Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Струтовский Максим Андреевич

Платформа для высокопроизводительного поиска зависимостей в базах данных

Бакалаврская работа

Научный руководитель:
к. ф.-м. н., доцент Михайлова Е.Г.

Консультант:
ассистент Чернышев Г.А.

Рецензент:
к. т. н., доцент Григорьева А.В.

Санкт-Петербург
2021

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems

Maksim Strutovskii

A platform for high-performance discovery of dependencies in databases

Graduation Thesis

Scientific supervisor:
Associate Professor, Ph.D. Elena Mikhaylova

Consultant:
Assistant George Chernishev

Reviewer:
Assistant Professor Anastasia Grigorieva

Saint-Petersburg
2021

Оглавление

1. Введение

Зависимости в базах данных представляют собой закономерности в данных. Владение подобной метаинформацией облегчает решение многих задач, таких как: нормализация схемы, очистка данных и оптимизация запросов [3, 25, 26]. Многие подвиды зависимостей, такие как рассматриваемые в данной работе функциональные зависимости, соответствуют известным в предметной области правилам, и зачастую эксперт может определить большинство зависимостей, имеющих место в данной таблице.

Если подобное исследование зависимостей недоступно, используется автоматическое обнаружение зависимостей — его целью является составление списка всех зависимостей, удерживающихся на заданной таблице. К сожалению, в данной постановке задача имеет сложность $\mathcal{O}(N \cdot 2^M)$, где N — количество кортежей, M — количество атрибутов [15].

Тем не менее, в области были предложены дюжины алгоритмов поиска функциональных зависимостей, и в каждом проблема решается уникальным способом. Таким образом, становится актуальной платформа:

1. позволяющая использовать эффективные реализации алгоритмов для непосредственного поиска зависимостей;

2. реализующая различные алгоритмы с использованием одинаковых технологий для их последующего сравнения и проведения прочих экспериментов.

При анализе предметной области было найдено единственное решение, подходящее на роль подобной платформы. Metanome — открытая платформа, на уровне бэкэнда реализованная на языке программирования Java, основной целью которой является анализ метаданных [8]. Несмотря на множество положительных сторон, Metanome не удовлетворяет вышеупомянутым требованиям в полной мере по следующим причинам:

- Программы, реализованные на языке Java, обладают достаточной производительностью при решении бизнес-задач, однако для решения задач, требующих высокопроизводительных вычислений, существуют другие языки и технологии, позволяющие добиться более высокой эффективности.

- Используемые при работе кода, написанного на Java, технологии наподобие динамической компиляции и сборки мусора вносят неопределенность в скорость работы программы, что усложняет задачу оценки производительности [14].

Таким образом, было решено спроектировать и реализовать альтернативную платформу для высокопроизводительного поиска функциональных зависимостей, получившую название Desbordante[1]. Чтобы избежать недостатков, присущих Metanome, было принято решение использовать для реализации язык программирования C++:

- Благодаря широким возможностям по низкоуровневой оптимизации и предоставлению программисту полного контроля над исполняемым кодом, C++ является одним из самых высокопроизводительных языков программирования. При этом C++ предоставляет обширный инструментарий для разработки масштабных, расширяемых проектов.

- Наличие полного контроля над исполняемым кодом также позволяет уменьшить флуктуации при проведении экспериментов.

1.1. Постановка задачи

Целью данной работы является реализация платформы для высокопроизводительного поиска функциональных зависимостей. Для достижения данной цели были поставлены следующие задачи:

1. Выполнить обзор алгоритмов поиска функциональных зависимостей, а также существующих решений.

[1] безграничный (исп.)

2. Спроектировать и реализовать платформу для высокопроизводительного поиска функциональных зависимостей.

3. Сравнить производительность реализованной платформы и существующего решения Metanome.

    - выработать метрики и показать превосходство предложенной платформы;
    - показать, что невозможно достичь аналогичного уровня производительности с помощью настройки Metanome;
    - исследовать причины повышенного уровня производительности.

4. С помощью разработанной платформы исследовать возможность уменьшения трудоёмкости алгоритмов засчёт ограничения максимального размера левой части зависимости

2. Обзор

2.1. Основные понятия

Определение 1. Функциональная зависимость (ФЗ) определяет связь между атрибутами отношения [13]. Говорят, что между двумя множествами атрибутов X и Y в таблице данных имеет место функциональная зависимость $X \to Y$, если любые два кортежа, совпадающие на атрибутах X, совпадают на атрибутах Y. В этом случае, X называется левой частью функциональной зависимости, а Y — правой.

