Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 22.Б08-мм

Оптимизация алгоритма SPLIT поиска дифференциальных зависимостей в рамках платформы Desbordante

Синельников Михаил Алексеевич

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
ассистент кафедры ИАС Г. А. Чернышев

Санкт-Петербург
2025

Оглавление

Введение

В настоящее время наблюдается быстрый рост количества данных. В связи с автоматизацией различных сфер жизни данные начинают играть всё более важную роль. Таким образом, эффективная обработка данных становится одной из ключевых задач в современном мире.

Одной из важных задач обработки данных является их профилирование [1], то есть извлечение метаданных. Метаданными является любая информация о рассматриваемых данных. Метаданными может быть как размер или название файла, так и различные закономерности, найденные в данных.

Пожалуй, самым известным типом закономерностей в табличных данных являются функциональные зависимости (FD) [6]. Функциональная зависимость $X \to Y$ (X, Y — множества атрибутов таблицы) удерживается, если каждому уникальному значению на атрибутах X соответствует ровно одно уникальное значение на атрибутах Y. Функциональные зависимости активно применяются в базах данных для улучшения их качества.

Однако в реальных данных зачастую содержатся ошибки, опечатки и различные форматы записи одних и тех же данных, что затрудняет поиск функциональных зависимостей, так как они требуют точного равенства значений [8]. В связи с этим были предложены различные варианты обобщения функциональных зависимостей, позволяющие учесть небольшие различия в равных по смыслу значениях.

Одним из таких вариантов обобщения являются дифференциальные зависимости (DD) [10]. В отличие от функциональных зависимостей, дифференциальные зависимости учитывают расстояние между значениями в атрибуте с помощью введения метрики для типа данных атрибута.

Desbordante[1] [2] — высокопроизводительный наукоёмкий профилировщик данных с открытым исходным кодом, позволяющий искать нетривиальные закономерности в табличных данных. На данный мо-

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

мент в Desbordante поддерживается поиск и валидация функциональных зависимостей, а также некоторых других видов зависимостей.

Мною в рамках Desbordante был реализован[2] [11] алгоритм SPLIT поиска дифференциальных зависимостей, описанный в статье [10]. Однако реализация оказалась недостаточно эффективной и требовала много времени для сравнительно небольших объёмов данных. Кроме того, уже после принятия моей реализации в Desbordante была опубликована статья [4], авторы которой предложили новый алгоритм поиска DD, а также реализовали алгоритм SPLIT для сравнения с новым алгоритмом. Открытая реализация[3] алгоритма SPLIT от авторов статьи [4] оказалась производительнее реализации в Desbordante. Таким образом, возникла необходимость оптимизировать алгоритм SPLIT в рамках платформы Desbordante.

[2] https://github.com/Desbordante/desbordante-core/pull/374

[3] https://github.com/TristonK/FastDD-Exp/blob/main/Exp-1/IE.zip

1. Постановка задачи

Целью работы является оптимизация алгоритма SPLIT поиска дифференциальных зависимостей. Для достижения цели были поставлены следующие задачи:

- провести обзор алгоритма SPLIT и его открытой реализации с целью поиска возможностей для оптимизации;
- реализовать оптимизации алгоритма;
- произвести сравнение производительности первоначальной и оптимизированной реализаций алгоритма.

2. Обзор

2.1. Основные определения

Все определения в данном подразделе повторяют определения из соответствующего подраздела моего отчёта [11] за осень 2023 года и взяты из работы [5].

Определение 2.1. Обозначим буквой R множество всех атрибутов в таблице, домен атрибута $A_i \in R$ обозначим как $dom(A_i)$.

Определение 2.2. Метрика d_A — функция, определённая на атрибуте $A \in R$, которая любой паре значений a_1, a_2 из атрибута сопоставляет расстояние между ними. При этом d_A обязана удовлетворять следующим условиям:

1. $d_A(a_1, a_2) \geq 0$;
2. $d_A(a_1, a_2) = 0 \Leftrightarrow a_1 = a_2$;
3. $d_A(a_1, a_2) = d_A(a_2, a_1)$,

где $a_1, a_2 \in dom(A)$.

Определение 2.3. Дифференциальная функция $A[\omega]$ ($\omega = [x, y], 0 \leq x \leq y$) — функция, определённая на атрибуте A с метрикой d_A и возвращающая булево значение, показывающее, выполняется ли утверждение $\forall a_1, a_2 \in dom(A)\ x \leq d_A(a_1, a_2) \leq y$.

Определение 2.4. Дифференциальной функцией $X[W_X]$ на множестве атрибутов $X = \{A_1, \ldots, A_m\}$ называется логическое умножение дифференциальных функций для каждого атрибута:

$$X[W_X] = A_1[\omega_1] \wedge A_2[\omega_2] \wedge \ldots \wedge A_m[\omega_m].$$

Если для пары строк таблицы значение дифференциальной функции $X[W_X]$ истинно, то говорят, что данная пара строк удовлетворяет $X[W_X]$.

