Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 23.М04-мм

Реализация поддержки приближенных зависимостей включения и зависимостей включения подобия в рамках профилировщика Desbordante

ЧИЖОВ Антон Игоревич

Отчёт по преддипломной производственной практике
в форме «Решение»

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

Санкт-Петербург
2025

Оглавление

Введение

Профилирование данных [1] является комплексным аналитическим процессом, цель которого заключается в выделении ключевых метаданных. Наиболее интересным для изучения является наукоёмкое профилирование, связанное с поиском зависимостей в данных. Наиболее известный пример — функциональные зависимости, которые являются ограничениями целостности в базах данных [3]. Данные зависимости, например, могут использоваться для проверки того, находится ли отношение в третьей нормальной форме [3].

Хотя ограничения целостности, такие как функциональные зависимости, успешно описывают семантику данных, некоторые данные в базе данных могут не соответствовать этим ограничениям. Одной из причин является тот факт, что данные могут поступать из различных независимых источников. В отличие от традиционных функциональных зависимостей, зависимости включения [7, 8] (Inclusion Dependency, IND) могут использовать разные отношения для проверки корректности данных.

Неформально, между двумя наборами колонок из двух (необязательно разных) отношений существует зависимость включения, если для каждого кортежа из первого набора атрибутов в первом отношении существует такой же кортеж из второго набора атрибутов во втором отношении.

Зависимости включения могут [2] использоваться для обнаружения несогласованности данных и восстановления целостности. Однако, данные зависимости не могут использоваться для обнаружения отсутствующих ограничений и проблем с целостностью данных. Для решения подобных задач могут использоваться расширения зависимостей включения, которые определяются подобно расширениям функциональных зависимостей.

Desbordante [5] представляет собой научно-ориентированный профилировщик данных с открытым исходным кодом[1]. Создание Desbordante было мотивировано недостатками существующей плат-

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

формы Metanome [4]. В Desbordante в качестве основного языка используется C++, а в Metanome — Java, из-за чего Metanome имеет не самую оптимальную производительность. Desbordante включает в себя алгоритмы для поиска различных примитивов и также предоставляет соответствующие интерфейсы для пользователей.

В данный момент Desbordante активно развивается как инструмент для анализа данных и требует расширения набора поддерживаемых примитивов, в том числе поддержку расширений зависимостей включения для добавления возможности обработки данных, содержащих ошибки и опечатки. Это послужило мотивацией для реализации поддержки новых типов закономерностей.

1. Постановка задачи

Целью данной работы является расширения поддержки зависимостей включения для работы с грязными данными. Для её выполнения были поставлены следующие задачи.

- Выполнить обзор предметной области и выбрать подходы к обработке неточностей в данных.
- Выбрать алгоритмы для реализации.
- Реализовать выбранные алгоритмы в рамках платформы Desbordante.
- Выполнить тестирование алгоритмов.
- Провести экспериментальное исследование алгоритмов.

2. Обзор предметной области

В данной главе вводятся базовые определения, выполняется обзор существующих алгоритмов и рассматриваются алгоритмы, выбранные для исследования и реализации.

Определение 1. ***Зависимость включения (IND)*** *определяется следующим образом:*

$$IND : R_1(X) \subseteq R_2(Y),$$

где X и Y это наборы атрибутов из отношений R_1 и R_2 соответственно. Это означает, что для каждого кортежа с атрибутами X в отношении R_1 существует такой же кортеж с атрибутами Y в отношении R_2.

Другими словами, $\forall t_1 \in \pi_X(R_1)\ \exists t_2 \in \pi_Y(R_2) : t_1 = t_2$.

Если множества X и Y содержат по одному атрибуту, то зависимость называется унарной; если больше одного — n-арной.

Определение 2. ***Приближённая зависимость включения (AIND)*** *определяется следующим образом:*

$$AIND : R_i(X) \subseteq_\epsilon R_j(Y),$$

где пороговое значение ϵ задаёт максимальное значение ошибки для функции g'_3, т.е. $g'_3(R_i(X) \subseteq R_j(Y)) \leq \epsilon$.

Пусть d это база данных над схемой базы данных R и r_i, $r_j \in d$ для соответствующих отношений R_i, $R_j \in \boldsymbol{R}$. Тогда функция g'_3 вычисляет соотношение минимального количества кортежей, которые необходимо удалить из r, чтобы $R_i(X) \subseteq R_j(Y)$ выполнялась над r.

$$g'_3(R_i(X) \subseteq R_j(Y)) =$$
$$= 1 - \frac{max\{|\pi_X(r')| : r' \subseteq r_i\ u\ (d - \{r_i\}) \cup \{r'\} \models R_i(X) \subseteq R_j(Y)\}}{|\pi_X(r_i)|}$$

r' это подмножество кортежей в r_i, которые не нарушают зависимость.

Определение 3. ***Зависимость включения подобия [10] (Similarity Inclusion Dependency, SIND)*** *определяется следующим образом:*

$$SIND : R_1[X] \subseteq_\sigma R_2[Y],$$

где:

1. *X и Y являются наборами атрибутов отношений R_1 и R_2, соответственно.*

2. $\forall t_1 \in \pi_X(R_1)\ \exists t_2 \in \pi_Y(R_2) : t_1 \approx_\sigma t_2$

   Другими словами, для каждого зависимого значения существует подобное ссылочное значение с учётом меры и порога ошибки.

Сравнение IND, AIND и SIND. Для сравнения различных зависимостей включений рассмотрим следующие таблицы:

name	**instructor**
Machine Learning	Dr. Smith
Data Structures	Dr. Johnson
Algorithms	Dr. Brown
Databases	Dr. Lee

Таблица 1: courses

student_id	**course_name**
1024	Machine Learning
1031	Machine Learnigg
1045	Data Structures
1077	Algoritms
1102	Databases
1140	Data Structurms

Таблица 2: enrollments

В таблица 1 определяется соответствие между курсами и преподавателями, которые ведут эти курсы, а в таблице 2 содержится информация о курсах, которые выбрали студенты. При этом часть студентов допустила опечатки в названиях курсов (ячейки с ошибочными значениями выделены красным цветом).

Предположим, что мы хотим исследовать следующую зависимость:

$$enrollment[course\_name] \subseteq course[name]$$

IND. Среди зависимых значений встречаются опечатки, из-за чего зависимость не выполняется. В таких случаях использование точных зависимостей включения не помогает анализировать данные.

AIND. В случае приближённых зависимостей необходимо подсчитать ошибку, с которой зависимость удерживается. Множество уникальных зависимых значений равняется шести, а количество неправильных уникальных значения равняется трём. Соответственно ошибка равняется $\frac{3}{6} = 0.5$. Соответственно для того, чтобы обнаружить данную зависимость необходимо использовать низкий порог ошибки. Однако использование настолько низкого порога ошибки на практике ведёт к большому количеству лишних зависимостей, которые не дают никакой дополнительной информации о данных, а скорее являются случайностью. Проблема заключается в том, что приближённые зависимости включения не учитывают особенности ошибок и не дают никакой дополнительной информации о них.

SIND. Для зависимости включения подобия будем использовать расстояние Левенштейна. Можно заметить, что в зависимой части в каждом ошибочном значении содержится максимум одна опечатка. Соответственно данная зависимость будет найдена с минимальным порогом ошибки, т.е. при $\sigma = 1$. Для работы с такими данными удобно использовать зависимости включения подобия с различными мерами схожести в зависимости от природы данных.

3. Обзор алгоритмов

3.1. Алгоритмы поиска IND и AIND

В работе [9] проведён полноценный обзор алгоритмов поиска IND. Алгоритмы можно разделить на n-арные и унарные.

Основная идея при реализации унарных алгоритмов заключается в отсекании кандидатов с помощью использования статистик колонок для того, чтобы избежать дорогостоящих проверок кандидатов. Некоторые алгоритмы используют инвертированные индексы, дисковая сортировка (`disk-based sort-merge-join`).

N-арные алгоритмы поиска IND начинают поиск с унарных зависимостей. Идея состоит в обходе решётки кандидатов. При обходе решётки происходит отсечение кандидатов (используется свойство антимонотонности). Существуют расширения данной идеи: обходы снизу вверх и сверху вниз, проверка кандидатов с использованием SQL.

На данный момент существует только один алгоритм для поиска AIND Mind [11]. Mind — это n-арный алгоритм поиска IND, который использует поуровневый обход по решётке снизу вверх с отсечением кандидатов. Он может принимать в качестве входных данных посчитанные ранее унарные зависимости включения. При этом авторы данной статьи приводят информацию о том, как можно обобщить данный алгоритм для поиска AIND. Также алгоритм Mind использует СУБД для генерации кандидатов и их проверки.

