Санкт-Петербургский государственный университет

Кафедра программной инженерии

Группа 23М07-мм

Реализация алгоритма поиска запрещающих ограничений в платформе Desbordante

Морозко Иван Дмитриевич

Отчёт по учебной практике
в форме «Решение»

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

Санкт-Петербург
2025

Оглавление

Введение

В последние годы все большее значение приобретают методы обработки и анализа данных. К сожалению, данные, основанные на экспериментах из реального мира, зачастую оказываются неидеальными. В них могут содержаться ошибки, такие как пропущенные значения, неправильные значения и т. п. Недостоверная информация может привести к принятию неверного решения, поэтому очистке данных в современном мире уделяется особое значение [16].

В этом вопросе может помочь профилирование данных — процесс извлечения метаданных из набора данных. Методы профилирования, представляющие собой поиск некоторой формально описанной закономерности или правила будем называть примитивами. Например, одними из самых известных примитивов являются функциональные зависимости (ФЗ) [11]. Благодаря профилированию данных можно узнать внутренние закономерности в данных, и, благодаря ним, сделать выводы о возможных ошибках [7].

Одним из таких примитивов являются запрещающие ограничения (ЗО). Данный примитив обобщает и расширяет существующий класс ограничений для очистки данных [5]. Каждое ЗО представляет собой отношение между предикатами, указывающее, какие комбинации значений атрибутов являются несогласованными. Автоматических поиск таких закономерностей является трудоёмкой задачей [1], однако, ЗО обладают рядом преимуществ, по сравнению с другими примитивам. Во-первых, ЗО позволяют выражать ограничения, не способные быть выраженными другими примитивами. Во-вторых, в недавних исследованиях, ЗО использовались в качестве стандартного языка для описания ограничений данных [6, 9, 12]. Например, HOLOCLEAN [12] — это инструмент для очистки данных, входными данными которого являются наборы приближённых ЗО.

Существует множество алгоритмов поиска ЗО. Более подробно они будут рассмотрены далее, однако среди них выделяется алгоритм FastADC [10]. Он позволяет находить приближённые запрещающие огра-

ничения на порядок быстрее своих конкурентов. Данный алгоритм полностью написан на языке Java, и представляет собой отдельный проект, не включённый ни в какие известные профилировщики данных, такие как Metanome[1], вследствие чего выявляется ряд проблем. Первая проблема — проблема производительности. Исследование [8] показывает, что производительности Java-приложений не всегда достаточно для вычислений, проводящихся на больших объемах данных. Вторая проблема — удобство конечных пользователей. Зачастую инструментами очистки и профилировки данных пользуются аналитики, далёкие от мира разработки программного обеспечения. В таком случае, необходимость самостоятельной сборки и конфигурации алгоритма может стать препятствием на пути удобного использования.

Для преодоления упомянутых проблем был создан Desbordante[2], открытая платформа для профилирования данных, написанная на C++. Данный инструмент не поддерживает запрещающие ограничения. Поэтому было решено реализовать и интегрировать в Desbordante алгоритм поиска приближенных запрещающих ограничений — FastADC. Это предоставит открытую, высокопроизводительную и удобную для пользователей реализацию данного алгоритма.

[1] hpi.de/naumann/projects/data-profiling-and-analytics/metanome-data-profiling.html
[2] https://desbordante.unidata-platform.ru/

1 Постановка задачи

Целью данной учебной практики является реализация алгоритма FastADC поиска приближенных запрещающих ограничений в платформе Desbordante. Для её достижения в прошлом семестре были выполнены следующие задачи:

1. Провести обзор предметной области
2. Провести сравнение существующих алгоритмов поиска ЗО
3. Провести обзор алгоритма FastADC

Поскольку текущая учебная практика является продолжением предыдущей, в этом семестре были поставлены следующие задачи:

1. Реализовать однопоточную версию алгоритма в платформе Desbordante
2. Сравнить производительность реализованного алгоритма с оригинальной реализацией [17]

2 Обзор

2.1 Базовые определения

Опишем основные понятия, использующиеся для описания запрещающих ограничений и алгоритма их поиска.

Используются стандартные определения из реляционной модели данных: R — отношение; $r \in R$ — атрибут; $s,\ t \in r$ — кортежи.

Определение 1 (Предикат)**.** Запрещающие ограничения определяются на основе предикатов. Каждый предикат p имеет форму $t.A_i\, op\, s.A_j$, где $t, s \in r$, $t \neq s$ (различные кортежи), $A_i, A_j \in R$, и $op \in \{<, \leq, >, \geq, =, \neq\}$. Сравнение проводится по одному и тому же атрибуту, если $i = j$. В противном случае, оно проводится между двумя атрибутами.

Определение 2 (Запрещающие ограничения)**.** ЗО φ на атрибуте r определяется следующим образом: $\forall t, s \in r$, $\neg(p_1 \wedge \ldots \wedge p_m)$, где $p_1, \ldots, p_m$ являются предикатами для кортежей t, s. Мы говорим, что φ удерживается на r, если для каждой пары кортежей $(t, s) \in r$ по крайней мере один из $p_1, \ldots, p_m$ не удовлетворяется. Пара кортежей (t, s) нарушает φ, если пара удовлетворяет каждому $p_i\ : i \in [1, m]$.

Определение 3 (Мера ошибки)**.** Значение меры ошибки g_1 для данного ЗО φ на r, обозначаемое как $g_1(\varphi, r)$, определяется как отношение числа пар кортежей, нарушающих φ, к общему числу пар кортежей в r. То есть, $g_1(\varphi, r) = \frac{|\{(t,s)|(t,s)\in r^2 \wedge (t,s) \text{ нарушает } \varphi\}|}{|r|^2-|r|}$.

Определение 4 (Приближённое запрещающее ограничение)**.** При заданном пороге ошибки ϵ, φ является приближённым ЗО на r, если и только если $g_1(\varphi, r) \leq \epsilon$.

Определение 5 (Минимальное запрещающее ограничение)**.** ЗО φ является минимальным, если не существует отличного от φ ограничения φ', такого что (a) множество предикатов φ' является строгим подмножеством множества предикатов φ, и (b) φ' является приближённым ЗО на r.

Таблица 1: Пример таблицы.

A	B	C
1	2	фрукт
2	1	овощ
2	4	овощ

Приведём несколько примеров ЗО, удерживающихся на таблице 1.