Определение 2.5. Дифференциальная функция $X[W_X]$ включает в себя дифференциальную функцию $Y[W_Y]$ ($X[W_X] \succeq Y[W_Y]$), если:

$$\forall A_i[\omega_i] \in X[W_X] \; \exists \; A_i[\omega_i'] \in Y[W_Y] : \omega_i' \subseteq \omega_i.$$

Заметим, что для того, чтобы $X[W_X] \succeq Y[W_Y]$, необходимо, чтобы $X \subseteq Y$. В самом деле, если $\exists A_i : A_i \in X, A_i \notin Y$, то для $A_i[\omega_i] \in X[W_X] \nexists A_i[\omega_i'] \in Y[W_Y]$. Также заметим, что если $X[W_X] \succeq Y[W_Y]$, то любая пара строк таблицы, удовлетворяющая $Y[W_Y]$, также удовлетворяет и $X[W_X]$.

Определение 2.6. Дифференциальная зависимость (DD) — утверждение $X[W_L] \to Y[W_R]$ между двумя дифференциальными функциями. Дифференциальная зависимость удерживается в таблице, если для любой пары строк таблицы из того, что она удовлетворяет $X[W_L]$, следует то, что она удовлетворяет $Y[W_R]$.

Определение DD похоже на определение включения в себя. Однако, если для того, чтобы $X[W_L] \succeq Y[W_R]$, необходимо, чтобы $L \subseteq R$, для того, чтобы DD удерживалась, это не требуется. Более того, наиболее интересными для рассмотрения являются зависимости с непересекающимися областями определения левой и правой дифференциальных функций ($L \cap R = \emptyset$).

Определение 2.7. Множество DD Σ_2 является логическим следствием (импликацией) множества DD Σ_1, если для любой таблицы из удерживания всех DD из Σ_1 следует удерживание всех DD из Σ_2.

Определение 2.8. Пусть Σ — множество дифференциальных зависимостей. DD $X[W_L] \to Y[W_R] \in \Sigma$ называется минимальной, если выполняются следующие условия:

1. $\nexists X'[W_X] \to Y[W_R] \in \Sigma : X'[W_X] \succeq X[W_L]$;
2. $\nexists X[W_L] \to Y'[W_Y] \in \Sigma : Y[W_R] \succeq Y'[W_Y]$.

Определение 2.9. Множество DD Σ' называется покрытием для множества DD Σ, если любая DD из Σ лежит в Σ' или является логическим следствием дифференциальных зависимостей из Σ'.

Определение 2.10. Покрытие Σ_c множества Σ называется минимальным, если не существует покрытия $\Sigma' \subseteq \Sigma_c$.

Таблица 1: Пример таблицы (взят из работы [5])

TID	**YoS**	**Gen**	**Edu**	**Sal**
1	25	2	4	15
2	25	1	3	18
3	28	1	3	15
4	32	2	2	12
5	30	1	1	15

Пример 2.1. В качестве примера рассмотрим таблицу 1. Возьмём в качестве метрики для каждой колонки $A \in R$ $d_A(x, y) = |x - y|$. Дифференциальная зависимость $YoS[0, 0] \to Edu[1, 1]$ удерживается, так как для любой пары строк, удовлетворяющих $YoS[0, 0]$ (то есть для пары строк $\{1, 2\}$), эта пара строк удовлетворяет и $Edu[1, 1]$. DD $YoS[2, 2] \to Sal[0, 0]$ не удерживается, так как пара строк $\{4, 5\}$ удовлетворяет $YoS[2, 2]$, но не удовлетворяет $Sal[0, 0]$.

Теперь возьмём в качестве множества дифференциальных зависимостей

$$\begin{aligned}\Sigma = \{&YoS[0, 5]Gen[0, 0] \to Edu[0, 0], YoS[0, 5] \to Edu[0, 2],\\ &YoS[0, 7] \to Edu[0, 0], Gen[0, 0]Sal[0, 0] \to YoS[3, 5],\\ &Gen[0, 0]Sal[0, 0] \to YoS[3, 7]\}.\end{aligned}$$

Множество DD

$$\Sigma_c = \{YoS[0, 7] \to Edu[0, 0], Gen[0, 0]Sal[0, 0] \to YoS[3, 5]\}$$

является минимальным покрытием Σ. Действительно, любая DD из Σ является логическим следствием DD из Σ_c. Например, если удерживает-

ся DD $YoS[0,7] \rightarrow Edu[0,0]$, то будет выполняться следующая цепочка следствий: пара строк удовлетворяет $YoS[0,5]Gen[0,0] \Rightarrow$ она удовлетворяет $YoS[0,7] \Rightarrow$ она удовлетворяет $Edu[0,0]$, значит, удерживается DD $YoS[0,5]Gen[0,0] \rightarrow Edu[0,0]$. Аналогично доказывается удерживание остальных DD из Σ. Также очевидно, что не существует покрытия меньшего по включению.

Пример 2.2. Пусть $DD_1 = d[1,6] \rightarrow a[0,150]$, $DD_2 = d[6,7] \rightarrow a[0,100]$, $DD_3 = d[1,7] \rightarrow a[0,150]$, $\Sigma = \{DD_1, DD_2, DD_3\}$. $\Sigma_c = \{DD_1, DD_2\}$ является минимальным покрытием Σ, так как DD_3 логически выводится из $\{DD_1, DD_2\}$, и, если убрать из Σ_c хотя бы одну из зависимостей, Σ_c перестанет быть покрытием.

