Санкт-Петербургский государственный университет

Кузин Яков Сергеевич

Выпускная квалификационная работа

Методы реализации поиска зависимостей порядка на основе формализации множеств

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование информационных систем»*

Основная образовательная программа *СВ.5162.2020 «Технологии программирования»*

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

Рецензент:
старший преподаватель, «Санкт-Петербургский государственный университет»
Смирнов К. К.

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

Санкт-Петербург
2024

Saint Petersburg State University

Yakov Kuzin

Bachelor's Thesis

Implementing order dependency discovery algorithm based on set-based axiomatization

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5162.2020 "Programming Technologies"*

Scientific supervisor:
C.Sc, docent E.G. Mikhailova

Reviewer:
position at "St. Petersburg State University" K.K. Smirnov

Consultant:
position at "St. Petersburg State University" G.A. Chernishev

Saint Petersburg
2024

Оглавление

Введение

У людей, работающих с данными, часто возникает задача профилирования этих данных. Им бывает полезна вспомогательная информация, которая раскрывает те или иные свойства обрабатываемых материалов. Например, это может быть дата создания и изменения файла или его заголовок и имя владельца. Подобная информация о данных называется метаданными, а процесс ее извлечения — профилированием данных [1].

Данные могут содержать разнообразные закономерности, которые можно найти при помощи различных алгоритмов и которые могут быть полезны многим специалистам [22]. Например, наличие закономерностей может служить фундаментом для создания гипотез и задания направлений новых исследований, что в первую очередь полезно ученым. Кроме того, они применяются в области очистки данных, оказывая существенное влияние на раскрытие ценности этих данных [21].

Подобные алгоритмы входят в состав разных инструментов для профилирования данных [2, 8, 15, 17]. Примером подобной программы может служить Metanome[1]. Согласно [9], этот проект, в частности, полезен аналитикам, поскольку он может быть использован в исследованиях, связанных с пониманием данных и получением заложенных в них знаний. Desbordante[2] служит другим примером инструмента, используемым в области профилирования данных. Он тоже может быть полезен аналитикам, но как и Metanome имеет намного более широкую аудиторию пользователей. Например, его могут использовать инженеры в области данных для обработки информации, полученной экспериментальным путем [33].

В данной работе будет рассмотрен алгоритм FASTOD [12] для выявления зависимостей порядка и предложена его эффективная реализация. Также будет предложен ряд следующих оптимизаций, описанных в разделе 3.5 и позволяющих добиться приемлемой для практического

[1]https://hpi.de/naumann/projects/data-profiling-and-analytics/metanome-data-profiling.html
[2]https://github.com/Desbordante/desbordante-core

использования скорости обработки входных данных:

1. Хранение данных в виде списка диапазонов.
2. Рациональный выбор представления партиции.
3. Автоматическая смена представления партиции.
4. Специальный алгоритм произведения партиций.

Эта реализация вошла в open-source проект Desbordante — высокопроизводительный и наукоемкий инструмент для профилирования данных, включающий в себя множество алгоритмов для обнаружения и проверки различных закономерностей. Она добавила новый поддерживаемый тип зависимостей, а именно зависимости порядка.

1. Постановка задачи

Целью настоящей работы является разработка эффективной реализации алгоритма FASTOD для поиска зависимостей порядка и выработка различных оптимизаций, позволяющих ускорить его работу на специальных наборах данных. Для ее достижения были поставлены следующие задачи:

1. Ознакомиться с предметной областью поиска зависимостей порядка. По результатам ознакомления написать обзор, выбрать алгоритм выделения этих зависимостей и обосновать данный выбор.

2. Реализовать базовую версию алгоритма FASTOD и добиться ускорения по сравнению с имеющейся Java реализацией.

3. Предложить оптимизации этой реализации, позволяющие добиться еще большего ускорения на специальных наборах данных.

4. Интегрировать алгоритм в проект Desbordante и добавить в него новый тип зависимостей (зависимости порядка). Создать Python привязки, позволяющие вызывать алгоритм из программ на языке Python, расширить CLI часть проекта и составить примеры, описывающие сущность зависимостей порядка, а также то, как ими можно пользоваться.

5. Проверить эффективность обеих реализаций как на обычных наборах данных, так и на имеющих специальный вид.

2. Обзор

В настоящее время имеется высокий интерес к анализу больших объемов данных. Эти данные зачастую имеют различные искажения, такие как пропущенные значения, дубликаты, а также некоторые другие несоответствия [6].

Вместе с увеличением объема анализируемых данных возрастает потребность в исследовании качества этих данных, а также различных оптимизаций алгоритмов, которые их обрабатывают.

Данная работа посвящена зависимостям порядка (Order Dependencies), так как это достаточно новый и перспективный тип зависимостей, вокруг которого в настоящее время ведется много исследований. Эти зависимости оказываются полезным во многих областях [12, 24], и потому их поддержка была необходима проекту Desbordante.

2.1. Зависимости порядка

Согласно [31], зависимости порядка (Order Dependencies, OD) впервые были представлены в 80-х годах в работе S. Ginsburg и R. Hull [20]. С того времени появилось достаточно много новых типов зависимостей. Более того, появилось разделение зависимостей на две группы согласно их формализации. В первой группе зависимости формализуются множествами атрибутов (Set-Based Axiomatization), а во второй списками (List-Based Axiomatization). Лексикографические (Lexicographical OD) и поточечные зависимости (Point-Wise OD) порядка служат примером зависимостей с разной формализацией: первые определяются на списках атрибутов, а вторые на множествах. Поточечные зависимости появились раньше и были предложены в работе “Order Dependency in the Relational Model” [20], в то время как лексикографические были представлены W. Ng в статье “An Extension of the Relational Data Model to Incorporate Ordered Domains” [26] в качестве более узкой и полезной в области баз данных альтернативы.

Частным случаем зависимостей порядка являются функциональные зависимости, вокруг которых было произведено множество исследова-

ний [14, 19, 25]. В отличие от первых, они не могут определить многие полезные связи между упорядоченными атрибутами, встречаемые в бизнес-данных [5], однако находят широкое применение во многих областях. Например, согласно [4, 11, 34], они применялись в обратном проектировании данных (data reverse engineering), оптимизации запросов и очистке данных. Интересным фактом является то, что зависимости порядка включают в себя функциональные зависимости, но при этом их поиск оказывается не сложнее (в алгоритмическом смысле) поиска вторых [12].

Другим примером специфичной формы зависимостей порядка служат зависимости совместимости упорядоченности (order compatibility dependencies), исследованные в работах [10] и [13]. В статье "Discovering Order Dependencies through Order Compatibility" [7] авторы использовали идею разделения зависимостей порядка на зависимости совместимости упорядоченности и функциональные зависимости, что позволило им добиться хороших результатов, однако это привело к потере некоторых зависимостей [16].

Примером более общей формы зависимостей порядка могут служить двусторонние зависимости порядка (bidirectional order dependencies), вокруг которых было проведено много исследований [13, 23, 29, 30]. Большой интерес к этому типу зависимостей наблюдается из-за возможности их использования в оптимизации запросов к базам данных, содержащих конструкцию ORDER-BY, а также симулирования подобных запросов [23]. В статьях [13] и [23] были представлены алгоритмы поиска двусторонних зависимости порядка, а в работах [29] и [30] предложены алгоритмы их распределенного поиска.

Наконец, Jaroslaw Szlichta и др. [12] утверждают, что зависимости порядка хорошо подходят для улучшения качества данных: их нарушение может свидетельствовать о том, что в данных содержатся ошибки. Кроме того, согласно [18], эти зависимости могут быть использованы различными оптимизаторами СУБД для оптимизации запросов.

Сами зависимости могут быть формализованы при помощи математических определений, которые изложены ниже и взяты из ста-

тей [12, 24].