Примитив (a) функциональная зависимость $A \to C$ может показать, что любые два кортежа, совпадающие на атрибуте A, совпадают на атрибуте C. Примитив (b) уникальная комбинация колонок (Unique Column Combination, UCC) может показать, что проекция по атрибутам A и B не имеет не имеет одинаковых кортежей. Оба этих примитива можно выразить в терминах ЗО:

(a) $\forall t, s \in r,\ \neg(t.A = s.A \wedge t.C \neq s.C)$

(b) $\forall t, s \in r,\ \neg(t.A = s.A \wedge t.B = s.B)$

(c) $\forall t, s \in r,\ \neg(t.A > s.B \wedge t.B \leq s.B)$

Так же, ЗО (c) показывает, что возможность использования не только операторов равенства/неравенства, но и числового сравнения, позволяет выражать и более сложные зависимости между данными.

2.2 Обзор существующих алгоритмов

Разработка алгоритмов поиска обычных и приближённых запрещающих ограничений ведётся с 2013 года с представления алгоритма FastDC [4], и постоянно открываются новые методы. Для оценки существующих алгоритмов поиска ЗО и выбора подходящего для реализации алгоритма в платформе Desbordante, было выделено несколько критериев отбора.

Критерии отбора

- Возможность поиска приближённых ЗО

  Несмотря на то, что нахождение приближённых ЗО является значительно более трудоёмкой задачей, именно они используются при очистке данных, в отличие от точных ЗО [5, 18].

- Возможность поиска гетерогенных ЗО

  Возможность использовать предикаты между различными атрибутами позволит увеличить объём информации, которую мы способны получить из данных.

- Нахождение минимальных ЗО

  Нахождение минимальных запрещающих ограничений предпочтительно, поскольку оно позволяет избежать избыточности. Алгоритмы, находящие все ЗО, во-первых, используют больше памяти, а во вторых, затрудняют очистку данных. Если ЗО φ удерживается на r, и существует ЗО ξ, следующее из φ, то в проверке ξ при очистке данных нет смысла.

- Наличие open-source реализации

  Наличие реализации с открытым кодом упрощает процесс реализации алгоритма на языке C++, и позволяет сравнить полученное решение с оригинальным.

- Эффективность по сравнению с предыдущими

  Самый эффективный алгоритм является и самым предпочтительным для реализации.

Оценка рассмотренных алгоритмов представлена на таблице 2. Алгоритмы представлены в порядке их выхода, от самого старого на первой строке, до state-of-the-art на последних.

Таблица 2: Оценка рассмотренных алгоритмов.

Алгоритм	Год	Приближённые ЗО	Гетерогенные ЗО	Минимальные ЗО	Открытый код	Эффективнее предыдущих	Константные ЗО
FastDC [4]	2013	-	+	+	+		+
A-FastDC [4]	2013	+	+	+	+	-	+
Hydra [3]	2017	-	+	+	+	+	-
BFastDC [14]	2018	+	+	+	-	+	+
DCFinder [15]	2019	+	+	+	+	+	-
ADCMiner [2]	2020	+	+	+	+	-	-
DCFinder+ [13]	2022	+	+	+	+*	+	-
FastADC [10]	2022	+	+	+	+	+	-

Возможность поиска ЗО, использующих в качестве одного из аргументов некое константное значение, была исследована, однако она не представляет должного интереса чтобы отметить её как необходимую. Данная колонка была помечена тёмно–зелёным цветом как не влияющая на выбор алгоритма.

Хотя авторами работы DCFinder+ была предоставлена открытая реализация[3], она, к сожалению, не поддерживает возможность поиска приближённых ЗО, описанную в статье. Таким образом, алгоритмами, удовлетворяющим всем критериям, стали DCFinder и FastADC. Однако последний является state-of-the-art алгоритмом. Показано, что FastADC быстрее DCFinder на порядок [10], вследствие чего и был выбран этот алгоритм.

2.3 Платформа Desbordante

Desbordante — это высокопроизводительный инструмент для профилирования данных с открытым исходным кодом, разработанный на C++ [8]. За счет использования низкоуровневых оптимизаций, таких как векторизация вычислений и специализированные аллокаторы, Desbordante обладает высокой производительностью.

Этот инструмент обладает удобным пользовательским интерфейсом, включая Python-привязки и frontend-клиент.

В настоящее время Desbordante активно развивается, постоянно пополняясь реализациями новых примитивов.

[3] https://github.com/eduardopena/fdcd

3 Алгоритм FastADC

На текущий момент все алгоритмы поиска приближённых ЗО следуют схеме, состоящей из двух этапов [4, 10, 13, 15]. На первом этапе по данному отношению R строится структура, называемая *набором свидетельств (evidence set, HC)*, а на втором этапе происходит поиск приближённых покрытий HC (подробности в разделе 3.3).

FastADC следует такой же схеме, но предлагает новое решение для построения HC, сначала создавая так называемый *набор подсказок* (clue set), а затем преобразуя его в HC, и предоставляя новый метод для нахождения приближённых ЗО из набора свидетельств — *приближённая инверсия свидетельства (Approximate Evidence Inversion, AEI)*.

Принцип работы алгоритма FastADC можно рассмотреть на Рис. 1.

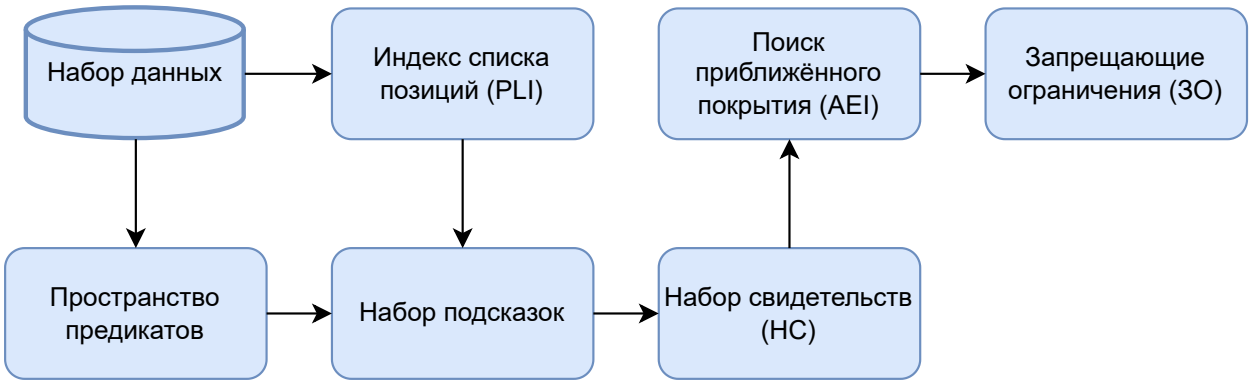

Рис. 1: Схема работы алгоритма FastADC.

