Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 20.Б08-мм

Методы реализации поиска зависимостей порядка на основе списковой формализации

Щека Дмитрий Вадимович

Отчёт по преддипломной практике
в форме «Решение»

Научный руководитель:
доцент кафедры информационно-аналитических систем, к. ф.-м. н. Михайлова Е. Г.

Консультант:
ассистент кафедры информационно-аналитических систем Чернышев Г. А.

Санкт-Петербург
2024

Оглавление

Введение

Профилирование данных [1] — это процесс извлечения некоторых метаданных из наборов данных. Метаданные могут быть как и относительно простой характеристикой (количество столбцов или пустых значений), так и сложными закономерностями, извлечение которых может являться трудоемкой задачей. Эти зависимости имеют формальное математическое описание и называются *примитивы*. Примерами таких примитивов могут служить функциональные зависимости [10], алгебраические ограничения [4], ассоциативные правила [2]. Все они имеют набор применений: от банального улучшения понимания устройства данных до оптимизации запросов баз данных.

Desbordante [6] — это инструмент с открытым исходным кодом для высокопроизводительного профилированния данных. Он уже объединяет в себе алгоритмы для поиска упомянутых примитивов и еще некоторых других, и этот список постоянно расширяется. Desbordante был вдохновлен проектом Metanome [5], который является не столько инструментом, сколько исследовательским прототипом. Metanome реализован на языке программирования Java, который в силу своих особенностей не позволяет достичь высоких результатов производительности. Desbordante же реализован на языке C++, и в некоторых случаях даже простой реализации того же алгоритма на C++ достаточно для увеличения его производительности.

Зависимость порядка (Order dependency, OD) — это зависимость между атрибутами (или списками атрибутов) отношения (в терминах реляционной модели), показывающая, что при упорядочивании значений атрибута (или списка атрибутов) в левой части, значения для правой части тоже будут упорядочены. Высокопроизводительный алгоритм для поиска такого вида зависимостей является хорошим дополнением к текущей функциональности Desbordante. А использование различных техник оптимизации позволило бы еще больше повысить производительность. Поэтому темой настоящей работы стала ускорение работы существующей реализации алгоритма ORDER в Desbordante.

1. Постановка задачи

Целью данной работы является повышение эффективности реализации алгоритма ORDER для поиска зависимостей порядка, с помощью различных оптимизаций, позволяющих улучшить производительность. Для ее достижения были поставлены следующие задачи:

1. Исследовать предметную область поиска зависимостей порядка и по результатам ознакомления написать обзор. На основе обзора сделать вывод о потенциальной полезности алгоритма ORDER, чтобы обосновать необходимость его ускорения.

2. Провести профилирование базовой версии алгоритма, чтобы обнаружить узкие места в реализации, которые требуют оптимизации.

3. Предложить оптимизации, основанные на замене структур данных и использовании библиотечных алгоритмов. Кроме того, предложить улучшения, основанные на многопоточности, для методов, где это является возможным.

4. Интегрировать алгоритм в проект Desbordante. Добавить привязку к Python, которая позволит использовать алгоритм в программах на языке Python. Составить пример, демонстрирующий работу алгоритма, на языке Python и добавить поддержку запуска алгоритма через CLI.

5. Сравнить эффективность работы алгоритма до и после оптимизаций, а также с реализацией из Metanome [5].

2. Обзор

Анализ больших данных приобретает все большую популярность. Пользователям интересны как просто некоторые свойства этих данных, так и возможность их очистки от различных неточностей и искажений.

В данной работе рассматриваются зависимости порядка (Order Dependencies, OD), которые существуют уже довольно давно [11], но в последние несколько лет стали вновь притягивать к себе внимание. Для них появились новые способы применения: поддержка целостности данных на основе ограничений целостности (Integrity Constraints) и оптимизация запросов баз данных.

Размер наборов данных со временем лишь увеличивается, что требует все более эффективных алгоритмов их исследования и обработки. Однако улучшение и создание новых алгоритмов требует хорошего понимания теории и принципов их работы.

2.1. Основные термины

Прежде чем описать основные виды зависимостей порядка, требуется ввести основную терминологию. Поэтому приведем определения, взятые из работы [13].

Пусть R — заголовок отношения, а r — тело отношения. A, B и C обозначают одиночные *атрибуты*, s и t обозначают *кортежи*. Значение атрибута A в кортеже t обозначается как t_A.

Определение 1. Унарной зависимостью порядка над r называется выражение вида $A \rightarrow_\theta B$, где $A, B \in R$, и $\theta \in \{\leq, <, =, >, \geq\}$. Эта зависимость порядка называется валидной зависимостью порядка, если для любых $s, t \in r$, $s[A]\,\theta\,t[A] \Rightarrow s[B]\,\theta\,t[B]$.

Замечание. Из этого определения также следует, что функциональные зависимости являются частным случаем зависимостей порядка, где θ это “=”.

Зависимости порядка определены на списках атрибутов (в отличие

от тех же функциональных зависимостей, которые определены на множествах атрибутов), потому что порядок атрибутов в левой и правой части зависимости имеет значение. Пусть **X, Y** — списки атрибутов. Тогда зависимость между такими списками выглядит как $\mathbf{X} \rightarrow_\theta \mathbf{Y}$. Иногда запись списка может быть явной: ABC — явная конкатенация одиночных атрибутов A, B и C. Запись $s[\mathbf{X}] \leq t[\mathbf{X}]$ означает, что кортеж полученный проекцией кортежа s на **X** *лексикографически* меньше или равен кортежу, который получился проекцией кортежа t на **X**.

Определение 2. Зависимость порядка *полностью нетривиальна*, если её списки атрибутов в левой и правой части не пересекаются.

2.2. Виды зависимостей

Существует несколько видов зависимостей порядка. Обычно, они незначительно отличаются друг от друга. Это позволяет модифицировать существующие алгоритмы для поиска одного из видов, чтобы искать остальные. Поэтому важно знать, какие еще зависимости порядка существуют, так как это поможет представить потенциальные расширения к алгоритму.

Лексикографические зависимости порядка (Lexicographical OD). Лексикографической зависимость называется так из-за порядка, заданного для проекций кортежей на список атрибутов. Рассмотрим таблицу 1. В ней выполняется зависимость $AB \rightarrow_< C$. Применим определение зависимости к кортежам t_3 и t_4: $(1, 1) < (1, 2) \Rightarrow 2 < 3$, поскольку сравнение лексикографическое, это следствие верно.

Таблица 1

	A	B	C
t_1	0	0	0
t_2	0	0	1
t_3	1	1	2
t_4	1	2	3
t_5	1	3	4

Точечные зависимости порядка (Point-Wise OD) [12]. Они отличаются от лексикографических только заданным порядком. Для них он определен следующим образом: $(s_1, ..., s_n) < (t_1, ..., t_n)$, если

$\forall i : 1 \leq i \leq n,\ s_i < t_i$.

Двунаправленные зависимости порядка (Bidirectional OD) [9]. Здесь для каждого атрибута в списке вводится направление его сортировки: восходящее или нисходящее. Так, например, зависимость $\overrightarrow{A} \rightarrow \overleftarrow{B}$ означает, что при восходящей сортировке A, атрибут B будет отсортирован в нисходящем порядке.