Определение 2. Если Y не зависит функционально ни от одного подмножества X, функциональную зависимость $X \to Y$ называют минимальной [13].

Поиск всех функциональных зависимостей в таблице сводится к нахождению минимальных зависимостей с правой частью, состоящей из одного атрибута, так как остальные функциональные зависимости можно вывести с помощью правил вывода Армстронга [2].

Недостаток такого определения функциональной зависимости заключается в том, что артефакты в реальных данных, такие как опечатки, отсутствующие значения и грязные данные, могут привести к ситуации, когда ФЗ не может быть выведена, хотя семантически она должна удерживаться. В качестве решения этой проблемы было сформулировано понятие приближённой функциональной зависимости, где под “приближённостью” подразумевается то, что небольшая часть кортежей может различаться на правой части при совпадении на левой. Таким образом, алгоритмы поиска приближённых ФЗ параметризуются значением максимальной погрешности e_{max}, служащим формальным критерием того, что не совпадает только “небольшая” часть кортежей, и зависимость-кандидат можно записать как приближённую ФЗ ($e(X \to Y) \leq e_{max}$). Стоит отметить, что приближённые ФЗ являются обобщением точных ФЗ: действительно, если взять $e_{max} = 0$, ни одна пара совпадающих на левой части кортежей не может различаться на правой части.

Определение 3. Пусть r — отношение, и $X \to Y$ —потенциальная приближённая ФЗ. Тогда погрешность вычисляется как [17]:

$$e(X \to Y, r) = \frac{|\{(t_1, t_2) \in r^2 | t_1[X] = t_2[X] \wedge t_1[Y] \neq t_2[Y]\}|}{|r|^2 - |r|}$$

Приближённая ФЗ $X \to Y$ удерживается на r, если $e(X \to Y, r) \leq e_{max}$.

2.2. Классификация алгоритмов

В работах, посвящённых поиску ФЗ, принято выделять три основных класса алгоритмов, по типу используемой стратегии поиска.

1. *Исследование частично упорядоченного множества возможных зависимостей.* Алгоритмы этого типа представляют множество всех возможных функциональных зависимостей в виде решётки, которую исследуют, отбрасывая неминимальные зависимости и исключая её части, исследование которых не приведёт к обнаружению новых зависимостей. Размерность этой решётки зависит экспоненциально от количества атрибутов, поэтому данный класс алгоритмов обладает низкой производительностью на “широких” таблицах. К данному классу можно отнести алгоритмы TANE [30], FUN [21], FDMine [33].

2. *Обработка множеств согласованности и разности кортежей.* Алгоритм попарно сравнивает кортежи, формируя пространство для поиска, после чего генерирует удерживающиеся зависимости. Сложность таких алгоритмов зависит квадратично от количества кортежей, и время их работы существенно снижается на “длинных” таблицах. Представителями данного типа являются Dep-Miner [18], FastFDs [32], FDep [12].

3. *Гибридный подход.* Современный гибридный подход был применён в алгоритме HyFD [24], чтобы разрешить проблему слабой масштабируемости существующих решений при росте числа атрибутов и кортежей. Его авторы предлагают разделить процесс поиска

ФЗ на две отдельные фазы: получение кандидатов на небольшом подмножестве таблицы и проверка кандидатов на всём отношении. Такая идея оказалась крайне плодотворной: авторы HyFD показали его превосходство над остальными алгоритмами, и в дальнейшем подход был расширен для поиска приближённых зависимостей. В работе [17] был предложен алгоритм Pyro, который и был реализован в рамках настоящей работы. Кроме того, подход был использован для поиска зависимостей в условии динамически обновляющихся данных. Авторами [10] описан процесс поиска и поддержания набора ФЗ при наличии запросов вида UPDATE, INSERT и DELETE, позволяющий избежать полного повторного вычисления ФЗ на всей таблице. Алгоритмы инкрементального поиска ФЗ являются особенно востребованными при работе с большими данными [5, 15].

В итоге, в рамках настоящей работы было решено реализовать два алгоритма:

1. TANE, будучи относительно простым классическим алгоритмом, хорошо подходит для проверки быстродействия C++, а также для отладки и оптимизации основных структур. Помимо того, TANE зачастую используется в работах при сравнении алгоритмов, поэтому его наличие необходимо для повторения соответствующих экспериментов.

2. Pyro на момент написания данной работы является последним предложенным алгоритмом. Реализация Pyro позволит не только провести массу экспериментов для исследования его нетривиального поведения, но и получить наиболее востребованный для индустриальных целей алгоритм.

