются моделью хранения данных. Однако то, как данные хранятся, очень сильно влияет на алгоритмы выполнения запросов над ними. В первом приближении, строчные системы хранят строки одной таблицы в одном файле друг за другом, это схематично изображено на рис. 1 зеленым цветом. В колоночных система значения каждой колонки хранятся отдельно от всех остальных. Это показано схематично на рис. 1 синим цветом. Такой подход позволяет реализовать большое количество оптимизаций. Примерами таких оптимизаций могут служить: выборочное чтение атрибутов, сжатие, векторизованные вычисления и др.

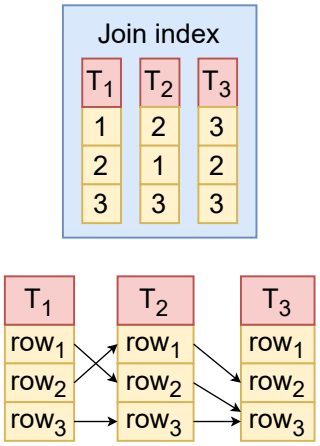

Рис. 2: Пример `join-index`.

Исполнители запросов со свободной поддержкой позиций способны использовать позиции кортежей вместо значений во время выполнения запроса. При этом пользователю безусловно необходимо получить

значения из таблицы, соответственно в какой-то момент исполнитель запросов должен сделать преобразование из позиций в значения. Такое преобразование называется материализацией. Момент преобразования, то есть место в плане конкретного запроса — точкой материализации. Стратегий материализации называются правила, указывающие, в каком месте плана материализация может происходить. Всего существует три стратегии материализации:

- ранняя: точка материализации не выше операторов фильтрации;
- поздняя: точка материализации не выше операций соединения;
- ультра-поздняя: точка материализации выше операций соединения.

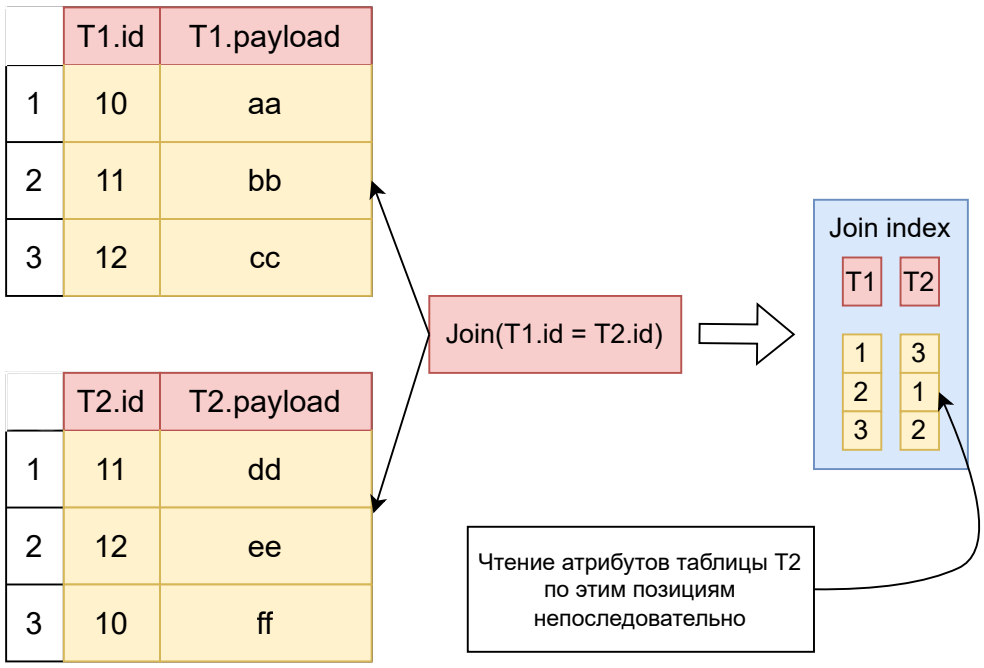

Рис. 3: Возникновение проблемы out-of-order probing.