Теперь заметим, что DD_1 не является минимальной DD, так как она не удовлетворяет условию 1 определения 2.8. Действительно, $DD_3 \in \Sigma$ и $d[1,7] \succeq d[1,6]$. Таким образом, если заменить DD_1 на DD_3 в Σ_c, множество $\Sigma'_c = \{DD_2, DD_3\}$ будет минимальным покрытием Σ, состоящим только из минимальных DD.

Одной из наиболее значимых задач поиска зависимостей в данных является поиск минимального покрытия всех зависимостей, удерживающихся в таблице. Минимальное покрытие удобно тем, что оно позволяет значительно уменьшить огромное по размеру множество удерживающихся зависимостей до достаточно небольшого множества, из которого можно логически вывести все остальные зависимости. При поиске дифференциальных зависимостей даже минимальное покрытие является очень большим по размеру, вследствие чего рассматривается задача поиска минимального покрытия, состоящего только из минимальных зависимостей.

2.2. Алгоритмы поиска дифференциальных зависимостей

В данном подразделе будут рассмотрены основные алгоритмы поиска DD.

Первый алгоритм поиска DD (алгоритм SPLIT) был представлен в 2011 году в статье [10], впервые вводящей дифференциальные зависимости. Более детально данный алгоритм будет рассмотрен в подразделе 2.3. Данный алгоритм был реализован мною в Desbordante[4]. Подробно о реализации алгоритма рассказано в моём отчёте [11] за осень 2023 года.

В 2015 году в статье [5] был представлен алгоритм AR-DDMiner, проводящий поиск DD с помощью ассоциативных правил (AR). В статье было показано, что данный алгоритм работает значительно быстрее, чем SPLIT. Однако AR-DDMiner имеет существенный недостаток — алгоритм находит зависимости из очень ограниченного класса, не беря в рассмотрение все возможные DD. Левая часть любой найденной DD является вырожденной, то есть интервал допустимых расстояний на каждом атрибуте является вырожденным отрезком:

$$X[W_L] = A_1[\omega_1] \wedge A_2[\omega_2] \wedge \ldots \wedge A_m[\omega_m], \forall i \ \omega_i = [x, x], x \geq 0.$$

В 2024 году в статье [4] был представлен алгоритм FastDD. Алгоритм проводит поиск DD, используя задачу нахождения вершинного покрытия (hitting set enumeration, set cover enumeration). Данная техника хорошо себя зарекомендовала в задачах поиска уникальных комбинаций колонок (UCC) [7], функциональных зависимостей (FD) [3] и запрещающих ограничений (DC) [9]. В секции экспериментов статьи показано, что алгоритм FastDD существенно производительнее алгоритма SPLIT. Немаловажным преимуществом алгоритма является присутствие открытой реализации как самого алгоритма FastDD[5], так и алгоритма SPLIT[6], использовавшегося для сравнения в экспериментах.

Несмотря на все преимущества алгоритма FastDD, в рамках данной работы было решено оптимизировать алгоритм SPLIT, так как его реализация уже присутствует в Desbordante и является менее производительной, чем реализация авторов статьи [4]. Реализация алгоритма

[4] https://github.com/Desbordante/desbordante-core/pull/374
[5] https://github.com/TristonK/FastDD
[6] https://github.com/TristonK/FastDD-Exp/blob/main/Exp-1/IE.zip

FastDD в рамках платформы Desbordante является возможным продолжением работы в следующем семестре.

2.3. Алгоритм SPLIT

Данный подраздел повторяет соответствующий подраздел моего отчёта [11] за осень 2023 года.

Алгоритм SPLIT предназначен для поиска минимального покрытия всех DD, удерживающихся в таблице, состоящего только из минимальных DD. Алгоритм имеет три основных стадии:

1. Построение пространства поиска;
2. Отсечение зависимостей, не минимальных по левой части (условие 1 определения 2.8);
3. Отсечение зависимостей, не минимальных по правой части (условие 2 определения 2.8), и зависимостей, логически выводящихся из остальных.

Первая составляющая алгоритма — построение пространства поиска дифференциальных функций. В работе [10] указаны некоторые возможные способы это сделать. Например,

$$\forall A_i \in R\ \Phi(A_i) = \{A_i[u, v] \mid 0 \leq u \leq v \leq D\},$$

где D — максимальное расстояние между значениями атрибута A_i. Соответственно, для нескольких атрибутов $X = \{A_1, \ldots, A_m\}$ пространством поиска является $\Phi(X) = \Phi(A_1) \times \ldots \Phi(A_m), A_i \in X$.

Вторая составляющая алгоритма — отсечение зависимостей, не минимальных по левой части. Более формально задача звучит так: из данного пространства поиска дифференциальных функций $\Phi(X)$, являющихся левыми частями DD с фиксированной правой частью $\phi_R[Y]$, выделить такое подмножество $\Phi'(X)$, что все DD $\phi_R[X] \to \phi_R[Y]$, где $\phi_R[X] \in \Phi'(X)$, удерживаются в таблице и $\nexists \phi'_R[X] \to \phi_R[Y] : \phi'_R[X] \succeq \phi_R[X]$ и $\phi'_R[X] \to \phi_R[Y]$ удерживается в таблице.