Определение 2.1. *Через $s[X]$ для данного множества атрибутов X обозначим набор, состоящий из значений кортежа s на X.*

Определение 2.2. *Пусть мы имеем схему отношения R и списки атрибутов $X, Y \in R$. Зависимость порядка $X \to \theta Y$, где $\theta \in \{\leq, <, =, >, \geq\}$, имеет место в экземпляре r над отношением R если, и только если для любых двух кортежей $s, t \in r$ выполнено $s[X]\theta t[X] \Rightarrow s[Y]\theta t[Y]$. Если θ опускается, подразумевается, что $\theta = <$.*

Определение 2.3. *Два кортежа s, t эквивалентны относительно данного множества атрибутов X, если $s[X] = t[X]$.*

Определение 2.4. *Пусть R — схема отношения, r — ее экземпляр. Классом эквивалентности (equivalence class) кортежа $t \in r$ относительно данного множества атрибутов X называется множество $E(t_X) = \{s \in r \mid s[X] = t[X]\}$.*

Определение 2.5. *Пусть R — схема отношения, r — ее экземпляр. Партицией (partition) Π_X относительно данного множества атрибутов X называется множество классов эквивалентности $\Pi_X = \{E(t_X) \mid t \in r\}$.*

Определение 2.6. *Сокращенной партицией (stripped partition) Π^*_X называется партиция, из которой исключены классы эквивалентности, состоящие из одного элемента.*

Определение 2.7. *Атрибут A является константой внутри каждого класса эквивалентности (constant within each equivalence class) относительно множества атрибутов X (эта зависимость обозначается через $X : [] \to_{cst} A$), если имеется зависимость порядка $X' \to X'A$ для любой перестановки X' элементов множества X.*

Определение 2.8. *Два атрибута A, B совместимы по порядку в каждом классе эквивалентности (order-compatible within each equivalence class) по отношению к множеству атрибутов X (эта зависимость*

обозначается через $X : A \sim B$*), если имеется зависимость порядка* $X'A \to X'B$*.*

Определение 2.9. *Зависимости вида* $X : [] \to_{cst} A$ *и* $X : A \sim B$ *называются каноническими зависимостями порядка (canonical order dependencies). Множество атрибутов* X *при этом называют контекстом (context).*

2.2. Алгоритмы поиска зависимостей порядка

По итогу совершенного научного поиска можно утверждать, что на данный момент имеется два алгоритма поиска точных зависимостей порядка: FASTOD и ORDER [24].

ORDER имеет факториальную временную сложность по количеству атрибутов в худшем случае, при этом данный алгоритм работает с зависимостями порядка, в основе которых лежат списки атрибутов. Согласно [12], для уменьшения своей времени работы ORDER намеренно опускает некоторые зависимости во время процедуры отсечения уровней (pruning strategies). Кроме того, некоторые найденные зависимости оказываются избыточными относительно зависимостей в канонической форме, которые можно найти при помощи алгоритма FASTOD.

Алгоритм FASTOD, напротив, работает с каноническими зависимостями порядка, в основе которых лежат множества атрибутов. Его идея заключается в полиномиальном отображении зависимостей порядка в их каноническую форму, что позволяет добиться более высокой производительности: алгоритм имеет экспоненциальную временную сложность по количеству атрибутов в худшем случае. Кроме того, FASTOD эффективно сужает пространство поиска зависимостей без потери полноты результирующего набора.

Отличие в формализации зависимостей играет большую роль. В статье [24] утверждается, что факториальная временная сложность по числу атрибутов алгоритма ORDER, работающего со списковой формализацией, неизбежна. Это происходит из-за возникновения огромного пространства поиска кандидатов, поскольку зависимости порядка опре-

деляются через списки атрибутов. В то же время в статье [12] утверждается, что алгоритм FASTOD, работающий с формализацией на основе множеств, имеет в худшем случае экспоненциальную временную сложность по числу атрибутов. Это становится возможно за счет существования отображения зависимостей порядка, формализуемых списками, в эквивалентные зависимости порядка, формализуемые множествами.

2.3. Выводы

В результате ознакомления с предметной областью было принято решение сконцентрироваться на канонических зависимостях порядка — достаточно новом типе зависимостей порядка, имеющим большое количество приложений, полезных многим специалистам в различных сферах. Этот выбор также мотивирован существованием полиномиального отображения [12] зависимостей порядка в их каноническое представление, которое позволяет добиться высокопроизводительного поиска полного набора соответствующих зависимостей.

Среди существующих алгоритмов выделения зависимостей порядка был выбран алгоритм FASTOD. Данный выбор обосновывается тем, что:

- FASTOD может достичь хорошей производительности за счет формализации, в основе которой лежат множества.
- Полнота набора зависимостей, которые находит FASTOD, выше, чем у современной альтернативы — алгоритма ORDER.

В результате анализа существующих решений поиска многих типов зависимостей было выявлено, что используемые в них методы направлены на решение соответствующей задачи исключительно с алгоритмической точки зрения. Однако эти задачи очень сложные с вычислительной стороны. Это приводит к появлению задач создания эффективных реализаций алгоритмов, которые смогут найти себя в промышленном использовании.

3. Реализация

Сначала опишем сам алгоритм FASTOD, а потом перейдем к предлагаемым в настоящей работе оптимизациям. FASTOD представляет из себя алгоритм обхода решетки. Он начинает поиск с единичного набора атрибутов, после чего продвигается к более крупным. Это продвижение идет через решетку включения множества поэтапно, что в результате дает разбиение работы алгоритма на уровни. Подобная стратегия является классической. Согласно [19], она и ее модификации применялась во многих алгоритмах, таких как DFD [3], FUN [27], TANE [32], FD_MINE [35] и других.

На каждом таком уровне происходит проверка кандидатов зависимостей на минимальность, после чего они превращаются в результирующие зависимости, если эта проверка прошла. При этом вместо самих кандидатов используется и сохраняется лишь вспомогательная информация о них. Она заложена в множества:

- $C_c^+(X) = \{A \in R \mid \forall_{B \in X} X \backslash \{A, B\} : [] \rightarrow_{cst} B$ не выполняется$\}$
- $C_s^+(X) = \{\{A, B\} \in X^2 \mid A \neq B$ и $\forall_{C \in X} X \backslash \{A, B, C\} : A \sim B$ не выполняется, и $\forall_{C \in X} X \backslash \{A, B, C\} : [] \rightarrow_{cst} C$ не выполняется$\}$

В дальнейшем из этих множеств могут быть получены сами кандидаты. Под R подразумевается исходное отношение. Определения и нотация взяты из статьи [12].

Таким образом, FASTOD использует стратегию поиска зависимостей от малого к большему. Она позволяет находить минимальные зависимости порядка (то есть те, которые нельзя вывести из других), а также сокращать пространство поиска новых зависимостей. Подобное сокращение можно наблюдать на решетке, изображенной на рисунке 1. Она состоит из узлов, каждый из которых соответствует контексту для возможной зависимости порядка. Пунктирной рамкой в решетке отображены узлы, которые были отброшены в результате процесса отсечения уровней для следующей ситуации. Пусть имеется зависимость $\{B\} : [] \rightarrow_{cst} A$ и множество атрибутов $X = \{A, B, C\}$. Тогда

$A \notin C_c^+(X \backslash C)$, а значит зависимость $\{B, C\} : [] \rightarrow_{cst} A$ не является минимальной и может быть отброшена.