Также Mind определяет свой алгоритм для поиска унарных зависимостей. Данный алгоритм похож на алгоритм Spider [6], но кандидаты-колонки группируются по типам. Поскольку Mind определяется в контексте СУБД, то информация о типах колонок известна в самом начале работы алгоритма. В случае, когда работа происходит вне СУБД, сначала необходимо вывести типы колонок. Но на практике колонки могут иметь смешанные типы, что может усложнить стадию валидации кандидатов.

Spider может быть обобщён для поиска AIND. Поэтому было реше-

но в рамках данной работы реализовать алгоритм SPIDER, модифицировав его. После реализации SPIDER'а решено реализовать MIND, поскольку данный алгоритм является единственным алгоритмом для поиска AIND.

3.2. Алгоритм Spider

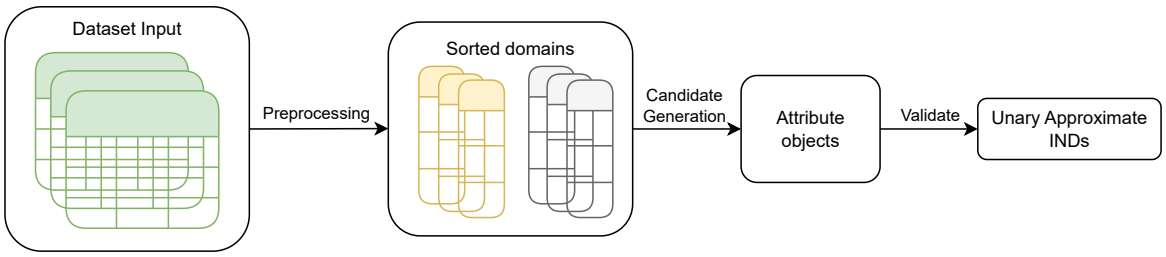

Рис. 1: Алгоритм Spider

На рисунке 1 показаны стадии работы алгоритма SPIDER. Сначала выполняется обработка входных данных, затем генерация кандидатов и их проверка.

1. Обработка входных данных.

На этой стадии SPIDER последовательно обрабатывает колонки входных таблиц. Обработка колонки заключается в поиске её отсортированного домена. По итогам работы алгоритма для каждой колонки создаётся отдельный колоночный файл на диске.

При этом, на данном этапе алгоритм может выполнять свопинг промежуточных результатов на диск при достижении ограничений по памяти с последующим их соединением (схожий подход используется операторами внешней сортировки в СУБД). Далее промежуточные результаты (отсортированный домен для части таблицы) будут называться партициями.

2. Генерация кандидатов.

Для каждой колонки создаётся объект атрибута. Он содержит:

- итератор по отсортированному домену значений колонки (т.е. итерация по строкам колоночного файла);

- два списка атрибутов — список исходных и список зависимых атрибутов.

Таким образом, каждый объект атрибута хранит информацию о том, какие атрибуты могут находится в левой и правой частях предполагаемой зависимости. Два списка необходимы для того, чтобы прекращать обработку объекта атрибута на самом раннем этапе. Объект атрибута считается обработанным, если либо все значения колоночного файла были прочитаны, либо оба списка стали пустыми.

На данной стадии, SPIDER инициализирует все объекты атрибутов с итераторами, указывающими на первые значения соответствующих колоночных файлов. Оба списка кандидатов заполняются всеми атрибутами за исключением текущего. Более продвинутая версия алгоритма может уточнять данные списки кандидатов, используя информацию о типах колонок (например, нет смысла проверять зависимость между числовым типом с плавающей запятой и целочисленным типом). Базовая версия работает со всеми значениями как со строками.

3. Проверка кандидатов

Объекты атрибутов хранятся в двоичной куче (min-heap), где значением вершины является текущее значение, на которое указывает итератор. SPIDER выполняет следующий алгоритм до тех пор пока куча не станет пустой:

- Из вершины кучи достаётся набор объектов атрибутов, имеющих одинаковое значение.

- Для каждого объекта атрибута, принадлежащего набору, его список зависимых атрибутов пересекается с набором. Другими словами, алгоритм удаляет атрибуты, которых нет в наборе из списка зависимых атрибутов. Соответствующие исходные атрибуты удаляются аналогичным образом (если для объекта атрибута A из списка кандидатов удаляется зависимый кандидат B, тогда для B удаляется исходный кандидат A).

- Для каждого необработанного объекта атрибута из набора, алго-

ритм продвигает итератор и вставлять объект атрибута в кучу.

3.3. Алгоритм Mind

Алгоритм Mind [11] является n-арным алгоритмом для поиска приближенных IND. Оригинальный алгоритм делится на:

- поиск унарных зависимостей;
- поиск n-арных зависимостей.

Фактически, вторая часть алгоритма не зависит от первой, поскольку для второй части алгоритма необходим исключительно список унарных зависимостей. Поэтому для поиска унарных зависимостей может использоваться любой алгоритм. В рамках данной работы используется модифицированный алгоритм поиска унарных зависимостей Spider.

```
1  # Input: d a database, and I_1 the set of unary INDs.
2  # Output: all INDs satisfied by d.
3  C_2 := gen_next(I_1);
4  i := 2;
5  while C_i != do
6    forall I ∈ C_i do
7      if d ⊨_ε I then
8        I_i := I_i ∪ {I};
9    C_{i+1} := gen_next(I_i);
10   i := i + 1;
11 end while
12 return ⋃_{j<i} I_j
```

Листинг 1: Алгоритм Mind.

Процесс проверки зависимостей осуществляется по уровням. Сначала формируется набор кандидатов, основываясь на зависимостях, найденных на предыдущем уровне. Затем происходит последовательная проверка кандидатов.

На Листинге 2 представлен алгоритм генерации кандидатов на языке SQL.

```
1  # Input: I_i, INDs of size i.
2  # Output: C_{i+1}, candidates of size i+1.
3  insert into C_{i+1}
4  select p.lhs.rel[p.lhs[1], p.lhs[2], ..., p.lhs[i], q.lhs[i]]
5         ⊆ p.rhs.rel[p.rhs[1], p.rhs[2], ..., p.rhs[i], q.rhs[i]]
6  from I_i p, I_i q
7  where p.lhs.rel = q.lhs.rel and p.rhs.rel = q.rhs.rel
8      and p.lhs[1] = q.lhs[1] and p.rhs[1] = q.rhs[1]
9      and ...
10     and p.lhs[i−1] = q.lhs[i−1] and p.rhs[i−1] = q.rhs[i−1]
11     and p.lhs[i] < q.lhs[i] and p.rhs[i] <> q.rhs[i]
12 for all I ∈ C_{i+1} do
13   for all J ≺ I and J of size i do
14     if J ∉ I_i then
15       C_{i+1} := C_{i+1} \ {I}
16     end if
17   end for
18 end for
```

Листинг 2: Алгоритм генерации кандидатов

Для генерации кандидатов $i+1$ уровня перебираются всевозможные комбинации зависимостей i уровня. Зависимости должны отличаться только последним атрибутом. На завершающем этапе для каждого кандидата проверяется, что все его подзависимости размера i выполняются. На Листинге 3 представлен упрощенный (более высокоуровневый) алгоритм генерации кандидатов.

```
1  # Input: I_i, INDs of size i.
2  # Output: C_{i+1}, candidates of size i+1.
3  forall (p, q) ∈ I_i × I_i:
4      if not p.startswith(q):
```

```
5           continue
6
7       if not (p.lhs.last_column() < q.lhs.last_column() and
8               p.rhs.last_column() ≠ q.rhs.last_column()):
9               continue
10
11      candidate := create_candidate(p, q)
12      if can_prune_candidate(candidate, prev_raw_inds):
13          continue
14
15      C_{i+1} := C_{i+1} ∪ {candidate}
16  end for
```

Листинг 3: Упрощенный алгоритм генерации кандидатов

Последняя часть алгоритма заключается в проверке валидности кандидата. В оригинальной статье [11] приводится следующий SQL запрос 4:

```
1  SELECT * FROM r
2  WHERE NOT EXISTS (
3      SELECT * FROM s
4      WHERE r.A_1 = s.B_1 AND ... AND r.A_n = s.B_n
5  );
```

Листинг 4: Проверка точного кандидата.

Фактически, для проверки точной зависимости необходимо выполнить проекцию обоих отношений (в одном случае выполняется проекция на атрибуты, которые используются в левой части зависимости, в другом — в правой части). Затем необходимо проверить, что все кортежи из левого отношения также есть и в правом отношении.

Для поиска приближенных зависимостей предлагается использовать следующие запросы 5:

```
1  # count error rows
2  select count(distinct r.A_1, ..., r.A_n) as disqualifying_rows
```

```
from r
where not exists (
    select * from s
    where r.A_1 = s.B_1 and ... and r.A_n = s.B_n
);
# calculate lhs cardinality
select count(distinct r.A_1, ..., r.A_n) from r;
```

Листинг 5: Проверка приближенного кандидата.