Наивная реализация данной части алгоритма — валидация каждой DD из пространства поиска и дальнейшее отсечение из получившегося множества удерживающихся зависимостей тех, которые не являются минимальными по левой части. Для валидации DD требуется $O(n^2)$ времени, где n — число строк в таблице, так как необходимо рассмотреть все пары строк и вычислить расстояние между значениями в этих строках. Если размер таблицы большой, то валидация DD становится самой затратной по времени частью алгоритма. Дальнейшие варианты алгоритма SPLIT представляют собой различные способы уменьшить число валидаций.

Основные варианты алгоритма:

1. Негативное отсечение;
2. Позитивное отсечение;
3. Гибридное отсечение;
4. Отсечение пар строк.

Негативное отсечение основывается на простом утверждении: если DD $\phi_1[X] \to \phi_R[Y]$ не удерживается в таблице, то и DD $\phi_2[X'] \to \phi_R[Y]$, где $\phi_2[X'] \succeq \phi_1[X]$, также не удерживается. Таким образом, если $\phi_1[X] \to \phi_R[Y]$ не удерживается в таблице, можно сразу отсечь некоторое множество зависимостей, которые не нужно валидировать.

Позитивное отсечение основывается на аналогичном утверждении, позволяя отсечь некоторое множество зависимостей, если $\phi_1[X] \to \phi_R[Y]$ удерживается в таблице.

Гибридное отсечение комбинирует негативное и позитивное отсечение, позволяя алгоритму работать быстрее как в случае, когда большинство DD из пространства поиска удерживается, так и в обратном случае.

Отсечение пар строк основывается на том факте, что множество пар строк, удовлетворяющих $\phi_2[X']$ является подмножеством множества пар строк, удовлетворяющих $\phi_1[X]$, если $\phi_1[X] \succeq \phi_2[X']$. Таким

ALGORITHM 1: Bottom-Up REDUCE(I, $\Phi(X)$, $\phi_R[Y]$)

Input: An instance I, a search space $\Phi(X)$ of $\phi_L[X]$ and a target $\phi_R[Y]$
Output: A minimal set Σ for all DDs determining $\phi_R[Y]$ under I

$\Sigma := \emptyset$
$\phi_p[Z] :=$ the last element removed from $\Phi(X)$
$\Phi_3(X) := \{\phi_v[V] \in \Phi(X) \mid V \subseteq Z, \phi_v[V] \succeq \phi_p[V]\}$
$\Phi_4(X) := \Phi(X) \setminus \Phi_3(X)$
if $I \not\models \phi_p[Z] \rightarrow \phi_R[Y]$ **then**
  $\Sigma := \Sigma \uplus$ REDUCE(I, $\Phi_4(X)$, $\phi_R[Y]$)
else
  $\Sigma := \Sigma \uplus \{\phi_p[Z] \rightarrow \phi_R[Y]\}$
  $\Sigma := \Sigma \uplus$ REDUCE(I, $\Phi_3(X)$, $\phi_R[Y]$)
  $\Sigma := \Sigma \uplus$ REDUCE(I, $\Phi_4(X)$, $\phi_R[Y]$)
end if
return Σ

Рис. 1: Негативное отсечение (взято из работы [10])

образом можно отсекать некоторые пары строк, что позволяет сократить время валидации DD.

Третья составляющая алгоритма — отсечение зависимостей, не минимальных по правой части, и зависимостей, логически выводящихся из остальных. После предыдущего этапа работы алгоритма для каждого атрибута Y и для каждой дифференциальной функции $\phi_R[Y] \in \Phi(Y)$ получено множество DD, минимальных по левой части. Для получения минимального покрытия достаточно удалить из множества DD, логически выводящиеся из остальных, то есть удалить такие DD $\phi_1[W] \rightarrow \phi_2[V]$, что $\exists \phi_L[X] \rightarrow \phi_R[Y] : \phi_L[X] \succeq \phi_1[W], \phi_2[V] \succeq \phi_R[Y]$.

ALGORITHM 3: Minimal COVER(R, I)

Input: A relation schema R, and an instance I of R.
Output: A minimal cover Σ for all DDs under I.