3.1 Пространство предикатов

Первым этапом алгоритма FastADC служит построение так называемого пространства предикатов P — множества всех допустимых предикатов на R. Пространство предикатов строится, используя все операторы сравнения для числовых атрибутов, и операторы равенства/неравенства для категориальных. При этом используются только *сравнимые* атрибуты, т.е либо одинаковые, либо атрибуты одинакового типа, между которыми есть не меньше 30% общих значений, что позволяет избегать неинтересных предикатов.

Пример пространства предикатов можно увидеть на таблице 3. Данное пространство предикатов построено для таблицы 1, обсуждённой в разделе 2.1.

Таблица 3: Пространство предикатов для таблицы 1.

$p_1 : t.A = s.A$	$p_2 : t.A \neq s.A$
$p_3 : t.A > s.A$	$p_4 : t.A < s.A$
$p_5 : t.A \geq s.A$	$p_6 : t.A \leq s.A$
$p_7 : t.A = s.B$	$p_8 : t.A \neq s.B$
$p_9 : t.A > s.B$	$p_{10} : t.A < s.B$
$p_{11} : t.A \geq s.B$	$p_{12} : t.A \leq s.B$
$p_{13} : t.B = s.B$	$p_{14} : t.B \neq s.B$
$p_{15} : t.B > s.B$	$p_{16} : t.B < s.B$
$p_{17} : t.B \geq s.B$	$p_{18} : t.B \leq s.B$
$p_{19} : t.C = s.C$	$p_{20} : t.C \neq s.C$

3.2 Набор подсказок и свидетельств

Дадим определение набора свидетельств.

Определение 6 (Свидетельство)**.** Свидетельство $evi(t, s)$ для пары кортежей $(t, s) \in r^2$ представляет собой множество предикатов $p \subset P$, удовлетворяющихся на паре кортежей (t, s).

Определение 7 (Набор свидетельств)**.** НС определяется как множество свидетельств по всем парам кортежей: $\{evi(t, s) : (t, s) \in r^2\}$.

Обычно НС реализуется с помощью битовых шкал (bit set), что позволяет эффективно выполнять алгоритм поиска приближённых покрытий, описанный в разделе 3.3. Поскольку $evi(t, s) = \bigcup_{A,B \in R} \{\text{предикаты } p : p = (t.A \ op \ s.B)\}$, где $op \in \{=, \neq\}$, либо $\in \{=, \neq, <, >, \geq, \leq\}$, то для кодирования предиката для каждой пары категориальных атрибутов используется два бита, а для пары числовых атрибутов — шесть [14, 15].

Однако построение НС из пространства предикатов с данным подходом может быть затруднительно. Как будет показано далее, FastADC строит НС гораздо быстрее существующих алгоритмов, ввиду введения

новой структуры — набора подсказок. Подсказка $clue(t, s)$ кодирует отношения между кортежами сравнимых пар атрибутов более сжато, чем набор свидетельств, требует меньше памяти для хранения, и в то же время биективна $evi(t, s)$.

$clue(t, s)$, как и $evi(t, s)$, тоже описывает все предикаты, удовлетворяющиеся на паре кортежей (t, s), однако кодирует **одним** битом предикаты для категориальной пары атрибутов: 0, если $t.A \neq s.B$ (или $t.A \neq s.A$), или 1, если $t.A = s.B$ (или $t.A = s.A$); и **двумя** битами для численных атрибутов: 00, если $t.A < s.B$ (или $t.A < s.A$), 01, если $t.A = s.B$ (или $t.A = s.A$), или 10, если $t.A > s.B$ (или $t.A > s.A$).

Набор подсказок строится следующим образом. Изначально каждой паре кортежей присваивается начальная подсказка c_0, после чего подсказки корректируются для пар кортежей, чьи подсказки отличаются от начальной. Эффективность этого процесса достигается за счёт использования индекса списка позиций (Position List Index, PLI)[4] отношения r. Такой подход значительно эффективнее стандартного метода, сравнивающего все пары кортежей.

Выбор c_0 влияет на эффективность: меньшее количество бит требует коррекции, если c_0 имеет много общих бит с фактическими подсказками. Исходя из предположения что большинство кортежей имеют различные значения атрибутов [15], c_0 выбирается как битовая шкала со всеми значениями равными нулю, т.е., для пары кортежей (t, s) и пары атрибутов (A, B) предполагаем, что $t.A \neq s.B$ для категориальных пар атрибутов и $t.A < s.B$ для числовых пар.

Например, в таблице 1 сравнимыми парами атрибутов являются пары $(A, A), (C, C), (A, B)$ и (B, B). Тогда подсказка $clue(2, 3)$ будет выглядеть следующим образом:

$$clue(2,3) = \underbrace{01}_{\text{AA}}\ \underbrace{1}_{\text{CC}}\ \underbrace{00}_{\text{AB}}\ \underbrace{00}_{\text{BB}}$$

что соответствует предикатам $\{p_1, p_{19}, p_{10}, p_{16}\}$, соответственно.

В худшем случае сложность алгоритма квадратична по отношению

[4]Подробнее про PLI написано в разделе 4.2.

к $|r|$, но на практике многие пары кортежей могут быть пропущены благодаря PLI. Также, скорость данного алгоритма в частности достигается за счёт попадания подсказок в кеш процессора. Каждая подсказка может быть скорректирована несколько раз, при этом при коррекции модифицируется максимум один бит для каждой пары атрибутов: бит 0 может быть изменён только на 1 для категориальных пар атрибутов, а 00 может быть изменён только на 01 или 10 для числовых пар.

Генерация доказательств из подсказок происходит через простое преобразование битов [10]. Сложность этого линейна по отношению к размеру набора подсказок, а мощность этого множества $\ll |r|^2$, поскольку многие пары кортежей могут производить одинаковые подсказки, как было показано авторами работы [10]. Тем самым, время преобразования незначительно по сравнению с временем создания набора подсказок.

3.3 Приближённые покрытия набора свидетельств

Для начала определим понятие обычного и приближенного покрытия НС.

Определение 8 (Покрытие НС)**.** Покрытие НС — это подмножество X множества P, такое что $\forall e \in \text{НС}\ X \cap e \neq \emptyset$.

Авторы статьи [4], положившей начало алгоритмам поиска приближённых запрещающих ограничений, ввели и доказали следующую теорему:

Теорема 1 (Допустимость точного ЗО)**.** ЗО $\varphi = \neg(\overline{p_1} \wedge \cdots \wedge \overline{p_m})$ допустимо на r, тогда и только тогда, когда множество $\{p_1, \ldots, p_m\}$ является покрытием НС.

С помощью данной теоремы находятся точные запрещающие ограничения. Авторы той же работы [4] также расширяют это определение для приближённых ЗО.