Приближенные зависимости порядка (Approximate OD) [3]. Пусть g — функция меры ошибки, e — порог ошибки, r — экзмепляр отношения. Тогда говорим, что зависимость порядка y является приближенной и валидной на r, тогда и только тогда, когда $g(y, r) < e$.

Частичные зависимости порядка (Partial OD) [7]. Здесь зависимость рассматривается не на всем наборе данных, а на некотором заданном его подмножестве.

2.3. Теория для алгоритма ORDER

Далее мы будем рассматривать θ равную только "$<$" и "$\leq$". Алгоритм легко может быть настроен, чтобы находить зависимости со знаками "$>$" и "$\geq$".

Теорема 1. $\mathbf{X} \rightarrow_{<} \mathbf{Y}$ валидна $\iff$ $\mathbf{Y} \rightarrow_{\leq} \mathbf{X}$ валидна.

Последняя теорема позволяет нам искать зависимости только для одного значения θ. Пусть мы теперь рассматриваем только θ со значением "$<$". Поэтому теперь запись $\mathbf{Y} \rightarrow \mathbf{X}$ подразумевает именно это значение θ.

Для проверки зависимостей на валидность нам потребуется два определения.

Определение 3. Кортежи s и t в отношении r образуют *merge* для пары списков атрибутов $(\mathbf{X}, \mathbf{Y})$, если $s[\mathbf{X}] \neq t[\mathbf{X}]$, но $s[\mathbf{Y}] = t[\mathbf{Y}]$.

Определение 4. Кортежи s и t в отношении r образуют *swap*

для пары списков атрибутов $(\mathbf{X}, \mathbf{Y})$, если $s[\mathbf{X}] < t[\mathbf{X}]$, но $s[\mathbf{Y}] > t[\mathbf{Y}]$.

Лемма 1. Зависимость порядка $\mathbf{X} \rightarrow_< \mathbf{Y}$ валидна, тогда и только тогда, когда для пары списков атрибутов $(\mathbf{X}, \mathbf{Y})$ нет *merge* и *swap*.

Минимальность является важным свойством зависимостей, потому что оно не только предлагает более короткую запись, но и помогает уменьшить пространство поиска. Определения минимальности являются основой для правил отсечения кандидатов, которые в случае алгоритма ORDER играют ключевую роль в сохранении высокой производительности.

Определение 5. Список атрибутов $\mathbf{X}$ называется *минимальным* тогда и только тогда, когда для двух непрерывных подсписков $\mathbf{V}$ и $\mathbf{W}$ в $\mathbf{X}$, где

1. $\mathbf{W}$ непосредственно предшествует $\mathbf{V}$ в $\mathbf{X}$, *или*
2. $\mathbf{W}$ следует (необязательно непосредственно) за $\mathbf{V}$ в $\mathbf{X}$,

$\mathbf{V} \rightarrow_\leq \mathbf{W}$ не является валидной.

Определение 6. Зависимость порядка $\mathbf{X} \rightarrow_< \mathbf{Y}$ минимальна тогда и только тогда, когда

1. $\nexists \mathbf{V} \in PREFIXES(\mathbf{X})$, таких что $\mathbf{V} \not\rightarrow_< \mathbf{Y}$, *и*
2. $\nexists \mathbf{W} \in PREFIXES(\mathbf{Y})$, таких что $\mathbf{X} \rightarrow_< \mathbf{W}$ валидна, *и*
3. $\mathbf{X}$ минимален, *и*
4. $\mathbf{Y}$ минимален,

где $PREFIXES(\mathbf{X})$ возвращает множество всех префиксов $\mathbf{X}$.

2.4. Алгоритмы поиска зависимостей порядка

Первым алгоритмом поиска лексикографических зависимостей порядка является алгоритм ORDER. Он ищет зависимости над решеткой

кандидатов, как и в алгоритме поиска функциональных зависимостей TANE [15]. Однако, в отличие от функциональных зависимостей, зависимости порядка определены на списках, а не множествах атрибутов. Это сильно увеличивает размер решетки, что заставляет использовать более агрессивные подходы к отсечению кандидатов. Это привело к тому, что алгоритм ORDER стал неполным, потому что он не находит зависимости следующих видов:

1. Алгоритм упускает зависимости вида $\mathbf{X} \to \mathbf{XY}$;
2. Если $\mathbf{X} \to \mathbf{Y}$ инвалидируется с помощью *swap*, то зависимость $\mathbf{X}A \to \mathbf{Y}B$ не рассматривается, из-за чего упускаются зависимости вида $\mathbf{X}A \to \mathbf{X}A\mathbf{Y}B$.

Описанная выше терминология в работе [8] стала называться *списковой формализацией* из-за представления частей зависимости как списков атрибутов. В этой же работе была предложена новая *формализация множеств*, которая основывалась на списковой, и новый алгоритм поиска зависимостей FASTOD. Он позволяет найти все зависимости порядка в таблице (то есть является полным в отличие от ORDER). Это стало возможным именно благодаря новой формализации, которая значительно уменьшила размер решетки кандидатов. Поскольку теперь обход решетки проходит быстрее, старые правила отсечения кандидатов были упразднены, что как раз и позволило алгоритму стать полным. Однако это не означает, что FASTOD во всем превосходит ORDER.

ORDER имеет очень агрессивную стратегию отсечения кандидатов, поэтому еще на ранних этапах он в состоянии понять, что продолжение поиска зависимостей не имеет смысла. FASTOD же продолжит обход решетки кандидатов в надежде найти еще зависимости, которых однако там может и не быть. Это приводит к тому, что на многих датасетах ORDER работает значительно быстрее, но находит намного меньше зависимостей. Это различие дает смысл применению обоих алгоритмов в подходящих для них ситуациях.

2.5. Выводы

После ознакомления с предметной областью было принято решение о необходимости поддержки и улучшения алгоритма ORDER в Desbordante. Зависимости порядка имеют полезные применения, а сам алгоритм ORDER может соревноваться с алгоритмом FASTOD, потому что их применимость зависит от конкретных наборов данных и цели их исследования. Дополнительно, алгоритм ORDER может быть расширен в будущем для поиска других видов зависимостей порядка, что еще больше увеличивает его ценность. Например, чтобы добавить в ORDER возможность находить двунаправленные зависимости, достаточно изменить метод проверки на валидность, который будет проходить партицию для правой части зависимости еще и в обратном порядке. FASTOD из-за больших различий в теории требует более масштабных модификаций для получения такой возможности.

3. Описание алгоритма

3.1. Общее описание

Алгоритм ORDER ищет все *минимальные полностью нетривиальные* зависимости порядка вида $\mathbf{X} \rightarrow_{<} \mathbf{Y}$.

Работа алгоритма основана на обходе решетки. Поиск происходит, начиная со списка атрибутов длины один, который далее удлиняется, при проходе решетки. Пример решетки для трех атрибутов можно увидеть на рисунке 1.