$\Sigma := \emptyset$
for each attribute $Y \in R$ **do**
  $X := R \setminus \{Y\}$
  for each $\phi_R[Y] \in \Phi(Y)$ **do**
    $\Sigma := \Sigma \cup$ REDUCE(I, $\Phi(X)$, $\phi_R[Y]$) {reduce step}
  end for
end for
repeat
  if exist $\phi_L[X] \rightarrow \phi_R[Y], \phi_w[W] \rightarrow \phi_v[V] \in \Sigma$ such that $X \subseteq W$, $\phi_L[X] \succeq \phi_w[X]$, $V \subseteq Y$ and $\phi_v[V] \succeq \phi_R[V]$ **then**
    remove $\phi_w[W] \rightarrow \phi_v[V]$ from Σ
  end if
until no changes on Σ
return Σ

Рис. 2: Поиск минимального покрытия (взято из работы [10])

3. Метод

В результате сравнения производительности реализации алгоритма SPLIT в Desbordante и открытой реализации авторов статьи [4] было выяснено, что последняя работает быстрее. Анализ открытой реализации показал, что её подходы к реализации алгоритма SPLIT значительно отличаются от подходов, использованных в реализации в Desbordante. Однако, в открытой реализации была найдена оптимизация, одинаково применимая к обеим реализациям. Подробнее об этой оптимизации рассказано в подразделе 3.4. Остальные оптимизации были найдены и реализованы после анализа недостатков реализации алгоритма в Desbordante.

Далее перечислены основные оптимизации, реализованные в алгоритме.

3.1. Перемещение метода IsFeasible

В статье [10] указано, что в пространство поиска должны попадать только те дифференциальные функции, которые удовлетворяются хотя бы одной парой строк. Для этой проверки в первоначальной реализации был реализован метод `IsFeasible`. Данный метод предполагает проверку условия для каждой пары строк, поэтому данная проверка является дорогой (сложность метода — $O(n^2)$). В первоначальной реализации данный метод вызывался для каждого кандидата на попадание в пространство поиска, что делало стадию 1 алгоритма достаточно долгой.

Было замечено, что среди всех рассматриваемых DF обычно довольно мало тех, которым не удовлетворяет ни одна пара строк. Поэтому было решено перенести исполнение данной проверки в стадию 2 перед непосредственным добавлением минимальной DD в список. В результате оптимизации стадия 1 значительно ускорилась, так как практически всё время работы данной стадии тратилось на выполнение метода `IsFeasible`. На время исполнения стадии 2 перенос проверки почти не повлиял, так как эта проверка выполняется не так часто.

3.2. Игнорирование атрибутов с пустым пространством поиска

В реализации алгоритма пространство поиска для каждой колонки задаётся пользователем с помощью таблицы специального вида. Однако в случае, если размер пространства поиска для колонки равен 0, данную колонку можно игнорировать, так как она не будет влиять на результат. В реализацию алгоритма был добавлен метод `CalculateIndexSearchSpaces`, конструирующий пространства поиска для каждой колонки и отбрасывающий колонки с пустыми пространствами поиска. В результате для некоторых датасетов и определяемых пользователем пространств поиска уменьшился размер потребляемой памяти, а также время работы.

3.3. Использование индексов списков позиций

В первоначальной реализации первым этапом являлся подсчёт всех расстояний между значениями в таблице. Для этого использовался метод `CalculateAllDistances`, вычисляющий расстояния для каждой колонки и для каждой пары строк и помещающий вычисленные значения в трёхмерный массив. Данный метод имеет сложность $O(n^2m)$ по времени и по памяти, где n — число строк, а m — число колонок. Таким образом, данная реализация является не очень эффективной как по времени, так и по памяти.

Для оптимизации подсчёта расстояний были использованы индексы списков позиций (Position List Index, PLI) [6]. Для каждой колонки PLI представляет собой множество кластеров уникальных значений в колонке. Вычисление PLI для каждой колонки имеет сложность O(n), что является достаточно быстрым по сравнению с подсчётом расстояний между всеми парами строк за $O(n^2)$. Для реализации PLI был создан класс `DistancePositionListIndex`. Поля класса:

- `value_mapping_` — словарь, хранящий для каждого уникального значения номер соответствующего кластера;

- `clusters_` — массив, хранящий для каждого кластера номер строки его представителя, а также размер кластера;
- `inverted_index_` — массив, хранящий для каждой строки номер кластера значения в этой строке.

Метод `CalculateAllDistances` был изменён: теперь метод вычисляет расстояния не между каждой парой строк, а между каждой парой кластеров уникальных значений. Для получения расстояния между каждой парой строк в дальнейшем необходимо узнать номера кластеров этих строк, записанных в PLI, и найти записанное в массив расстояние между соответствующими кластерами. Поскольку кластеров уникальных значений в реальных данных обычно значительно меньше, чем число строк, полученный трёхмерный массив расстояний имеет гораздо меньший размер, чем в первоначальной реализации, что уменьшает потребляемую алгоритмом память. Время вычисления расстояний вследствие этого также уменьшилось.

3.4. Подсчёт полезных пар строк

Основное время алгоритма тратит метод `VerifyDD`, проверяющий, удерживается ли DD в таблице. Метод имеет сложность $O(n^2)$, так как он требует для каждой пары строк проверок, удовлетворяет ли пара строк нужным DF. Однако в реальности результаты проверок для пар строк часто совпадают из-за небольшого размера пространства поиска для каждой колонки.

Пространство поиска для каждой колонки задаётся пользователем и является небольшим. Результирующее пространство поиска является декартовым произведением одноколоночных пространств поиска, поэтому его размер значительно больше.