Определение 9 (Функция cnt)**.** Разные пары кортежей могут производить одинаковые свидетельства. Функция $cnt(e)$ для свидетельства e обозначает число пар кортежей (t, s), таких что $evi(t, s) = e$.

Определение 10 (Приближённое покрытие НС)**.** Приближенное покрытие НС это такое подмножество $X \subset P$, что для всех $e \in$ НС, таких что $X \cap e \neq \emptyset$, $\sum cnt(e) \geq (1-\epsilon) \times (|r|^2 - |r|)$.

Теорема 2 (Валидность приближённого ЗО)**.** При данном пороге ошибки ϵ, приближённое ЗО $\varphi = \neg(\overline{p_1} \wedge \cdots \wedge \overline{p_m})$ допустимо на r, тогда и только тогда, когда $\{p_1, \ldots, p_m\}$ является приближённым покрытием НС.

Иными словами, множество инвертированных предикатов ЗО должно пересекаться с «достаточным» количеством свидетельств (не обязательно со всеми).

Прежде чем перейти к методу приближенной инверсии доказательств (AEI), введем некоторые обозначения.

- Для ЗО $\phi = \neg(p_1 \wedge \ldots \wedge p_m)$ и свидетельства e, $\phi \subseteq e$, означает что набор предикатов ϕ, т.е., $\{p_1, \ldots, p_m\}$, является подмножеством e. Очевидное следствие: $\phi \subseteq e(t,s) \Rightarrow \phi$ нарушается парой кортежей (t,s) (но не обязательно наоборот).

- Говорим, что ЗО ϕ *покрывает* $e(t,s)$, если $\phi \nsubseteq e$, и ϕ не нарушается парой кортежей (t,s).

- Для двух ЗО ϕ и ϕ' мы пишем $\phi \subseteq \phi'$, если набор предикатов ϕ является подмножеством набора предикатов ϕ'. Очевидное следствие: ϕ' не является минимальным, если $\exists \phi : \phi \subsetneq \phi'$ и ϕ действителен.

- $\phi \cup \{p\}$ обозначает ЗО, получаемый добавлением нового предиката p к ϕ. Данный процесс называется *уточнением* ϕ.

3.4 Алгоритм AEI

Основная идея алгоритма AEI выглядит следующим образом. Определятся начальный набор ЗО Σ, который постепенно модифицируется, и в итоге становится ответом. Этот набор инициализируется запрещающими ограничениями, состоящими из одного предиката:

$\Sigma \overset{init}{=} \{\varphi : \varphi = \neg(p)\ \forall p \in P\}$. На каждой итерации алгоритма обрабатывается одно свидетельство e из НС следующим образом: для каждого $\varphi \in \Sigma$ проверяется, *покрывает* ли оно e. Если φ *не покрывает* свидетельство, алгоритм *пытается покрыть* φ e: к φ добавляются новые предикаты из P, которые не входят в e, создавая новые кандидатные ЗО. После каждого добавления нового предиката к φ проверяется, остается ли полученное ЗО минимальным. Если нет, ЗО исключается дальнейшего рассмотрения. Итерации повторяются пока каждое свидетельство не будет рассмотрено.

Однако данное описание сможет найти только точные ЗО, поскольку при генерации кандидатов приближенных ЗО необязательно разрешать все нарушения ЗО относительно каждого свидетельства e. Вместо этого должно учитываться количество ($cnt(e)$) для каждого e при генерации или отсечении кандидатов.

Модификации для нахождения приближённых ЗО

Каждое кандидатное ЗО φ хранится в паре с множеством предикатов *cand*, которые могут быть использованы для *уточнения* φ. Множество таких пар называется Ψ.

Как и раньше, на каждой итерации рассматривается свидетельство $e \in$ НС. Все кандидатные ЗО из Ψ делятся на две части в зависимости от того, покрывают ли они e. ЗО которые не покрывают e, помещаются в Ψ^-, в то время как остальные остаются в Ψ.

Затем AEI пытается покрыть ЗО $\varphi \in \Psi^-$, как и ранее описанный алгоритм, но в этот раз сначала с использованием *cnt* проверяется, может ли этот путь привести к корректному ЗО. Вычисляется количество оставшихся свидетельств, которые могут быть покрыты оставшимися предикатами, т.е. верхняя граница накопленного количества, и данное число сравнивается с N — суммарным числом свидетельств, которые обязаны быть покрытыми (инициализируется выражением $(1-\epsilon)\times(|r|^2-|r|)$ из определения 10). Если полученное число меньше N, то из *cand* удаляется e. Если после этого *cand* представляет собой пустое множество, то φ больше невозможно *уточнять*. Происходит проверка того что φ

является минимальным и приближённым ЗО, и φ добавляется в Σ. Происходит рекурсивный вызов, которому передаётся Ψ^- и следующее свидетельство.

В ветви, где e покрыто, собирается отдельное множество предикатов $\{\bigcup_{\{\_,cand\}\in\Psi^-} cand \setminus e\}$. Для каждого предиката p из этого множества формируется новое кандидатное ЗО путём *уточнения* $\varphi \in \Psi$: $\varphi' = \varphi \cup \{p\}$, и определяется набор $cand'$ с предикатами из $cand$, определённых на тех же парах атрибутов что и p. Если такие предикаты существуют, то Ψ обновляется — в него добавляется новая пара $\{\varphi', cand'\}$, если, конечно, φ' минимальна в Ψ, т.е $\nexists \varphi \in \Psi : \varphi \subseteq \varphi'$.

Если же $cand'$ представляет собой пустое множество, происходит то же самое что и в первой ветви: проверка того, что φ' является минимальным и приближённым ЗО; добавление φ' в Σ.

Обе ветви пройдены. N уменьшается на $cnt(e)$ и метод рекурсивно вызывается, в этот раз для Ψ.

Более подробно алгоритм AEI можно рассмотреть в работе [10].

4 Реализация

Были реализованы необходимые классы для построения пространства предикатов, индекса списка позиций, набора подсказок, набора свидетельств и приближенных инверсий свидетельств.

Рассмотрим ближе отдельные блоки алгоритма.

4.1 Пространство предикатов

Пространство предикатов представляет собой список классов *Predicate*. Для построения класса *Predicate*, описывающего предикат $p = t.A\,op\,s.B$, были построены классы *ColumnOperand* и *Operator*.

Алгоритм построения пространства предикатов проходится по каждой паре атрибутов и вычисляет, являются ли атрибуты сравнимыми. Напомним, что *сравнимые* атрибуты, это либо одинаковые атрибуты, либо атрибуты одинакового типа, между которыми есть не меньше 30% общих значений. Были реализованы необходимые функции, возвращающие метрику сравнимости для атрибутов категориальных и числовых типов.

