Санкт-Петербургский государственный университет

***Смирнов Александр Андреевич***

Выпускная квалификационная работа

# Реализация алгоритма для поиска зависимостей включения в рамках платформы Desbordante

Уровень образования: бакалавриат

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

Основная образовательная программа *СВ.5006.2019 «Математическое обеспечение и администрирование информационных систем»*

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

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

Рецензент:
старший преподаватель кафедры информационно-аналитических систем К. К. Смирнов

Санкт-Петербург
2023

Saint Petersburg State University

***Aleksandr Smirnov***

Bachelor's Thesis

# Implementation of inclusion dependency discovery algorithm in Desbordante platform

Education level: bachelor

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

Programme *CB.5006.2019 "Software and Administration of Information Systems"*

Scientific supervisor:
C.Sc., Senior researcher E. G. Mikhailova

Consultant:
Assistant G. A. Chernishev

Reviewer:
Senior lecturer K. K. Smirnov

Saint Petersburg
2023

# Оглавление

**Введение** **4**

**1. Постановка задачи** **6**

**2. Поиск зависимостей включения: обзор** **7**

2.1. Основные понятия . . . . . 7

2.2. Обзор существующих алгоритмов . . . . . 8

2.3. Существующие инструменты . . . . . 9

2.4. Desbordante . . . . . 11

**3. State-of-the-Art: алгоритм Faida** **13**

3.1. Принцип работы . . . . . 13

**4. Архитектура решения** **18**

**5. Исследование производительности и оптимизации** **20**

5.1. Буферизация . . . . . 21

5.2. Выбор хеш-таблицы . . . . . 22

5.3. Векторизация . . . . . 23

5.4. Распараллеливание . . . . . 25

**6. Эксперименты: сравнение с Metanome** **27**

6.1. Исследовательские вопросы и метрики . . . . . 27

6.2. Условия эксперимента . . . . . 27

6.3. Эксперимент . . . . . 28

6.4. Результаты . . . . . 31

6.5. Угрозы валидности . . . . . 33

**Заключение** **35**

**Список литературы** **36**

# Введение

В настоящее время многие компании сталкиваются с задачами обработки большого количества данных. Данные обрабатываются, перерабатываются и исследуются бизнес-аналитиками, специалистами по машинному обучению, учеными. Однако, если данных оказывается слишком много, при работе с ними могут возникнуть затруднения. В таких случаях может быть полезно профилирование данных — совокупность методов, направленная на извлечение метаданных из набора данных [1] (иначе — получение данных о данных). Благодаря профилированию можно обнаружить различные закономерности, сокрытые в данных, что позволяет сделать некоторые выводы о структуре этих данных и исправить различные ошибки. Существует множество различных методов профилирования, каждый из которых представляет поиск некоторой формально описанной закономерности или правила. Такие закономерности будем называть примитивами. Например, один из самых широко известных примитивов — функциональная зависимость [10]. Примитивы позволяют узнать внутреннюю структуру таблицы, понять, как связаны между собой разные таблицы, они полезны [1] при очистке и интеграции данных, оптимизации и нормализации баз данных. Кроме того, нахождение закономерностей в данных полезно и учёным для анализа результатов экспериментов, специалистам по машинному обучению для выбора параметров моделей, аналитикам для выдвижения гипотез. Один из таких полезных примитивов — зависимости включения. Неформально, между двумя столбцами (возможно из разных) таблиц удерживается зависимость, если множество всех значений одного столбца содержится в множестве всех значений из другого. Такие зависимости применяются в различных алгоритмах работы с базами данных: например, самое часто упоминаемое в литературе применение — поиск первичных ключей [8]. Также зависимости включения используются [12] при оптимизации запросов, проектировании баз данных, интеграции данных.

Для поиска зависимостей включения был разработан ряд алгорит-

мов, сравнение производительности которых было произведено в статье [12]. Среди этих алгоритмов выделяется Faida — алгоритм приближенного поиска зависимостей включения. В отличие от других алгоритмов, он с небольшой вероятностью может выдавать ложноположительные результаты, однако это позволяет ему работать в несколько раз быстрее точных state-of-the-art алгоритмов. Реализация Faida, а также реализации многих других алгоритмов профилирования были собраны в платформе Metanome — инструменте для профилирования данных, полностью написанном на Java. Инструмент ориентирован на работу с данными и позволяет запускать различные алгоритмы профилирования на входных наборах данных. Однако исследования [5] показали, что производительности реализаций на Java не всегда достаточно для требовательных к ресурсам вычислений на больших объемах данных. Кроме того, перед началом использования Metanome необходимо правильно сконфигурировать и развернуть на машине пользователя, что может быть затруднительно для аналитиков, которые не так тесно связаны с миром разработки.

Чтобы избавиться от описанных недостатков, была разработана платформа Desbordante — инструмент для профилирования данных с открытым исходным кодом, идейный наследник Metanome, написанный на языке C++. Инструмент поддерживает несколько примитивов, в том числе и функциональные зависимости, однако на момент начала настоящей работы поддержка зависимостей включения отсутствовала. Таким образом, было принято решение реализовать алгоритм приближенного поиска зависимостей включения Faida на языке C++, и внедрить код алгоритма в инструмент. Благодаря этому будет создана производительная open-source реализация этого алгоритма.

# 1. Постановка задачи

Целью работы является является создание высокопроизводительной реализации алгоритма поиска зависимостей включения “Faida” в рамках профайлера данных Desbordante. Для её выполнения были поставлены следующие задачи:

1. произвести краткий обзор предметной области и работы алгоритма Faida;
2. реализовать алгоритм на языке C++;
3. исследовать производительность и оптимизировать код алгоритма;
4. провести эксперименты: сравнить полученную реализацию с существующей Java-реализацией из Metanome.

# 2. Поиск зависимостей включения: обзор

## 2.1. Основные понятия

Введём основные понятия, связанные с областью зависимостей включения и алгоритмами их поиска.

**Определение 1** (**Зависимости включения**). *Пусть $r$ и $s$ — два отношения (не обязательно различных) в терминах реляционной модели данных. Пусть $R = (R_1, ..., R_k)$ и $S = (S_1, ..., S_m)$ — соответствующие им заголовки. Пусть $\bar{R} = (R_{i_1}, ..., R_{i_n})$ и $\bar{S} = (S_{i_1}, ..., S_{i_n})$ — n-арные комбинации* ***различных*** *атрибутов. Говорят [8], что комбинация $\bar{R}$ содержится в $\bar{S}$, или удерживается зависимость включения (далее — ЗВ) $\bar{R} \subseteq \bar{S}$ , если для каждого кортежа $t_r \in r$ найдется кортеж $t_s \in s$, такой, что $t_r[\bar{R}] = t_s[\bar{S}]$. При этом важно отметить, что обе комбинации $\bar{R}$ и $\bar{S}$ содержат ровно по n атрибутов.*

**Определение 2.** *В условиях предыдущего определения набор атрибутов $\bar{R}$ называется [8] зависимым (dependent), а $\bar{S}$ — исходным (referenced).*

**Определение 3.** *Зависимость включения, у которой как в левой, так и в правой части содержится по одному атрибуту, называют [12] унарной. Зависимость, у которой в левой и правой части содержится больше, чем по одному атрибуту, называют [12] n-арной.*

**Определение 4** (**Точный поиск ЗВ**). *Точный поиск ЗВ определяется [8] как нахождение всего множества ЗВ, которые удерживаются на заданном наборе отношений.*

**Определение 5** (**Приближенный поиск ЗВ**). *Приближенный поиск ЗВ определяется [8] как нахождение приближенного множества всех ЗВ, которые удерживаются на заданном наборе отношений. Приближенное множество может и содержать ложно-положительные результаты, и не содержать некоторые положительные результаты.*

## 2.2. Обзор существующих алгоритмов

Нахождение всего множества зависимостей включения для заданного набора данных (набора отношений) — одна из самых вычислительно сложных [12] задач профилирования данных. Чтобы оптимизировать процесс поиска, разные исследовательские группы разработали ряд алгоритмов, краткое описание принципа работы которых было приведено в работе [12]. Авторы отметили, что каждый из алгоритмов предоставляет хотя бы одно нововведение, которое затем используется в последующих алгоритмах. Также в этой работе было приведено подробное сравнение их производительности на различных входных данных.