После чего для проверки валидности кандидата, достаточно поделить количество уникальных кортежей ($disqualifying\_rows$), которые не содержатся в правом отношении на мощность левого множества ($lhs\_cardinality$). Неформально, значение ошибки для зависимости включения можно определить как количество строк, которые нужно удалить из левого отношения для того (независимо от их количества появлений), чтобы получить датасет, на котором зависимость выполняется.

3.4. Алгоритм Sawfish

Алгоритм Sawfish (**S**imilarity **AW**are **F**inder of **I**nclusion dependencies via a **S**egmented **H**ashindex) [10] является единственным известным на данный момент алгоритмом для поиска унарных зависимостей включения подобия.

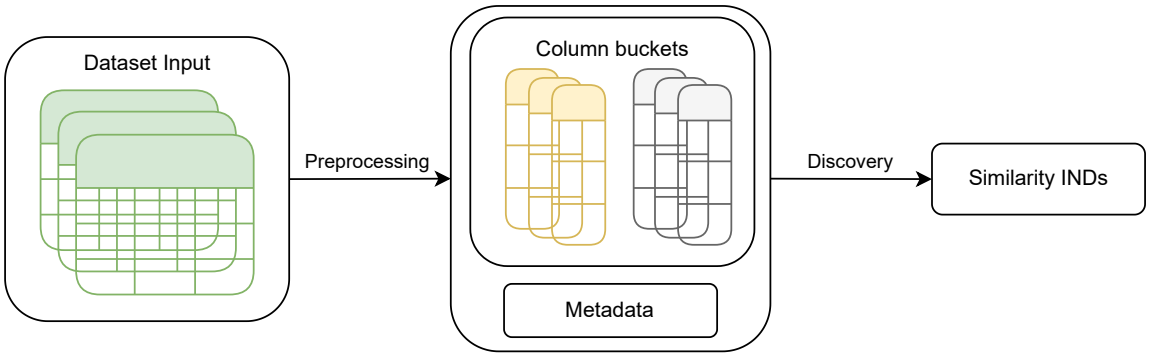

Рис. 2: Алгоритм Sawfish

На рисунке 2 показаны стадии работы алгоритма Sawfish. На первом этапе входные данные преобразуются в специальный формат, при этом извлекаются дополнительные метаданные о колонках. На втором этапе происходит генерация кандидатов и их проверка с использованием инвертированного индекса, позволяющего эффективно находить подобные строки.

Данный алгоритм может использовать любые меры подобия, которые удовлетворяют следующим свойствам:

1. иметь возможность отсекать пары значений на основе упорядоченной числовой характеристики;

2. иметь возможность отсекать пары значений на основе неравенства их подстрок.

Второе свойство необходимо для создания инвертированного индекса, где подстроки указывают на свои родительские строки. А если две строки не имеют общих подстрок, то они считаются несхожими.

Среди других мер схожести можно выделить расстояние Хэмминга, косинусное сходство (Cosine similarity) и коэффициент Сёренсена (Dice similarity).

Далее в данной секции рассматривается этапы работы алгоритма.

3.4.1. Обработка данных

На первом этапе входные данные преобразуются в специальный формат, а также извлекаются дополнительные метаданные.

Для каждого столбца все уникальные значения группируются по их длине. В случае расстояния Левенштейна длина измеряется в количестве символов в строке, а в случае коэффициента Жаккара — в количестве токенов в строке. При этом отбрасываются колонки, которые содержат значения, длина которых более 50 символов или более 10 токенов, в случае работы с токенами. Также, для каждого столбца собирается информация о максимальной и минимальной длине его значений.

Алгоритмы для работы с зависимостями включения могут использоваться для работы с большими данными, из-за чего могут возникать проблемы с недостатком оперативной памяти. Для таких случаев используется контроль памяти и выгрузка колонок значений на диск. Если на этапе обработки входных данных алгоритм достигает ограничения по памяти, то алгоритм выгружает самый большой столбец на диск.

Данные на диске хранятся в отдельных файлах (файлы группируются по индексу колонки и длине значений) для последующего чтения без необходимости повторного чтения всей таблицы. После полной обработки входных данных проводится удаление дупликатов выгруженных на диск данных, так как в них могут присутствовать дублирующиеся данные из-за раннего сброса буферов.

Также, на этапе обработки входных данных происходит отсеивание неподходящих кандидатов. Например, отсеиваются кандидаты с пустыми столбцами, рефлексивные зависимости. Также, в случае расстояния Левенштейна исключаются кандидаты, в которых все значения короче порогового значения. Отбрасываться также могут кандидаты, которые не могут быть истинными из-за разницы в длинах значений.

После отсеивания оставшиеся кандидаты упорядочиваются, назначая каждой ссылочной колонке все потенциально зависимые от неё колонки.

3.4.2. Поиск зависимостей

Основной принцип `Sawfish` заключается в том, что для отсечения кандидата `SIND` достаточно найти всего один контрпример. В отличие от зависимостей включения, в случае зависимостей включения подобия для каждого зависимого значения должно существовать хотя бы одно похожее значения в ссылочном столбце.

Для уменьшения числа проверок, `Sawfish` использует инвертированный индекс. Для ссылочного столбца строится инвертированный индекс, после чего для каждого значения из зависимого столбца выполняется поиск подобного значения по данному индексу.

3.4.3. Инвертированный индекс

Инвертированный индекс хранит отображение уникальных подстрок на их исходные строки. В зависимости от режима работы данные хранятся в разном виде.

- В режиме работы с токенами индекс сопоставляет токены и их исходные строки.
- В режиме работы с расстоянием Левенштейна каждая строка разбивается на сегменты, а индекс сопоставляет сегменты и их исходные строки.

В случае работы с токенами алгоритм построения индекса очевиден, поэтому далее более подробно рассмотрим построение индекса в случае работы с расстоянием Левенштейна.

Сначала генерируются позиции сегментов, при этом позиции достаточно вычислить один раз для конкретного значения длины. Длина сегментов делается практически одинаковой. Если есть остаток при делении длины строки на количество сегментов, то более короткие сегменты размещаются в начале.

Подстроки извлекаются на основе сгенерированных позиций, при этом каждая подстрока сопоставляется со своей исходной строкой. Построение индекса должно происходить последовательно для возможности использования скользящего окна по длинам. Поэтому наборы сегментов хранятся отдельно и независимо друг от друга.

Важной особенностью инвертированного индекса является возможность использования скользящего окна по длинам: достаточно сравнивать только зависимые значения со значениями внутри индекса в пределах допустимого интервала. Итерация по длинам значений начинается с самых длинных к самым коротким. При переходе от одной длины к другой происходит удаление ненужного индекса и происходит загрузка нового.

Благодаря скользящему окну нет необходимости строить весь индекс (если столбец перестаёт быть ссылочным, то индексация прекра-

щается досрочно), а индексы занимают меньше памяти.

3.5. Выводы

По итогам обзора выбраны подходы к обработке неточностей данных, а именно добавить поддержку приближённых зависимостей включения и зависимостей включения подобия, поскольку именно эти типы зависимостей включения позволяют обрабатывать грязные данные.

Для поддержки приближённых зависимостей включения выбрано два алгоритма:

- SPIDER — для поиска унарных зависимостей включения,
- MIND — для поиска n-арных зависимостей включения.

Для поиска зависимостей включения подобия принято решение использовать алгоритм SAWFISH.

4. Реализация

4.1. Алгоритм Spider

В реализации от Metanome обработка колонок входных таблиц происходит последовательно в одном потоке. То есть по входной таблице с n колонками выполняется n полных проходов по таблице, где на i-ой итерации создаётся колоночный файл для i-ой колонки.

На практике, создание колоночных файлов можно выполнять одновременно, за один проход по входной таблице. При этом заполнение структур данных, в которых хранятся данные для колоночных файлов может выполняться параллельно.

В реализации от Metanome по итогам обработки очередной колонки выполняется объединение всех её партиций в один колоночный файл. Объединение файлов может быть сделано "на лету", на стадии валидации кандидатов (в данном случае итератор должен инкапсулировать логику объединения партиций).

Также, в случае когда алгоритм не достигает ограничения по памяти, можно не создавать колоночные файлы на диске, а хранить партиции в памяти. Благодаря этому, можно избежать дорогостоящих операций ввода-вывода.

2. Генерация кандидатов.

В подразделе 3.2 упоминалось, что список кандидатов может быть уточнён благодаря информации о типах. Однако для того, чтобы использовать информацию о типах необходимо сделать дополнительный проход по значениям колонок, чтобы определить её тип.

С другой стороны, можно добавить более простую проверку — проверку максимального и минимального значения. Данная проверка не требует выполнения дополнительного прохода по таблице, поскольку эта информация может быть получена на этапе создания колоночных файлов.

3. Поиск приближённых IND.