Для оптимизации была предложена ещё одна стадия алгоритма — подсчёт полезных пар строк. Для этого был создан метод `CalculateTuplePairs`. Данный метод для каждой пары строк проверяет удовлетворение каждой DF из одноколоночных пространств поиска. Резуль-

татом каждой проверки является 1 или 0, где 1 показывает, что пара удовлетворяет DF, а 0 — что она не удовлетворяет DF. Все биты результатов проверки записываются в битсет. Поскольку размер пространств поиска для каждой колонки небольшой, то и размер битсета, равный сумме размеров одноколоночных пространств поиска, является небольшим, и сам битсет вычисляется быстро. Если для разных пар строк полученные битсеты совпадают, то любая проверка этих пар на любой DF из результирующего пространства поиска будет выдавать одинаковый результат, поэтому одна из этих пар строк является бесполезной. В результате множество пар строк с уникальными битсетами будет являться множеством полезных пар строк, использующихся для дальнейших стадий алгоритма. В реальности данное множество пар строк значительно меньше множества всех пар строк, из-за чего стадия 2 работает в разы быстрее. Хотя добавленная стадия алгоритма с ростом числа строк и колонок начинает занимать больше времени, алгоритм всё равно начинает работать быстрее.

4. Эксперимент

Для тестирования производительности алгоритма были использованы его первоначальная реализация в Desbordante, открытая реализация алгоритма[7] [4] и оптимизированная реализация в Desbordante. Для тестирования была взята самая быстрая версия алгоритма — с отсечением пар строк (`IE-Hybrid`). Эксперименты проводились на ноутбуке с установленной операционной системой Linux Ubuntu 24.04, 8 ГБ оперативной памяти и 8 процессорами Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40 GHz (максимальная частота — 4.1 GHz). При проведении экспериментов был выключен файл подкачки, сброшены кэши и зафиксирована максимальная частота процессоров. Все представленные показатели времени работы алгоритмов являются усреднёнными значениями по пяти замерам. Для каждого атрибута использовалось по 5 дифференциальных функций для формирования пространства поиска.

4.1. Сравнение производительности

В таблице 2 представлено время работы различных реализаций алгоритма SPLIT на разных датасетах. Колонка 'Перв.' показывает время работы первоначальной реализации в Desbordante; колонка 'Откр.' показывает время работы открытой реализации; колонка 'Опт.' показывает время работы оптимизированной реализации в Desbordante. ML означает, что исполнение прервалось из-за недостатка оперативной памяти; TL означает, что алгоритм не завершил свою работу за 30 минут.

Из таблицы можно легко увидеть, что оптимизированная реализация алгоритма SPLIT в Desbordante значительно превосходит первоначальную и открытую реализации как по времени работы, так и по затрачиваемой памяти.

В таблице 3 представлены ускорения, полученные на различных датасетах. Колонка 'Перв./Опт.' показывает отношение времени работы

[7] https://github.com/TristonK/FastDD-Exp/blob/main/Exp-1/IE.zip

Таблица 2: Время работы (с)

Датасет	#колонок	#строк	#DD	Перв.	Откр.	Опт.
adult	5	2000	9	11.017	1.007	0.255
adult	5	4000	10	17.550	3.221	0.957
adult	5	6000	11	25.231	6.799	2.146
adult	5	8000	11	42.632	11.747	3.798
adult	5	10000	9	ML	18.097	5.964
adult	5	32561	5	ML	186.394	64.308
adult	6	32561	8	ML	315.868	74.713
adult	7	32561	14	ML	498.309	86.449
adult	8	32561	19	ML	658.240	100.774
adult	9	32561	37	ML	ML	203.924
abalone	5	4177	9	4.129	1.281	0.814
abalone	6	4177	18	8.019	1.828	0.999
abalone	9	4177	117	170.815	ML	1.770
digits	6	1797	0	1.481	1.263	0.330
digits	9	1797	16	8.061	1227.355	2.356
flights	10	1000	17	2.867	ML	0.060
flights	15	1000	61	TL	ML	46.940
neighbors10k	7	10000	12	ML	ML	5.027

первоначальной и оптимизированной реализаций алгоритма; колонка 'Откр./Опт.' показывает отношение времени работы открытой и оптимизированной реализаций алгоритма.

Как видно из таблицы, оптимизированная реализация алгоритма SPLIT в Desbordante стала производительней открытой реализации, среднее ускорение работы — 3.86.

4.2. Сравнение времени работы стадий алгоритма

На рисунке 3 представлена диаграмма, показывающая время работы различных стадий оптимизированного алгоритма SPLIT. Для данного эксперимента алгоритм был запущен на датасете adult с использованием его первых 5 столбцов и различного числа строк. Как показано на диаграмме, почти всё время работы алгоритма занимает подсчёт полезных пар строк. Время работы данной стадии растёт квадратично в