Один из наиболее ресурсозатратных этапов работы многих алгоритмов — валидация кандидатов [8]. Кандидатом называют потенциальную зависимость включения, то есть такую зависимость, про которую на данном шаге алгоритма нельзя сказать, что она заведомо не удерживается. Иными словами, которая потенциально может удерживаться, но в этом ещё предстоит убедиться на шаге валидации. Каждый алгоритм представляет свой собственный, оптимизированный подход к валидации. При этом большая часть алгоритмов для этих целей использует [12] SQL-запросы и встроенные в базы данных алгоритмы.

Алгоритмы поиска ЗВ можно разделить на два типа [12]. Первый тип алгоритмов ищет только унарные зависимости включения, после чего завершает работу. Второй тип ищет всё множество зависимостей, которые удерживаются в заданном датасете — унарные, 2-арные, ..., n-арные до такого натурального n, что ни одной (n+1)-арной зависимости уже не удерживается. Алгоритмы такого типа называют алгоритмами n-арного поиска (n-ary IND discovery). Согласно статье [12], среди алгоритмов поиска унарных зависимостей хорошей производительностью отличается распределенный алгоритм Sindy [13], который наиболее эффективен при запуске в многопоточном режиме. Отличные результаты показывает и алгоритм DeMarchi [4], который, однако, полностью хранит все вспомогательные данные в оперативной памяти, поэтому не подойдет для обработки большого объема данных. Среди алгоритмов

точного n-арного поиска авторы статьи выделили state-of-the-art алгоритм Binder [7] — он показал наилучший результат при запуске на большинстве датасетов. Binder также может быть использован и для поиска унарных зависимотей. Среди всего множества алгоритмов выделяется Faida [8] — на текущий момент единственный алгоритм *приближенного* n-арного поиска. Faida может выдавать ложно-положительные результаты, но при этом работает значительно быстрее, чем Binder и другие существующие алгоритмы.

## 2.3. Существующие инструменты

### 2.3.1. Metanome

Metanome[1] — инструмент для профилирования данных с открытым исходным кодом, написанный на языке Java. Он поддерживает несколько различных примитивов профилирования, таких как функциональные зависимости, порядковые зависимости, уникальные наборы столбцов, и, в том числе, зависимости включения. Для каждого из таких примитивов поддерживается несколько алгоритмов поиска соответствующих им зависимостей. Metanome удобен тем, что даёт возможность пользователю работать с различными примитивами и алгоритмами их поиска, объединив их в одном инструменте [3]. Это позволяет быстро получить информацию из нужного набора данных — достаточно загрузить его в платформу, и становится возможным запускать на нём различные алгоритмы с помощью единого элемента управления. Таким образом, задав набор данных единожды, можно с помощью одного инструмента запустить на нём поиск функциональных зависимостей, зависимостей включения и других зависимостей. Чтобы сделать этот процесс более удобным, Metanome оснащён frontend-клиентом, который позволяет совершать эти операции с использованием графического интерфейса. Также фреймворк предоставляет единый интерфейс для разработки новых алгоритмов на базе платформы, тем самым позволяя сразу интегрировать их в свою экосистему. Таким образом, инструмент

[1] hpi.de/naumann/projects/data-profiling-and-analytics/metanome-data-profiling.html

полезен как для прикладных аналитиков, так и для разработчиков новых примитивов и алгоритмов.

Ещё одно достоинство Metanome состоит в том, что в его кодовой базе содержатся [3] Java-реализации всех алгоритмов, которые он предоставляет пользователям. В том числе инструмент содержит реализации алгоритмов поиска зависимостей включения. Эти реализации, в частности, были использованы авторами упомянутой ранее работы [12] для сравнения производительности алгоритмов между собой. Среди доступных реализаций есть и код алгоритма Faida, ссылка на который содержится [8] в оригинальной статье. В процессе исследования было установлено, что на текущий момент это единственная реализация алгоритма, которая существует в открытом доступе.

### 2.3.2. Другие инструменты для профилирования данных

В ходе исследования было выяснено, что Metanome — единственный инструмент, который поддерживает функциональность поиска зависимостей включения. Далее будут кратко рассмотрены несколько интересных инструментов для профилирования данных.

- Pandas Profiling [19] — open-source инструмент для анализа данных, представленный в виде библиотеки для языка Python. Инструмент ориентирован на разведочный анализ и предоставляет ряд классических методов профилирования, например, определяет типа столбца, вычисляет простые статистики, корреляцию. Однако инструмент не поддерживает более сложные примитивы профилирования, такие как функциональные зависимости или зависимости включения.

- OpenClean [18] — open-source библиотека для языка Python, нацеленная на профилирование и очистку данных. В качестве одной из возможностей библиотека позволяет запускать алгоритмы, реализованные в Metanome. Однако на момент подготовки настоящей работы инструмент не поддерживал ни одного из алгоритмов поиска зависимостей включения.

## 2.4. Desbordante

Desbordante [5, 6] — новый инструмент для профилирования данных с открытым исходным кодом, написанный на языке C++. Его назначение во многом совпадает с таковым у Metanome — оба инструмента объединяют различные примитивы и соответствующие им алгоритмы в один инструмент, предоставляя удобный интерфейс для работы с ними. Как и Metanome, инструмент оснащён frontend-клиентом, который обеспечивает удобную работу с исследуемыми наборами данных и позволяет визуализировать полученные результаты. Однако у Desbordante есть два важных преимущества.

**Производительность.** Важное отличие от Metanome состоит в том, что последний целиком реализован на Java. Предыдущие исследования показали [5], что для сложных вычислительных задач, коими являются алгоритмы поиска зависимостей, производительности языка Java может быть недостаточно, особенно если входные данные имеют большой объём. По этой причине Desbordante написан на языке C++. Такое решение, во-первых, позволяет увеличить производительность алгоритмов за счёт использования преимуществ компилируемого языка с ручным контролем памяти, и, во-вторых, предоставляет возможность применения низкоуровневых оптимизаций, таких как векторизация вычислений с использованием SIMD-расширений процессора, реализация собственных аллокаторов.

**Доступность.** Desbordante, как и Metanome, призван обеспечить простой доступ к работе с алгоритмами профилирования. В первую очередь это полезно для аналитиков, учёных и других специалистов, которые не занимаются разработкой и мало знакомы со сложными системами сборки. Тем не менее, Metanome имеет довольно непростую модульную структуру, поэтому установка на компьютер пользователя может вызвать затруднения и занять некоторое время. В Desbordante эта проблема решается следующим образом:

- код алгоритмов и консольного интерфейса находятся в одном репозитории и собираются в один исполняемый файл;
- реализована поддержка работы с инструментом через Python-байндинги;
- система доступна онлайн[2]: можно опробовать функциональность инструмента не прибегая к его установке.

В настоящее время инструмент находится в стадии активной разработки и не поддерживает всего множества реализованных в Metanome примитивов. Поэтому основной задачей по развитию инструмента является как добавление уже реализованных в Metanome алгоритмов, реализуя их на C++ и по возможности оптимизируя, так и реализация новых примитивов и алгоритмов из последних научных работ.