Алгоритм Spider может быть обобщён для того, чтобы искать не

только точные унарные IND, но и приближённые. Для этого объект атрибута должен для всех исходных объектов атрибутов подсчитывать количество совпадающих значений.

Важно отметить, что в случае поиска приближённых IND нельзя использовать возможность раннего терминирования, из-за чего стадия валидации может существенно замедлиться. Поэтому стоит отдельно рассматривать случай поиска точных и приближённых IND.

Реализация.

Для того, чтобы добавить новый алгоритм в DESBORDANTE, необходимо реализовать класс с общей функциональностью для нового примитива и затем реализовать класс для нового алгоритма, унаследовав его от класса с общей функциональность для данного примитива.

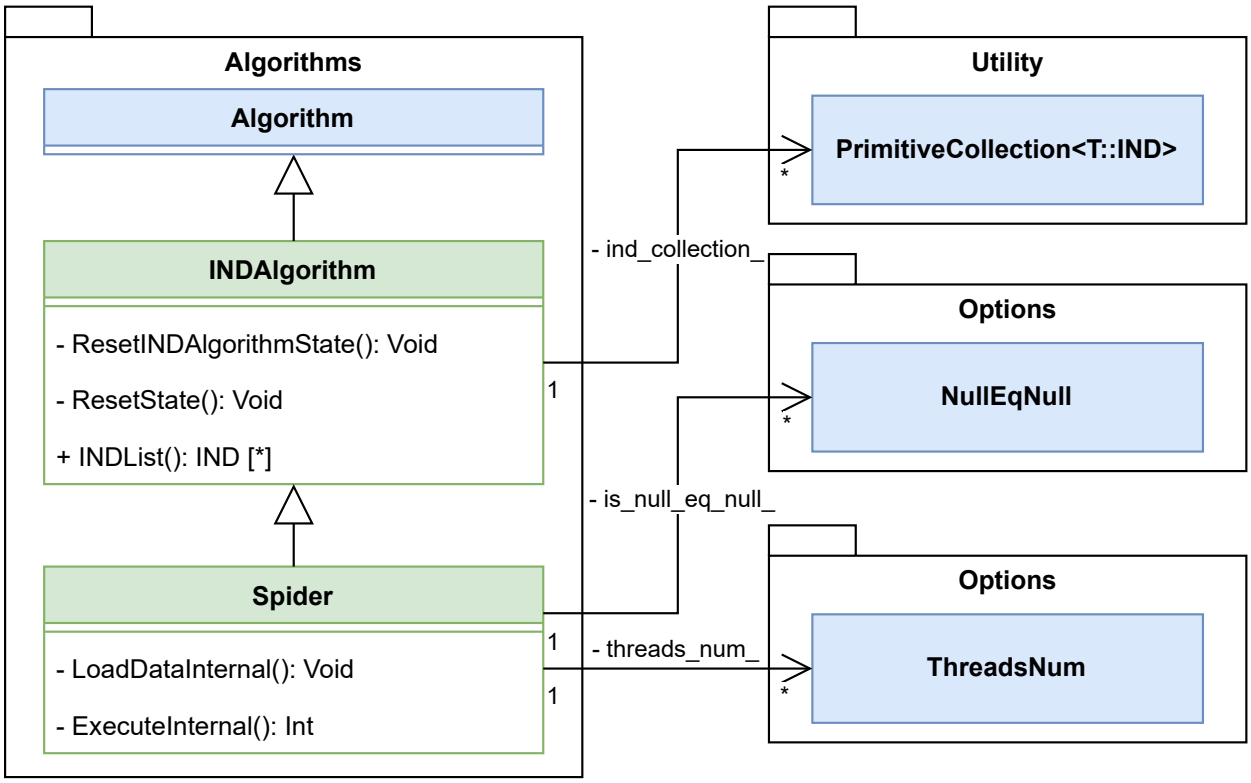

Рис. 3: Иерархия классов алгоритмов поиска IND.

На Рис. 3 представлена диаграмма, показывающая иерархию наследование классов алгоритмов поиска зависимостей включения. Зелёным цветом обозначены классы, реализованные в рамках данной работы. Наиболее важными методами при реализации нового алгоритма являются `LoadDataInternal`, `ExecuteInternal` и `ResetState`. Метод

`LoadDataInternal` необходим для загрузки данных, с которыми происходит дальнейшая работа. На этом шаге происходит обработка входных таблиц, по итогам которой создаются отсортированные домены для всех колонок в случае алгоритма SPIDER. Также поскольку загрузка данных выполняется в отдельном шаге, ещё до начала выполнения, впоследствии можно запускать один и тот же алгоритм, выставляя новые опции. При этом нет необходимости выполнять наиболее ресурсозатратный шаг — загрузку данных. В функции `ExecuteInternal` содержится основная логика самого алгоритма, в случае алгоритма SPIDER это генерация и валидация кандидатов. Функция `ResetState` необходима для выполнения сброса состояния алгоритма, после выполнения которой алгоритм должен вернуться к состоянию когда данные были загружены. Это может быть полезно при необходимости повторного выполнения алгоритма, но с другими конфигурационными параметрами.

Абстрактный класс `INDAlgorithm` предоставляет общий интерфейс для алгоритмов поиска IND. Интерфейс данного алгоритма аналогичен другим аналогичным классам, реализованным в DESBORDANTE. Интерфейс `INDAlgorithm` содержит метод для получения результатов работы алгоритма, то есть списка зависимостей включения. Реализует метод `ResetState`, внутри которого происходит очищение списка зависимостей и вызов абстрактного метода `ResetINDAlgorithmState`.

На стадии загрузки данных SPIDER выполняет последовательную обработку входных таблиц, при этом каждая таблица обрабатывается за один проход. Для каждой таблицы создаётся набор доменов. Домен представляется в виде набора партиций. Партиция может либо находиться в оперативной памяти, либо храниться на диске. Свопинг партиции на диск происходит только при достижении ограничения по памяти.

Класс домена колонки инкапсулирует логику чтения с партиции, предоставляя единый интерфейс для итерации по значениям партиций. Благодаря этому можно выполнять соединение таблиц “на лету”, прямо во время обработки объектов атрибутов. Благодаря чему можно избегать дополнительного взаимодействия с диском.

Создание доменов выполняется параллельно. Чтение таблицы с диска выполняется поблочно (размер блока выбирается в зависимости от заданного пользователем параметра ограничения памяти). Блок представляет из себя набор из колонок, обработка которых происходит в отдельных потоках. Таким образом, максимальная параллельность алгоритма ограничена количеством колонок в таблице.

Для подсчёта количества используемой памяти выполняется проход по всем строкам, хранящимся в партиции и подсчитывается примерная оценка используемой памяти. Данная проверка дорогостоящая, поэтому она выполняется как можно реже. Для этого подсчитывается текущее среднее потребление памяти, благодаря которой можно достаточно точно оценить сколько ещё блоков может быть обработано.

На этапе выполнения алгоритма SPIDER взаимодействует с объектами атрибутов `INDAttribute` и `AINDAttribute` для поиска точных и приближённых зависимостей (в зависимости от заданного пользователем порога ошибки). Оба класса наследуются от общего класса `Attribute`, реализующего общую логику работы с атрибутом. При этом алгоритм не работает с абстрактным базовым классом `Attribute`, поскольку классы `INDAttribute` и `AINDAttribute` предоставляют специфичный интерфейс для взаимодействия. Базовый класс реализует логику сравнения с другими атрибутами и хранит итератор домена.

Класс `INDAttribute` реализует логику объекта атрибута, представленного в главе 3.2. То есть данный объект хранит два списка индексов кандидатов и отсекает кандидатов по мере обработки атрибута. А класс `AINDAttribute` содержит только вектор счётчиков, которые подсчитывают количество совпадающих значений с каждым из кандидатов. По итогам обработки всех объектов атрибута не происходит отсечения кандидатов, а валидность кандидатов определяется после обработки всех доменов (кандидат валиден, если ошибка меньше порога возможной ошибки).

В разделе 4.1 упоминалась возможная модификация стадии генерации кандидатов с их отсечением на основе максимального и минимального значения. Данная оптимизация в среднем даёт хорошее ускорение

стадии валидации кандидатов, однако практически не влияет на общее время работы алгоритма. Это связано с тем, что стадия валидации кандидатов занимает в среднем около $\frac{1}{10}$ от времени выполнения стадии препроцессинга. Поэтому данная модификация не использовалась в итоговом решении.

4.2. Алгоритм Mind

Важно отметить, что алгоритм в оригинальной статье определяется в терминах реляционных баз данных. То есть, для генерации и проверки кандидатов используются SQL запросы. При этом данное решение имеет несколько недостатков:

- Некорректная работа в случае с нулевыми и пустыми значениями;
- Отсутствие возможности повторно использовать промежуточные вычисления для проверки кандидатов, из-за чего время проверки кандидатов может значительно меняться.