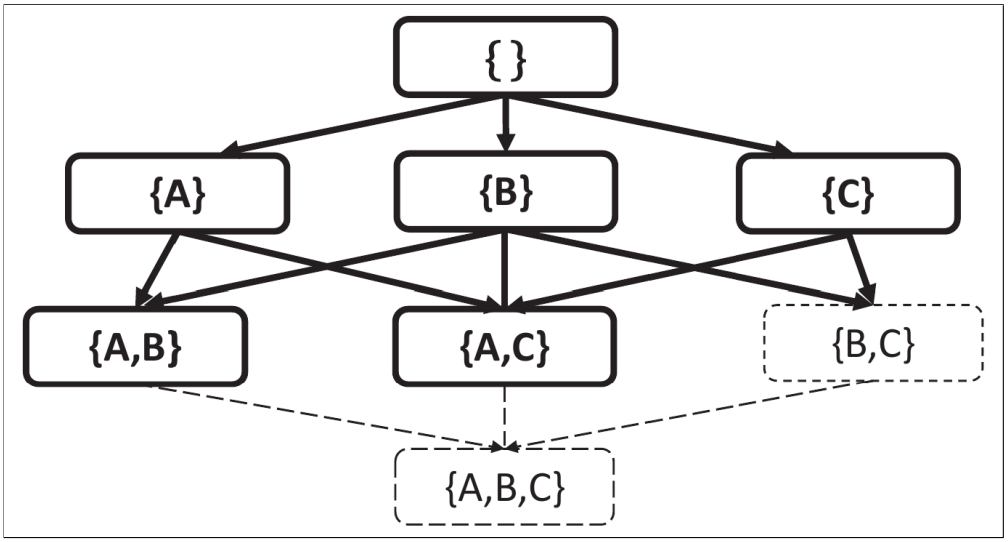

Рис. 1: Решетка для атрибутов A, B, C (изображение взято из исходной работы [12])

Далее будут описаны ключевые этапы работы алгоритма FASTOD, а в разделах 3.5 и 3.6 будут изложены собственные разработки автора данной работы. А именно интеграция алгоритма в проект Desbordante и ряд оптимизаций этого алгоритма.

3.1. Поиск зависимостей

Сначала производится заполнение кандидатов, описанное алгоритмом 1, который взят из оригинальной статьи [12].

Следующим этапом служит поиск зависимостей, описанный алгоритмом 2, который также взят из статьи [12]. В первом вложенном цикле идет поиск зависимостей вида $X : [] \rightarrow_{cst} A$, а во втором — вида $X : A \sim B$.

Алгоритм 1: Заполнение кандидатов

Data: L_i — текущий уровень, R — исходное отношение

for $X \in L_i$ **do**
  $C_c^+(X) = \cap_{A \in X} C_c^+(X \backslash A)$
  if $i = 2$ **then**
    $\forall_{A,B \in R^2, A \neq B}\ C_s^+(\{A, B\}) = \{A, B\}$
  else if $i > 2$ **then**
    $C_s^+(X) = \{\{A, B\} \in \cup_{C \in X} C_s^+(X \backslash C) \mid$
     $\forall_{D \in X \backslash \{A,B\}} \{A, B\} \in C_s^+(X \backslash D)\}$
  end
end

Алгоритм 2: Поиск зависимостей

Data: L_i — текущий уровень, R — исходное отношение, N — результирующий набор зависимостей

for $X \in L_i$ **do**
  for $A \in X \cap C_c^+(X)$ **do**
    if $X \backslash A : [] \rightarrow_{cst} A$ **then**
      Add $X \backslash A : [] \rightarrow_{cst} A$ to M
      Remove A from $C_c^+(X)$
      Remove all $B \in R \backslash X$ from $C_c^+(X)$
    end
  end
  for $\{A, B\} \in C_s^+(X)$ **do**
    if $A \notin C_c^+(X \backslash B)$ *or* $B \notin C_c^+(X \backslash B)$ **then**
      Remove $\{A, B\}$ from $C_s^+(X)$
    else if $X \backslash \{A, B\} : A \sim B$ **then**
      Add $X \backslash \{A, B\} : A \sim B$ to M
      Remove $\{A, B\}$ from $C_s^+(X)$
    end
  end
end

3.2. Отсечение уровней

Отсечение уровня (level pruning) представляет собой удаление на нем множеств атрибутов, являющихся базовыми множествами для кандидатов. Для этого используются знания, накопленные во время поиска зависимостей на текущем уровне. Эта очистка обеспечивает дополнительную оптимизацию и реализуется по следующему правилу: множество атрибутов X удаляется из текущего уровня, если оба множества $C_c^+(X)$ и $C_s^+(X)$ являются пустыми. При этом удаление происходит на всех уровнях, кроме первого.

3.3. Вычисление следующего уровня

Каждый уровень вычисляется через предыдущий, при этом первый включает в себя все атрибуты из исходного отношения. Вычисление уровней описано в алгоритме 3, который также взят из статьи [12]. В нем используется подпрограмма singleAttrDiffBlocks(L_i), которая разбивает текущий i-ый уровень на блоки: два множества атрибутов принадлежат одному блоку, если они отличаются только одним атрибутом и имеют общее подмножество мощности $(i-1)$. То есть блоки имеют вид $Y \cup K$ и $Y \cup N$.

Алгоритм 3: Вычисление следующего уровня

```
   Data: L_i — текущий уровень, L_{i+1} — следующий уровень
1  L_{i+1} = []
2  for {Y ∪ K, Y ∪ N} ∈ singleAttrDiffBlocks(L_i) do
3  |   X = Y ∪ {K, N}
4  |   if ∀_{A∈X} X\A ∈ L_i then
5  |   |   Add X to L_{i+1}
6  |   end
7  end
```

3.4. Описание алгоритма через блоки

Собрав все вышеизложенное воедино, алгоритм может быть описан следующими блоками:

1. Первый уровень включает в себя все атрибуты из исходного отношения, он же является текущим на первой итерации.
2. Идет выполнение пунктов 3–5 до того момента, пока текущий уровень не станет пустым.
3. На текущем уровне идет поиск всех зависимостей.
4. Сокращается пространство поиска зависимостей за счет отсечения текущего уровня.
5. Вычисляется следующий уровень, который впоследствии становится текущим.

3.5. Оптимизации алгоритма

В результате исследования многих наборов данных было выявлено, что многие из них содержат большое число колонок, состоящих из образующих диапазоны значений. Алгоритм FASTOD, в свою очередь, активно работает не с самими колонками, а с построенными по ним партициями (множествами классов эквивалентности). Именно они играют ключевую роль в работе алгоритма. FASTOD использует их особый вид — сокращенные партиции (stripped partitions). Их отличие от обычных заключаются в том, что они не содержат в себе классы эквивалентности единичной мощности. Все нижеизложенные оптимизации алгоритма будут связаны с этими партициями и являются собственными разработками автора данной работы (как и интеграция алгоритма со всеми его улучшениями в проект Desbordante).

3.5.1. Хранение данных в виде списка диапазонов

Стандартное представление партиций заключается в хранении списка индексов значений. Эти индексы разбиваются на группы, каждая из которых соответствует некоторому классу эквивалентности. Такое представление в некоторых случаях влечет за собой излишнее потребление памяти. Например, рассмотрим атрибут $A = (0, 0, 0, 5, 0, 0, 0)$. Его стандартное представление будет включать две группы индексов, образующих классы эквивалентности $[0] = (0, 1, 2, 4, 5, 6)$ и $[5] = (3)$. На этом примере наглядно видно, что последовательности индексов $(0, 1, 2)$ и $(4, 5, 6)$ образуют диапазоны. По мере увеличения числа строк в исходном наборе данных подобные последовательности могут разрастаться до огромных размеров. Их хранение не только увеличивает расход памяти, но и влечет за собой издержки времени выполнения при копировании представления.

Первая оптимизация решает эти проблемы за счет иного представления партиций. Оно подразумевает хранение индексов в виде диапазонов. Возвращаясь к предыдущему примеру, в новом представлении классы эквивалентности имеют следующий вид: $[0] = ([0–2], [4–6])$ и $[5] = ([5–5])$.