[2] https://desbordante.unidata-platform.ru

# 3. State-of-the-Art: алгоритм Faida

Эксперименты показали [12], что на объемных наборах данных даже самые производительные алгоритмы точного поиска ЗВ требуют значительного времени, чтобы завершить работу. Для решения этой проблемы был разработан [8] алгоритм Faida — алгоритм *приближенного* n-арного поиска ЗВ. Как и алгоритмы точного поиска, Faida гарантированно находит все множество ЗВ (то есть гарантирует полноту результата), но также может дополнительно выдавать ложно-положительные результаты. Эксперименты показали [8, 12], что алгоритм, однако, в подавляющем большинстве случаев выдаёт точный ответ. При этом, благодаря использованию приближенного подхода алгоритм гораздо быстрее завершает работу. Таким образом, Faida незначительно жертвует корректностью общего результата, при этом показывая значительный прирост производительности в сравнении со всеми точными алгоритмами. Благодаря этому свойству алгоритм был выбран для реализации в рамках данной работы.

## 3.1. Принцип работы

Работа алгоритма делится на три стадии: препроцессинг, генерация кандидатов и валидация (IND test). Схема работы алгоритма представлена на Рис. 1.

**Препроцессинг.** Это первая фаза работы алгоритма. Алгоритм построчно читает входные таблицы, хеширует все значения и записывает полученное представление данных на диск. При этом для каждой колонки из таблицы создаётся отдельный файл, в который последовательно записываются её хешированные значения. Таким образом, входной датасет подготавливается для дальнейшей работы, все значения приводятся к единому представлению в виде хешей. Далее алгоритм работает только с полученными хешированными значениями, исходный набор данных не используется. Стоит отметить, что возможные коллизии хешей на этом этапе — одна из причин, по которой алгоритм может

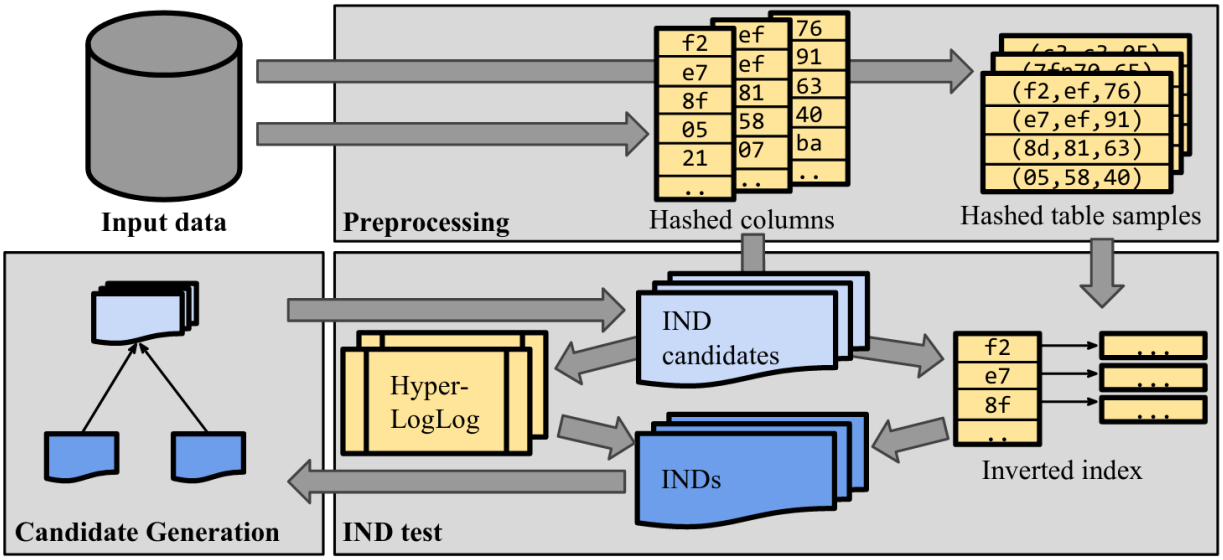


Рис. 1: Принцип работы алгоритма Faida. Источник [8].

выдавать ложноположительный результат. Также на этапе препроцессинга для каждой таблицы создаётся выборка — подмножество строк таблицы, подобранных по определённому правилу. Выборка строится таким образом, чтобы в ней оказались либо все уникальные значения из каждой колонки, либо как минимум $n$ таких значений, если у колонки уникальных значений больше чем $n$. Число $n$ — параметр алгоритма, который задаётся при запуске. Выборка будет нужна для дальнейшей работы, поэтому также хешируется и целиком записывается на диск в отдельный файл.

**Генерация кандидатов.** Сначала генерируются унарные кандидаты, множество которых представляет собой всевозможные пары колонок в таблице, за исключением пар с одинаковыми колонками. Затем происходит процесс валидации кандидатов, который будет описан ниже, и по его результатам генерируется множество уже n-арных кандидатов. На этом множестве задан частичный порядок, поэтому его удобно представлять в виде алгебраической решетки (lattice). Поскольку это множество очень велико, при наивном подходе к его генерации придется проверить очень много кандидатов. Поэтому, как и ряд предшествующих алгоритмов, Faida использует Apriori-подобную гене-

рацию. В этом случае решетка кандидатов генерируются по уровням, что позволяет отсечь заведомо ложных кандидатов, не прибегая к лишним дорогостоящим валидациям. На первом уровне решетки находятся ранее найденные унарные зависимости. На основании этого множества генерируются 2-арные кандидаты. Затем кандидаты проверяются, и по результатам валидации из них выделяется множество тех, которые на самом деле являются 2-арными зависимостями. С помощью этого множества генерируется множество кандидатов 3-го уровня и процесс повторяется. В общем виде процесс выглядит так: с помощью множества найденных n-арных зависимостей включения генерируется множество (n+1)-арных кандидатов. Кандидаты проходят процесс валидации, в результате которого выводится множество (n+1)-арных зависимостей. Процесс повторяется до тех пор, пока множество (n+1)-арных кандидатов непусто. При генерации кандидатов активно используется свойство анти-монотонности [8].

**Валидация.** Для валидации кандидатов алгоритм использует две структуры данных — инвертированный индекс и HyperLogLog. Рассмотрим их подробнее.

HyperLogLog [11] (HLL для краткости) — структура данных, которая позволяет приближенно оценить количество уникальных элементов в множестве. Особенность структуры состоит в том, что она хорошо работает с множествами очень большой мощности — результат приближения считается за константное время и не зависит от размера исходного множества. Потребляемая структурой память тоже константна. Утверждается [8], что если $X$ и $Y$ — колонки или комбинации колонок, а $s(X)$ и $s(Y)$ — множества уникальных значений в этих колонках, то $X \subseteq Y$ тогда и только тогда, когда $|s(Y)| = |s(X) \cup s(Y)|$. Значения $|s(Y)|$ и $|s(X) \cup s(Y)|$ могут быть получены с помощью HyperLogLog. Таким образом, HyperLogLog позволяет за константное время проверить, удерживается ли зависимость $X \subseteq Y$. Однако при таком подходе пришлось бы инициализировать по две структуры для каждого проверяемого кандидата. Поэтому авторы алгоритма адаптировали структу-

ру HyperLogLog для поиска зависимостей включения, что позволило поддерживать по одной структуре на колонку (или их комбинацию). Нужно отметить, что HyperLogLog требует, чтобы исследуемое множество было представлено в виде набора целых чисел, поэтому для работы с ней используются хешированные значения из файлов колонок, которые были получены на этапе препроцессинга. Структура имеет числовой параметр — точность (accuracy), с которой будет приближен результат. Чем больше точность, тем больше памяти будет занимать структура. Этот параметр является одним из параметров алгоритма Faida и задаётся при запуске.

Однако точность приближения с помощью HyperLogLog заметно снижается, если в исследуемом множестве мало уникальных элементов — например, такое может наблюдаться в столбцах со значениями категориальных признаков. Для обработки кандидатов в таких случаях используется вторая структура данных — инвертированный индекс.