2.3. Проприетарные решения

Насколько известно автору, не существует коммерческих решений, нацеленных на эффективную реализацию алгоритмов поиска ФЗ. Вместо

этого, подобный функционал включён в индустриальное ПО, предназначенное для управления данными [1]. Ниже приведена сводная информация по возможностям таких инструментов касательно поиска ФЗ:

- SAP Information Steward позволяет запустить Dependency profile task с целью найти удерживающиеся в данной таблице приближённые ФЗ, левая часть которых состоит только из одного атрибута [28]. Такое ограничение делает невозможным поиск зависимостей с несколькими атрибутами в левой части, зато значительно уменьшает трудоёмкость задачи. Кроме того, есть возможность задать ФЗ вручную — тогда инструмент проверит её корректность и выведет количество кортежей, на которых заданная зависимость нарушается.

- Oracle Warehouse Builder [23], Informatica Data Quality [16] обладают теми же возможностями, что и SAP Information Steward, с одним дополнительным ограничением: автоматически обнаруживаемые зависимости могут иметь только один атрибут в правой части. Описанным ограничением можно пренебречь, так как множество зависимостей вида $1 \to 1$ тривиально преобразуется в зависимость $1 \to N$.

- Microsoft SQL Server Data Profiling Task — процесс, который можно запустить в СУБД Microsoft SQL Server [29]. Данный процесс позволяет проверить множество вручную заданных точных и приближённых ФЗ, причём некоторые из перечисленных кандидатов будут отброшены, например, $AB \to C$, если уже известно, что $A \to C$. Пусть такое правило и позволяет ускорить перебор, оно не может предоставить уровень производительности, аналогичный алгоритмам поиска ФЗ. Как и SAP Information Steward, данный инструмент позволяет задавать максимальное и выводить текущее количество кортежей, противоречащих зависимости.

- Talend Open Studio предоставляет возможность задать и проверить на имеющихся данных ФЗ вида $1 \to 1$ [31]. Кроме того, вы-

водится доля кортежей, противоречащих зависимости.

Подводя итоги, практически все перечисленные проприетарные решения предоставляют возможность проверки заданных вручную ФЗ, помимо того большая часть решений предлагает функционал по автоматическому поиску зависимостей, либо ограниченный лишь одним атрибутом в левой части, либо использующий неэффективный наивный подход с перебором (Microsoft SQL Server). Таким образом, представленные инструменты не соответствуют требованиям, сформулированным в данной работе, и не могут рассматриваться в качестве альтернативы к проектируемой платформе.

2.4. Исследовательские решения

Помимо множества коммерческих инструментов, существует набор исследовательских решений, посвящённых профилированию и исследованию данных. Среди них встречаются проекты, частично предназначенные для поиска ФЗ:

- Data Auditor — инструмент для исследования качества и семантики данных, предложенный в работе [7]. Data Auditor принимает на вход ФЗ и порождает немного отличающиеся от неё ФЗ, по которым генерируется список условных функциональных зависимостей (conditional functional dependencies), удерживающихся в данной таблице. Таким образом, основной идеей является получение условных ФЗ, соответствующих рассматриваемой таблице, по заданной вручную ФЗ, которой данная таблица по той или иной причине не удовлетворяет.

- RuleMiner — система для обнаружения ограничений целостности в данных, описанная авторами [27]. В работе рассматривается понятие ограничений отрицания (denial constraints), обобщающих ФЗ, первичные ключи и другие правила, и описывается приложение, позволяющее автоматически находить ограничения отрицания и,

следовательно, ФЗ. Тем не менее, используемые алгоритмы являются ещё более ресурсоёмкими: кластеру из 20 машин может потребоваться около суток для обработки таблицы, состоящей из 13 атрибутов и $1 \cdot 10^6$ кортежей [6].

- FDTool — приложение для автоматического поиска полного набора функциональных зависимостей на заданном отношении [11]. Инструмент был спроектирован для декомпозиции "длинных" и "узких" таблиц с медицинскими данным через применение ФЗ. Для достижения поставленной цели авторы выбрали алгоритм FDMine и реализовали его вместе с модулями ввода-вывода и манипуляции результатами на языке Python. Несмотря на то, что инструмент решает рассматриваемую в настоящей работе задачу, он не отвечает сформулированным требованиям. Во-первых, FDTool не был спроектирован с целью его последующего расширения альтернативными алгоритмами, во-вторых, выбор FDMine — эффективного только на специфичных датасетах алгоритма — и Python, обладающего сравнительно низкой производительностью, ведёт к высокому времени работы FDTool в общем случае.