3.5.2. Рациональный выбор представления партиции

Представление партиций в виде списка диапазонов индексов оказывается эффективным только в том случае, когда атрибут содержит большое число подряд идущих одинаковых значений. В остальных случаях наблюдается обратный эффект: маленькие (а порой даже вырожденные) диапазоны начинают увеличивать не только время работы алгоритма, но и расход памяти.

Вторая оптимизация направлена на решение этой проблемы. Она заключается в рациональном выборе типа первоначальной партиции: в подходящих случаях выбирается новое, а в остальных стандартное. Этот выбор основывается на информации, полученной в результате предварительного анализа исходных данных: каждая колонка прове-

ряется на наличие в ней большого числа подряд идущих одинаковых значений. Если эта колонка удовлетворяет данному требованию, она помечается как подходящая для нового представления партиций.

3.5.3. Автоматическая смена представления партиции

В ходе работы алгоритма диапазоны, служащие основой для нового представления партиций, не увеличиваются. Более того, они, как правило, постепенно сужаются и в итоге могут превратиться в вырожденные диапазоны. Поэтому начиная с определенного момента новое представление партиций теряет свою эффективность. Появляется большое число маленьких диапазонов, замедляющих работу алгоритма и влекущих излишнее потребление памяти.

Третья оптимизация решает эту проблему. Ее идея заключается в динамической смене представления партиций. Как только число маленьких диапазонов по отношению к числу больших превысит некоторый порог, новое представление партиций преобразуется в стандартное. Эта операция требует небольших временных затрат, однако предотвращает замедление времени исполнения и чрезмерное потребление памяти в будущем. Подобная смена типа представления позволяет получить значительный прирост производительности на первых этапах работы с партициями. Кроме того, она обеспечивает отсутствие замедления после того, как новое представление перестанет давать подобный эффект.

3.5.4. Специальный алгоритм произведения партиций

Новое представление партиций позволяет добиться значительного прироста эффективности алгоритма. Однако его специальный вид плохо подходит для стандартного алгоритма произведения партиций. Поэтому было принято решение разработать другой алгоритм, оптимизированный под данный вид представления. Основная идея этого алгоритма заключается в быстром пересечении диапазонов и пропуска пересечений, которые не дадут никакого результата.

Для каждого атрибута строится особый список. Он состоит из пар

“значение-диапазон”. Диапазон в этой паре несет в себе информацию о том, где располагаются подряд идущие значения, соответствующие первому элементу данной пары. Этот список сортируется по диапазонам, в результате чего получается список, который был назван S. Далее для каждого такого списка создается таблица соответствий T. Она отображает каждый элемент из каждого содержащегося в списке S диапазона в индекс, соответствующий позиции диапазона в этой коллекции. Описанные выше структуры позволяют избежать большого числа ненужных пересечений диапазонов.

Рассмотрим следующий пример: пусть мы хотим пересечь некоторый диапазон d с упорядоченным списком S. Наивный подход подразумевает пересечение d со всеми диапазонами, содержащимися в S. Вместо этого мы сначала находим два индекса. Первый из них указывает на диапазон, содержащий левый конец d, а второй на диапазон, содержащий правый конец d. После чего мы находим диапазоны, чьи индексы лежат между найденными индексами, и пересекаем d только с ними. Получается набор диапазонов, каждому элементу которого сопоставляется то же значение, что сопоставлялось соответствующему диапазону из списка S. Это и дает результат пересечения диапазона d с S. Аналогичным образом происходит пересечение произвольного списка диапазонов L с упорядоченным списком S: происходит пересечение каждого диапазона из L с S и дальнейшее объединение результатов в одну структуру.

Подобная стратегия пересечения хорошо применима к новому представлению партиций. Оно может быть представлено в виде списка пар “значение-диапазон” путем сопоставления каждому диапазону индексов соответствующего ему значения. А значит мы можем пересекать произвольную партицию, представленную в виде списка диапазонов, с заранее построенной и отображенной в специальный вид партицией.

В качестве поясняющего примера рассмотрим следующую ситуацию. Пусть мы имеет атрибут $Q = (0, 0, 1, 1, 0, 0, 2)$, построенную по нему партицию Π_Q и ее представление в виде пар “значение-диапазон” $L_Q = \{(0, [0–1]); (0, [4–5]); (1, [2–3]); (2, [6–6])\}$. Упорядоченным по диапазонам

списком будет служить $S_Q = \{(0, [0–1]); (1, [2–3]); (0, [4–5]); (2, [6–6])\}$, а таблицей соответствий будет служить $T_Q = \{(0 \to 0); (1 \to 0); (2 \to 1); (3 \to 1); (4 \to 2); (5 \to 2); (6 \to 3)\}$. Пусть мы имеем партицию Π_W в новом представлении $\{[0–1], [5–6], [2–4]\}$. Первый класс эквивалентности K_1 образован диапазонами [0–1], [5–6], а второй K_2 диапазоном [2–4]. Для вычисления произведения $\Pi_{W \cup Q} = \Pi_W * \Pi_Q$ необходимо пересечь каждый класс эквивалентности из Π_W с построенным по атрибуту Q упорядоченным списком S_Q.

Сначала пересечем класс K_1, состоящий из диапазонов $r_1 = [0–1]$ и $r_2 = [5–6]$, с S_Q. Для этого пересекаем сначала r_1 с S_Q. Вычисляем индексы диапазонов, содержащие левый и правый края r_1 соответственно: $T_Q[0] = 0$, $T_Q[1] = 0$. Получили единственный индекс 0, который указывает на диапазон [0–1]. Его пересечение с r_1 дает $[0–1] \cap [0–1] = [0–1]$. Этому диапазону будет сопоставлено то же значение, что соответствовало диапазону [0–1] в списке S_Q, то есть 0. Аналогичные действия производим для r_2: $T_Q[5] = 2$, $T_Q[6] = 3$; $[4–5] \cap [5–6] = [5–5]$, $[6–6] \cap [5–6] = [6–6]$. Полученным в результате пересечений диапазонам будут сопоставлены значения 0 и 2 соответственно. Таким образом, $N_1 = K_1 \cap S_Q = (r_1 \cap S_Q) \cup (r_2 \cap S_Q) = \{(0, [0–1]); (0, [5–5]); (2, [6–6])\}$. Вышеизложенный процесс пересечения класса K_1 с S_Q наглядно изображен на рисунке 2.