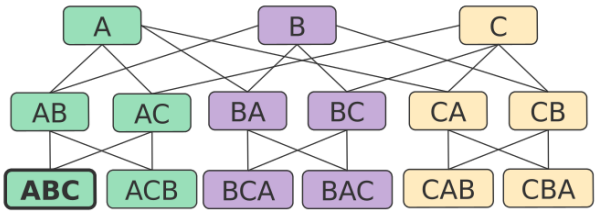

Рис. 1: Решетка для трех атрибутов, рисунок взят из [13]

Псевдокод алгоритма ORDER представлен в листинге 1.

```
1 CreateSingleColumnSortedPartitions();
2 previous_candidate_sets = {};
3 current_candidate_sets = {};
4 lattice = new ListLattice(single_attributes);
5 while (lattice is not empty) do
6     current_candidate_sets = UpdateCandidateSets(previous_candidate_sets);
7     ComputeDependencies(lattice.GetCurrentLevel(), current_candidate_sets);
8     Prune(lattice, current_candidate_sets);
9     lattice = GenerateNextLevel(lattice);
```

Листинг 1: ORDER

Сначала алгоритм создает *Sorted Partitions* для каждого отдельного атрибута таблицы. Описание *Sorted Partitions* будет представлено в следующей секции. После происходит инициализация множеств кандидатов: нам требуется две переменные, поскольку вычисления на текущем уровне решетки иногда будут зависеть от того, какие кандидаты были

на предыдущем уровне. Из одиночных атрибутов создается решетка. Далее начинается основной цикл алгоритма.

Алгоритм завершается в случае, когда решетка окажется пуста. Такое может случиться, потому что в методе *Prune* из нее убираются неперспективные узлы. Для каждого уровня решетки выполняется несколько операций:

1. На основе кандидатов предыдущего уровня мы создаем новых, путем добавления в список правой части зависимостей нового атрибута. Учитывая правила отсечения и определение минимальности, некоторые кандидаты отсеиваются;

2. Добавляются новые кандидаты, появляющиеся путем разбиения узлов решетки на два списка атрибутов, которые становятся левой и правой частью зависимости. Затем эти зависимости проверяются на валидность, результат влияет на то, останутся они в множестве кандидатов, будут удалены или добавлены в список валидных зависимостей;

3. Удаляются некоторые узлы из решетки, если они более не смогут породить кандидатов;

4. Создается следующий уровень решетки, путем добавления нового атрибута в конец списка атрибутов каждого узла.

3.2. Отсортированные партиции

Отсортированные партиции (Sorted Partitions, SP, далее просто партиции) — это структура данных, представляющая собой список классов эквивалентности, в которых находятся индексы кортежей для списка атрибутов $\mathbf{X}$. Внутри партиций эти индексы расположены в том же порядке, в каком бы находились кортежи после их сортировки для данного списка атрибутов. То есть в классах эквивалентности находятся индексы, для которых значения кортежей равны и, соответственно, расположены последовательно после сортировки.

Таблица 2: Таблица с информацией о доставке, адаптированная из [13]

t_{id}	Weight	Shipment cost	Days
0	5	14	2
1	10	22	6
2	3	10	4
3	10	25	7
4	5	14	2
5	20	40	8

На примере Таблицы 2 рассмотрим, как выглядят партиции для каждого атрибута в отдельности:
$SP(Weight) = (\{2\}, \{4, 0\}, \{3, 1\}, \{5\})$
$SP(Shipment\,cost) = (\{2\}, \{0, 4\}, \{1\}, \{3\}, \{5\})$
$SP(Days) = (\{0, 4\}, \{2\}, \{1\}, \{3\}, \{5\})$

На их примере можно проиллюстрировать, как определения *merge* и *swap* работают в терминах партиций.

Зависимость $\mathbf{Y} \to \mathbf{X}$ инвалидируется с помощью *swap*, если $SP(\mathbf{Y})$ и $SP(\mathbf{X})$ содержат индексы в разном порядке для любого набора перестановок индексов внутри классов эквивалентности. Из этого следует, что *swap* имеется, если среди k первых индексов есть такие, что присутствуют в одной партиции, но отсутствуют среди k первых индексов в другой для любых перестановок индексов внутри классов эквивалентности. Наоборот, зависимость $\mathbf{Y} \to \mathbf{X}$ не содержит *swap*, если можно расположить индексы внутри классов эквивалентности так, что индексы в $SP(\mathbf{Y})$ и $SP(\mathbf{X})$ имеют одинаковый порядок.

Зависимость $Days \to Weight$ содержит *swap*, достаточно взять k равное единице, чтобы это заметить. В партиции для $Weight$ первым индексом является двойка, а в партиции для $Days$ на первом месте может быть как ноль, так и четыре, но никак не двойка.

Зависимость $Weight \to Shipment\,cost$ не содержит *swap*. Мы можем подобрать нужные перестановки в $SP(Days)$, поменяв местами индексы ноль и четыре во втором классе эквивалентности и один и три в третьем классе эквивалентности. Тем самым порядок индексов в двух

партициях совпадает.

Зависимость $\mathbf{Y} \to \mathbf{X}$ содержит *merge*, если есть такая пара индексов, что в $SP(\mathbf{Y})$ находятся в разных классах эквивалентности, а в $SP(\mathbf{X})$ находятся в одном.

Зависимость $Shipment\,cost \to Weight$ содержит *merge*, потому что индексы один и три в $SP(Shipment\,cost)$ находятся в разных классах эквивалентности, а в $SP(Weight)$ в одном классе.

На этих знаниях и основывается алгоритм проверки валидности зависимости. Он проверяет, что k первых индексов в паре партиций одинаковы, с постепенным увеличением k.

3.3. Вычисление произведения партиций

Произведение позволяет получать партиции для списков атрибутов. Например: $SP(AB) = SP(A) \times SP(B)$, причем операция является некоммутативной. Говоря просто, мы уточняем порядок для значений класса эквивалентности из A с помощью порядка этих значений внутри B. Алгоритм произведения проходит по классам из $SP(A)$, если это класс размера один, то он просто вставляется в новую партицию. Если же размер больше одного, то индексы этого класса ищутся в $SP(B)$, чтобы затем добавить их в результат в том же порядке, что и в $SP(B)$ (то есть класс эквивалентности может разделиться на несколько, если эти индексы были в разных классах в $SP(B)$). Пример можно увидеть на рис. 2

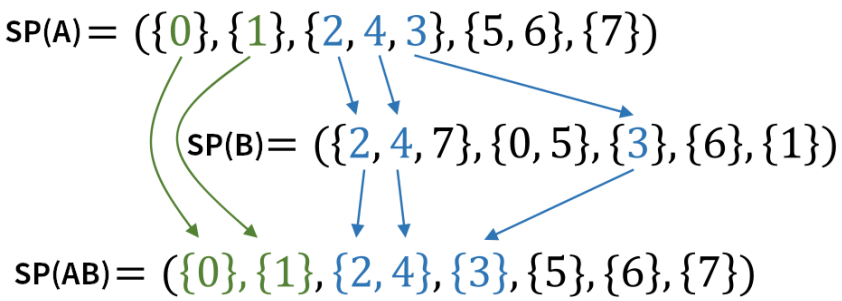

Рис. 2: Пример вычисления произведения партиций из [13]