*Инвертированный индекс* — структура данных, которая отображает каждое значение из таблицы на множество колонок, в которых это значение содержится. Поиск унарных ЗВ с помощью инвертированного индекса впервые был представлен в алгоритме [4]. Подход основан на том, что инвертированный индекс позволяет легко найти все колонки, которые содержат в себе какую-то колонку $X$. Для этого ищутся все множества колонок, в которых присутствует колонка $X$, а затем пересекаются. Повторяя процесс каждой колонки $X$ из набора данных, можно получить список всех удерживающихся ЗВ. Однако при обработке больших данных этот подход может стать ресурсозатратным — инвертированный индекс становится слишком большим и не помещается в оперативную память. Поэтому в алгоритме Faida инвертированный индекс строится на на небольшой выборке строк из таблицы, которая была сгенерирована на этапе препроцессинга. Причём, поскольку Faida работает только с хешированными значениями, ключами инвертированного индекса являются не сами значения, а их хеши.

Сам процесс валидации происходит следующим образом. На вход алгоритму поступают все кандидаты на текущем уровне алгебраиче-

ской решетки. Затем происходит процесс заполнения структур: производится чтение ранее созданных файлов колонок, и структуры данных заполняются хешированными данными из этих файлов. После этого происходит проверка кандидатов следующим образом. Если кандидат покрывается инвертированным индексом, то есть если все уникальные значения из колонок кандидата содержатся в выборке, то он тестируется с помощью инвертированного индекса и добавляется к общему ответу. В этом случае результат тестирования можно назвать точным с точностью до коллизий хешей. В противном случае, если все уникальные значения колонок кандидата не поместились в выборку и, соответственно, не покрываются инвертированным индексом, алгоритм запускает уже приближенную валидацию с использованием структур HyperLogLog.

Начиная с 2-арных кандидатов, в левой и правой частях ЗВ появляются комбинации из нескольких атрибутов (например, $AB \subseteq CD$), и в общем случае алгоритмам поиска ЗВ нужно проверять на включение множества, состоящие из кортежей. Например, в случае Faida ему пришлось бы заполнять структуры данных кортежами значений. Однако поскольку Faida изначально работает только с хешированными значениями, подобные кортежи тоже состоят из хешей, и каждый такой кортеж легко представить в виде единичного хеша – можно объединить содержащиеся в кортеже хеши в один хеш с помощью XOR и циклического сдвига. Таким образом, при появлении кандидатов арности $n \geq 2$, алгоритм продолжает заполнять структуры единичными значениями-хешами, и процесс тестирования многоарных кандидатов практически ничем не отличается от унарных.

Таким образом, алгоритм комбинирует две структуры, которые устраняют недостатки друг друга: инвертированный индекс даёт точный результат при валидации кандидатов, содержащих небольшое количество уникальных значений, а HyperLogLog, напротив, эффективно тестирует кандидатов, в колонках которых содержится много уникальных значений. Параметры этих структур являются параметрами алгоритма и задаются при запуске.

# 4. Архитектура решения

Высокоуровневая структура кода алгоритма представлена на Рис. 2.

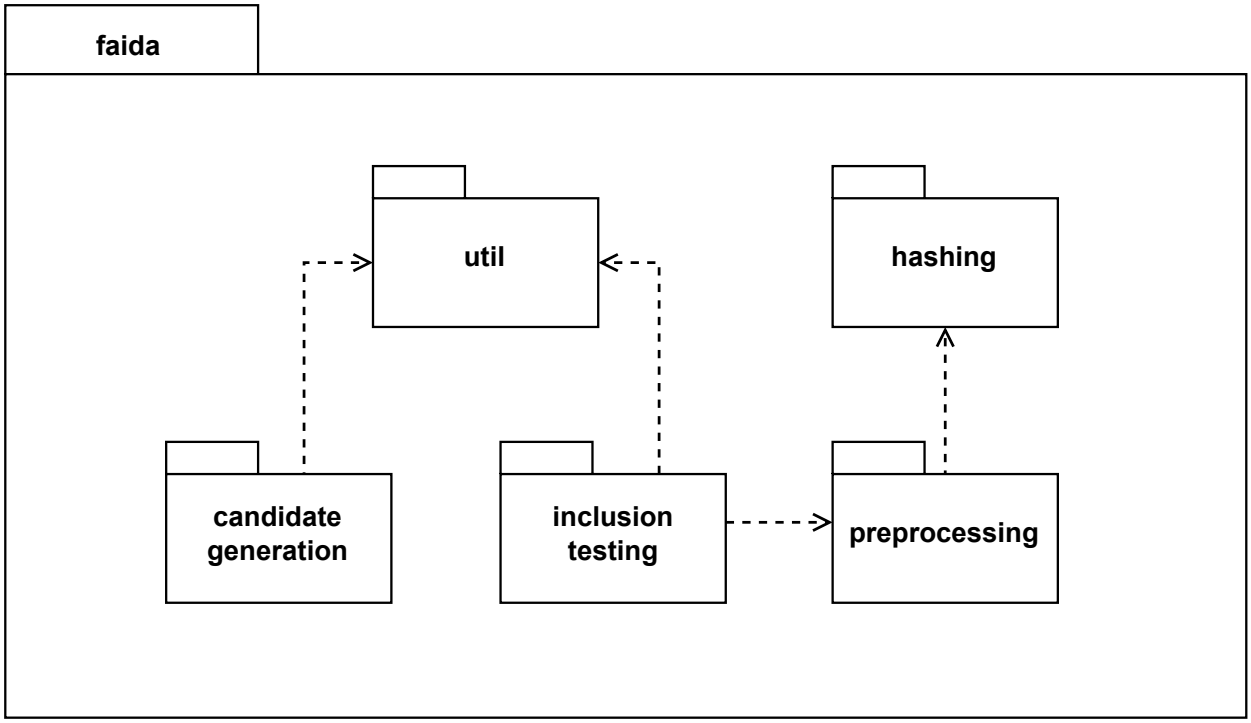


Рис. 2: Диаграмма пакетов.

В структуре кода отражены три пакета, которые соответствуют трем стадиям работы алгоритма: `preprocessing` отвечает за препроцессинг, `candidate generation` отвечает за генерацию кандидатов, `inclusion testing` — за валидацию кандидатов. При этом пакет `preprocessing` использует код из пакета `hashing` для того, чтобы хешировать данные на этапе обработки. В качестве хеш-функции была выбрана murmurhash3 [2]. Также в этом пакете хранится информация о внутреннем представлении обработанных данных и методы для работы с ними (классы `HashedColumnStore` и `BlockIterator`). Пакеты `candidate generation` и `inclusion testing` зависят от пакета `util`, в котором находятся основные структуры — классы, представляющие зависимоси включения (`SimpleIND`) и комбинации колонок (`SimpleCC`). Пакет `inclusion testing` также зависит от пакта `preprocessing`, потому что процессу валидации необходимо знать о внутреннем представлении данных. Также в этом пакете находятся реализации описанных в секции 3.1

структур данных — инвертированного индекса (класс `InvertedIndex`) и HyperLogLog (класс `HLL`).

Основная логика алгоритма находится в методе `ExecuteInternal()` класса `Faida`. Этот класс унаследован от класса `Primitive` — базового класса для всех алгоритмов профилирования, которые реализуются в рамках инструмента Desbordante. В классе `Primitive` объявлен абстрактный метод `ExecuteInternal` — классы-потомки переопределяют его и реализуют в нём свою логику. Также при реализации алгоритма был использован класс `CSVParser` — он представляет парсер входных файлов и также используется многими другими алгоритмами.

# 5. Исследование производительности и оптимизации

Производительность алгоритма была исследована с помощью профилировщика perf. В результате было обнаружено, что подавляющую часть времени работы алгоритма занимает процесс заполнения структур, который является частью этапа валидации.