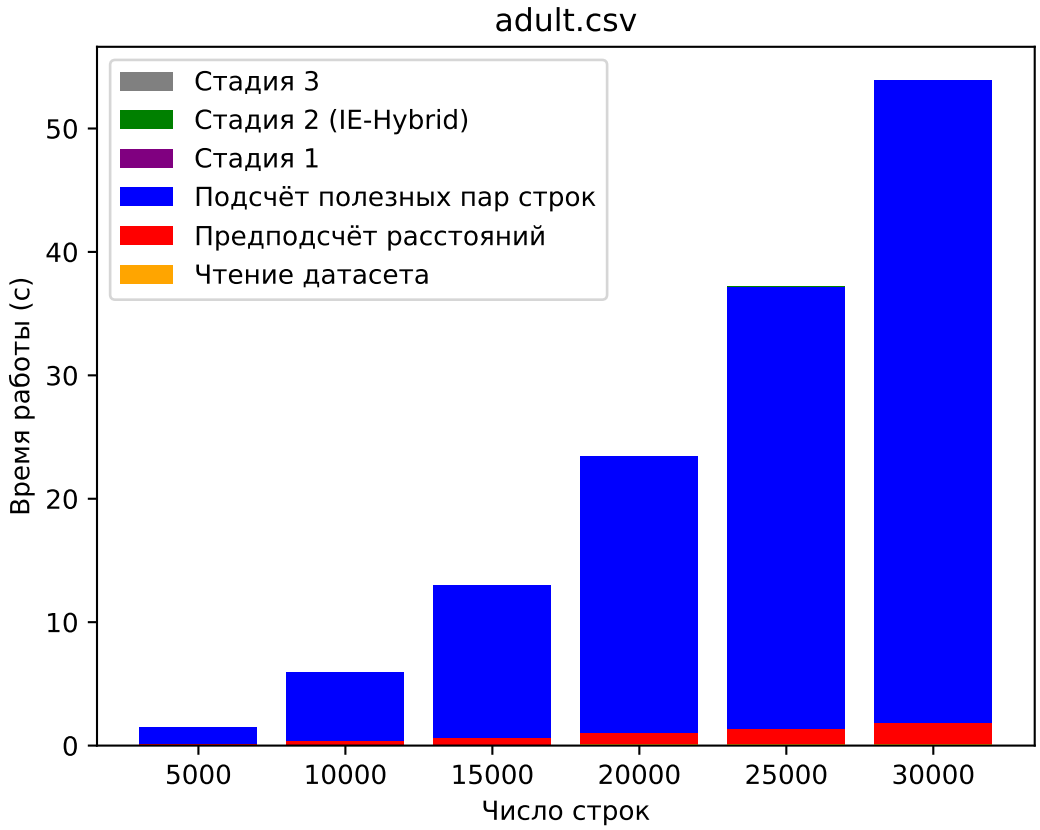

Рис. 3: Сравнение времени работы стадий алгоритма в зависимости от числа строк

Таблица 3: Ускорение

Датасет	#колонок	#строк	Перв./Опт.	Откр./Опт.
adult	5	2000	43.20	3.95
adult	5	4000	18.34	3.37
adult	5	6000	11.76	3.17
adult	5	8000	11.22	3.09
adult	5	10000	-	3.17
adult	5	32561	-	2.90
adult	6	32561	-	4.23
adult	7	32561	-	5.76
adult	8	32561	-	6.53
adult	9	32561	-	-
abalone	5	4177	5.07	1.57
abalone	6	4177	8.03	1.83
abalone	9	4177	96.51	-
digits	6	1797	3.83	6.73
digits	9	1797	3.42	520.95
flights	10	1000	47.78	-
flights	15	1000	-	-
neighbors10k	7	10000	-	-

зависимости от числа строк, что находит своё отражение в диаграмме.

На рисунке 4 представлена диаграмма, показывающая время работы стадий оптимизированного алгоритма в зависимости от числа атрибутов. Для данного эксперимента алгоритм был запущен на датасете adult с использованием всех его строк и различного числа столбцов. На диаграмме видно, что с ростом числа столбцов время работы подсчёта полезных пар строк растёт незначительно, в то время как время работы стадии 2 начинает расти очень быстро, почти достигая время работы подсчёта пар строк на 9 колонках. Эта тенденция подтверждается тем фактом, что время работы подсчёта пар строк линейно зависит от числа колонок, а время работы стадии 2 зависит экспоненциально от числа колонок.