При реализации методов для генерации кандидатов и проверки их валидности использовались аналогичные алгоритмы без использования SQL.

На Рис. 4 представлена иерархия классов для алгоритма Mind. Алгоритм Mind использует алгоритм Spider для майнинга унарных зависимостей включения, а для поиска зависимостей большей размерности применяется непосредственно алгоритм Mind. На каждом уровне выполняется генерация кандидатов, после чего происходит их проверка. Алгоритм генерации кандидатов является хорошо известным и не будет рассматриваться далее.

Наибольший интерес представляет реализация проверки валидности кандидатов. Алгоритм отличается в случае проверки точной зависимости включения и приближенной. На Листинге 6 представлен алгоритм для проверки точных зависимостей:

```
1 Input: d a database, I — IND candidate.
2 Output: true, if I holds, false otherwise.
```

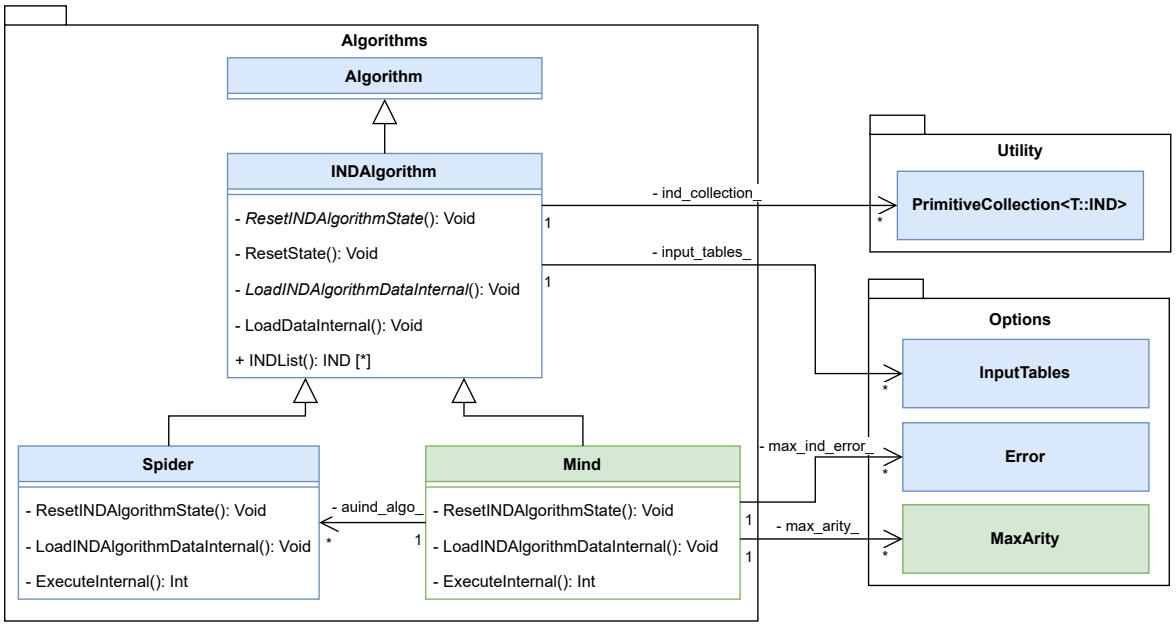

Рис. 4: Иерархия классов для алгоритма Mind.

```
3
4  lhs_stream := get_projected_stream(d, I.lhs.table, I.lhs.indices)
5  rhs_stream := get_projected_stream(d, I.rhs.table, I.rhs.indices)
6  rhs_hash_set := create_hash_set(rhs_stream)
7
8  forall lhs_row ∈ lhs_stream do
9      if lhs_row ∉ rhs_hash_set then
10         return false
11     end if
12 end forall
13
14 return true
```

Листинг 6: Алгоритм проверки кандидата IND

Сначала выполняется создание хэш таблицы для проекции правого отношения на $I.rhs.indices$. После чего выполняется проход по всем кортежам проекции левого отношения на $I.lhs.indices$.

На Листинге 7 представлен алгоритм для проверки приближенных зависимостей:

```
1  Input: d a database, I — IND candidate, ε — error threshold.
2  Output: (holds, error), where holds is true iff I holds.
3
4  lhs_stream := get_projected_stream(d, I.lhs.table, I.lhs.indices)
5  rhs_stream := get_projected_stream(d, I.rhs.table, I.rhs.indices)
6  rhs_hash_set := create_hash_set(rhs_stream)
7  lhs_hash_set := create_hash_set(lhs_stream)
8
9  lhs_cardinality := cardinality(lhs_hash_set)
10 disqualify_row_limit := ⌊lhs_cardinality * ε⌋ + 1
11 disqualify_row_count := 0
12
13 forall lhs_row ∈ lhs_hash_set do
14     if lhs_row ∉ rhs_hash_set then
15         disqualify_row_count := disqualify_row_count + 1
16         if disqualify_row_count == disqualify_row_limit then
17             return (false, None)
18         end if
19     end if
20 end forall
21
22 error := disqualify_row_count/lhs_cardinality
23 if error ≤ ε then
24     return (true, error)
25 else
26     return (false, None)
27 end if
```

Листинг 7: Алгоритм проверки кандидата AIND

В отличие от оригинального алгоритма, здесь используется раннее терминирование в случае, когда количество ошибок достигает лимита. Также, при проверке приближенной зависимости необходимо подсчитывать мощность левого спроецированного множества. Из-за чего хэш

таблица создается не только для правого отношения, а также и для левого.

4.3. Алгоритм INDVerifier

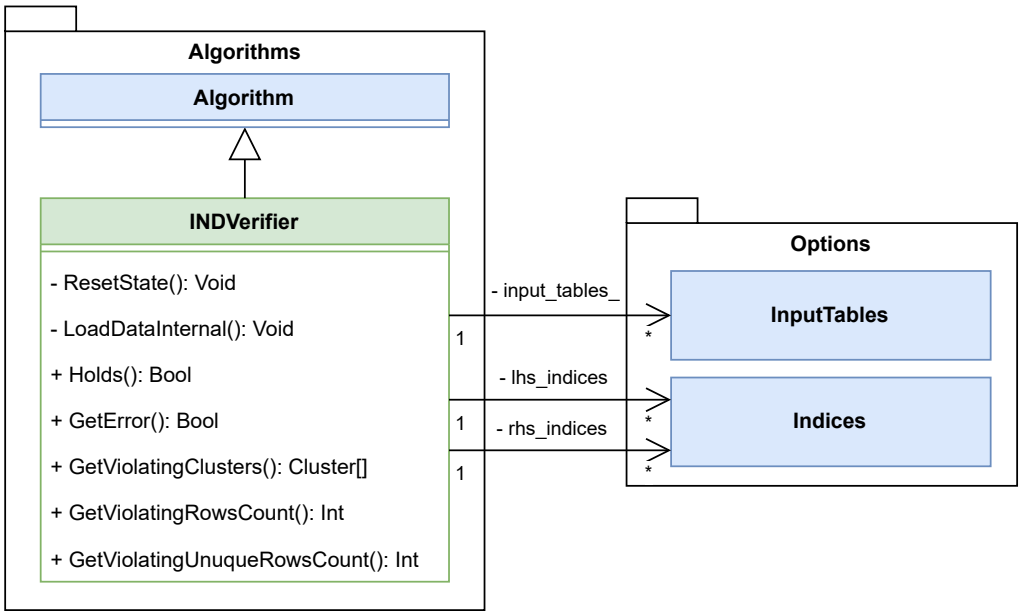

Рис. 5: Иерархия классов для алгоритма INDVerifier.

Иерархия классов представлена на Рис. 5. Алгоритм принимает одну/две таблицы, а также индексы левой части зависимости и индексы для правой части. Если указана одна таблица, то зависимость проверяется в рамках одного отношения, в противном случае $lhs\_indices$ относятся к первой таблице, а $rhs\_indices$ — ко второй.

Для реализации алгоритма верификации IND используется схожий алгоритм валидации, как и в алгоритме Mind, о котором говорилось ранее. Однако, в данном случае также происходит сбор статистики о кластерах, на которых зависимость не выполняется. Верификатор возвращает список из кластеров, где кластер (т.е. набор индексов) соответствует одному уникальному значению в левой части отношения.

Для пользователя наибольший интерес представляют те зависимости, на которых зависимость практически полностью удерживается, то есть для которых верификатор находит мало кластеров.

Алгоритм INDVerifier реализован на языке C++, а также добавлен интерфейс к нему для взаимодействия через Python, подобно аналогичным алгоритмам верификации в Desbordante.