Рассмотрим этот алгоритм подробнее. Его первоначальный вариант был взят из существующей Java-реализации, и упрощенная версия этого алгоритма представлена на листинге 1. На вход алгоритму подаются таблица, инвертированный индекс и структура `levelColumnCombnations`: в ней хранятся все комбинации колонок, которые присутствуют на текущем уровне кандидатов, то есть из каждого кандидата выделяются зависимая и исходная комбинации колонок. При этом структура `levelColumnCombnations` не только хранит все комбинации колонок, но ещё и отображает эти комбинации на соответствующие им структуры HLL. Для каждой строки таблицы алгоритм проводит следующие действия. Сначала из каждого файла-колонки, полученного на этапе препроцессинга, читается по одному хешу, и из прочитанных хешей формируется одномерный массив – хешированное представление строки таблицы (строки 3–4). Затем алгоритм итерируется по множеству комбинаций колонок: для каждой из комбинаций вычисляется хеш проекции текущей строки на эту комбинацию в терминах реляционной модели, как было описано в разделе 3.1 (строки 8–11). Затем полученный хеш добавляется в инвертированный индекс или HLL (строки 12–16): если значение хеша найдено в инвертированном индексе, то к списку комбинаций колонок, в котором этот хеш содержится, добавляется текущая комбинация. Если же значение не найдено, то это значит, что текущая комбинация колонок не покрывается инвертированным индексом, поэтому хеш проекции строки для неё добавляется в HLL.

**Листинг 1** Заполнение структур

```
Require: table, invertedIndex, levelColumnCombs
 1: for row in table do
 2:   hashedRow ← ArrayOfIntegers()
 3:   for colFile in table.colFiles do
 4:     hashedRow[i] ← ReadNextValue(colFile)
 5:   end for
 6:   for (columnCombination, hll) levelColumnCombs do
 7:     combinedHash ← 0
 8:     for columnIndex in columnCombination do
 9:       combinedHash ← Rotl(combinedHash)
10:       combinedHash ← Xor(combinedHash, hashedRow[columnIndex])
11:     end for
12:     if invertedIndex.HasKey(hash) then
13:       invertedIndex.Insert(hash, columnCombination)
14:     else
15:       hll.Insert(hash)
16:     end if
17:   end for
18: end for
```

## 5.1. Буферизация

Недостаток описанного подхода в том, что процесс итерации по всем комбинациям колонок происходит для каждой строки таблицы. Для решения этой проблемы было предложено использовать буферизацию. Буферизованный код представлен на листинге 2.

Теперь основной цикл идёт не по строкам таблицы, а по чанкам — подмножествам строк, идущих друг за другом в таблице. Поэтому из файлов-колонок данные читаются не по одному значению, а сразу блоками значений (строка 4), то есть захватывается несколько строк одновременно. Прочитанные блоки объединяются в чанк — двумерный массив, который представляет подмножество хешированных строк таблицы. Первый индекс в этом массиве — номер колонки, а второй — номер строки в этом чанке. То есть, чтобы получить к i-му значению j-й строки чанка, нужно обратиться к элементу `hashedRowsChunk[i][j]`. Таким образом, итерируясь в цикле по всем комбинациям колонок, алгоритм обрабатывает не одну строку, как в предыдущем случае, а сразу весь чанк (строки 6–18). Сначала для всего чанка считаются хеши проекций строк на текущую комбинацию колонок (строки 8–10), а за-

тем происходит заполнение соответствующих структур полученными значениями. За счёт такого подхода уменьшается количество итераций внешнего цикла алгоритма (строка 1). Кроме того, данные внутри чанка расположены рядом друг с другом, что увеличивает количество попаданий в кеш процессора.

**Листинг 2** Заполнение структур, буферизованный вариант

```
Require: table, invertedIndex, levelColumnCombs
 1: for chunk in table do
 2:     hashedChunk ← ArrayOfIntegers2D(table.numColumns, chunkSize)
 3:     for colFile in table.colFiles do
 4:         hashedChunk[i] ← ReadNextBlock(colFile, chunkSize)
 5:     end for
 6:     for (columnCombination, hll) in levelColumnCombs do
 7:         combinedHashes ← ArrayOfInts(chunkSize)
 8:         for columnIdx in columnCombination do
 9:             combinedHashes ← RotlXor(combinedHashes, hashedChunk[columnIdx])
10:         end for
11:         for hash in combinedHashes do
12:             if invertedIndex.HasKey(hash) then
13:                 invertedIndex.Insert(hash, columnCombination)
14:             else
15:                 hll.Insert(hash)
16:             end if
17:         end for
18:     end for
19: end for
```

## 5.2. Выбор хеш-таблицы

Стоит отметить, что с инвертированным индексом происходит две операции (листинг 2): это поиск хеша в нём (строка 12) и добавление новых колонок, в которых данный хеш содержится (строка 13). Поэтому в Java-реализации из Metanome инвертированный индекс представлен как хеш-таблица с целочисленными ключами. Она отображает хеш-значения на множества комбинаций колонок, в которых содержатся соответствующие значения хешей. Эти множества также представлены хеш-таблицами. Причем для этих целей вместо хеш-таблиц из стандартной библиотеки Java используются специальные, оптимизированные под целочисленные индексы быстрые хеш-таблицы из библиотеки

`fastutil` [17].

В ходе работы было выяснено, что хеш-таблица из стандартной библиотеки C++ не обеспечивает достаточную производительность для данной части алгоритма. Поэтому для реализации алгоритма на C++ тоже было решено использовать сторонние хеш-таблицы. Для C++ существует множество различных хеш-таблиц, сравнение производительности которых приведено в исследовании [14]. В алгоритме, приведенном на листинге 2, для хеш-таблицы инвертированного индекса критична скорость поиска (строка 12) — если у таблицы $N$ строк и на текущем уровне содержится $M$ комбинаций колонок, то операция поиска в инвертированном индексе произойдёт $N * M$ раз. Поэтому для этой цели была выбрана хеш-таблица `emhash7` из семейства `emhash` [16] — она оптимизирована для хранения целочисленных значений, а также, согласно исследованию [14], обладает быстрым поиском. В качестве хеш-таблицы, которая представляет множество комбинаций колонок, была выбрана таблица `emhash2` из семейства `emhash` — в этом случае критична скорость вставки.

## 5.3. Векторизация

Заметим, что для удобства записи в строке 9 листинга 2 процесс вычисления хешей для проекций строк представлен в векторной записи: операции `ROTL` (циклический сдвиг влево) и `XOR` проводятся не для единичных хешей, как в листинге 1, а сразу для всего чанка внутри функции `RotlXor()`. При наивной реализации функции `RorlXor()`, представленной на листинге 3 алгоритм итерируется по вектору хешей и для каждого значения выполняет инструкции `ROTL` и `XOR`, чтобы вычислить значение объединенного хеша. Причем циклический сдвиг влево происходит на один бит, как было сделано в оригинальной имплементации. Поэтому было предложено векторизовать эти вычисления с помощью SIMD-расширения процессора.

Теперь сразу несколько элементов массива combinedHashes загружаются в векторный регистр (листинг 4, строки 2–3), и на этом реги-

**Листинг 3** RotlXor

**Require:** $combinedHashes, hashedChunk[colIdx]$
1: $rowIdx \leftarrow 0$
2: **for** $hash$ in $combinedHashes$ **do**
3: $hash \leftarrow Rotl(hash)$
4: $hash \leftarrow Xor(hash, hashedChunk[colIdx][rowIdx])$
5: $rowIdx \leftarrow rowIdx + 1$
6: **end for**

стре выполняются векторные операции `ROTL` и `XOR`. Таким образом, хеш проекций нескольких строк, которые поместились в векторный регистр, будет вычисляться за то же самое количество инструкций, за которое в обычной версии вычислялся хеш только одной строки. В частности, в текущей реализации алгоритма хеши представлены 64-битными целыми числами, и процессор, поддерживающий векторное расширение AVX2, предоставляет 256-битные векторные регистры. Таким образом, за одно и то же количество инструкций будет вычислено 4 хеша вместо одного. Однако векторные расширения до AVX2 включительно не поддерживают векторный вариант инструкций циклического сдвига `ROTL`, поэтому возникла необходимость сделать эквивалентную замену этой инструкции на три других: два побитовых сдвига и логическое “ИЛИ”. Исходные значения сдвигаются на один бит влево, на n-1 бит вправо, где n — размер хеша в битах, и потом к ним применяется логическое “ИЛИ”. Таким образом, вместо двух инструкций будет выполнено четыре: `XOR` и три инструкции, заменяющие `ROTL`. Стоит отметить, что расширение AVX-512, следующее за AVX2, позволяет использовать уже цельную инструкцию `ROTL`, не прибегая к замене.