Таким образом, рассмотренные выше исследовательские решения либо не решают основную задачу — автоматический поиск набора ФЗ, — либо не соответствуют требованиям эффективности и расширяемости.

2.5. Metanome

Metanome — открытая платформа, на уровне бэкэнда реализованная на языке программирования Java, основной целью которой является анализ метаданных [8]. Данный инструмент — единственный, согласно знаниям автора, инструмент, предоставляющий пользователю:

1. структуру для разработки и тестирования алгоритмов поиска зависимостей,

2. поддержку разнообразных типов данных и подключаемых СУБД,

3. фронтенд, доступный через браузер.

С точки зрения разработчика, Metanome обладает достаточным набором возможностей. Предоставляются все необходимые интерфейсы для разработки алгоритмов, нацеленных на поиск: функциональных и приближённых зависимостей, зависимостей включения и порядка (inclusion, order dependencies), уникальных наборов колонок (unique column combination) и т.д. Кроме того, Metanome содержит реализацию многих структур данных, используемых в поиске, таких как: партиции, наборы разности и согласованности и различные структуры для обхода исследуемого пространства кандидатов. Наконец, в платформу встроен функционал для обработки реляционных данных.

С точки зрения аналитика данных, Metanome является инструментом для получения и анализа метаинформации в данных. На ввод можно подавать текстовые файлы с данными или получать таблицы через соединение с СУБД. Реализована масса алгоритмов, упакованных в формате jar, которые подключаются к платформе перемещением jar-файла в соответствующую директорию. По результатам работы алгоритма собирается высокоуровневая статистика, которая подаётся пользователю вместе с простой визуализацией найденных зависимостей.

3. Архитектура Desbordante

Desbordante — консольное приложение, предоставляющее на данный момент базовый функционал по выводу списка ФЗ (Приближённых и Точных), удерживающихся на заданном csv-файле. Предполагаемый сценарий использования выглядит следующим образом:

1. Пользователь вызывает из терминала приложение, устанавливая через его аргументы используемый алгоритм, параметры алгоритма, файл с таблицей и параметры датасета, такие как разделительный символ и наличие заголовка.

2. Desbordante считывает файл и переводит таблицу во внутреннее сжатое представление (`ColumnLayoutRelationData`).

3. Управление передаётся алгоритму, который, следуя своей логике и используя специфичные структуры данных, накапливает множество ФЗ.

4. Наконец, в консоль выводится множество найденных ФЗ и некоторые статистики, полезные для анализа работы алгоритма, например, время его работы.

Классы:

- `CSVParser` — предназначен для приведения поступающего csv-файла в двумерный массив строк, представляющий таблицу.

- `ColumnData`, `RelationData`, `ColumnLayoutRelationData`, `RelationalSchema` — сущности, хранящие данные о схеме и теле отношения.

- `Column`, `Vertical` — классы, позволяющие работать с множествами атрибутов.

- `FD`, `PartialFD` — соответствуют точным и приближённым ФЗ. Наличие специальных классов, отведённых под эти роли, позволяет

добиться в алгоритмах единого представления ФЗ, например, для функционирования описанной ниже системы верификации.

- `FDAlgorithm` — класс, обобщающий поведение алгоритма поиска ФЗ.

- `PositionListIndex` — необходим для работы с партициями.

В процессе написания алгоритмов стало очевидно, что необходима автоматическая система валидации алгоритмов, позволяющая проверять корректность вывода алгоритма не только однократно, по завершении разработки, но и регулярно при внесении изменений, оптимизирующих скорость работы и исправляющих ошибки, в код. В итоге, в приложение был добавлен тестирующий модуль, позволяющий проверить алгоритм тремя способами:

1. Проверить корректность результатов работы алгоритма на небольших синтетических датасетах, где известен набор удерживающихся ФЗ.

2. Удостовериться в консистентности работы на одном и том же датасете, то есть проверить совпадение набора ФЗ, обнаруживаемого новой версией алгоритма и протестированной старой. Для ускорения процесса при первом запуске протестированного алгоритма на конкретной таблице вычисляется и записывается контрольная сумма Adler-32 [19] для строкового представления найденного набора ФЗ — так нет необходимости в запуске обеих версий алгоритма и в хранении самого набора ФЗ.

3. При разработке нового алгоритма сверить его результирующий набор ФЗ с набором, выдаваемым уже реализованным и проверенным алгоритмом.