3.4. Проверка на валидность

Метод проверки на валидность обнаруживает, если в паре партиций присутствует хотя бы один *swap*. Это происходит с помощью сравнения наборов индексов в классах эквивалентности линейным поиском. Псевдокод метода проверки валидности представлен в листинге 2.

```
1  ValidityType CheckForSwap(SP lhs_partition, SP rhs_partition):
2      ValidityType res = ValidityType::valid;
3      int l_i = 0, r_i = 0;
4      bool next_l = true, next_r = true;
5      SP::EquivalenceClass l_eq_class, r_eq_class;
6      while (l_i < lhs_partition.size() and r_i < rhs_partition.size()) do
7          if (next_l)
8              l_eq_class = lhs_partition[l_i];
9          if (next_r)
10             r_eq_class = rhs_partition[r_i];
11         if (l_eq_class.size() < r_eq_class.size())
12             if (not SubsetSetDifference(l_eq_class, r_eq_class))
13                 return ValidityType::swap;
14             else
15                 res = ValidityType::merge;
16                 ++l_i;
17                 next_l = true;
18                 next_r = false;
19         else
20             if (not SubsetSetDifference(r_eq_class, l_eq_class))
21                 return ValidityType::swap;
22             else
23                 ++r_i;
24                 next_r = true;
25                 if (l_eq_class.empty())
26                     ++l_i;
27                     next_l = true;
28                 else
29                     next_l = false;
30     return res;
```

Листинг 2: CheckForSwap

Функция *SubsetSetDifference* проверяет, является ли левый класс эквивалентности подмножеством правого, и если является, то вычитает его из правого. Если же левый класс не является подмножеством правого, то возвращает *false*.

Действительно, как и говорилось в предыдущей секции, метод проверяет, что индексы можно расположить в одном и том же порядке, если переставлять их внутри классов эквивалентности. Поэтому метод заключается в том, что берутся последовательно классы эквивалентности и в них ищутся одни и те же индексы. Если какой-то индекс найти не удается, значит среди k первых индексов есть несовпадение, что является определением *swap* из прошлой секции.

3.5. Отсечение кандидатов

Ранее уже было упомянуто, что, в связи с большим размером решетки, алгоритм использует агрессивное отсечение кандидатов. Они позволяют избежать большого количества проверок на валидность, при этом теряются только зависимости конкретных видов, которые в списковой формализации не считаются важными. Основные правила отсечения:

1. $\mathbf{X} \not\to_< \mathbf{Y} \Rightarrow \mathbf{XV} \not\to_< \mathbf{Y}$;
2. $\mathbf{X} \to_< \mathbf{Y}$ валидна $\Rightarrow \mathbf{X} \to_< \mathbf{YW}$ валидна;
3. Пусть $\mathbf{X} \not\to_< \mathbf{Y}$ была инвалидирована с помощью *swap*. Тогда $\mathbf{X} \not\to_< \mathbf{Y} \Rightarrow \mathbf{XV} \not\to_< \mathbf{YW}$;
4. Пусть $\mathbf{X}$ содержит только уникальные значения. Тогда $\mathbf{X} \to_< \mathbf{Y}$ валидна $\Rightarrow \mathbf{XV} \to_< \mathbf{YW}$ валидна.

Списки атрибутов $\mathbf{X}, \mathbf{Y}, \mathbf{V}, \mathbf{W}$ не пересекаются, и только $\mathbf{V}, \mathbf{W}$ могут быть пустыми.

От них отделяется правило отсечения для случая зависимости с *merge*. Кандидаты инвалидированные *merge*, в отличие от кандидатов инвалидированных *swap*, могут породить валидные зависимости. Например, пусть $A \not\to_< B$ инвалидирована только с помощью *merge*. Тогда мы можем сказать, что $A\mathbf{X} \not\to_< B$, потому что *merge* останется. Но может существовать такой $\mathbf{Y}$, что $A\mathbf{X} \to_< B\mathbf{Y}$ валидна. И эта зависимость

не будет найдена, если мы уберем (A, B) из кандидатов. Поэтому тут требуется другой подход. Пусть мы в добавок к тому, что $A \not\to_< B$ инвалидируется с помощью *merge*, знаем, что зависимости вида $A \to_< B\mathbf{Y}$ ($\mathbf{Y}$ не пуст) не имеет смысла проверять. Под "не имеет смысла проверять" понимается, что кандидата такой зависимости нет и он не сможет породиться из текущих. Тогда нам не имеет смысла проверять зависимости вида $A\mathbf{X} \not\to_< B\mathbf{Y}$. Пример такой ситуации можно наблюдать в таблице 3

Таблица 3: $A \not\to_< B$, $AX \not\to_< B$, $AX \to_< BY$.

A	B	X	Y
0	1	0	2
1	1	0	3
2	2	1	4
2	3	2	4
3	4	3	4

Еще один способ уменьшения количества кандидатов может быть применен еще в самом начале алгоритма во время создания партиций для одиночных атрибутов. Если после создания партиции оказывается, что она состоит из всего одного класса эквивалентности, то любая зависимость, где этот атрибут является единственным в левой части, будет валидна. Наличие всего одного класса эквивалентности означает, что все значения атрибута равны. Из определения зависимости для знака "$<$" можно понять, почему так происходит: $s[A] < t[A] \Rightarrow s[B] < t[B]$. Для равных значений левое сравнение всегда будет ложным, что означает истинность импликации при любом следствии.

4. Базовая реализация и профилирование

На момент начала работы над дипломной работой прототип реализации алгоритма ORDER уже присутствовал в Desbordante, но не был интегрирован. Он был реализован автором данной дипломной работы ранее. Чтобы приступить к применению техник оптимизации, нужно сначала улучшить текущее качество кода, полноценно интегрировать его в Desbordante и провести исследование его производительности. В связи с эти были выделены следующие шаги:

1. Провести рефакторинг существующей реализации;
2. Интегрировать алгоритм в Desbordante, добавив привязку к Python и поддержку запуска в CLI;
3. Провести профилирование алгоритма.

4.1. Рефакторинг

Изначальная реализация алгоритма была темой учебной практики шестого семестра. Итогом стал прототип реализации, который корректно находил все зависимости, но по эффективности уступал CLI версии Metanome. Реализация получилась громоздкой и трудно-читаемой, поэтому в первую очередь было решено провести рефакторинг. Это позволило лучше охватить существующие недочеты: например, при использовании хэш-таблицы иногда возникал повторный поиск, хотя можно было делать это однократно, использовав итератор. Недочеты были исправлены, диаграмму классов итоговой реализации можно увидеть на рис. 3.

Алгоритм ORDER наследуется от базового класса алгоритма, который предоставляет базовый интерфейс для всех алгоритмов. Принцип работы методов Order уже был рассмотрен в секции 3. Экземпляр Order хранит в себе указатель на решетку, набор партиций и указатель на типизированное отношение (знание типов требуется для сортировки кортежей).