Проблема out-of-order probing возникает только в планах с поздней и ультра-поздней материализацией, как правило вследствие операции соединения (оператор `Join`). В этой работе сосредоточимся только на случае поздней материализации и эффективном выполнении соединения. Позиционный `Join` (то есть `Join`, принимающий на вход отношения, заданные набором позиций) использует структуру данных `join-index` [18] для представления результата. `Join-index` хранит последовательность

наборов позиций, где каждый набор хранит сопоставление кортежей, заданных позициями в разных таблицах. Его пример можно видеть на рис. 2.

Теперь рассмотрим конкретный пример возникновения проблемы непоследовательного чтения, изображенный на рис. 3. Таблицы `T1` и `T2` имеют атрибуты с названием `id`, по которым происходит соединение. Операция соединения принимает на вход два набора позиций, соответствующих `T1` и `T2`. Оба набора упорядочены по возрастанию. После операции соединения, поскольку значения в атрибутах `T1.id` и `T2.id` находятся в разном порядке, как минимум один из наборов позиций в `join-index` окажется перемешанным. Перемешанность позиций в дальнейшей приведет к непоследовательному чтению при материализации атрибутов по этим позициям, что и называется проблемой out-of-order probing.

2.2 Решения, представленные в исследовательской литературе

В исследовательской литературе существует большое количество работ, посвященных исследованию проблемы непоследовательного чтения. Опишем далее наиболее важные из них.

Авторы работы [14] описывают два алгоритма соединения: `Jive-Join` и `Slam-Join`. Основная идея решения проблемы out-of-order probing, предложенная в этих алгоритмах, заключается в сортировке позиций перед чтением. `Jive-Join` сначала разбивает наборы позиций на партиции такие, что каждая предыдущая партиция содержит позиции меньшие, чем любая следующая. Далее сортирует позиции внутри одной партиции при чтении. `Slam-Join` в свою очередь реализует алгоритм внешней сортировки слиянием для позиций.

В работе [2] авторы описывают алгоритм `RadixJoin`. Его основная идея заключается в том, что для смягчения негативного эффекта непоследовательно чтения не обязательно читать все позиции последовательно. Достаточно [7], чтобы перемешанные позиции были локальными, в

частности умещались в блок жесткого диска.

В работе [9] представлен алгоритм `RARE-Join`. Он использует те же идеи, что `Jive-Join`, однако особым образом обрабатывает само чтение по позициям беря в расчет выполнение на SSD. В работе не представлено экспериментальное исследование, а только стоимостная модель.

Авторы работы [16] описывают алгоритм `FlashJoin`. `FlashJoin` сначала считывает только те атрибуты, которые участвуют в предикате соединения. Далее выполняет соединение и эффективно материализует нужные в запросе выше атрибуты, сортируя `join-index` особым образом.

В работе [4] авторы проводят исследование возможностей внутреннего параллелизма SSD. Далее, используя СУБД PostgreSQL, экспериментально показывают, что алгоритмы, эффективно использующие многопоточные операции ввода/вывода гораздо производительные остальных. В частности, авторы описывают алгоритм соединения, использующий B^+-дерево вместо хеш-таблицы для поиска совпадающего кортежа.

Авторы работы [15] представляют алгоритм `DigestJoin`. Основной вклад данной работы заключается в двух алгоритмах эффективной материализации атрибутов: `Page-based`, `Graph-based`. В первом подходе позиции сортируются перед чтением так, чтобы в результате получилась последовательность, разбитая на партиции, как в `JiveJoin`. Это быстрее, чем полная сортировка, и позволяет избегать чтения одних и тех же страниц дважды. Второй подход моделирует задачу чтения значений по позициям как задачу на графе и предлагает эффективное решение для нее.

В работе [13] представлен алгоритм соединения `ParaHashJoin`. Это многопоточный алгоритм, которые опирается на внутренний параллелизм SSD. Авторы описывают как эффективно и многопоточно материализовывать атрибуты, участвующие в предикате соединения, по позициям, а так же как эффективно строить хеш-таблицу для выполнения соединения. Материализация результирующего `join-index` реализована так же, как в `DigestJoin`.