4.2 Индекс списка позиций (PLI)

Каждый PLI строится для атрибута $A \in R$. При этом предварительно значения атрибута хешируются, что позволяет говорить о значении кортежа на атрибуте A как об обычном числе — ключе, не заботясь о настоящем типе атрибута.

Определение 11 (Кластер)**.** Кластером в контексте PLI мы называем пару $\langle k, l \rangle$, где ключ k является значением в атрибуте A, а l — множество всех кортежей, имеющих то же значение k в атрибуте A.

Определение 12 (Индекс списка позиций (PLI))**.** Для атрибута A, класс PLI является тройкой:

- Множество ключей уникальных значений атрибута A
- Список кластеров
- Отображение ключ $\rightarrow$ кластер

Для числового атрибута A, кластеры в PLI дополнительно сортируются по ключу k в порядке убывания, что позволит оптимизировать построение набора подсказок путём использования двоичного поиска.

Данный класс был реализован, наряду с методом хеширования значений атрибутов таблицы.

4.3 Набор подсказок и свидетельств

Для построения набора подсказок было реализовано два вспомогательных класса *SingleClueSetBuilder* и *CrossClueSetBuilder*. Набор подсказок строится по описанному ранее алгоритму в разделе 3.2, принимая в себя пространство предикатов и PLI в качестве входных данных. В цикле по каждой паре атрибутов строится промежуточный набор подсказок, после чего данный промежуточный набор добавляется в итоговый.

Было реализовано два класса, поскольку логика построения промежуточного набора подсказок для разных атрибутов отличается от логики построения для одинаковых атрибутов.

Набор свидетельств принимает в качестве входных данных набор подсказок и пространство предикатов, и строит свидетельства через простое преобразование битов [10].

4.4 AEI

Помимо классов *DenialConstaint* и *DenialConstaintSet*, нужных для представления итогового результата, для реализации алгоритма AEI, описанной в разделе 3.4, были реализованы вспомогательные классы *PredicateOrganizer*, *Closure* и *DCCandidateTrie*.

- *PredicateOrganizer*

  Данный класс отвечает за преобразование и упорядочивание предикатов для оптимизации процесса инверсии свидетельств.

  Определение 13 (Покрытие предиката)**.** Покрытием предиката называется количество свидетельств, в которых присутствует данный предикат.

  Для каждого предиката класс вычисляет его покрытие, и перестраивает порядок предикатов таким образом, чтобы предикаты с меньшим покрытием обрабатывались первыми. Это позволяет уменьшить размер пространства поиска и повысить эффективность AEI, так как потенциально проблемные предикаты будут рассматриваться в первую очередь.

- *Closure*

  Данный класс принимает множество предикатов в качестве входных данных, и добавляет к нему все предикаты которые можно получить из данного множества. Например, к предикатам $A < B$ и $B < C$, *Closure* добавит предикат $A < C$ как транзитивный, предикат $A \leq B$ как импликацию $A < B$, и все симметричные им. Это необходимо, поскольку все логически вытекающие предикаты должны быть учтены при проверке минимальности и валидности запрещающих ограничений.

- *DCCandidateTrie*

  Данный класс реализует префиксное дерево (trie) для эффективных операций над множеством кандидатов ЗО, таких как определение того, содержит ли дерево кандидат, являющийся подмножеством данного кандидата, или удаление из дерева всех кандидатов, являющихся обобщениями текущего свидетельства.

4.5 Диаграмма классов

На Рис. 2 представлена UML–диаграмма классов, описывающая структуру решения.

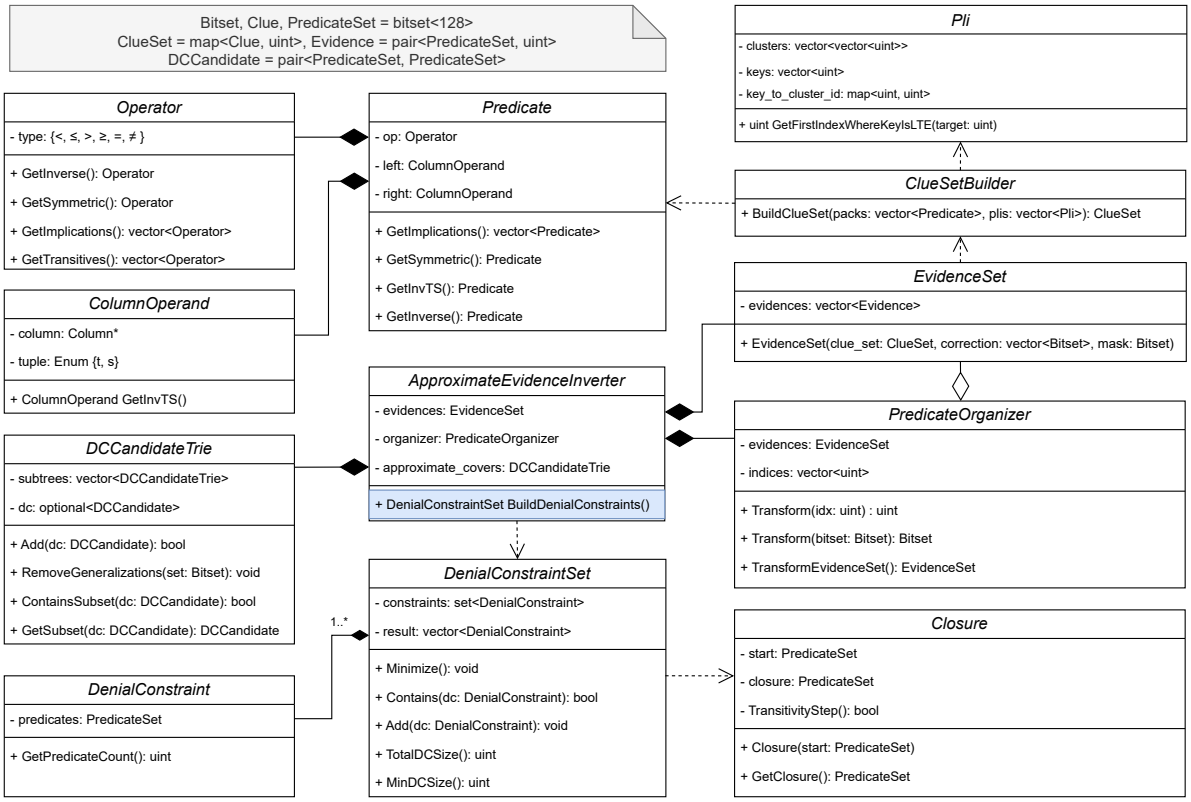

Рис. 2: UML–диаграмма классов алгоритма FastADC.