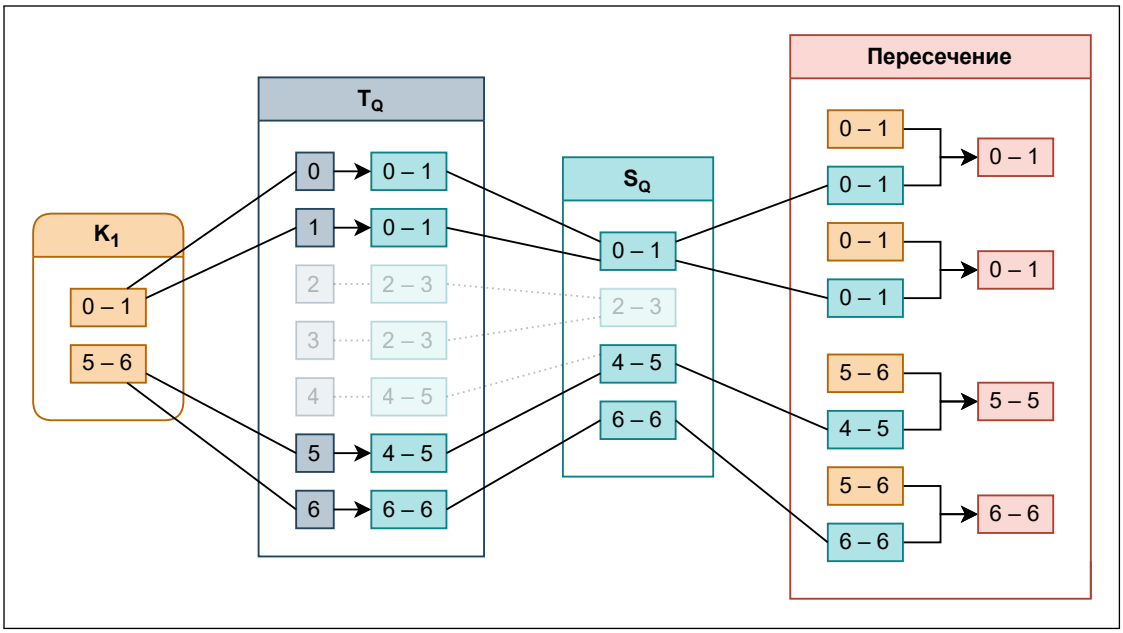

Рис. 2: Пересечение класса K_1 с отсортированным списком S_Q

Алгоритм 4: Произведение партиций

Data: T_Q — таблица соответствий для атрибута Q, Π_W — первая партиция, S_Q — отсортированный список, построенный по второй партиции Π_Q

Result: P — партиция $\Pi_{W \cup Q}$

```
P ← ∅;
for K ∈ Π_W do
    N ← ∅;
    for [d_b; d_l] ∈ K do
        b ← T_Q[d_b];
        l ← T_Q[d_l];
        G ← S_Q[b ... l];
        for (v, [r_b; r_l]) ∈ G do
            r ← [d_b; d_l] ∩ [r_b; r_l];
            Add (v, r) to N;
        end
    end
    Sort N;
    D ← Extract equivalence classes from N;
    Exclude equivalence classes with cardinality of 1 from D;
    Add each class from D to P;
end
return O;
```

Аналогичным образом вычисляем пересечение второго класса эквивалентности K_2, состоящего из единственного диапазона $r_3 = [2\text{–}4]$, с S_Q. $T_Q[2] = 1$, $T_Q[4] = 2$; $[2\text{–}3] \cap [2\text{–}4] = [2\text{–}3]$, $[4\text{–}5] \cap [2\text{–}4] = [4\text{–}4]$; $N_2 = K_2 \cap S_Q = r_3 \cap S_Q = \{(1, [2\text{–}3]); (0, [4\text{–}4])\}$.

Следующим этапом служит сортировка по первому элементу пары результатов пересечения классов эквивалентности с S_Q и выделение из полученных упорядоченных наборов классов эквивалентности. Проведение данных шагов с N_1 дает два класса эквивалентности, представление которых через индексы имеет вид $([0\text{–}1], [5\text{–}5])$ и $([6\text{–}6])$ соответственно. В случае с N_2 имеем два класса, которые могут быть представлены как $([4\text{–}4])$ и $([2\text{–}3])$. Далее идет исключение классов эквивалентности единичной мощности. В результате в результирующий список классов попадают лишь $([0\text{–}1], [5\text{–}5])$ и $([2\text{–}3])$. Этот список и является результа-

том произведения партиций.

Весь этот процесс также изложен в алгоритме 4 в более общей и компактной форме.

3.6. Интеграция в проект Desbordante

Алгоритм FASTOD со всеми его оптимизациями был интегрирован в Desbordante. Для этого были добавлены все необходимые структуры в ядро проекта, а также созданы различные Python привязки[3], позволяющие вызывать алгоритм из программ на языке Python. В частности, это позволило автору добавить алгоритм в CLI часть проекта, что, в свою очередь, дало возможность работать с этим алгоритмом из терминала. Кроме того, были составлены примеры[4], в которых подробно описывается сущность зависимостей порядка, а также то, как ими можно пользоваться.

В листинге 1 изложен процесс вызова алгоритма из программы на языке Python. В строках 3–4 указываются входной набор данных и максимальное время ожидания результирующего набора зависимостей. Строки 6–7 служат для инициализации алгоритма и его запуска, а 10–18 для получения зависимостей и вывода их в терминал.

[3]https://github.com/Desbordante/desbordante-core/tree/main/src/python_bindings/od
[4]https://github.com/Desbordante/desbordante-core/tree/main/examples

Листинг 1: Пример вызова алгоритма в Python

```
import desbordante

TABLE = 'salary.csv'
TIME_LIMIT_SECONDS = 3

algo = desbordante.od.algorithms.Fastod()
algo.load_data(table=(TABLE, ',', True))
algo.execute(time_limit=TIME_LIMIT_SECONDS)

asc_ods = algo.get_asc_ods()
desc_ods = algo.get_desc_ods()
simple_ods = algo.get_simple_ods()
all_ods = asc_ods + desc_ods + simple_ods

print('ods:', len(all_ods))

for od in all_ods:
    print(od)
```

4. Эксперименты

В результате проведения научного поиска была найдена реализация алгоритма FASTOD на языке Java, которая доступна по ссылке[5]. В экспериментах производилось ее сравнение с созданной автором данной работы реализацией того же алгоритма, а также оптимизированной версией этой реализации.

В ходе проведения экспериментов было выявлено, что время работы алгоритма не изменялось более чем на 1% для C++ реализаций и более чем на 3% для Java реализации в ходе каждого запуска алгоритма. Кроме того, запуск некоторых тестов занимал достаточно большое количество времени. В связи с этим было принято решение взять в качестве результата каждого эксперимента значение, полученное усреднением результатов трех запусков алгоритма на соответствующих наборах данных.

В ходе экспериментов не учитывалась фаза предварительной обработки данных. Этот этап зависит от скорости чтения и записи диска, а также анализатора и обработчика данных. Созданная реализация использует внутренние сервисы Desbordante, поэтому автор данной работы не пытался их оптимизировать. Кроме того, Java реализация учитывает только время работы самого алгоритма, то есть не учитывает этап подгрузки и обработки данных. Поэтому было принято решение сосредоточиться только на этапе работы самого алгоритма, а этап предварительной обработки данных оставить для дальнейших исследований.

Алгоритм реализован на C++ 17 и является частью open-source проекта Desbordante (pull request 355[6]). Весь код открыт для публичного просмотра и доступен по ссылке[7].

[5]https://github.com/leveretconey/cocoa

[6]https://github.com/Desbordante/desbordante-core/pull/355

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

4.1. Условия экспериментов

Оценка производительности проводилась на ПК со следующим аппаратным и программным обеспечением. Аппартаное обеспечение: Intel® Core™ i7-11800H CPU @ 2.30GHz (8 cores), 16GB DDR4 3200MHz RAM, 512GB SSD SAMSUNG MZVL2512HCJQ-00BL2. Программное обеспечение: Kubuntu 23.10, Kernel 6.5.0-14-generic (64-bit), gcc 13.2.0, 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).

4.2. Исходные данные

Найденная Java реализация алгоритма в отличие от созданной C++ реализации может работать только с наборами данных, имеющих строго определенную структуру и состоящими из целочисленных значений. В связи с этим исходные данные приводились к соответствующему виду, на котором уже проводилось тестирование. Помимо этого у определенного числа наборов данных удалялись некоторые столбцы и строки, чтобы добиться достаточно быстрого завершения работы алгоритма. Все исходные данные могут быть найдены по ссылке[8]. Они также описаны в таблице 1, в которой отражены основные характеристики набора данных, а также его короткое название.

4.3. Исследовательские вопросы

Для того, чтобы оценить качество предлагаемого решения, было необходимо понять следующее:

1. Насколько существенно алгоритм FASTOD, реализованный на C++, оказался эффективнее Java реализации.

2. Как сильно предложенные оптимизации базовой C++ версии оказали влияние на производительность алгоритма.