**Листинг 4** RotlXor, векторизованный вариант

**Require:** $combinedHashes, hashedChunk[colIdx]$
1: $rowIdx \leftarrow 0$
2: **for** $hashVector$ in $combinedHashes$ **do**
3: $vectReg \leftarrow LoadVect(hashVector)$
4: $vectReg \leftarrow RotlVect(vectReg)$
5: $vectReg \leftarrow XorVect(vectReg, hashedChunk[colIdx][rowIdx])$
6: $hashVector \leftarrow StoreVect(vectReg)$
7: $rowIdx \leftarrow rowIdx + vectorSize$
8: **end for**

**Листинг 5** Замена ROTL на эквивалентные инструкции

**Require:** $vectReg, n$
1: $shiftedLeft \leftarrow ShiftLeft(vectReg, 1)$
2: $shiftedRight \leftarrow ShiftRight(vectReg, n-1)$
3: $vectReg \leftarrow Or(shiftedLeft, shiftedRight)$

## 5.4. Распараллеливание

Несмотря на предыдущие оптимизации, в большинстве случаев процесс заполнения структур всё ещё остается узким местом алгоритма. Рассмотрим цикл на листинге 2, строках 6–18. Для каждого чанка таблицы этот цикл итерируется по всем парам вида *(комбинация колонок, HLL)*, в каждой итерации выполняя вычисление объединенных хешей (строки 8–10) и вставку в структуры данных (строки 12–16). Как показало профилирование, именно этот цикл занимал много времени, поэтому было решено заменить последовательное выполнение цикла на параллельное. Однако наивным образом распараллелить эту часть кода оказалось невозможно: проблема заключается в методе `invertedIndex.Insert()`. Поскольку, как уже упоминалось, каждую комбинацию колонок можно закодировать целым числом, инвертированный индекс представлен в виде `map<int, set<int> >` — он отображает хеш на множество комбинаций, в которых этот хеш содержится. Таким образом, если при вставке два разных потока получат одинаковый хеш-ключ и попытаются выполнить вставку в соответствующее этому ключу множество, то может произойти гонка. Для решения этой проблемы было предложено два варианта. Первый — для каждого множества `set<int>` создать собственный `mutex` (либо spinlock), и тогда вставку комбинации колонок в такое множество можно защитить критической секцией. Второй вариант — заменить `set<int>` на lock-free bit vector, который реализован с помощью atomic-переменных. В этом случае операция выставления i-го бита эквивалента вставке i-ой комбинации в множество. При этом операция атомарна, то есть описанной выше гонки не произойдет. Преимущество такого подхода в том, что отсутствуют издержки на системные вызовы, которые могут иметь место при работе с мьютексами. Есть и недостаток — большее потреблении памяти: если в первом под-

ходе все `set<int>` изначально пусты, то в lock-free реализации каждый bit vector должен быть полностью аллоцирован, чтобы зарезервировать место под все возможные индексы комбинаций колонок.

# 6. Эксперименты: сравнение с Metanome

## 6.1. Исследовательские вопросы и метрики

В экспериментах производится сравнение C++-реализации Faida с уже существующей его реализацией на Java из инструмента Metanome. Как и в большинстве статей на тематику алгоритмов профилирования [8, 12], в качестве основной метрики производительности будет рассмотрено время работы алгоритмов. Цель экспериментального исследования - ответить на два исследовательских вопроса.

- **RQ1:** *Как предложенные оптимизации влияют на производительность C++-реализации?*
- **RQ2:** *Во сколько раз оптимизированная C++-версия превосходит по производительсти Java-версию из Metanome?*

## 6.2. Условия эксперимента

В процессе экспериментов каждый алгоритм был запущен 5 раз, после чего в качестве результата времени его работы было взято среднее арифметическое из этих запусков. Кроме того, поскольку алгоритмы активно работают с диском, для чистоты эксперимента при каждом запуске сбрасывался дисковый кеш.

### 6.2.1. Наборы данных

Для экспериментов было подобрано шесть наборов данных, каждый из которых интересен своими отличными от других характеристиками. Наборы и их различные параметры представлены в таблице 1. *Brazilian E-commerce* — набор реальных данных, который содержит некоторую информацию о продажах, а также довольно длинные строки с отзывами покупателей. *FitBit Fitness Tracker Data* интересен тем, что, во-первых, содержит достаточно много атрибутов, и во-вторых, огромное количество зависимостей включения. *TPC_H* — широко известный синтети-

ческий набор данных, часто используется для тестирования производительности баз данных. В таблице он представлен в двух вариантах с различным размером (scale-factor-ом). *haIndGen* — ещё один синтетический набор, его отличительная особенность в том что он содержит зависимости большой арности (до 7-й включительно). И, наконец, *CIPiblicHighway* — набор реальных данных, который содержит большое количество null-значений.

| **Название** | **Размер** | **Тип** | **#Таблиц** | **#Колонок** | **#Строк** | **#Унарных ЗВ** | **#N-арных ЗВ** |
|---|---|---|---|---|---|---|---|
| Brazilian E-Commerce [15] | 126.3 MB | Реальный | 9 | 52 | 1.6 M | 23 | 23 |
| FitBit Fitness Tracker Data | 330 MB | Реальный | 18 | 259 | 8.1 M | 8798 | ? (ML) |
| TPC-H (SF=1) | 1.1 GB | Синтетич. | 8 | 61 | 8.7 M | 96 | 99 |
| TPC-H (SF=10) | 11.1 GB | Синтетич. | 8 | 61 | 86.6 M | 97 | 103 |
| haINDgen (generator) | 862 MB | Синтетич. | 2 | 36 | 6.1 M | 18 | 221 |
| CIPublicHighway | 28.3 MB | Реальный | 1 | 18 | 0.4 M | 65 | 440 |

Таблица 1: Характеристики наборов данных

### 6.2.2. Характеристики тестового стенда

Эксперименты проводились на тестовом стенде со следующими характеристиками:

- CPU AMD Ryzen 7 4800H @ 2.90GHz x 8 cores (16 threads);
- 32 GiB RAM;
- SSD A-Data S11 Pro AGAMMIXS11P-512GT-C 512GB;
- OS Ubuntu 22.04;
- конфигурация Java: openjdk 19.0.1 2022- 10-18 OpenJDK Runtime Environment (build 19.0.1+10-Ubuntu-1ubuntu122.04);
- gcc (Ubuntu 11.3.0-1ubuntu1 22.04) 11.3.0, компиляция производилась с опцией -O3.

## 6.3. Эксперимент

Чтобы убедиться, что каждая предложенных из оптимизаций улучшает производительность, было создано пять реализаций на C++, в

каждой из которых постепенно добавлялись усовершенствования. Таким образом, самая первая реализация Default(2) аналогична Java-реализации из Metanome и не содержит ни одной из оптимизаций. В следующей версии HTab(3) была добавлена сторонняя хеш-таблица. На следующем шаге HTab+Buff(4) дополнительно была добавлена буферизация. Затем, в реализации HTab+Buff+SIMD(5) ко всем предыдущим оптимизациям были добавлены SIMD вычисления. И, наконец, последняя реализация HTab+Buff+SIMD+Par(6) представляет распараллеленную версию реализации HTab+Buff+SIMD(5). Параллельная версия запускалась на 16 потоках. Стоит отметить, что из двух параллельных реализаций, предложенных ранее, в экспериментах участвовала реализация с мьютексами. Это было сделано для более честного сравнения, потому что версия с мьютексами занимает столько же памяти, сколько и остальные, в отличие от lock-free варианта, который занимает чуть больше. Также, для сравнения, в экспериментах участвовала Java-реализация алгоритма из Metanome(1).