Пример совместного использования алгоритмов. Рассмотрим пример использования пайплайна из алгоритма для поиска приближенных зависимостей включения и алгоритма верификации.

id	customer_id	product
1	101	Laptop
2	102	Phone
3	103	Tablet
4	104	Monitor
5	108	Keyboard
6	201	Mouse
7	102	Charger

Таблица 3: Orders

id	name	country
101	Alice	USA
102	Bob	UK
103	Charlie	Canada
104	David	Germany
105	Eve	France

Таблица 4: Customers

Для демонстрации примера используются таблицы *orders* 3 и *customers* 4. Рассмотрим скрипт для построения пайплайна на Рис. 8, написанный на языке Python с использованием библиотеки *desbordante*:

```
1  import desbordante
2  import pandas as pd
3
4  def print_clusters(algo, table, indices):
5      print("Number of clusters violating IND:",
6            f"{len(algo.get_violating_clusters())}")
7      for i, cluster in enumerate(algo.get_violating_clusters(),
8                                  start=1):
9          print(f"#{i} cluster:")
10         for el in cluster:
11             values = " ".join([f"{table[table.columns[idx]][el]}"
                   for idx in indices])
12             print(f"{el}: {values}")
13
```

```
14
15 TABLE_NAMES = ['orders.csv', 'customers.csv']
16 TABLES = [pd.read_csv(name) for name in TABLE_NAMES]
17 ERROR = 0.4
18
19 print(f"Run AIND algorithm with error threshold {ERROR}")
20 algo = desbordante.aind.algorithms.Default()
21 algo.load_data(tables=TABLES)
22 algo.execute(error=ERROR)
23
24 print("List of found AINDs:")
25 inds = algo.get_inds()
26 ainds = list(filter(lambda i: i.get_error() != 0.0, inds))
27 for aind in ainds:
28     print(aind)
29
30 print("Run AIND Verifier for each AIND:")
31 for aind in ainds:
32     (lhs_col, lhs_indices) = aind.get_lhs().to_index_tuple()
33     (rhs_col, rhs_indices) = aind.get_rhs().to_index_tuple()
34     lhs_table = TABLES[lhs_col]
35     rhs_table = TABLES[rhs_col]
36
37     verifier = desbordante.aind_verification.algorithms.Default()
38     verifier.load_data(tables=[lhs_table, rhs_table])
39     verifier.execute(lhs_indices=lhs_indices,
40                      rhs_indices=rhs_indices)
41     print("AIND:", aind)
42     print_clusters(verifier, lhs_table, lhs_indices)
43     print()
```

Листинг 8: Скрипт для совместно использования алгоритмов

В рамках пайплайна сначала происходит выполнение алгоритма для

поиска приближенных зависимостей (строки 20-22), а также для каждой приближенной зависимости запускается алгоритм верификации с выводом информации о кластерах (строки 31-43), на которых зависимость нарушается. На Листинге 9 представлен вывод после выполнения скрипта:

```
1  Run AIND algorithm with error threshold 0.4
2  List of found AINDs:
3  (orders, [customer_id]) -> (customers, [id]) with error = 0.33
4  (customers, [id]) -> (orders, [customer_id]) with error = 0.2
5
6  Run AIND Verifier for each AIND
7  AIND: (orders, [customer_id]) -> (customers, [id]) with error =
       0.33
8  Number of clusters violating IND: 2
9  #1 cluster:
10 5: 201
11 #2 cluster:
12 4: 108
13
14 AIND: (customers, [id]) -> (orders, [customer_id]) with error =
       0.2
15 Number of clusters violating IND: 1
16 #1 cluster:
17 4: 105
```

Листинг 9: Вывод скрипта при выполнении AIND алгоритма

Алгоритму майнинга приближенных зависимостей удалось найти две приближенные зависимости. Вторая зависимость не представляет интереса, поскольку *customers.id* является первичным ключом. Первая же зависимость должна удерживаться, поскольку в данном случае *orders.customer_id* является ссылкой на первичный ключ *customers.id*. Очевидно, что пятый и четвертый кортежи являются некорректными, поскольку они ссылаются на некорректный индекс.

4.4. Алгоритм Sawfish

В главе 4.1 подробно рассказывалось о добавлении алгоритма в существующую иерархию наследования. Для добавления алгоритма Sawfish используется аналогичная логика.

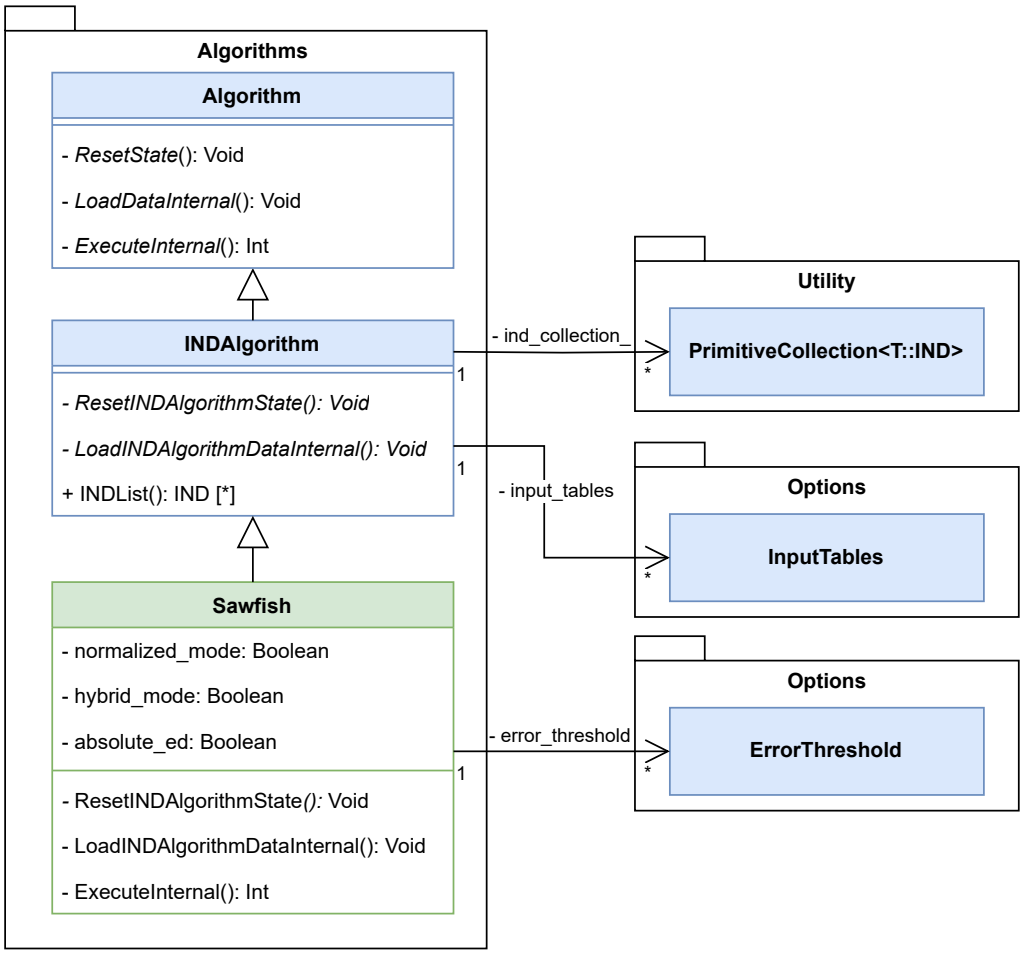

Рис. 6: Иерархия классов для алгоритма Sawfish.

На Рис. 6 представлено место алгоритма Sawfish в иерархии классов. Алгоритм реализует три метода. В методе `ResetINDAlgorithmState` выполняется сброс внутренних структур алгоритма. А методы `LoadINDAlgorithmDataInternal` и `ExecuteInternal` содержат логику для выполнения обработки входных данных и поиска зависимостей включения, соответственно.

Алгоритм также содержит параметры для конфигурации режима его работы, которые необходимы для проверки подобия значений.

5. Эксперименты

В экспериментальном исследовании сравнивается реализация алгоритмов Spider и Sawfish с реализацией от Metanome.

5.1. Методология

В экспериментальном исследовании сравнивается модифицированная реализация алгоритма Spider и тот же алгоритм, реализованный в Metanome. Время выполнения алгоритма является основной метрикой для измерения производительности алгоритмов. В экспериментах рассматривается среднее время выполнения за 10 запусков. Для очистки кэша файловой системы перед новым запуском алгоритма выполнялась команда `/proc/sys/vm/drop_caches`. Во время выполнения выполнения экспериментов все тяжеловесные процессы были отключены и фиксировалась частота процессора.

Характеристики системы. Для экспериментов использовалась система со следующими характеристиками: AMD Ryzen 7 4800H CPU @2.90GHz x 8 cores (16 threads), 32 GiB RAM, SSD A-Data S11 Pro AGAMMIXS11P512GT-C 512GB, running Ubuntu 24.04.1.