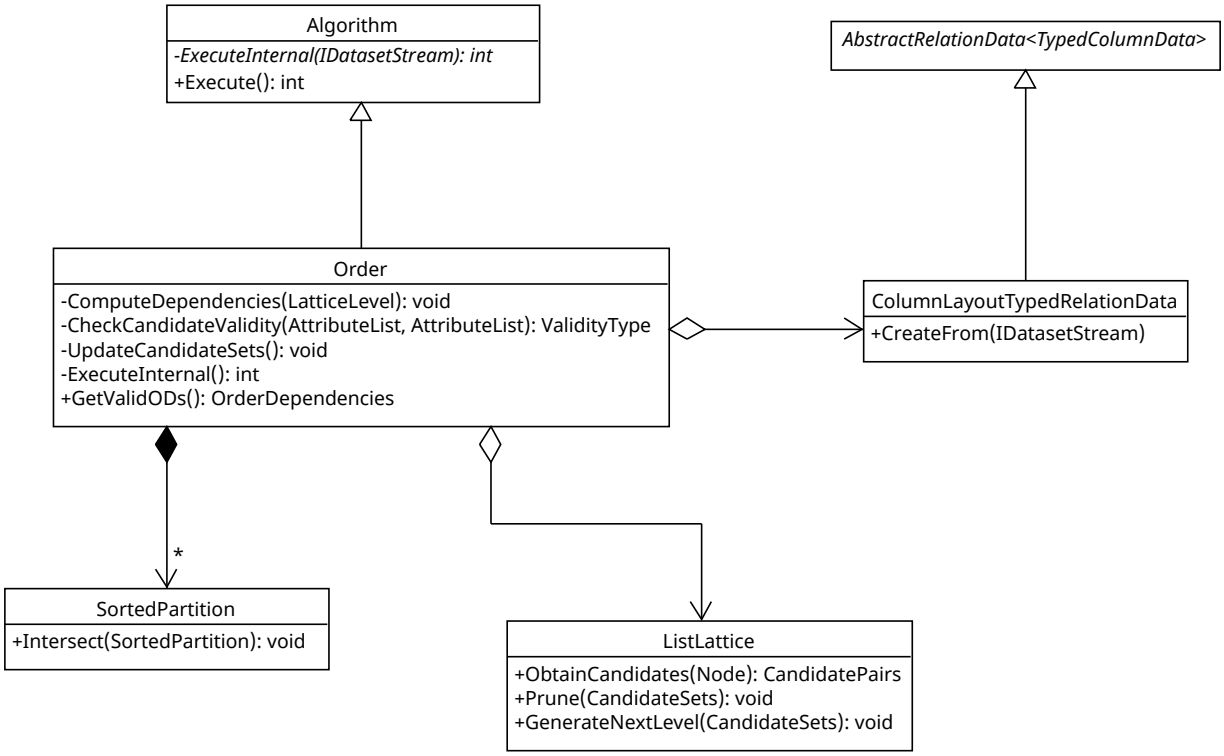

Рис. 3: Диаграмма классов ORDER

Рефакторинг позволил избавиться от громоздкости изначальной реализации. Например, максимальное количество строк в одном файле уменьшилось с примерно восьмисот до четырехсот. Классы алгоритма теперь имеют более высокую связность и соблюдают принцип единственности ответственности.

4.2. Интеграция в Desbordante

Проект Desbordante рассчитан не только на пользователей, владеющих C++. Чтобы предоставить способ использования для более широкой аудитории, интеграция в Desbordante проводилась в три шага:

1. Была добавлена привязка к Python. Она позволяет использовать алгоритм в сценариях на языке Python. Для ее реализации в Desbordante используется библиотека pybind11[1].

2. Была добавлена возможность запуска алгоритма из CLI версии

[1] https://github.com/pybind/pybind11

Desbordante. Привязка к Python является необходимым условием для добавления алгоритма в CLI часть.

3. Были добавлены примеры запуска алгоритма, позволяющие получить интуитивное понимание работы алгоритма и определения зависимости порядка.

Пример использования алгоритма на языке Python можно увидеть в листинге 3.

```
1  import desbordante
2
3  TABLE = 'shipping.csv'
4
5  algo = desbordante.od.algorithms.Order()
6  algo.load_data(table=(TABLE, ',', True))
7  algo.execute()
8
9  list_ods = algo.get_list_ods()
10
11 print('ods:', len(list_ods))
12
13 for od in list_ods:
14     print(od.lhs, "->", od.rhs)
```

Листинг 3: Пример вызова алгоритма в Python

В пятой строке создается экземпляр алгоритма ORDER. После чего в шестой в него передаются данные о таблице, чтобы алгоритм начал процесс ее чтения. В следующей строке происходит запуск алгоритма. Результаты можно получить, вызвав метод get_list_ods(), как показано в девятой строке. Наконец, в строках 13–14 производится печать найденных зависимостей, так как, благодаря способу привязки, зависимости представляются в виде полноценных объектов, которые поддерживают операции языка Python.

Возвращаясь к реализации в целом, можно сказать что ее уже можно использовать как основу для разработки дальнейших техник оптими-

зации. Чтобы обнаружить конкретные части алгоритма, нуждающиеся в ускорении, было решено использовать профилирование.

4.3. Профилирование

Профилирование производилось с применением Flamegraph[2], который позволяет визуализировать стек вызовов и обнаружить тем самым узкие места в программе.

Основные затраты времени алгоритма приходятся на три метода:

1. **CreateSingleColumnSortedPartitions**, который создает партиции для каждой одиночной колонки;
2. **CheckForSwap**, который проверяет зависимость на валидность на основе пары партиций;
3. **SortedPartition::Intersect**, который находит произведение (пересечение) двух партиций.

В зависимости от характеристик набора данных акцент смещается с одного метода на другой. Например, произведение партиций совершается тем чаще, чем больше зависимостей порядка находится в таблице. А в данных, в которых отсутствуют зависимости, этот метод практически не вызывается.

В **CreateSingleColumnSortedPartitions** основные затраты времени приходятся на сортировку значений столбцов и работу с хэш-таблицей. В **CheckForSwap** большая часть времени уходит на операцию нахождения разности множеств **SubsetSetDifference** и также работу с хэш-таблицей.

Рассмотрим два примера результатов Flamegraph. Сначала рассмотрим случай, когда в данных отсутствуют зависимости порядка на рис. 4.

Тут можно наблюдать, что почти всё время работы алгоритма занимает метод CreateSingleColumnSortedPartitions, метод проверки валидности лишь малую часть и совсем отсутствует вызов метода для вычисления произведения партиций. Благодаря агрессивному отсечению

[2] https://github.com/brendangregg/FlameGraph

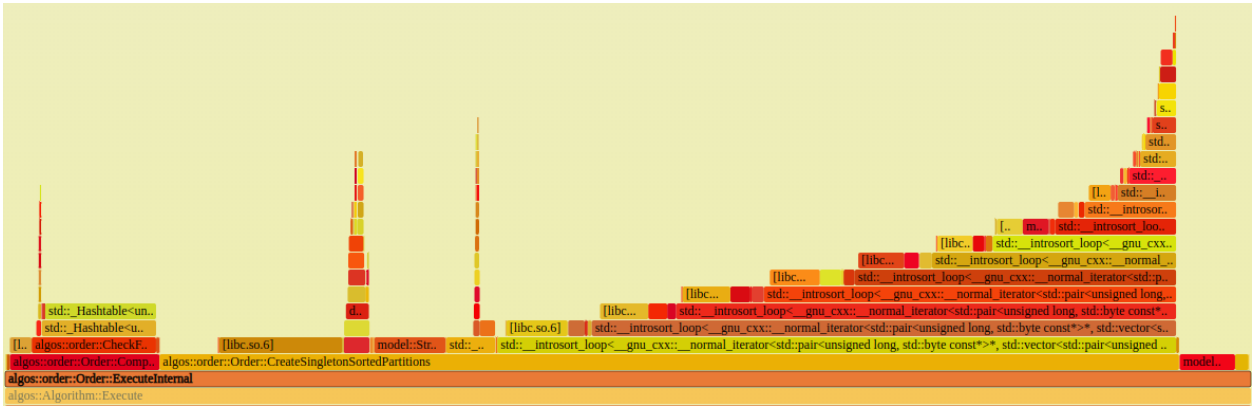

Рис. 4: Flamegraph для данных без зависимостей

кандидатов, алгоритм на ранних уровнях решетки прекращает вычисление зависимостей, поэтому мы имеем такое малое количество проверок на валидность.