В работе [3] авторы описывают алгоритм `Bt-Join`. Он аналогичен

обычному соединению с использованием хеш-таблицы, но вместо хеш-таблицы использует B-дерево, у которого в листьях хранятся одновременно необходимые значения и позиции. Авторы так же исследуют влияние на производительность различных алгоритмов материализации во время выполнения операции соединения.

Авторы работы [1] представляют алгоритм `Advanced Block Nested Loop Join` или `ANLJ`. Целью авторов было разработать алгоритм, который не будет писать промежуточные результаты на запоминающее устройство и при этом будет так же эффективен, как алгоритмы, пишущие на диск. Такой алгоритм позволяет увеличить время жизни твердотелых накопителей не теряя при этом сильно в производительности.

В работе [6] представлен алгоритм соединения `DaC-Join`. `DaC-Join` использует эффективный многопоточный алгоритм материализации атрибутов, участвующих в предикате соединения. Для выполнения самого соединения, `DaC-Join` строит иерархию хеш-таблиц. В этой иерархии таблицы верхнего уровня заполняются параллельно, а после параллельно обходятся алгоритмом, чтобы выполнить соединение.

2.3 Выводы

В таблице 1 отображена свободная информация об алгоритмах, представленных в работах, посвященных эффективному выполнению `Join`ов в контексте проблемы непоследовательного чтения. Опишем её атрибуты:

- “Год”: год публикации работы, опиывающей алгоритм.
- “Эффективнее предыдущих”: есть ли в работе экспериментальное исследование, показывающее, что алгоритм эффективнее каких-либо известных ранее.
- “Параллельность”: использует ли алгоритм многопоточность.

Таблица 1: Алгоритмы соединения и их свойства.

	Год	Эффективнее предыдущих	Параллельность	SSD-специфичен	Используется сортировка позиций	Используется разбиение по “ведрам“	Не требует изменения кода менеджера буферов
Jive-Join	1996	+	–	–	+	+	+
Slam-Join	1996	+	–	–	+	–	+
Radix Join	1999	?	–	–	–	+	+
RARE-Join	2008	?	–	+	–	+	+
FlashJoin	2009	+	–	+	+	–	+
B+-Join	2011	?	+	+	–	–	+
DigestJoin	2013	?	–	+	+	–	?
ParaHashJoin	2013	?	+	+	+	–	+
Bt-Join	2015	+	–	+	–	–	+
ABNL Join	2017	–	–	+	–	+	+
DaC-Join	2019	+	+	+	–	+	+

- “SSD-специфичен”: использует ли алгоритм оптимизации, специфичные для SSD.
- “Используется сортировка позиций”: сортирует ли алгоритм позиции перед материализацией по ним.
- “Используется разбиение по ведрам”: разбивает ли алгоритм позиции на ведра (*buckets*, партиции) перед материализацией по ним.
- “Не требует изменения кода менеджера буферов”: необходимо ли для реализации алгоритмов изменять код менеджера буферов (*buffer manager* [10]).

Исходя из описанного выше можно охарактеризовать положение дел в области проблемы непоследовательного чтения следующим образом:

- различных алгоритмов очень много, но во многих используются похожие идеи;
- мало какие алгоритмы сравнивались друг с другом;
- а если сравнивались, то экспериментальное исследование в разных работах проводилось в разном окружении.

Как следствие, не получится выбрать какой-нибудь алгоритм в качестве самого эффективного в настоящий момент и просто реализовать его. Тем не менее, то, какие техники используют авторы алгоритмов, чтобы добиться высокой производительности позволяет сделать следующие выводы:

- чтобы алгоритму быть эффективным, он должен использовать специфичные оптимизации под SSD;
- при этом, среди таких специфических оптимизаций вероятно самая важная — параллельность.

На основе обзора был разработан дальнейший план решения проблемы:

1. Реализовать FlashJoin и RARE-Join, как базовые алгоритмы. Эти алгоритмы используют концептуально разные подходы, на которых основано большинство следующих алгоритмов.
2. Реализовать DaC-Join, как скорее всего наиболее эффективный алгоритм в настоящее время.
3. Провести экспериментальное исследование реализаций, из его результатов уже делать выводы о дальнейшей работе.