[8]https://github.com/Sched71/Desbordante-OD-Data

Таблица 1: Описание наборов данных

Набор данных	Короткое имя	#Колонки	#Строки	Размер (МБ)	#OD	#FD	#OCD
graduation_dataset_norm_15c.csv	G	15	4424	0.14	333	1	332
Anonymize_norm.csv	A	23	38480	4.28	115400	7774	107626
PFW_2021_public_norm.csv	P	18	100000	7.51	1240	48	1192
Spotify_Dataset_V3_norm.csv	S1	11	651936	35.96	1645	171	1474
diabetes_binary_norm.csv	D3	14	67136	2.18	0	0	0
spotify-2023_norm.csv	S2	20	953	0.06	198327	17915	180412
diabetes_binary2_norm.csv	D4	12	236378	6.99	0	0	0
Dataset_norm.csv	D1	15	175028	13.64	858	68	790
file.csv	F	20	52955	7.08	3134	70	3064
DOSE_V2_norm.csv	D2	16	46797	5.18	5912	592	5320
merged_data_norm.csv	M	4	11509051	276.06	0	0	0
Test_norm.csv	T	12	89786	3.17	326	12	314
openpowerlifting_norm.csv	O	13	386414	33.50	1079	19	1060
youtube_norm.csv	Y	12	161470	13.35	1752	96	1656
superstore_norm.csv	S3	20	51290	6.47	45916	2098	43818

Для этого были поставлены следующие исследовательские вопросы:

RQ1 Правда ли, что прямой перенос кода (без внесения существенных изменений) имеющейся Java реализации алгоритма на язык C++ принесет значительный прирост производительности?

RQ2 Насколько существенно предложенное представление партиций влияет на ускорение работы алгоритма на наборах данных с большим числом колонок с диапазонами?

RQ3 Снижает ли использование предложенного представления партиций эффективность алгоритма на обычных наборах данных? Каковы накладные расходы этого представления?

RQ4 Как изменяется эффективность оптимизированной C++ реализации алгоритма в зависимости от числа колонок в исходных данных?

RQ5 Насколько существенно уменьшение объема потребляемой памяти обоих C++ реализаций алгоритма по сравнению с Java реализацией?

4.4. Проведенные эксперименты

Для ответа на исследовательские вопросы было произведено 5 экспериментов (по одному на каждый вопрос с соответствующим номером).

4.4.1. Эксперимент 1

В первом эксперименте производилось сравнение базовой реализации алгоритма с Java реализацией. Его результаты отражены в таблице 3 и демонстрируют существенный прирост производительности: реализованная на C++ базовая версия алгоритма может быть быстрее Java реализации вплоть до 8 раз, при этом в среднем она оказывается быстрее в 4 раза.

4.4.2. Эксперимент 2

Второй эксперимент затрагивает специальные наборы данных с колонками, содержащими большое число диапазонов, и показывает возможное ускорение на них. Результаты эксперимента отражены в таблице 3 и демонстрируют весомый прирост производительности оптимизированной версии алгоритма относительно базовой: она оказывается быстрее в 1.3–1.8 раз.

4.4.3. Эксперимент 3

В третьем эксперименте производилось сравнение базовой реализации алгоритма с оптимизированной. Его результаты отражены в таблице 3 и демонстрируют тот факт, что время работы на обычных наборах данных оптимизированной версии алгоритма не только не увеличивается, но даже немного уменьшается. Это наблюдение показывает, что предложенные оптимизации не только не ухудшают эффективность алгоритма на обычных наборах данных, но и позволяют добиться весомого ускорения на специальных наборах данных.

4.4.4. Эксперимент 4

Четвертый эксперимент направлен на исследование вопроса зависимости времени работы алгоритма от числа колонок в исходных данных. Был выбран некоторый набор входных данных, включающий как обычные наборы данных, так и специальные (содержащие колонки с большим числом диапазонов). Из каждого входного набора было исключено соответствующее число колонок из его начала, в результате чего получались входные данные с необходимым числом колонок. Результаты эксперимента отражены на рисунках 3 и 4 и демонстрируют превосходство C++ реализаций алгоритма над Java реализацией независимо от числа колонок в исходных данных.

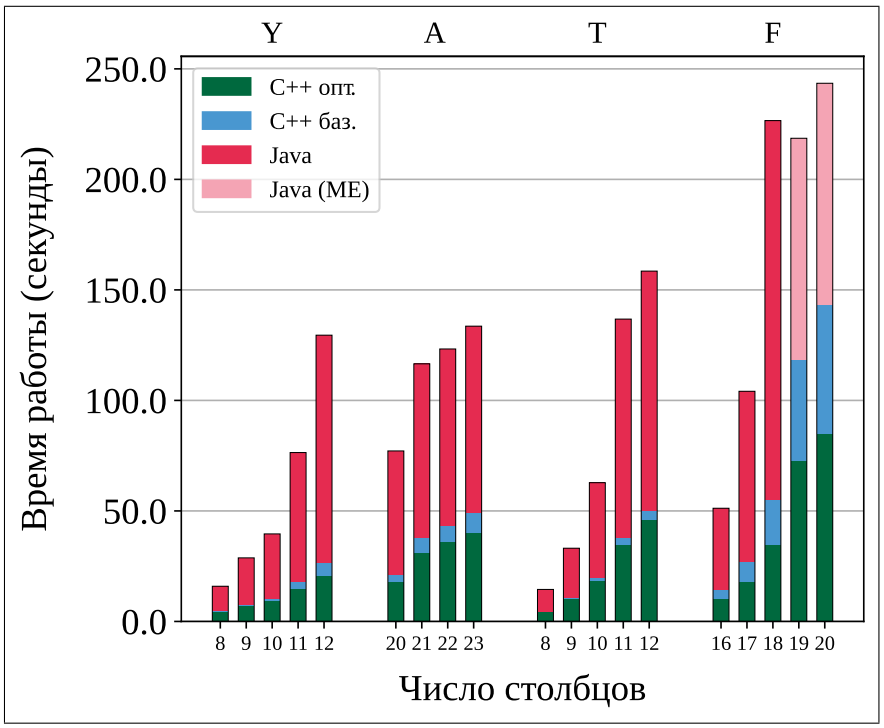

Рис. 3: Масштабируемость по числу столбцов (часть 1)

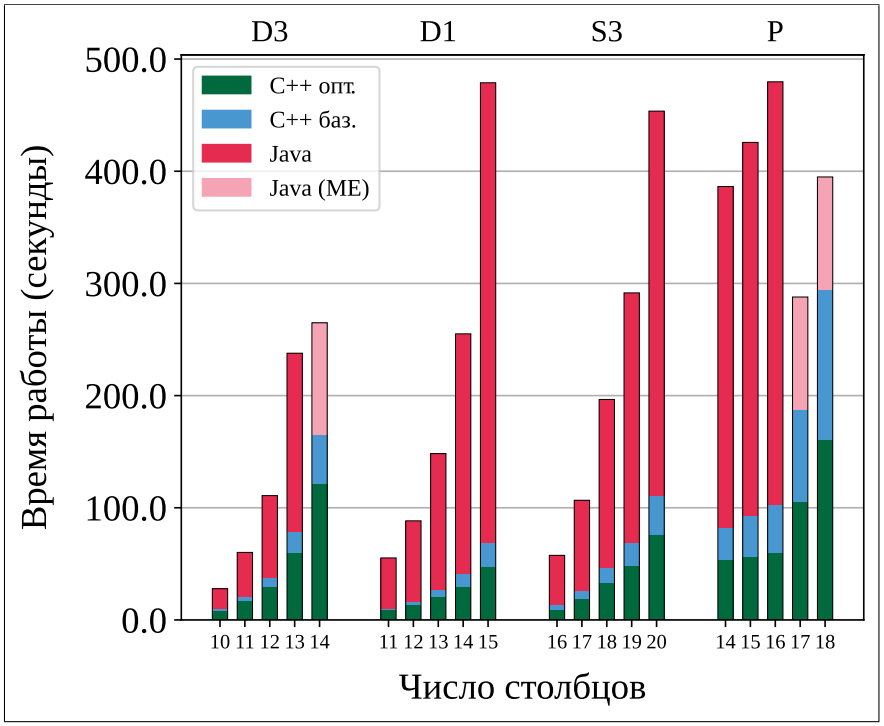

Рис. 4: Масштабируемость по числу столбцов (часть 2)