Как было упомянуто в секции 3.1, алгоритм Faida имеет два параметра. Один из них задает размер выборки, а второй — точность приближения HyperLogLog. В зависимости от различных значений параметров меняется точность алгоритма, но также и время его работы. В экспериментах было решено использовать такие же параметры, как в экспериментах из оригинальной статьи – значение первого параметра было установлено равным 500, а точность HLL — 0.1%. Как показали эксперименты авторов, эти параметры обеспечивают достаточную точность, при этом не нанося сильный ущерб производительности.

Для каждой из реализаций было получено усредненное значение времени выполнения. При этом, кроме общего времени работы, было измерено время работы основных этапов — это препроцессинг и валидация, частью которой является процесс заполнение структур. Полученные значения представлены в таблице 2, а также показаны на Рис. 3. Легенда рисунка следующая: сверху на оси OX указаны наборы данных, снизу на оси OX для каждого набора обозначены реализации. Реализации сгруппированы по наборам данных: на каждом из них был

запущен весь набор реализаций. На оси OY отображено время работы. Стоит отметить, что общий график разделен на два таким образом, что на одном представлена одна часть наборов, на другом вторая. Это было сделано для того, чтобы выбрать разный масштаб оси OY: поскольку на разных данных время работы алгоритмов сильно различается, такое представление более наглядно.

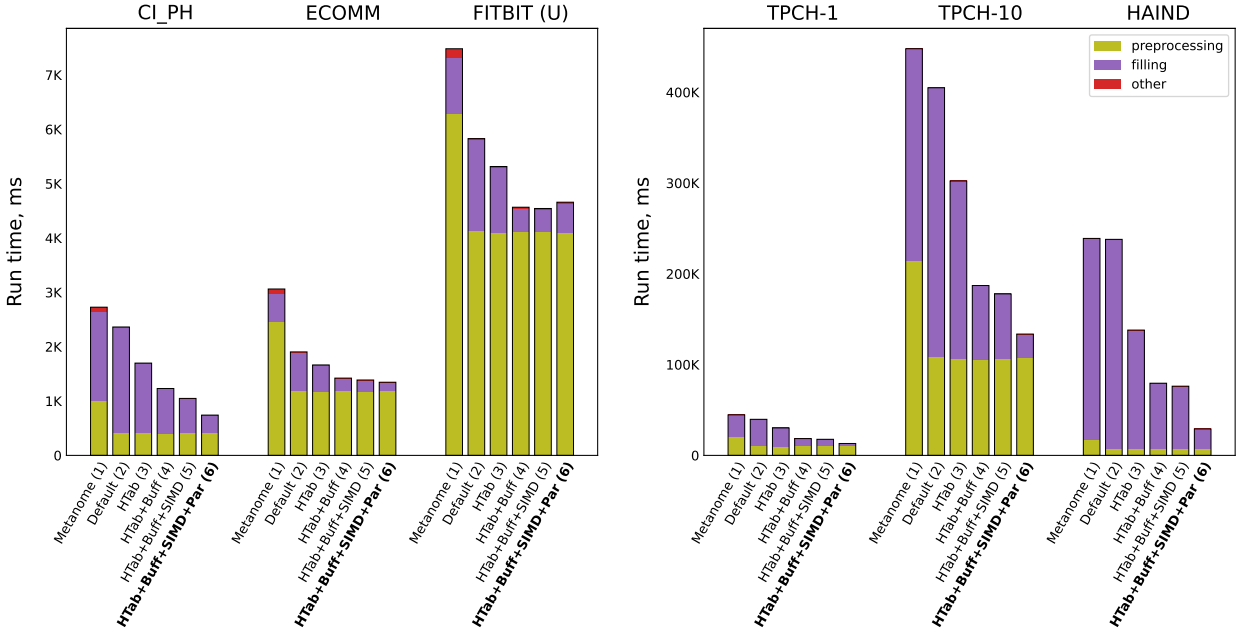


Рис. 3: Среднее время работы реализаций алгоритма Faida на различных наборах данных.

| | CIPH | FIT(U) | HAIND | ECOMM | TPCH-1 | TPCH-10 |
|---|---|---|---|---|---|---|
| Metanome (1) | 2.73 | 7.48 | 239.01 | 3.06 | 44.66 | 447.95 |
| Default (2) | 2.36 | 5.83 | 237.98 | 1.90 | 39.75 | 404.94 |
| HTab (3) | 1.70 | 5.32 | 138.04 | 1.66 | 30.25 | 302.27 |
| H+Buf (4) | 1.23 | 4.57 | 79.51 | 1.42 | 18.51 | 187.10 |
| H+B+SIMD (5) | 1.05 | 4.54 | 76.24 | 1.39 | 17.73 | 177.88 |
| H+B+S+Par(6) | 0.74 | 4.66 | 29.13 | 1.35 | 13.12 | 133.65 |

Таблица 2: Среднее время работы реализаций алгоритма Faida на различных наборах данных, в секундах.

Отдельный интерес может вызвать точность результатов алгоритма, поскольку он является приближенным и может выдавать ложноположительные результаты. Однако в данных экспериментах точность

не измерялась, потому что она была измерена ранее в работах [8, 12], и ни одна из предложенных оптимизаций на неё не влияет.

## 6.4. Результаты

По результатам экспериментов можно заключить, что каждая из оптимизаций даёт положительный результат и ускоряет алгоритм. При этом использование сторонней хеш-таблицы (HTab(3)) и буферизация (HTab+Buff(4)) дают более заметный прирост производительности, чем добавление SIMD-вычислений (HTab+Buff+SIMD(5)). Последовательную версию HTab+Buff+SIMD(5), включающую в себя три оптимизации, рекомендуется использовать на одноядерной машине. Наилучший результат среди всех показала параллельная версия HTab+Buff+SIMD+ +Par(6), которая содержит все предложенные оптимизации — именно эта версия рекомендуется к использованию при наличии на машине нескольких ядер. Коэффициент ускорения относительно Java-реализации для этих двух версий указан в таблице 3.

| | CIPH | FIT(U) | HAIND | ECOMM | TPCH-1 | TPCH-10 |
|---|---|---|---|---|---|---|
| Sequential(5) | 2.60 | 1.65 | 3.14 | 2.21 | 2.52 | 2.52 |
| Parallel(6) | 3.68 | 1.61 | 8.20 | 2.27 | 3.40 | 3.35 |

Таблица 3: Общее ускорение.

Нужно отметить, что в случае набора данных FITBIT ни Java-реализация, ни C++-реализация не смогли завершить поиск всех N-арных зависимостей. На втором уровне решетки находится более пяти миллионов 2-арных зависимостей, и при генерации кандидатов третьего уровня заканчивается оперативная память. Поэтому на графике для этого набора время работы приведено только для поиска унарных зависимостей (пометка U).

Теперь обратим внимание на время работы каждого из этапов — препроцессинга и валидации, в который входит в процесс заполнения структур. Предложенные оптимизации влияют только на этап заполнения, который соответствует алгоритму 2, и заметно ускоряют его на

всех наборах данных. Поэтому будет интересно рассмотреть, во сколько раз относительно Java-версии был ускорен этот этап для параллельной HTab+Buff+SIMD+Par(6) и наилучшей последовательной HTab+Buff+ +SIMD(5) реализаций. Коэффициент указан в таблице 4. Время работы этого этапа существенно зависит от свойств набора данных, таких как количество зависимостей, колонок, количества уникальных значений и других параметров, поэтому между разными наборами оно заметно различается. Также стоит отметить, что на производительность этого этапа влияет скорость чтения с диска — алгоритму нужно читать файлы колонок. Кроме того, на графике видно, что в случае FITBIT параллельная версия работает медленнее последовательной. Это вызвано спецификой набора данных — он содержит много одинаковых значений, и поэтому в процессе заполнения часто срабатывают блокировки мьютексов. Также был проведён эксперимент с ранее описанной параллельной lock-free реализацией, и в этом случае такой проблемы не возникало.

| | CIPH | FIT(U) | HAIND | ECOMM | TPCH-1 | TPCH-10 |
|---|---|---|---|---|---|---|
| Sequential(5) | 2.64 | 2.66 | 3.22 | 2.48 | 3.11 | 3.27 |
| Parallel(6) | 5.47 | 1.92 | 10.20 | 3.64 | 8.14 | 9.11 |

Таблица 4: Ускорение процесса заполнения структур.