Теперь рассмотрим случай, когда зависимости порядка присутствуют на рис. 5.

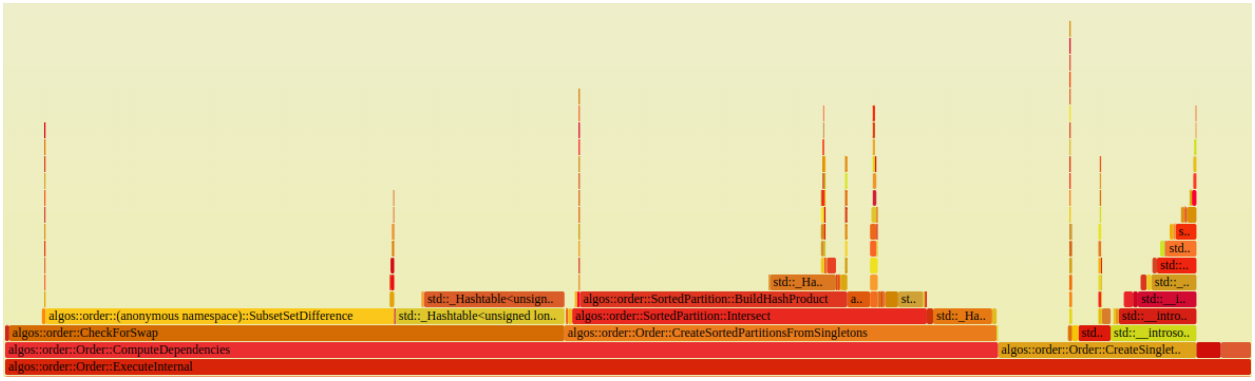

Рис. 5: Flamegraph для данных с зависимостями

Здесь уже видно, что акцент сместился на методы проверки на валидность и вычисления произведения партиций.

Учитывая эти наблюдения, можно попытаться найти подход к оптимизации алгоритма.

5. Оптимизации

Оптимизации для алгоритма основаны на использовании более эффективных структур данных и алгоритмов сортировки из библиотеки Boost[3], а также применении многопоточности. Многопоточность применяется в реализации ORDER впервые, то есть отсутствует в реализации Metanome.

5.1. Сортировка

Сортировка значений происходит в начале алгоритма, чтобы получить представление данных в виде партиций.

Оптимизировать этот этап работы алгоритма имеет особый смысл, потому что он происходит в самом начале работы алгоритма для любого набора данных. Таким образом, его оптимизация даст прирост в производительности на всех наборах данных.

В качестве рассматриваемых вариантов сортировок были взяты сортировки из стандартной библиотеки C++ и библиотеки Boost.Sort[4], которая предоставляет набор различных сортировок (как параллельных, так и последовательных). Для однопоточного исполнения была взята сортировка flat_stable_sort, которая характеризуется хорошей производительностью и низким потреблением памяти. Для многопоточного исполнения алгоритма по тем же причинам была взята сортировка block_indirect_sort.

5.2. Валидация кандидатов

Для валидации зависимости $\mathbf{X} \rightarrow \mathbf{Y}$ берутся партиции для списков атрибутов $\mathbf{X}$ и $\mathbf{Y}$. Затем в них последовательно берется по классу эквивалентности, в которых происходит поиск одинаковых элементов (алгоритм валидации основан на том, что среди первых k индексов должны быть только одинаковые). Структурой, подходящей для боль-

[3] https://www.boost.org/

[4] https://www.boost.org/doc/libs/master/libs/sort/doc/html/index.html

шого количества поисков является unordered_set, которым в программе представлен класс эквивалентности. Мы рассматриваем реализации unordered_set из стандартной библиотеки C++ и из библиотеки Boost, которая называется unordered_flat_set и характеризуется высокой производительностью. Возможно, использование реализации из Boost может привести к увеличению производительности алгоритма в целом.

5.3. Вычисление произведения партиций

Вычисление произведения происходит тем чаще, чем больше мы имеем зависимостей в текущем наборе данных. Вычисление происходит с использованием хэш-таблиц. Поэтому мы рассматриваем замену unordered_map из стандартной библиотеки на unordered_flat_map из библиотеки Boost, которая может принести потенциальное увеличение производительности на данных с большим количеством зависимостей.

5.4. Многопоточность

Многопоточность применялась к произведению партиций и проверке на валидность. Поскольку проверка на валидность происходит чаще вычисления произведения партиций, то её многопоточная реализация имела более высокий приоритет.

Идея заключается в том, что процедура проверки завершается в тот же момент, как только обнаруживается *swap*, поэтому можно запустить ее для разных частей пары партиций одновременно. Как только один из потоков обнаружит *swap*, остальные могут прекратить свое выполнение. Однако поскольку алгоритм работает с парой партиций, разбить задачу на несколько становится тяжело из-за наличия классов эквивалентности с разным количеством элементов. Разбиение задач происходит по количеству индексов, а не количеству классов эквивалентности. Однако требуется показать, что, рассматривая партиции по частям, алгоритм все еще останется корректен. Для этого воспользуемся теоремой ниже.

Будем считать, что все индексы внутри классов эквивалентности отсортированы, чтобы можно было опустить уточнение, что часть

утверждений верна “для некоторой перестановки индексов внутри классов эквивалентности”.

Теорема 2. Для партиций $SP(X)$ и $SP(Y)$ отсутствует *swap* $\iff$ если взять любую часть этих партиций с индексами на позициях от i до $i+j$ $(\forall i, j \in \mathbb{N})$, то среди них будет отсутствовать *swap*.

Доказательство основано на том, что при отсутствии *swap* для $SP(X)$ и $SP(Y)$, k первых индексов из $SP(X)$ должны находиться среди первых k индексов $SP(Y)$ $\forall k \in \mathbb{N}$. Взяв $k = i$, а затем $k = i+j$ можно понять, что и на промежутке от i до $i+j$ индексы должны совпадать. Для доказательства в обратную сторону достаточно взять промежуток длинной равный количеству всех индексов.

Точка разбиения выбирается отсчетом равного количества индексов в каждой партиции. Имеется несколько случаев, в зависимости от точки разбиения:

Случай 1. Отсутствие наложения. В случае отсутствия наложения классов эквивалентности разбиение проводится тривиальным образом:

$$\begin{array}{cccc|cc} \{ & 0 & 1 & \} & 2 & 3 \\ & 1 & 0 & & 3 & 2 \end{array}$$

Случай 2. Наложение классов. В случае наличия наложения классов разбить прямо в точке разбиения становится проблематично, потому что теперь требуется выделить общий набор индексов и разделить классы эквивалентности на несколько, что для некоторых случаев может лишить нас *merge* и усложнить поиск *swap*. Чтобы этого избежать, можно сдвинуть точку разбиения, чтобы получить случай проще, но тогда разбиение будет неравномерным.