Конфигурация Java, используемая Metanome: openjdk 21.0.7 2025-04-15, OpenJDK Runtime Environment (build 21.0.7+6-Ubuntu-0ubuntu124.04), OpenJDK 64-Bit Server VM (build 21.0.7+6-Ubuntu-0ubuntu124.04, mixed mode, sharing). Конфигурация, используемая Desbordante: gcc (Ubuntu 13.3.0-6ubuntu2 24.04) 13.3.0, ldd (Ubuntu GLIBC 2.39-0ubuntu8.4) 2.39. Компиляция Desbordante происходила с опцией `-O3`.

5.2. Алгоритм Spider

5.2.1. Наборы данных

В экспериментах используется 3 реальных датасета из различных областей и 3 синтетических, информация о которых представлена в

Имя	Размер	Тип	Таблицы	Атрибуты	Строки	unary IND
Brazilian E-Commerce[2]	126.3 МБ	Реальный	9	52	1.6 M	23
FitBit Fitness Tracker Data	330 МБ	Реальный	18	259	8.1 M	8798
TPC-H (SF=1)	1.1 ГБ	Синтетический	8	61	8.7 M	96
TPC-H (SF=10)	11.1 ГБ	Синтетический	8	61	86.6 M	97
haINDgen (generator)	862 МБ	Синтетический	2	36	6.1 M	18
CIPublicHighway	28.3 МБ	Реальный	1	18	0.4 M	65

Таблица 5: Информация о наборах данных для экспериментов

таблице 5. Главной целью при выборе датасетов было собрать набор данных, каждый из которых будет обладать особенностями. `Brazilian E-Commerce` — это реальный набор данных о продажах, который содержит довольно длинные строки с отзывами пользователей. `FitBit Fitness Tracker Data` — это широкий набор данных, который содержит большое количество IND. `CIPublicHighway` — это разреженная таблица с большим процентов нуллов. `TPC-H` — широко используемый синтетический набор данных, для которого мы рассматриваем два коэффициента масштаба: 1 и 10. Последний набор данных `haINDgen` генерируется утилитой, которая может создавать данные с n-арным числом IND, где n велико.

5.2.2. Экспериментальное исследование

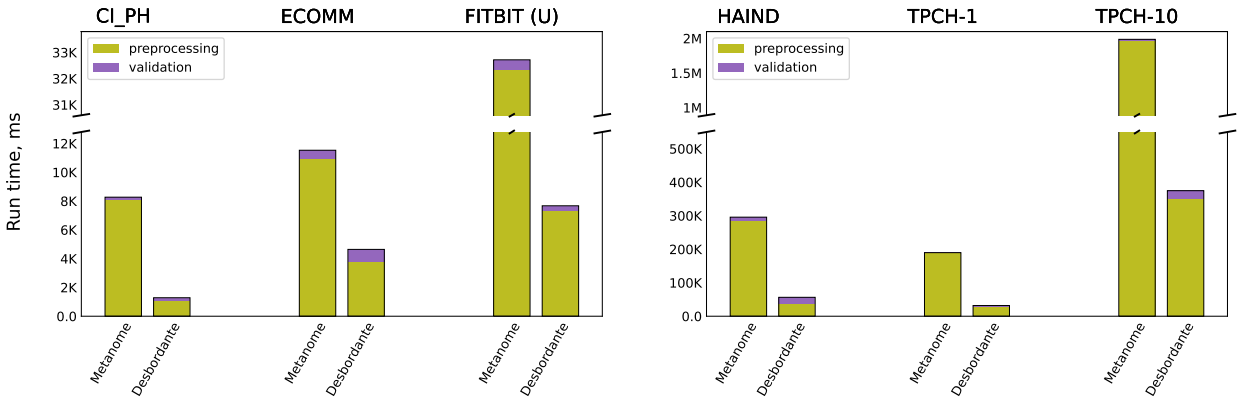

Рис. 7: Время выполнения Spider'а для Metanome и Desbordante.

На Рис. 7 представлена диаграмма со сравнением времени выполнения между модифицированной версией Spider из Desbordante, запущенной в 8 потоков и однопоточной версией от Metanome.

В таблице 6 представлено соотношение времени выполнения между

	CI_PH	ECOMM	FITBIT (U)	HAIND	TPCH-1	TPCH-10
Desbordante / Metanome	6.45	2.49	4.27	5.24	5.93	5.30

Таблица 6: Spider — Среднее ускорение

реализациями. Среднее ускорение составило примерно 4.95. Для большинства датасетов ускорение составляет порядка 5-6 раз. Наибольшее ускорение даёт модификация, связанная с параллельным заполнением структур доменов, поскольку эта стадия занимает наибольшее количество времени. Также заметный прирост скорости даёт отсутствие принудительного свопинга колоночных файлов после обработки.

Однако можно заметить, что для датасета `Brazilian E-Commerce` было получено наименьшее ускорение — порядка двух с половиной раз. Это связано с тем, что таблицы из данного датасета в среднем имеют мало атрибутов, из-за чего параллельное заполнение структур доменов колонок не даёт существенного прироста в скорости. Также поскольку таблицы имеют небольшой размер по сравнению с таблицами из других датасетов, то добавление нового значения в структуры доменов занимает мало времени. Для хранения значений партиции используется `std::set`, вставка в который будет занимать больше времени при увеличении количества элементов.

5.3. Алгоритм Sawfish

Имя	**Размер**	**Тип**	**Таблицы**	**Атрибуты**	**Строки**
CENSUS	126.3 МБ	Реальный	1	42	199 тыс
WIKIPEDIA	302 МБ	Реальный	2	12	8.1 млн
TPC-H (SF=1...16)	SF · 1.1 ГБ	Синтетический	8	61	SF · 8.7 млн
IMDB	5.7 ГБ	Реальный	22	94	761 млн

Таблица 7: Информация о наборах данных для экспериментов

Для тестирования алгоритма Sawfish используется три публично доступных датасета: два реальных (`CENSUS` и `WIKIPEDIA`) и один синтетический (`TPC-H`).

На Рис. 8 представлено сравнение времени обработки данных для разных реализаций в случае Desbordante с реализацией от

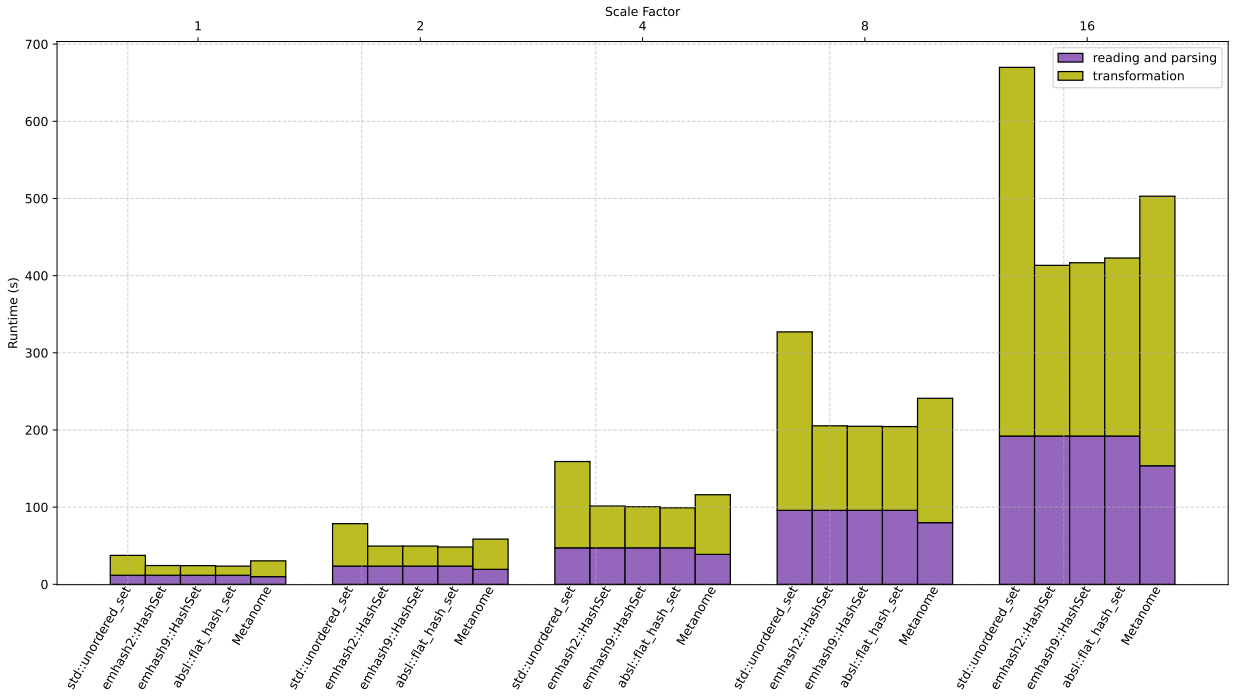

Рис. 8: Время обработки входных данных (TPC-H) для алгоритма Sawfish в зависимости от коэффициента масштаба.