Синим цветом выделена основная функция построения набора запрещающих ограничений. Серым цветом выделена пояснительная записка с указанием используемых псевдонимов.

5 Результаты

Для того чтобы оценить производительность полученной реализации были проведены эксперименты. Измерялось время вычисления набора свидетельств, время работы алгоритма AEI и затраты памяти на наборах данных, представленных на таблице 4. Обе реализации выводят на экран время работы соответствующих частей без учёта обработки входного файла, что и использовалось в качестве показателя производительности по времени. Память же измерялась на отдельных запусках, где каждые 100 миллисекунд собиралась информация о затратах памяти командой `ps -o rss= -p $PID`. Для каждого набора данных проводилось десять запусков, с полученных данных бралось среднее значение с доверительным интервалом 95%. Desbordante компилировался с флагом оптимизации `-O3`. Авторами реализации на Java не была предоставлена дополнительная информация о конфигурации виртуальной машины Java, поэтому была использована LTS версия JDK с настройками виртуальной машины по умолчанию.

Эксперименты запускались на ненагруженном тестовом стенде, представленном на таблице 5, без других тяжеловесных процессов. Поскольку сравнивались однопоточные реализации, процессы связывались с конкретным ядром посредством команды `taskset -c $CORE`, и для данного ядра фиксировалась максимальная частота посредством команды `cpupower -c $CORE frequency-set`. Так же, для минимизации влияния операционной системы отключался файл подкачки через `swapoff -a` и между запусками очищались кеши путём записи `3` в файл `/proc/sys/vm/drop_caches`.

Таблица 4: Характеристики наборов данных, использовавшихся для проведения экспериментов.

Название	**Атрибуты**	**Строки**	**Размер**	**# ЗО**
Airport	11	55113	5.74 Мб	122
Atom	13	147067	7.47 Мб	3835
Food	12	200000	71.80 Мб	65
SPStock	6	122496	4.02 Мб	315
Neighbors	7	10000	647.51 Кб	2236

Таблица 5: Характеристики тестового стенда.

Компонент	**Спецификация**
Процессор	12th Gen Intel Core i7-12800H @ 4.7 GHz
Оперативная память	32GiB DDR5 @ 4800 MHz
Запоминающее устройство	NVMe SED Samsung 512GB
Операционная система	Ubuntu 22.04.5 LTS
Ядро Linux	6.8.0-45-generic
Версия OpenJDK	21.0.3 2024-04-16 LTS
Версия GCC	11.4.0

Результаты представлены на Рис. 3, Рис. 4, Рис. 5 и Рис. 6.

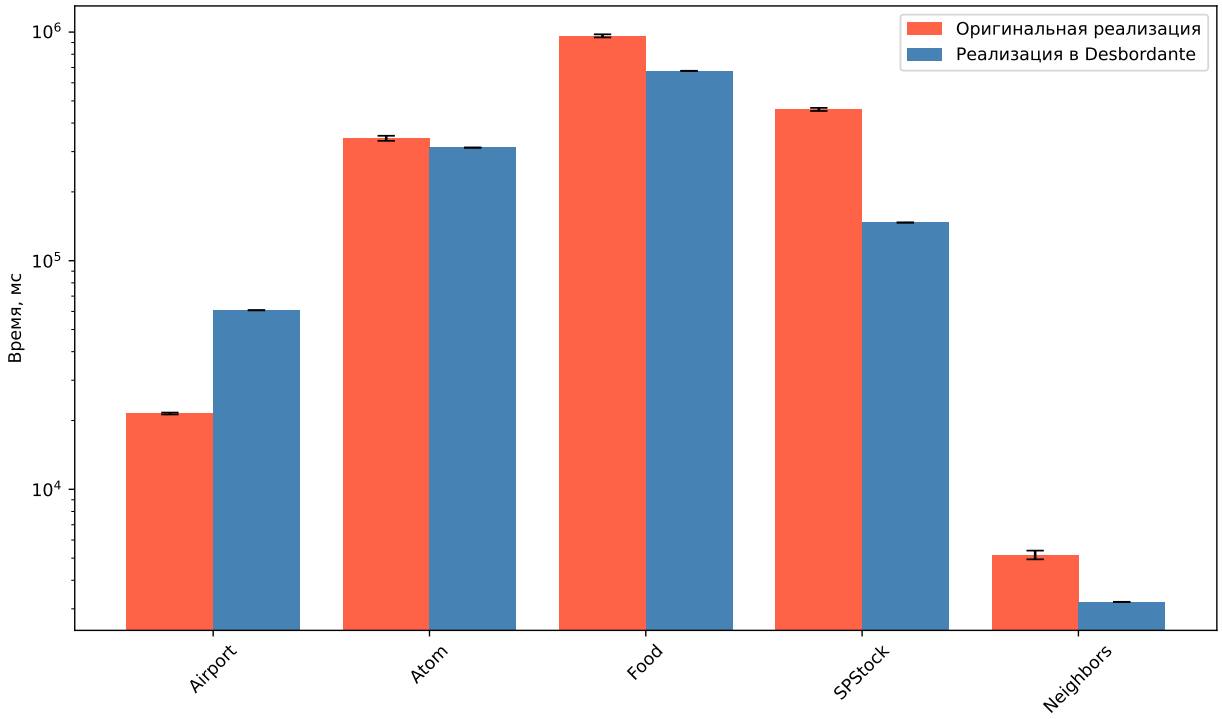

Рис. 3: Суммарное время выполнения FastADC.