Данный подход к разбиению оказался слишком трудоемким, что не позволило получить прирост производительности от многопоточности. Поэтому подход к распараллеливанию этой части изменился. Задача без всякого разбиения может быть решена двумя потоками: первый бу-

дет проходить партиции с начала, а второй с конца. Рассмотрим пример: $SP(A) = (\{2\}, \{0, 4\}, \{1\}, \{3\}, \{5\})$ и $SP(B) = (\{2\}, \{4\}, \{0\}, \{3, 5\}, \{1\})$. При обычной проверке, которая начинается с начала партиций, алгоритм бы дошел до четвертого класса эквивалентности в $SP(B)$ и третьего в $SP(A)$, после чего завершился с результатом *swap*. При проходе с конца алгоритм завершился бы на первой итерации с тем же результатом. Тогда запуск потоков с двух сторон может потенциально улучшить производительность метода: как только один из них обнаружит *swap*, то сообщит об этом второму, чтобы тот завершился, и вернет результат.

Вычисление произведения партиций тоже подверглось частичной параллелизации. Одним из шагов в вычислении произведения является построение хэш-таблицы, которая хранит классы эквивалентности, которые пришлось разбить на несколько. Пусть возьмем $SP(AB) = SP(A) \times SP(B)$. Ключом в таблице является номер класса эквивалентности в $SP(A)$, а значением список классов эквивалентности, на которые его пришлось разбить в соответствии с классами эквивалентности из $SP(B)$. Можно параллельно составлять несколько таких таблиц, если просматривать части $SP(B)$ параллельно. Затем нужно будет лишь собрать результаты из всех получившихся таблиц.

6. Эксперименты

В ходе исследования предметной области была обнаружена реализация алгоритма ORDER в инструменте Metanome. Ее сравнение производилось с реализацией автора данной работы, а также с оптимизированной версией этой реализации.

Фаза предварительной обработки данных не учитывалась в экспериментах, потому что ее оптимизация не является целью данной работы. Реализация в Metanome так же не учитывает время обработки данных, поэтому оптимизацию этого этапа было решено оставить для дальнейших исследований.

Эксперименты показали, что время работы алгоритмов не изменялось более чем на 1% и 3% для C++ и Java реализаций соответственно. Поэтому в качестве результата каждого эксперимента было решено взять усредненное значение времени работы для трех запусков алгоритма на каждом наборе данных.

Используемые наборы данных представлены в Таблице 7.

Алгоритм реализован на C++ 17 и является частью open-source проекта Desbordante (pull request 294[5]). Код алгоритма с применением всех оптимизаций можно найти по ссылке[6].

6.1. 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, openjdk 11.0.21 2023-10-17, OpenJDK Runtime Environment (build 11.0.21+9-post-Ubuntu-0ubuntu123.10), OpenJDK 64-Bit Server VM (build 11.0.21+9-post-Ubuntu-0ubuntu123.10, mixed mode, sharing).

[5]https://github.com/Desbordante/desbordante-core/pull/294

[6]https://github.com/Sched71/Desbordante/tree/final_measure

6.2. Исследовательские вопросы

Чтобы оценить, насколько предложенная реализация превосходит по производительности реализацию в Metanome и насколько предложенные оптимизации влияют на производительность реализации автора, были поставлены следующие вопросы:

RQ1 Может ли реализация на языке C++ без существенных изменений в сравнении с реализацией в Metanome иметь более высокую производительность?

RQ2 Насколько оптимизации с использованием средств библиотеки Boost смогут повлиять на производительность реализации на языке C++?

RQ3 Как повлияет применение многопоточности на производительность реализации на языке C++?

RQ4 Насколько существенным может быть уменьшение объема потребляемой памяти однопоточной реализации на языке C++ в сравнении с реализацией в Metanome?

6.3. Проведенные эксперименты

Для ответа на исследовательские вопросы было проведено четыре эксперимента (по эксперименту на каждый вопрос с тем же номером).

Итоговые результаты эксперимента можно наблюдать в Таблице 8. Там присутствуют результаты измерений времени работы алгоритма с применением сразу всех техник оптимизации.

6.3.1. Эксперимент 1

В этом эксперименте мы сравнили базовые реализации алгоритма из Desbordante и Metanome. Результаты можно наблюдать в Таблице 8.

Эксперименты показали, что базовая реализация на языке C++ превосходит реализацию на языке Java по производительности в большинстве случаев. Наблюдается повышение производительности вплоть до

девяти раз, в среднем же производительность улучшилась в четыре раза.

Поскольку есть наборы данных, где реализация в Metanome имеет более высокую производительность, можно сказать, что требуются дополнительные оптимизации алгоритма.

6.3.2. Эксперимент 2

В этом эксперименте мы сравнили базовую реализацию на C++ с реализациями, в которых были применены оптимизации с использованием средств из библиотеки Boost. Оптимизации применялись как по отдельности, так и одновременно.

Было произведено сравнение boost::unordered_flat_map (flat_map), boost::unordered_flat_set (flat_set), и boost::block_indirect_sort (sort) с их стандартными аналогами — std::unordered_map, std::unordered_set, и std::sort. Соотношения времени выполнения представлены в Таблице 4.

Таблица 4: Улучшения от оптимизаций в сравнении с базовой версией

Набор данных	flat_map	flat_set	sort	Вместе
Diabetes	1.017x	4.954x	0.969x	6.253x
Pfw	1.019x	1.850x	1.015x	2.258x
Ditag	1.026x	1.267x	1.546x	2.189x
Credit	1.003x	1.128x	1.421x	1.727x
Epic	1.023x	1.480x	1.268x	2.128x
Modis	1.025x	1.592x	1.341x	2.550x
Bay	1.026x	1.360x	1.503x	2.785x

По результатам экспериментов можно сделать вывод, что использование unordered_flat_map не принесло существенного улучшения, в отличие от использования сортировок из библиотеки Boost и использования unordered_flat_set. Кроме того, одновременное применение всех оптимизаций дает еще больший прирост производительности, чем произведение результатов прироста при отдельном применении.

6.3.3. Эксперимент 3

В третьем эксперименте мы сравнили производительность многопоточной реализации ORDER и базовой реализации. Результаты можно увидеть в Таблице 5.

Таблица 5: Многопоточная версия ORDER в сравнении с базовой

Данные	C++ (база)	C++ (8 потоков)	Улучшение
Diabetes	3696	3694	1.00x
Pfw	472	432	1.09x
Ditag	7219	7135	1.01x
Credit	5279	5335	0.99x
Epic	3438	3002	1.14x
Modis	9339	8471	1.10x
Bay	18466	18171	1.01x

Прирост производительности произошел только на наборах данных, в которых присутствуют зависимости порядка. Это логично, учитывая, что многопоточность затрагивает шаги алгоритма, связанные с проверкой на валидность и вычислением произведения партиций, которые выполняются тем чаще, чем больше зависимостей есть в наборе.