3 Эксперименты: out-of-order probing на SSD

Сначала покажем, что проблема непоследовательного чтения все ещё актуальна даже на современных твердотелых накопителях. Для того чтобы показать наличие проблемы в минимальном окружении, был разработан следующий эксперимент. Сгенерируем простые таблицы, в которых нет ничего кроме идентификатора, уникального для каждой строчки, и полезной нагрузки. При этом в разных таблицах сделаем разное упорядочивание идентификаторов: идентификаторы отсортированы, перемешаны случайно, перемешаны внутри одной ведра. Размер одного ведра выберем 4КБ, то есть равным странице SSD [8].

Листинг 1: Использующийся в эксперименте запрос.

```
1 SELECT T2.id, T2.payload
2 FROM T1 JOIN T2 ON T1.id = T2.id
```

Возьмем простой запрос, изображенный на листинге 1, с одним соединением и зафиксируем порядок идентификаторов по возрастанию в левой таблице (`T1`). Для правой же будем рассматривать разные варианты упорядочивания. В зависимости от порядка идентификаторов в `T2` будет меняться порядок позиций в `join-index` и, соответственно, при чтении по этим позициям значений порядок чтения будет разным. В качестве метрик выберем время выполнения всего запроса и шага материализации отдельно. Такой эксперимент покажет, насколько негативно порядок позиций влияет на производительность системы.

Чтобы минимизировать влияние случайных факторов на работу PosDB, были предприняты следующие шаги. Файл подкачки был выключен используя команду `swapoff -a`. Частота процессора была зафиксирована на максимальном значении с помощью утилиты `cpupower`. Между запусками одного и того же запроса сбрасывались кеши Linux с помощью записи “3” в `proc/sys/vm/drop_caches`, а так же внутренние кеши диска с помощью утилиты `blockdev`. PosDB был скомпилирован с уровнем оптимизаций `-O3`. Запрос выполнялся десять раз, а в качестве

результата бралось среднее значение с уровнем доверия 95%.

Аппаратное и программное обеспечение системы, на которой выполнялись эксперименты: AMD Ryzen 7 6800H @ 4.785GHz (8 cores, 16 threads), 2×16 GiB RAM DDR5 4800 MT/s, 1TB NVME M.2 SAMSUNG MZVL21T0HCLR-00BL2, Artix Linux, Kernel 6.6.9-artix1-1, GCC 13.2.1.