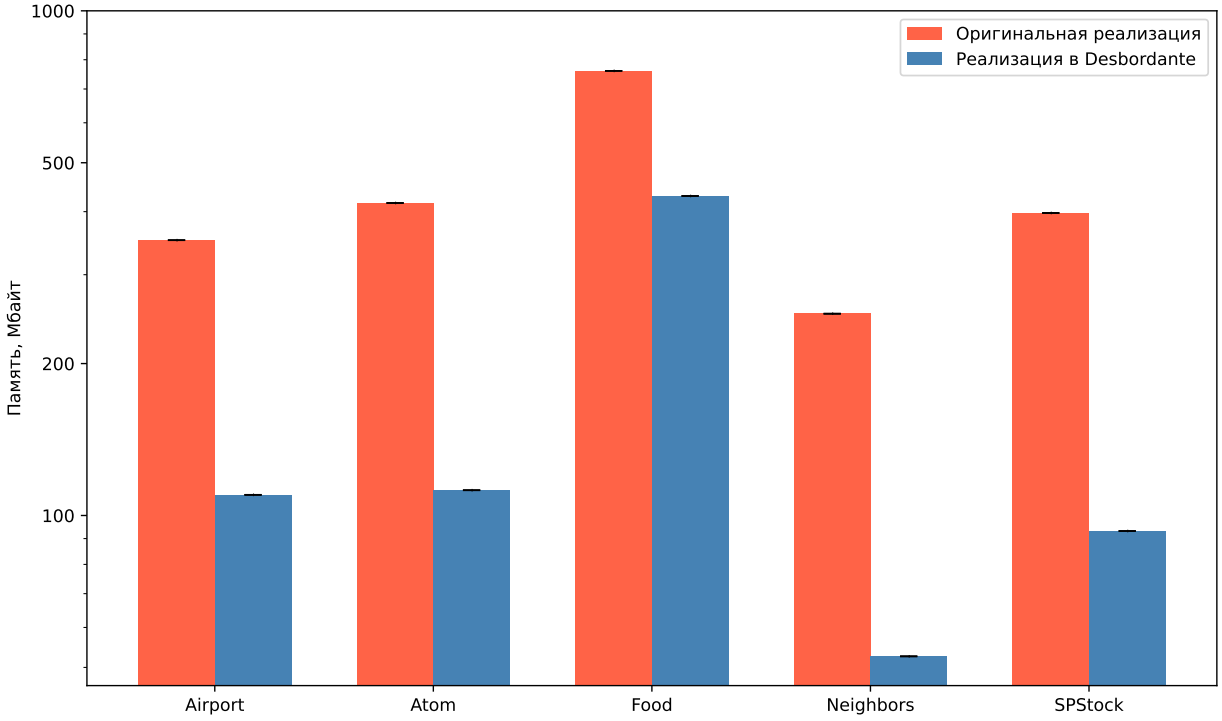

Рис. 4: Затраты памяти.

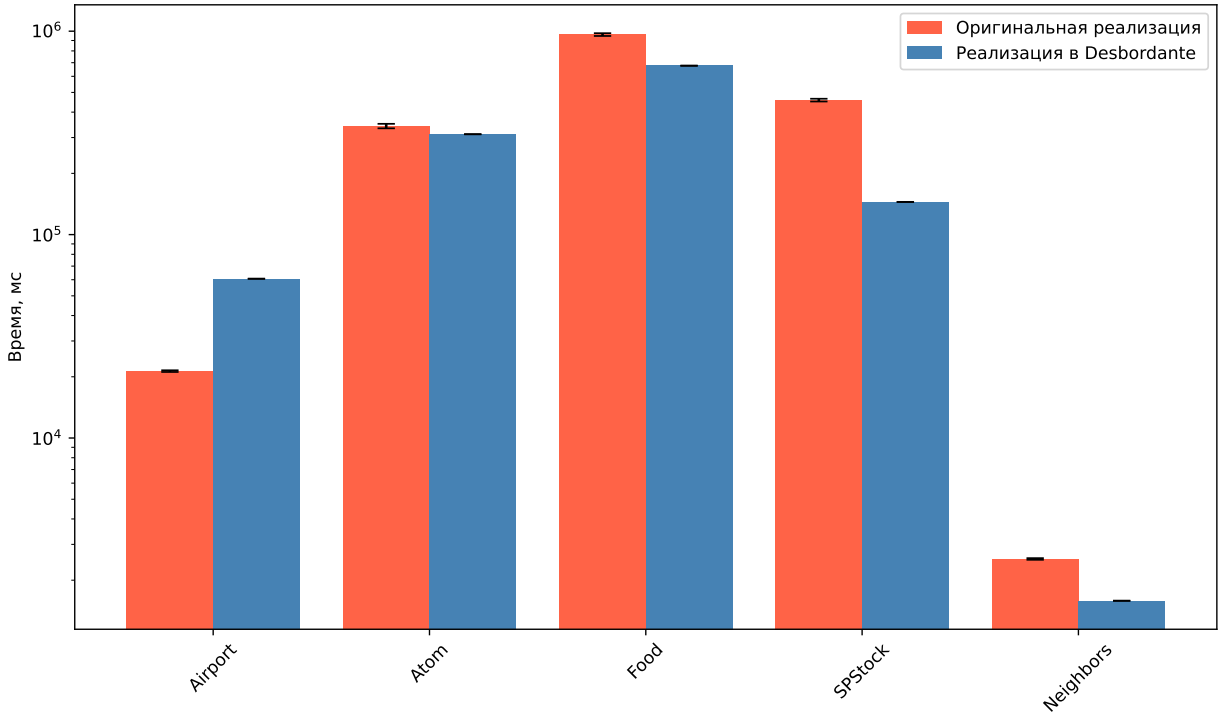

Рис. 5: Время построения набора свидетельств.