Отдельного обсуждения заслуживает этап препроцессинга, описанный в разделе 3.1: во всех реализациях время его работы одинаково, потому что к нему не было применено никаких оптимизаций. Было выяснено, что это довольно прямолинейный этап, на производительность которого в основном влияют три аспекта: парсер входных файлов, скорость чтения/записи на диск и хеш-функция. Рассмотрим их подробнее. Предложенная C++-реализация привязана к текущему внутреннему парсеру Desbordante, поэтому на момент выполнения работы не было возможности его ускорить или заменить на другой. Однако также были проведены эксперименты с более эффективным сторонним парсером, в этом случае удалось заметно увеличить производительность. Существенно влияет на производительность и скорость диска: алгоритм

не только читаем входной файл, но и в то же время записывает полученные хешированные значения в файлы колонок и выборки. Кроме того, важна хорошая хеш-функция, которая может быть медленной: от её качества зависит процент коллизий, что влияет на точность алгоритма.

### 6.4.1. Вывод

Таким образом, на поставленнные исследовательские вопросы можно ответить следующим образом:

- **RQ1:** *Как предложенные оптимизации влияют на производительность C++-реализации?*

  Можно заключить, что в большинстве случаев все предложенные оптимизации дают прирост производительности, и версия, содержащая все эти оптимизации, существенно превосходят неоптимизированную C++-версию.

- **RQ2:** *Во сколько раз оптимизированная C++-версия превосходит по производительсти Java-версию из Metanome?*

  При упомянутых ранее характеристиках тестового стенда прирост производительности составил среднем 3.7 раза, а максимально — 8 раз.

## 6.5. Угрозы валидности

Данное экспериментальное исследование имеет следующие угрозы валидности:

- На стадии препроцессинга оба алгоритма активно работают с диском, производят чтение и запись. Поэтому на итоговое время их работы значительно влияет скорость диска. При проведении экспериментов был использован быстрый SSD диск, но при использовании других, более медленных дисков, например, HDD, время работы алгоритмов может сильно увеличиться.

- На время работы алгоритмов влияют характеристики набора данных: например, количество уникальных значений в колонках, количество колонок, количество удерживающихся зависимостей. В экспериментах алгоритмы были протестированы на данных с разнообразными характеристиками, но невозможно гарантировать, что рассмотрены все случаи.

- При запуске java-реализации алгоритмов мз Metanome не варьировались параметры виртуальной машины Java. В теории, подстройка параметров могла повлиять на общую производительность, однако предыдущие исследования [5] показали, что это не даёт существенного улучшения.

# Заключение

В ходе работы над созданием высокопроизводительной реализации алгоритма Faida в рамках профайлера данных Desbordante были выполнены следующие задачи:

- произведён краткий обзор предметной области зависимостей включения и алгоритма Faida, выделены три основные стадии работы алгоритма: препроцессинг, генерация кандидатов и валидация;
- алгоритм реализован на языке C++, в качестве эталонной реализации была взята Java-реализация алгоритма из Metanome;
- исследована производительность кода и реализованы оптимизации:
  - сторонняя хеш-таблица,
  - буферизация,
  - векторизация с использованием SIMD-инструкций,
  - распараллеливание;
- проведено сравнение производительности C++-реализации с существующей Java-реализацией алгоритма из Metanome. Эксперименты показали, что все предложенные оптимизации улучшают производительность, и оптимизированная C++-реализация работает в среднем в 3.7 раз быстрее Java-реализации.

На основе результатов работы была подготовлена статья [9], которая была принята на конференцию FRUCT'23 (публикуется в IEEE Xplore и индексируется в Scopus). Код алгоритма доступен[3] на Github.

[3] https://github.com/alexandrsmirn/Desbordante/tree/faida-tests-final-parallel/src/algorithms/inclusion_dependencies/faida

# 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] Appleby Austin. SMHasher. — URL: https://github.com/aappleby/smhasher (дата обращения: 2022-05-1).

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

[4] De Marchi Fabien, Lopes Stéphane, Petit Jean-Marc. Efficient Algorithms for Mining Inclusion Dependencies // Advances in Database Technology — EDBT 2002 / Ed. by Christian S. Jensen, Simonas Šaltenis, Keith G. Jeffery et al. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2002. — P. 464–476.

[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). — 2021. — P. 344–354.

[6] Desbordante: from benchmarking suite to high-performance science-intensive data profiler (preprint) / George A. Chernishev, Michael Polyntsov, Anton Chizhov et al. // CoRR. — 2023. — Vol. abs/2301.05965. — arXiv : 2301.05965.

[7] Divide & Conquer-Based Inclusion Dependency Discovery / Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, Felix Naumann // Proc. VLDB Endow. — 2015. — feb. — Vol. 8, no. 7. — P. 774–785. — URL: https://doi.org/10.14778/2752939.2752946.

[8] Fast Approximate Discovery of Inclusion Dependencies / Sebastian Kruse, Thorsten Papenbrock, Christian Dullweber et al. // Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings / Ed. by Bernhard Mitschang, Daniela Nicklas, Frank Leymann et al. — Vol. P-265 of LNI. — GI, 2017. — P. 207–226. — URL: https://dl.gi.de/20.500.12116/629.

[9] Fast Discovery of Inclusion Dependencies with Desbordante (paper accepted) / Alexander Smirnov, Anton Chizhov, Ilya Shchuckin et al. // 2023 33th Conference of Open Innovations Association (FRUCT). — 2023.

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

[11] HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm / Philippe Flajolet, Éric Fusy, Olivier Gandouet, Frédéric Meunier // AofA: Analysis of Algorithms / Ed. by Philippe Jacquet. — Vol. DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07) of DMTCS Proceedings. — Juan les Pins, France : Discrete Mathematics and Theoretical Computer Science, 2007. — jun. — P. 137–156. — URL: https://hal.inria.fr/hal-00406166.

[12] Inclusion Dependency Discovery: An Experimental Evaluation of Thirteen Algorithms / Falco Dürsch, Axel Stebner, Fabian Windheuser et al. — CIKM '19. — New York, NY, USA : Association for Computing Machinery, 2019. — P. 219–228. — URL: https://doi.org/10.1145/3357384.3357916.

[13] Kruse Sebastian, Papenbrock Thorsten, Naumann Felix. Scaling Out the Discovery of Inclusion Dependencies // Datenbanksysteme

für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings / Ed. by Thomas Seidl, Norbert Ritter, Harald Schöning et al. — Vol. P-241 of LNI. — GI, 2015. — P. 445–454. — URL: https://dl.gi.de/20.500.12116/2421.

[14] Leitner-Ankerl Martin. Comprehensive C++ Hashmap Benchmarks 2022. — URL: https://martin.ankerl.com/2022/08/27/hashmap-bench-01 (дата обращения: 2022-05-1).

[15] Olist, Sionek André. Brazilian E-Commerce Public Dataset by Olist. — 2018. — URL: https://www.kaggle.com/dsv/195341.

[16] fast and memory efficient open addressing c++ flat hash table/map. — URL: https://github.com/ktprime/emhash (дата обращения: 2022-05-1).

[17] fastutil: Fast & compact type-specific collections for Java. — URL: https://fastutil.di.unimi.it/ (дата обращения: 2022-05-1).

[18] openclean — Data Cleaning for Python. — URL: https://github.com/VIDA-NYU/openclean (дата обращения: 2022-05-1).

[19] pandas-profiling. — URL: https://github.com/ydataai/pandas-profiling (дата обращения: 2022-05-1).