4.4.5. Эксперимент 5

Пятый эксперимент направлен на исследование вопроса потребления памяти всеми реализациями алгоритма FASTOD. Его результаты представлены в таблице 2. Они демонстрируют существенное уменьшение потребления памяти как базовой, так и оптимизированной версии C++ реализации алгоритма по сравнению с Java реализацией. Первая выигрывает у нее в 2.9–3.9 раза, а вторая в 2.4–2.9 раза. Важно отметить, что оптимизированная версия потребляет больше памяти, чем базовая. Однако это необходимая затрата для обеспечения более высокой эффективности алгоритма.

Эксперимент также показывает чрезмерное потребление оперативной памяти Java реализацией. На некоторых наборах данных она заканчивает свою работу с ошибкой нехватки оперативной памяти. C++ реализации, в свою очередь, успешно справились со всеми рассматриваемыми входными данными.

Таблица 2: Общие результаты потребления памяти

Данные	Java (ГБ)	C++ баз. (ГБ)	C++ опт. (ГБ)	Java vs C++ баз.	Java vs C++ опт.
G	3.150	0.809	1.160	3.894x	2.716x
A	3.087	1.035	1.270	2.983x	2.431x
P	*ME*	9.321	14.304	∞	∞
S1	11.919	3.478	4.504	3.427x	2.646x
D3	*ME*	8.296	14.124	∞	∞
S2	0.762	0.261	0.266	2.920x	2.865x
D4	*ME*	7.233	12.062	∞	∞
D1	6.878	2.374	3.556	2.897x	1.934x
F	*ME*	6.031	8.941	∞	∞
D2	1.062	0.071	0.126	14.958x	8.423x
M	9.508	2.519	2.595	3.775x	3.664x
T	2.537	0.760	0.994	3.338x	2.552x
O	*ME*	8.109	11.215	∞	∞
Y	3.168	0.839	1.248	3.776x	2.538x
S3	7.142	2.698	3.915	2.647x	1.824x

4.5. Результаты

Общие результаты экспериментов отражены в таблице 3. Строки, окрашенные в серый цвет, выделяют наборы данных, которые содержат

колонки с большим числом диапазонов (RB-колонки). ME обозначает, что на соответствующем наборе данных не удалось получить время работы алгоритма из-за возникновения ошибки нехватки оперативной памяти.

Предложенные эксперименты явно продемонстрировали, что базовая и оптимизированная C++ реализации алгоритма обладают более высокой производительностью по сравнению с Java реализацией. Кроме того, они более эффективны по памяти: на многих наборах данных Java реализация алгоритма не может корректно закончить свою работу в силу нехватки ресурсов компьютера, на котором производилось тестирование, в то время как обе C++ реализации хорошо справляются с данной задачей.

Эксперименты также показывают, что новое представление партиций, являющееся основным компонентом оптимизированной версии алгоритма, позволяет не только добиться существенного ускорения относительно базовой версии на специальных входных данных, но и не проиграть в эффективности на обычных наборах. Следует отметить, что оптимизированная версия алгоритма потребляет больше памяти, чем базовая. Однако эти затраты все равно существенно ниже, чем у Java реализации.

Таблица 3: Общие результаты

Набор данных	#Колонки	#RB-колонки	Java (секунды)	C++ баз. (секунды)	C++ опт. (секунды)	Уск. (баз.)	Уск. (опт.)	Уск. (итог)
G	15	8	50.326	17.083	14.184	2.946x	1.204x	3.548x
A	23	3	135.004	49.309	40.137	2.738x	1.229x	3.364x
P	18	8	ME	294.900	160.967	∞	1.832x	∞
S1	11	2	724.101	93.216	71.644	7.768x	1.301x	10.107x
D3	14	11	ME	166.503	121.866	∞	1.366x	∞
S2	20	2	7.177	5.187	4.994	1.384x	1.039x	1.437x
D4	12	8	ME	189.312	142.254	∞	1.331x	∞
D1	15	3	478.885	69.236	47.129	6.917x	1.469x	10.161x
F	20	15	ME	143.485	84.933	∞	1.689x	∞
D2	16	2	23.348	7.176	6.676	3.254x	1.075x	3.497x
M	4	4	49.791	16.723	10.438	2.977x	1.602x	4.770x
T	12	6	158.503	50.214	45.864	3.157x	1.095x	3.456x
O	13	7	ME	264.255	182.810	∞	1.446x	∞
Y	12	5	129.516	26.755	20.810	4.841x	1.287x	6.224x
S3	20	10	453.583	110.795	75.537	4.094x	1.467x	6.005x

Заключение

Была разработана эффективная реализация алгоритма FASTOD для поиска зависимостей порядка, предложены ее различные оптимизации и, как результат, выполнены следующие задачи:

1. Произведено знакомство с предметной областью поиска зависимостей порядка. Написан обзор, выбран алгоритм FASTOD выделения этих зависимостей и обоснован данный выбор.

2. Реализована базовая версия алгоритма FASTOD и получено ускорение вплоть до 8 раз (в среднем в 4 раза) по сравнению с имеющейся Java реализацией.

3. Предложены оптимизации этой реализации, позволяющие добиться ещё большего ускорения (в 1.3–1.8 раз относительно базовой версии) на специальных наборах данных.

4. Произведена интеграция алгоритма в проект Desbordante и добавлен в него новый тип зависимостей (зависимости порядка). Созданы Python привязки, позволяющие вызывать алгоритм из программ на языке Python, расширена CLI часть проекта и составлены примеры, описывающие сущность зависимостей порядка, а также то, как ими можно пользоваться.

5. Проверена эффективность обеих реализаций как на обычных наборах данных, так и на имеющих специальный вид. Получено итоговое ускорение работы алгоритма вплоть до 10 раз и уменьшение потребления памяти до 8 раз.

Алгоритм был интегрирован в проект Desbordante (pull request 355[9]), а предложенные подходы опубликованы в статье “Order in Desbordante: Techniques for Efficient Implementation of Order Dependency Discovery Algorithms” [28], представленной на 35-ой конференции ассоциации открытых инноваций FRUCT, проводившейся в Финляндии (г. Тампере). Ее работы публикуются IEEE и индексируются в Scopus.

[9]https://github.com/Desbordante/desbordante-core/pull/355

Список литературы

[1] Abedjan Ziawasch, Golab Lukasz, Naumann Felix. Profiling relational data: a survey // VLDB J. — 2015. — Vol. 24, no. 4. — P. 557–581. — URL: https://doi.org/10.1007/s00778-015-0389-y.

[2] Abedjan Ziawasch, Golab Lukasz, Naumann Felix. Data Profiling: A Tutorial // Proceedings of the 2017 ACM International Conference on Management of Data. — SIGMOD '17. — New York, NY, USA : Association for Computing Machinery, 2017. — P. 1747–1751. — URL: https://doi.org/10.1145/3035918.3054772.

[3] Abedjan Ziawasch, Schulze Patrick, Naumann Felix. DFD: Efficient Functional Dependency Discovery // Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014 / Ed. by Jianzhong Li, Xiaoyang Sean Wang, Minos N. Garofalakis et al. — ACM, 2014. — P. 949–958. — URL: https://doi.org/10.1145/2661829.2661884.