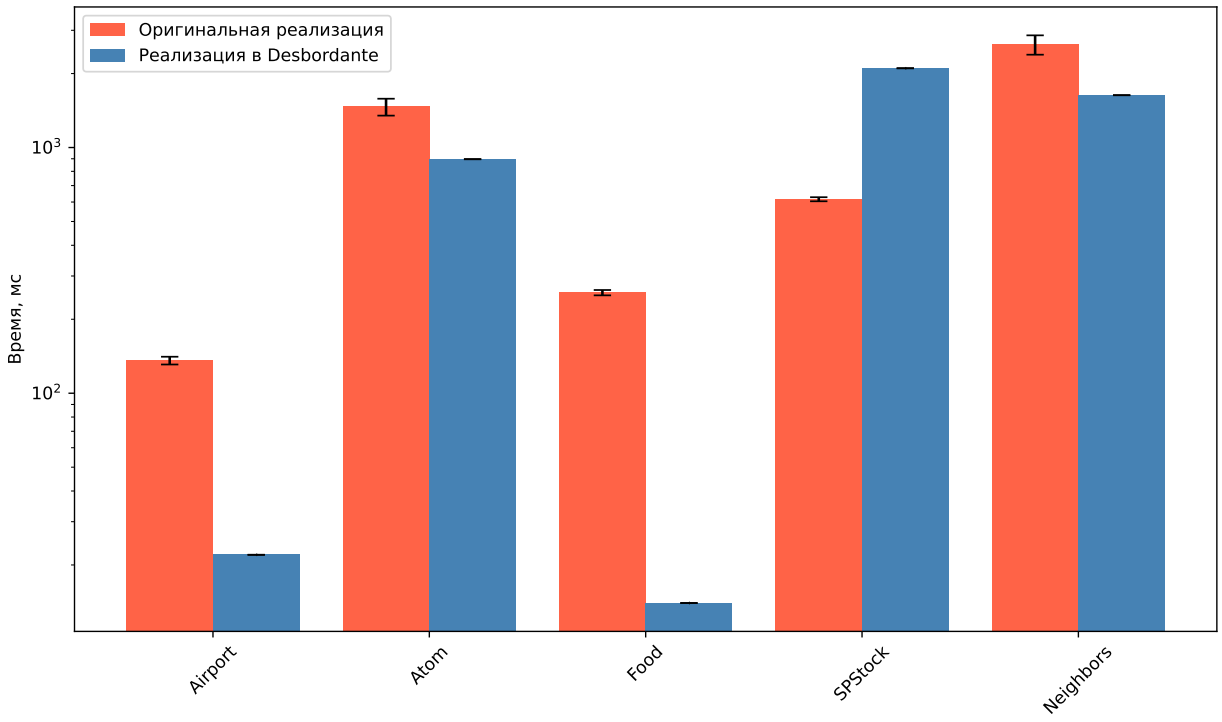

Рис. 6: Время выполнения алгоритма AEI.

Можно заметить, что реализация в Desbordante показала ускорение по сумме времени построения набора свидетельств и выполнения AEI на четырёх из пяти наборов данных. Ускорение составило от 1.1 раза на наборе данных Atom до 3.13 раза на наборе данных SPStock. В среднем ускорение составило 1.81 раза. На наборе данных Airport реализация Desbordante оказалась медленнее в 2.85 раз.

Экономия по памяти составила от 1.78 раза на наборе данных Food до 7.17 раза на наборе данных Neighbors. В среднем экономия составила 4.05 раза.

При построении набора свидетельств ускорение составило от 1.1 раза на наборе данных Atom до 3.17 раза на наборе данных SPStock. В среднем ускорение составило 2.03 раза.

При выполнении алгоритма AEI ускорение составило от 1.61 раза на наборе данных Neighbors до 18.33 раза на наборе данных Food. В среднем ускорение составило 6.23 раза.

Реализация алгоритма FastADC в платформе Desbordante не оптимизирована. На будущее отводится исследование возможностей оптими-

зации алгоритма и реализация многопоточной версии.

6 Заключение

В предыдущем семестре были выполнены следующие задачи:

- Проведён обзор предметной области
- Выбран набор критериев, по которым проведён обзор алгоритмов поиска ЗО
- Проведён обзор алгоритма FastADC

В результате текущей проделанной работы были выполнены следующие задачи:

- Реализован алгоритм FastADC в платформе Desbordante
- Проведено сравнение реализованного алгоритма с оригинальной реализацией
  - Получено ускорение по времени в среднем в 1.81 раз
  - Получена экономия по памяти в среднем в 4.05 раза

Код находится в репозитории[5].

[5] https://github.com/Desbordante/desbordante-core/pull/470

Список литературы

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

[2] Livshits Ester, Heidari Alireza, Ilyas Ihab F., Kimelfeld Benny. Approximate Denial Constraints. — 2020. — 2005.08540.

[3] Bleifuß Tobias, Kruse Sebastian, Naumann Felix. Efficient denial constraint discovery with hydra // Proc. VLDB Endow. — 2017. — Vol. 11, no. 3. — P. 311–323. — URL: https://doi.org/10.14778/3157794.3157800.

[4] Chu Xu, Ilyas Ihab F., Papotti Paolo. Discovering denial constraints // Proc. VLDB Endow. — 2013. — Vol. 6, no. 13. — P. 1498–1509. — URL: https://doi.org/10.14778/2536258.2536262.

[5] Chu Xu, Ilyas Ihab F., Papotti Paolo. Holistic data cleaning: Putting violations into context // 2013 IEEE 29th International Conference on Data Engineering (ICDE). — 2013. — P. 458–469.

[6] Cleaning inconsistencies in information extraction via prioritized repairs / Ronald Fagin, Benny Kimelfeld, Frederick Reiss, Stijn Vansummeren // Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. — PODS '14. — New York, NY, USA : Association for Computing Machinery, 2014. — P. 164–175. — URL: https://doi.org/10.1145/2594538.2594540.

[7] Conditional Functional Dependencies for Data Cleaning / Philip Bohannon, Wenfei Fan, Floris Geerts et al. // 2007 IEEE 23rd International Conference on Data Engineering. — 2007. — P. 746–755.

[8] 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.

[9] Fan Wenfei, Geerts Floris, Wijsen Jef. Determining the Currency of Data // ACM Trans. Database Syst. — 2012. — Vol. 37, no. 4. — 46 p. — URL: https://doi.org/10.1145/2389241.2389244.

[10] Fast approximate denial constraint discovery / Renjie Xiao, Zijing Tan, Haojin Wang, Shuai Ma // Proc. VLDB Endow. — 2022. — Vol. 16, no. 2. — P. 269–281. — URL: https://doi.org/10.14778/3565816.3565828.

[11] 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.

[12] HoloClean: holistic data repairs with probabilistic inference / Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, Christopher Ré // Proc. VLDB Endow. — 2017. — Vol. 10, no. 11. — P. 1190–1201. — URL: https://doi.org/10.14778/3137628.3137631.

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

[14] Pena Eduardo H. M., de Almeida Eduardo Cunha. BFASTDC: A Bitwise Algorithm for Mining Denial Constraints // Database and Expert Systems Applications / Ed. by Sven Hartmann, Hui Ma, Abdelkader Hameurlain et al. — Cham : Springer International Publishing, 2018. — P. 53–68.

[15] Pena Eduardo H. M., de Almeida Eduardo C., Naumann Felix. Discovery of approximate (and exact) denial constraints // Proc. VLDB Endow. — 2019. — Vol. 13, no. 3. — P. 266–278. — URL: https://doi.org/10.14778/3368289.3368293.

[16] Rahm Erhard, Do Hong Hai. Data Cleaning: Problems and Current Approaches. // IEEE Data(base) Engineering Bulletin. — 2000. — Vol. 23. — P. 3–13. — URL: https://api.semanticscholar.org/CorpusID:260972099.

[17] RangerShaw. FastADC: An Efficient Algorithm for Dependency Constraints. — https://github.com/RangerShaw/FastADC. — 2024. — GitHub repository.

[18] Towards an End-to-End Human-Centric Data Cleaning Framework / El Kindi Rezig, Mourad Ouzzani, Ahmed K. Elmagarmid et al. // Proceedings of the Workshop on Human-In-the-Loop Data Analytics. — HILDA '19. — New York, NY, USA : Association for Computing Machinery, 2019. — 7 p. — URL: https://doi.org/10.1145/3328519.3329133.