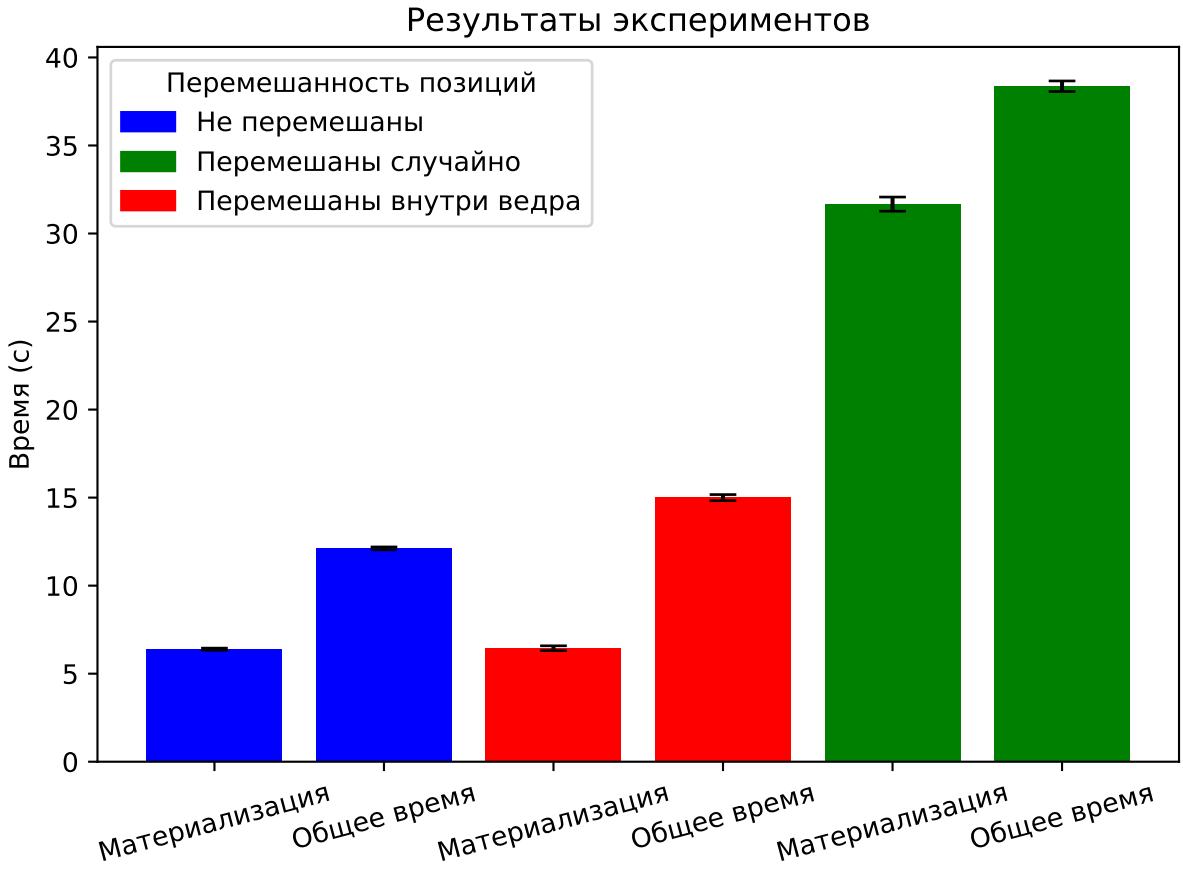

Рис. 4: Результаты эксперимента.

Результаты эксперимента представлены на рис. 4. Можно видеть, что порядок позиций, по которым происходит материализация, сильно влияет на время выполнения запроса. Запрос, где позиции перемешаны случайным образом, выполняется почти в 4 раза дольше. Так же можно видеть, что при изменении упорядоченности позиций меняется на самом деле только время выполнения шага материализации, весь запрос до материализации во всех случаях выполняется примерно семь секунд. Данные результаты показывают наличие проблемы даже при выполнении на современном SSD и важность её исправления. При этом так же видно, что перемешанные внутри одной ведра значения почти

не влияют на время выполнения запроса и вообще не влияют на шаг материализации. Это показывает, насколько важна локальность данных и что в случае SSD проблема на самом деле возникает не из-за непоследовательно чтения как такового, а из-за чтения одних и тех же страницы более одного раза.

Заключение

В ходе работы были достигнуты следующие результаты.

1. Выполнен обзор существующих подходов к решению проблемы out-of-order probing и смежных с ней. Выделены особенности имеющихся подходов.

2. Проведено экспериментальное исследование, показывающее актуальность проблемы даже при использовании современного аппаратного обеспечения. А именно:

   - написаны сценарии на языке Python для генерации данных и визуализации результатов экспериментов;
   - разработан и реализован набор экспериментов внутри дискового колоночного исполнителя запросов PosDB;
   - показано, что при наличии out-of-order probing запрос выполняется в разы медленнее, чем при последовательном чтении. При этом, если позиции перемешаны внутри одной страницы SSD, то накладные расходы сильно меньше.

3. Разработан план дальнейшего исследование проблемы out-of-order probing, а именно:

   (a) реализовать FlashJoin и RARE-Join в качестве базовых алгоритмов;

   (b) реализовать DaC-Join, как наиболее эффективный алгоритм в настоящее время;

   (c) провести экспериментальное сравнение реализаций, предложить улучшения.