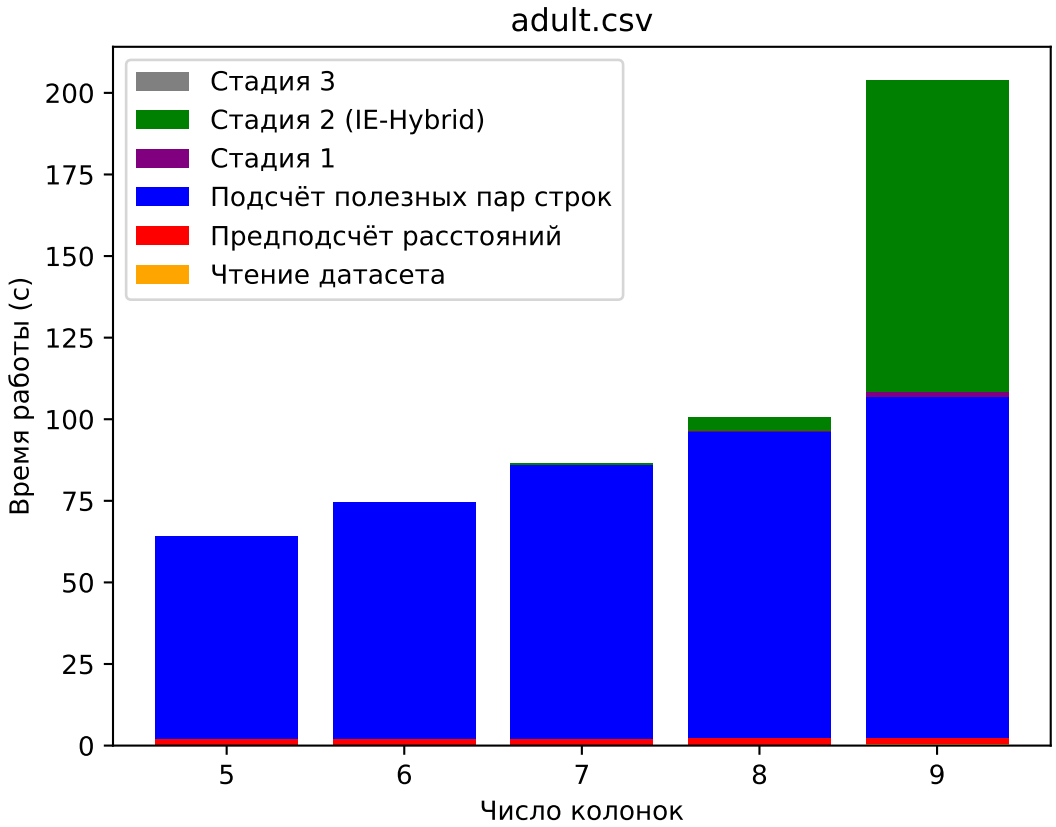

Рис. 4: Сравнение времени работы стадий алгоритма в зависимости от числа колонок

Заключение

В ходе работы были выполнены следующие задачи:

- сделан обзор алгоритма SPLIT и его открытой реализации с целью поиска возможностей для оптимизации;
- реализованы оптимизации алгоритма;
- проведено сравнение производительности первоначальной и оптимизированной реализаций алгоритма.

Исходный код алгоритма доступен на GitHub[8,9].

[8] https://github.com/Desbordante/desbordante-core/pull/439 (дата обращения: 2 января 2025 г.).

[9] https://github.com/Desbordante/desbordante-core/pull/506 (дата обращения: 2 января 2025 г.).

Список литературы

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

[2] 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.

[3] Discovering Functional Dependencies through Hitting Set Enumeration / Tobias Bleifuß, Thorsten Papenbrock, Thomas Bläsius et al. // Proc. ACM Manag. Data. — 2024. — mar. — Vol. 2, no. 1. — 24 p. — URL: https://doi.org/10.1145/3639298.

[4] Efficient Differential Dependency Discovery / Shulei Kuang, Honghui Yang, Zijing Tan, Shuai Ma // Proc. VLDB Endow. — 2024. — may. — Vol. 17, no. 7. — P. 1552–1564. — URL: https://doi.org/10.14778/3654621.3654624.

[5] Efficient Discovery of Differential Dependencies Through Association Rules Mining / Selasi Kwashie, Jixue Liu, Jiuyong Li, Feiyue Ye // Databases Theory and Applications / Ed. by Mohamed A. Sharaf, Muhammad Aamir Cheema, Jianzhong Qi. — Cham : Springer International Publishing, 2015. — P. 3–15.

[6] Functional dependency discovery: an experimental evaluation of seven algorithms / Thorsten Papenbrock, Jens Ehrlich, Jannik Marten et al. // Proc. VLDB Endow. — 2015. — jun. — Vol. 8, no. 10. — P. 1082–1093. — URL: https://doi.org/10.14778/2794367.2794377.

[7] Hitting set enumeration with partial information for unique column combination discovery / Johann Birnick, Thomas Bläsius, To-

bias Friedrich et al. // Proc. VLDB Endow. — 2020. — jul. — Vol. 13, no. 12. — P. 2270–2283. — URL: https://doi.org/10.14778/3407790.3407824.

[8] Kruse Sebastian, Naumann Felix. Efficient discovery of approximate dependencies // Proc. VLDB Endow. — 2018. — mar. — Vol. 11, no. 7. — P. 759–772. — URL: https://doi.org/10.14778/3192965.3192968.

[9] Pena Eduardo H. M., Porto Fabio, Naumann Felix. Fast Algorithms for Denial Constraint Discovery // Proc. VLDB Endow. — 2022. — dec. — Vol. 16, no. 4. — P. 684–696. — URL: https://doi.org/10.14778/3574245.3574254.

[10] Song Shaoxu, Chen Lei. Differential Dependencies: Reasoning and Discovery // ACM Trans. Database Syst. — 2011. — aug. — Vol. 36, no. 3. — 41 p. — URL: https://doi.org/10.1145/2000824.2000826.

[11] Синельников Михаил. Реализация алгоритма поиска дифференциальных зависимостей в рамках платформы Desbordante. — 2024. — URL: https://github.com/Desbordante/desbordante-core/blob/main/docs/papers/SPLIT-MichaelSinelnikov-2023autumn.pdf.