6.3.4. Эксперимент 4

В четвертом эксперименте мы сравнили использование памяти оптимизированной однопоточной реализации C++ с версией из Metanome. Результаты проведенных экспериментов представлены в Таблице 6.

Таблица 6: Metanome vs Desbordante потребление памяти (ORDER)

Данные	C++ (MB)	Java (MB)	Улучшение
Diabetes	177.94	429.37	2.413x
Pfw	150.69	265.62	1.762x
Ditag	3715.66	5951.69	1.601x
Credit	5607.87	7333.7	1.307x
Epic	662.78	1309.7	1.976x
Modis	2614.75	6240.43	2.386x
Bay	5690.77	7345.08	1.290x

Эксперименты показали, что наша реализация использует меньше памяти по сравнению с реализацией из Metanome. Если быть точным, была достигнута экономия памяти в 1,3-2,4 раза, в зависимости от набора данных.

Заключение

Была реализована высокопроизводительная версия алгоритма ORDER для поиска зависимостей порядка, путем использования различных техник оптимизации. Как результат, выполнены следующие задачи:

1. Произведено ознакомление с предметной областью поиска зависимостей порядка, после чего написан обзор. Обосновано решение об оптимизации алгоритма ORDER.

2. Произведен рефакторинг и профилирование базовой версии алгоритма в качестве подготовки к оптимизации.

3. Предложены оптимизации: как на основе структур данных и алгоритмов из библиотеки Boost, так и с применением многопоточности.

4. Произведена интеграция в Desbordante, путем добавления привязки к Python и поддержки CLI запуска. Также написан пример использования на языке Python, дающий интуитивное понимание зависимостей порядка и принципа работы алгоритма.

5. Было произведено сравнение с реализацией в Metanome, базовая реализация ORDER показала увеличение производительности до 9 раз, версия с оптимизациями до 25 раз.

Алгоритм интегрирован в проект Desbordante (pull request 294[7]). Часть предложенных подходов к оптимизации опубликована в статье “Order in Desbordante: Techniques for Efficient Implementation of Order Dependency Discovery Algorithms” [14], представленной на 35-ой конференции ассоциации открытых инноваций FRUCT, проводившейся в Финляндии (г. Тампере). Ее работы публикуются IEEE и индексируются в Scopus.

[7] https://github.com/Desbordante/desbordante-core/pull/294

Список литературы

[1] Abedjan Ziawasch, Golab Lukasz, Naumann Felix. Data profiling // 2016 IEEE 32nd International Conference on Data Engineering (ICDE). — 2016. — P. 1432–1435.

[2] Aggarwal Charu C., Han Jiawei. Frequent Pattern Mining. — Springer Publishing Company, Incorporated, 2014. — ISBN: 3319078208.

[3] Approximate Order Dependency Discovery / Yifeng Jin, Zijing Tan, Weijun Zeng, Shuai Ma // 2021 IEEE 37th International Conference on Data Engineering (ICDE). — 2021. — P. 25–36.

[4] Brown Paul G., Hass Peter J. BHUNT: Automatic Discovery of Fuzzy Algebraic Constraints in Relational Data // Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29. — VLDB '03. — VLDB Endowment, 2003. — P. 668–679.

[5] Data Profiling with Metanome / Thorsten Papenbrock, Tanja Bergmann, Moritz Finke et al. // Proc. VLDB Endow. — 2015. — aug. — Vol. 8, no. 12. — P. 1860–1863. — URL: https://doi.org/10.14778/2824032.2824086.

[6] Desbordante: a Framework for Exploring Limits of Dependency Discovery Algorithms / Maxim Strutovskiy, Nikita Bobrov, Kirill Smirnov, George Chernishev // 2021 29th Conference of Open Innovations Association (FRUCT). — 2021. — P. 344–354.

[7] Dong Jirun, Hull Richard. Applying Approximate Order Dependency to Reduce Indexing Space // Proceedings of the 12th ACM SIGMOD International Conference on Management of Data. — ACM Press, 1982. — P. 119–127.

[8] Effective and Complete Discovery of Order Dependencies via Set-Based Axiomatization / Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab et al. // Proc. VLDB Endow. — 2017. — mar. — Vol. 10, no. 7. — P. 721–732. — URL: https://doi.org/10.14778/3067421.3067422.

[9] Expressiveness and Complexity of Order Dependencies / Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, Calisto Zuzarte // Proc. VLDB Endow. — 2013. — sep. — Vol. 6, no. 14. — P. 1858–1869. — URL: https://doi.org/10.14778/2556549.2556568.

[10] Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms / Thorsten Papenbrock, Jens Ehrlich, Jannik Marten et al. // Proc. VLDB Endow. — 2015. — . — Vol. 8, no. 10. — P. 1082–1093. — URL: https://doi.org/10.14778/2794367.2794377.

[11] Ginsburg Seymour, Hull Richard. Order dependency in the relational model // Theoretical Computer Science. — 1983. — Vol. 26, no. 1. — P. 149–195. — URL: https://www.sciencedirect.com/science/article/pii/0304397583900841.

[12] Ginsburg Seymour, Hull Richard. Sort Sets in the Relational Model // J. ACM. — 1986. — may. — Vol. 33, no. 3. — P. 465–488. — URL: https://doi.org/10.1145/5925.5929.

[13] Langer Philipp, Naumann Felix. Efficient order dependency detection // The VLDB Journal. — 2016. — 04. — Vol. 25.

[14] Order in Desbordante: Techniques for Efficient Implementation of Order Dependency Discovery Algorithms / Yakov Kuzin, Dmitriy Shcheka, Michael Polyntsov et al. // 2024 35th Conference of Open Innovations Association (FRUCT). — 2024. — P. 413–424. — Paper accepted.

[15] Tane: An Efficient Algorithm for Discovering Functional and Approximate Dependencies / Yká Huhtala, Juha Kärkkäinen, Pasi Porkka, Hannu Toivonen // The Computer Journal. — 1999. — Vol. 42, no. 2. — P. 100–111.

Таблица 7: Описание наборов данных.

Набор ланных	#Колонок	#Строк	Размер (MB)	#OD
diabetes_binary_BRFSS2021.csv	22	236379	17.0	0
PFW_2021_public.csv	22	100001	14.7	17
DITAG.csv	5	4339917	299.0	0
creditcard_2023.csv	31	568631	324.8	0
EpicMeds.csv	10	1281732	56.8	9
modis_2000-2019_Australia.csv	15	5081220	410.4	7
bay_wheels_data_wrangled.csv	8	5022834	660.1	0

Таблица 8: Общие результаты (время в секундах)

Набор данных	Java	C++ (база)	C++ (опт.)	Улучшение (база vs Java)	Улучшение (опт. vs база)	Улучшение (опт. vs Java)
Diabetes	3.331	3.696	0.586	0.901x	6.307x	5.684x
Pfw	1.361	0.472	0.201	2.883x	2.348x	6.771x
Ditag	16.833	7.219	3.111	2.331x	2.320x	5.410x
Credit	32.585	5.279	3.097	6.172x	1.704x	10.521x
Epic	8.649	3.438	1.508	2.515x	2.279x	5.735x
Modis	88.069	9.339	3.407	9.430x	2.741x	25.849x
Bay	33.158	18.466	6.472	1.795x	2.853x	5.123x