Список литературы

[1] Advanced Block Nested Loop Join for Extending SSD Lifetime / Hongchan Roh, Mincheol Shin, Wonmook Jung, Sanghyun Park // IEEE Trans. on Knowl. and Data Eng. — 2017. — apr. — Vol. 29, no. 4. — P. 743–756. — URL: https://doi.org/10.1109/TKDE.2017.2651803.

[2] Boncz Peter A., Manegold Stefan, Kersten Martin L. Database Architecture Optimized for the New Bottleneck: Memory Access // Proceedings of the 25th International Conference on Very Large Data Bases. — VLDB '99. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1999. — P. 54–65.

[3] Bt-Join: A Join Operator for Asymmetric Storage Device / Neusa L. Evangelista, José de Aguiar M. Filho, Angelo Brayner, Namom Alencar // Proceedings of the 30th Annual ACM Symposium on Applied Computing. — SAC '15. — New York, NY, USA : Association for Computing Machinery, 2015. — P. 988–993. — URL: https://doi.org/10.1145/2695664.2695785.

[4] Chen Feng, Lee Rubao, Zhang Xiaodong. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing // 2011 IEEE 17th International Symposium on High Performance Computer Architecture. — 2011. — P. 266–277.

[5] 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.

[6] DaC-Join: A join operator for improving database performance on modern hardware / Namom Alencar, Angelo Brayner, José Maria Monteiro, José de Aguiar Moraes Filho // Concurrency and Computation: Practice and Experience. — 2019. — Vol. 31, no. 17. — P. e5180. — e5180 cpe.5180. https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5180.

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

[8] Fantastic SSD Internals and How to Learn and Use Them / Nanqinqin Li, Mingzhe Hao, Huaicheng Li et al. // Proceedings of the 15th ACM International Conference on Systems and Storage. — SYSTOR '22. — New York, NY, USA : Association for Computing Machinery, 2022. — P. 72–84. — URL: https://doi.org/10.1145/3534056.3534940.

[9] Fast Scans and Joins Using Flash Drives / Mehul A. Shah, Stavros Harizopoulos, Janet L. Wiener, Goetz Graefe // Proceedings of the 4th International Workshop on Data Management on New Hardware. — DaMoN '08. — New York, NY, USA : Association for Computing Machinery, 2008. — P. 17–24. — URL: https://doi.org/10.1145/1457150.1457154.

[10] Garcia-Molina Hector, Ullman Jeffrey D., Widom Jennifer. Database Systems: The Complete Book. — 2 edition. — USA : Prentice Hall Press, 2008. — ISBN: 9780131873254.

[11] 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.

[12] Hellerstein Joseph M., Stonebraker Michael, Hamilton James. Architecture of a Database System // Found. Trends Databases. — 2007. —

feb. — Vol. 1, no. 2. — P. 141–259. — URL: https://doi.org/10.1561/1900000002.

[13] Lai Wenyu, Fan Yulei, Meng Xiaofeng. Scan and Join Optimization by Exploiting Internal Parallelism of Flash-Based Solid State Drives // Web-Age Information Management / Ed. by Jianyong Wang, Hui Xiong, Yoshiharu Ishikawa et al. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. — P. 381–392.

[14] Li Zhe, Ross Kenneth A. Fast Joins Using Join Indices // The VLDB Journal. — 1999. — apr. — Vol. 8, no. 1. — P. 1–24. — URL: https://doi.org/10.1007/s007780050071.

[15] Optimizing Nonindexed Join Processing in Flash Storage-Based Systems / Yu Li, Sai Tung On, Jianliang Xu et al. // IEEE Transactions on Computers. — 2013. — Vol. 62, no. 7. — P. 1417–1431.

[16] Query Processing Techniques for Solid State Drives / Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah et al. // Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data. — SIGMOD '09. — New York, NY, USA : Association for Computing Machinery, 2009. — P. 59–72. — URL: https://doi.org/10.1145/1559845.1559854.

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

[18] Valduriez Patrick. Join Indices // ACM Trans. Database Syst. — 1987. — jun. — Vol. 12, no. 2. — P. 218–246. — URL: https://doi.org/10.1145/22952.22955.