[4] Approximate Discovery of Functional Dependencies for Large Datasets / Tobias Bleifuß, Susanne Bülow, Johannes Frohnhofen et al. // Proceedings of the 25th ACM International on Conference on Information and Knowledge Management. — CIKM '16. — New York, NY, USA : Association for Computing Machinery, 2016. — P. 1803–1812. — URL: https://doi.org/10.1145/2983323.2983781.

[5] Business-Intelligence Queries with Order Dependencies in DB2 / Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz et al. // Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014 / Ed. by Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis et al. — OpenProceedings.org, 2014. — P. 750–761. — URL: https://doi.org/10.5441/002/edbt.2014.81.

[6] Chu Xu, Ilyas Ihab F., Papotti Paolo. Holistic data cleaning: Putting violations into context // 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013 / Ed. by Christian S. Jensen, Christopher M. Jermaine, Xiaofang Zhou. — IEEE Computer Society, 2013. — P. 458–469. — URL: https://doi.org/10.1109/ICDE.2013.6544847.

[7] Consonni Cristian et al. Discovering Order Dependencies through Order Compatibility // EDBT'19 / Ed. by Melanie Herschel et al. — OpenProceedings.org, 2019. — P. 409–420.

[8] Data Profiling / Z. Abedjan, L. Golab, F. Naumann, T. Papenbrock. Synthesis Lectures on Data Management. — Morgan & Claypool Publishers, 2018. — ISBN: 9781681734477. — URL: https://books.google.ru/books?id=LEF7DwAAQBAJ.

[9] 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. — May. — P. 344–354.

[10] Discovering Domain Orders via Order Dependencies / Reza Karegar, Melicaalsadat Mirsafian, Parke Godfrey et al. // 38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022. — IEEE, 2022. — P. 1098–1110. — URL: https://doi.org/10.1109/ICDE53745.2022.00087.

[11] DynFD: Functional Dependency Discovery in Dynamic Datasets / Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse et al. // Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019 / Ed. by Melanie Herschel, Helena Galhardas, Berthold Reinwald et al. — OpenProceedings.org, 2019. — P. 253–264. — URL: https://doi.org/10.5441/002/edbt.2019.23.

[12] Effective and Complete Discovery of Order Dependencies via Set-based Axiomatization / Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab et al. // Proc. VLDB Endow. — 2017. — Vol. 10, no. 7. — P. 721–732. — URL: http://www.vldb.org/pvldb/vol10/p721-szlichta.pdf.

[13] Effective and complete discovery of bidirectional order dependencies via set-based axioms / Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab et al. // VLDB J. — 2018. — Vol. 27, no. 4. — P. 573–591. — URL: https://doi.org/10.1007/s00778-018-0510-0.

[14] Efficient Discovery of Functional and Approximate Dependencies Using Partitions / Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, Hannu Toivonen // Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998 / Ed. by Susan Darling Urban, Elisa Bertino. — IEEE Computer Society, 1998. — P. 392–401. — URL: https://doi.org/10.1109/ICDE.1998.655802.

[15] Ehrlinger Lisa, Wöß Wolfram. A Survey of Data Quality Measurement and Monitoring Tools // Frontiers Big Data. — 2022. — Vol. 5. — P. 850611. — URL: https://doi.org/10.3389/fdata.2022.850611.

[16] Errata Note: Discovering Order Dependencies through Order Compatibility / Parke Godfrey, Lukasz Golab, Mehdi Kargar et al. // CoRR. — 2019. — Vol. abs/1905.02010. — arXiv : 1905.02010.

[17] Evaluation of freely available data profiling tools for health data research application: a functional evaluation review / Ben Gordon, Clara Fennessy, Susheel Varma et al. // BMJ open. — 2022. — 05. — Vol. 12. — P. e054186.

[18] Expressiveness and Complexity of Order Dependencies / Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, Calisto Zuzarte // Proc. VLDB Endow. — 2013. — Vol. 6, no. 14. — P. 1858–1869. — URL: http://www.vldb.org/pvldb/vol6/p1858-szlichta.pdf.

[19] 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: http://www.vldb.org/pvldb/vol8/p1082-papenbrock.pdf.

[20] 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.

[21] Ilyas Ihab F., Chu Xu. Trends in Cleaning Relational Data: Consistency and Deduplication // Found. Trends databases. — 2015. — Oct.. — Vol. 5, no. 4. — P. 281–393. — URL: http://dx.doi.org/10.1561/1900000045 (дата обращения: 2019-07-09).

[22] Ilyas Ihab F., Chu Xu. Data Cleaning. — New York, NY, USA : Association for Computing Machinery, 2019. — ISBN: 978-1-4503-7152-0.

[23] Jin Yifeng, Zhu Lin, Tan Zijing. Efficient Bidirectional Order Dependency Discovery // 36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020. — IEEE, 2020. — P. 61–72. — URL: https://doi.org/10.1109/ICDE48307.2020.00013.

[24] Langer Philipp, Naumann Felix. Efficient order dependency detection // VLDB J. — 2016. — Vol. 25, no. 2. — P. 223–241. — URL: https://doi.org/10.1007/s00778-015-0412-3.

[25] Mannila Heikki, Räihä Kari-Jouko. On the Complexity of Inferring Functional Dependencies // Discret. Appl. Math. — 1992. — Vol. 40, no. 2. — P. 237–243. — URL: https://doi.org/10.1016/0166-218X(92)90031-5.

[26] Ng Wilfred. An extension of the relational data model to incorporate ordered domains // ACM Trans. Database Syst. — 2001. — sep. —

Vol. 26, no. 3. — P. 344–383. — URL: https://doi.org/10.1145/502030.502033.

[27] Novelli Noël, Cicchetti Rosine. FUN: An Efficient Algorithm for Mining Functional and Embedded Dependencies // Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings / Ed. by Jan Van den Bussche, Victor Vianu. — Vol. 1973 of Lecture Notes in Computer Science. — Springer, 2001. — P. 189–203. — URL: https://doi.org/10.1007/3-540-44503-X_13.

[28] 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.

[29] Saxena Hemant, Golab Lukasz, Ilyas Ihab F. Distributed Implementations of Dependency Discovery Algorithms // Proc. VLDB Endow. — 2019. — Vol. 12, no. 11. — P. 1624–1636. — URL: http://www.vldb.org/pvldb/vol12/p1624-saxena.pdf.

[30] Schmidl Sebastian, Papenbrock Thorsten. Efficient distributed discovery of bidirectional order dependencies // VLDB J. — 2022. — Vol. 31, no. 1. — P. 49–74. — URL: https://doi.org/10.1007/s00778-021-00683-4.

[31] Szlichta Jaroslaw, Godfrey Parke, Gryz Jarek. Fundamentals of Order Dependencies // Proc. VLDB Endow. — 2012. — jul. — Vol. 5, no. 11. — P. 1220–1231. — URL: https://doi.org/10.14778/2350229.2350241.

[32] TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies / Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, Hannu Toivonen // Comput. J. — 1999. — Vol. 42, no. 2. — P. 100–111. — URL: https://doi.org/10.1093/comjnl/42.2.100.

[33] Unidata. Data profiling, и с чем его едят. — 2022. — URL: https://habr.com/ru/companies/unidata/articles/667636/ (дата обращения: 18.04.2024).

[34] Yao Hong, Hamilton Howard J. Mining functional dependencies from data // Data Min. Knowl. Discov. — 2008. — Vol. 16, no. 2. — P. 197–219. — URL: https://doi.org/10.1007/s10618-007-0083-9.

[35] Yao Hong, Hamilton Howard J., Butz Cory J. FD_Mine: Discovering Functional Dependencies in a Database Using Equivalences // Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan. — IEEE Computer Society, 2002. — P. 729–732. — URL: https://doi.org/10.1109/ICDM.2002.1184040.