Metanome. На этапе обработки данных можно выделить две основных задачи:

- чтение данных с диска и обработку значений;
- преобразование данных в специальный формат.

Первая задача находится за пределами рассматриваемого алгоритма, из-за чего далее время выполнения данной задачи не будет учитываться. При этом можно отметить, что в случае Metanome чтение данных с диска и обработка значений выполняется немного быстрее.

Задача трансформации входных данных в колоночные представления является здесь наиболее важной. От выбора реализации хэш-таблицы, в которой будет храниться колоночное представление, зависит итоговое время трансформации. Наиболее плохие результаты показывает `std::unordered_set` и `boost::unordered_set`. Использование сторонних хеш-таблиц позволяет получить значительный прирост.

Можно заметить, что время преобразования растёт линейно в зависимости от коэффициента масштаба. Наихудшие результаты показыва-

	SF 1	SF 2	SF 4	SF 8	SF 16
std::unordered_set	25.8	55.0	111.8	231.3	477.7
emhash2::HashSet	12.6	25.9	54.2	109.6	221.2
emhash9::HashSet	12.5	26.0	53.3	108.9	224.5
absl::flat_hash_set	11.9	24.7	51.9	108.6	230.7
metanome	20.5	39.2	77.3	161.1	349.3

Таблица 8: Время преобразования (в секундах) для разных реализаций и коэффициентов масштаба TPCH

ет хэш-таблица из стандартной библиотеки. При этом в среднем наилучшее время показывает `absl::flat_hash_set`, а именно в среднем трансформация работает в $1.5 - 1.7$ раза быстрее. А `emhash2` и `emhash9` хэш-таблицы показывают всегда сопоставимые значения с реализацией от `absl`.

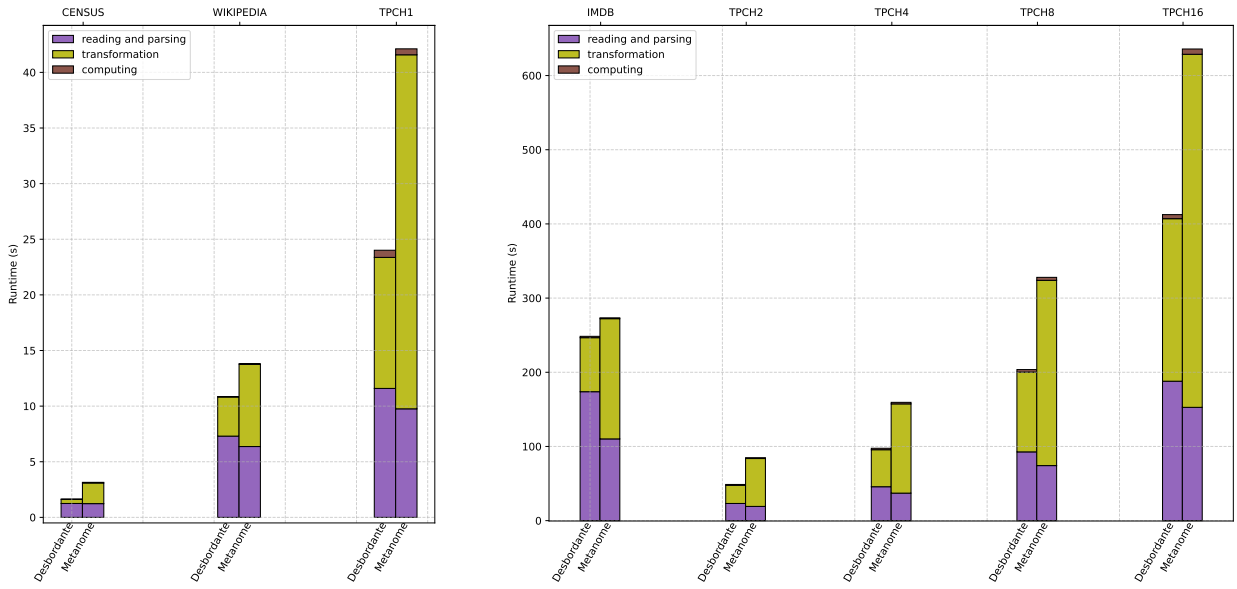

Рис. 9: Время выполнения Sawfish для Metanome и Desbordante.

На Рис. 9 представлена диаграмма со сравнением времени выполнения между реализациями алгоритма Sawfish в Desbordante и Metanome.

	CENSUS	WIKIPEDIA	IMDB	TPCH1	TPCH2	TPCH4	TPCH8	TPCH16
Desbordante / Metanome	4.97	2.09	2.19	2.60	2.56	2.36	2.29	2.15

Таблица 9: Sawfish - Общее ускорение (без чтения и парсинга значений)

В таблице 6 представлено соотношение времени выполнения между реализациями. Наибольшее ускорение наблюдается на датасете `CENSUS` — почти в 5 раз быстрее, что объясняется меньшим объёмом данных. Остальные датасеты демонстрируют устойчивое ускорение от 2.09 до 2.60.

Особенно стоит отметить, что даже на больших и сложных датасетах, таких как `IMDB` и `TPCH` (при больших значениях коэффициента масштаба), ускорение сохраняется в пределах 2.1–2.3.

Таким образом, Desbordante демонстрирует существенное преимущество по времени выполнения по сравнению с Metanome в рамках алгоритма Sawfish на всех протестированных наборах данных.

Заключение

По итогам работы были получены следующие результаты.

- Выполнен обзор предметной области и выбраны расширения зависимостей включения для обработки неточностей в данных (AIND, SIND).
- Реализованы алгоритмы для работы с расширениями зависимостей включения, а именно:
  - алгоритм поиска унарных зависимостей Spider и обобщён для поиска приближённых зависимостей;
  - алгоритм поиска приближенных зависимостей Mind;
  - алгоритм верификации приближённых зависимостей включения INDVerifier и алгоритм для создания пайплайнов;
  - добавлены интерфейсы для алгоритмов для взаимодействия через Python.
- Выполнено экспериментальное исследование алгоритмов, а именно:
  - выполнено сравнение производительности с Metanome для алгоритмов Spider и Sawfish.

Код реализации открыт и доступен в репозитории[3].

[3] https://github.com/Desbordante/desbordante-core (дата обращения: 30 мая 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: http://dx.doi.org/10.1007/s00778-015-0389-y.

[2] Chomicki Jan, Marcinkowski Jerzy. Minimal-change integrity maintenance using tuple deletions // Information and Computation. — 2005. — Vol. 197, no. 1. — P. 90–121. — URL: https://www.sciencedirect.com/science/article/pii/S0890540105000179.

[3] Codd E. F. Further Normalization of the Data Base Relational Model // Research Report / RJ / IBM / San Jose, California. — 1971. — Vol. RJ909. — URL: https://api.semanticscholar.org/CorpusID:45071523.

[4] Data profiling with metanome / Thorsten Papenbrock, Tanja Bergmann, Moritz Finke et al. // Proceedings of the VLDB Endowment. — 2015. — aug. — Vol. 8, no. 12. — P. 1860–1863. — URL: http://dx.doi.org/10.14778/2824032.2824086.

[5] 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). — IEEE, 2021. — may. — URL: http://dx.doi.org/10.23919/FRUCT52173.2021.9435469.

[6] Efficiently Detecting Inclusion Dependencies / Jana Bauckmann, Ulf Leser, Felix Naumann, Veronique Tietz // 2007 IEEE 23rd International Conference on Data Engineering. — 2007. — P. 1448–1450.

[7] Fagin Ronald. Horn clauses and database dependencies (Extended Abstract) // Symposium on the Theory of Computing. — 1980. — URL: https://api.semanticscholar.org/CorpusID:6285434.

[8] Fagin Ronald. A Normal Form for Relational Databases That is Based on Domains and Keys // ACM Trans. Database Syst. — 1981. — sep. — Vol. 6, no. 3. — P. 387–415. — URL: https://doi.org/10.1145/319587.319592.

[9] Inclusion Dependency Discovery: An Experimental Evaluation of Thirteen Algorithms / Falco Dürsch, Axel Stebner, Fabian Windheuser et al. // Proceedings of the 28th ACM International Conference on Information and Knowledge Management. — CIKM '19. — New York, NY, USA : Association for Computing Machinery, 2019. — P. 219–228. — URL: https://doi.org/10.1145/3357384.3357916.

[10] Kaminsky Youri, Pena Eduardo, Naumann Felix. Discovering Similarity Inclusion Dependencies // Proceedings of the ACM on Management of Data. — 2023. — 05. — Vol. 1. — P. 1–24.

[11] Marchi Fabien De, Lopes Stéphane, Petit Jean-Marc. Unary and n-ary inclusion dependency discovery in relational databases // Journal of Intelligent Information Systems. — 2009. — Vol. 32. — P. 53–73. — URL: https://api.semanticscholar.org/CorpusID:14322668.