Платформа Desbordante написана полностью на языке C++ стандарта 17 с активным применением средств стандартной библиотеки и библиотеки boost. В качестве системы сборки выбран CMake, что

облегчает процесс развёртывания на разных системах. Для валидации алгоритмов и написания юнит-тестов используется инфраструктура Google Test. В качестве логгирующего модуля была выбрана библиотека easyloggingcpp в силу простоты её освоения и интеграции, соответствия возникающим задачам и наличия регулярных обновлений. Наконец, используется система контроля версий git, а сам исходный код доступен на GitHub[2].

[2]https://github.com/Mstrutov/Desbordante/

4. Подготовка экспериментов

4.1. Вопросы экспериментального исследования

Для исследования Pyro и производительности Desbordante были сформулированы следующие исследовательские вопросы (ИВ):

1. ИВ1: Удалось ли добиться улучшения в плане времени исполнения и потребляемой памяти при реализации алгоритма на C++?
2. ИВ2: Возможно ли достичь аналогичного ускорения для реализации на Java с помощью настройки среды исполнения? То есть, можно ли получить скорость работу, близкую к C++, подобрав параметры виртуальной машины и прочие настройки, не изменяя исходный код?
3. ИВ3: Как можно объяснить различия в измеренных метриках?
4. ИВ4: Каким образом на производительность влияет изменение максимального размера левой части?

4.2. Среда для проведения экспериментов

Датасет	Источник	Строки	Атрибуты	Размер	#ФЗ	#NULL
Adult	uci[3]	32K	15	3.5 MB	78	0, 0%
BreastCancer	uci[3]	500	30	118 KB	11835	0, 0%
CIPublicHighway	datasa[4]	427K	18	27 MB	143	3.5M, 46.3%
EpicMeds	epf[5]	1.3M	10	55 MB	17	280K, 2.2%
EpicVitals	epf[5]	1.2M	7	33 MB	2	0, 0%
Iowa1KK	mydata.iowa.gov[6]	1M	24	210 MB	1585	1M, 4.2%
LegacyPayors	epf[5]	1.4M	4	21 MB	6	0, 0%
Neighbors100K	SDSS[7]	100K	7	6.4 MB	15	0, 0%
SG_Bioentry	BioSQL[8]	184K	8	24 MB	19	184K, 11.1%

Таблица 1: Обзор отобранных датасетов

[3]http://archive.ics.uci.edu/ml/index.php
[4]https://data.sa.gov.au/
[5]https://www.epa.gov/
[6]https://www.opendatanetwork.com/dataset/mydata.iowa.gov/m3tr-qhgy
[7]https://www.sdss.org/dr13/
[8]https://biosql.org/wiki/Downloads

В таблице 1 представлен набор отношений, используемых для проведения экспериментов. При отборе отношений особое внимание было уделено двум критериям:

1. отношения должны состоять из реальных данных;
2. отношения должны быть взяты из разных источников и обладать различными характеристиками.

Характеристики отношений, такие как: размерность, количество ФЗ и доля NULL-значений — также представлены в таблице.

Все эксперименты были выполнены с использованием алгоритма Pyro, поскольку, во-первых, понимание характеристик более современного и эффективного алгоритма является более ценным, и, во-вторых, более сложная логика алгоритма открывает больше направлений для разноплановых экспериментов.

В качестве платформы для исполнения кода был использован настольный ПК с операционной системой Pop!_OS 20.10 64-bit, компилятором C++ gcc 10.2.0, виртуальными машинами Java: OpenJDK 64-bit Server VM 11.0.9.1, GraalVM CE 21.0.0 — и комплектующими: Intel(R) Core(TM) i5-7600K CPU (4 ядра) @ 3.80GHz, 16GB DDR4 2133MHz RAM, 2TB HDD WD20EZAZ.

Наконец, была исключена возможность возникновения дополнительной нагрузки на систему через, например, исполняющиеся параллельно тяжеловесные процессы или системные обновления.

1. Во-первых, одинаковая нагрузка системы при проведении экспериментов с обеими платформами обеспечивает равные условия, в которых результаты зависят только от эффективности самих реализаций, а не от внешних факторов.
2. Во-вторых, отсутствие сторонних процессов, влияющих на результаты, облегчает формирование и проверку гипотез, объясняющих экспериментально полученные результаты.

3. Напоследок, так как поиск ФЗ — ресурсоёмкая задача, следует ожидать, что при индустриальном использовании под неё будет выделена отдельная платформа.

4.3. Оцениваемые метрики

Базовый набор, используемый в большинстве работ для оценки эффективности алгоритмов, состоит из двух метрик: времени работы и требуемой памяти. Очевидно, лучшим алгоритмом в отдельно взятом сценарии будет считаться такой алгоритм, который корректно закончит работу в условиях ограниченной памяти за наименьшее время.

Кроме того, в данной работе уделено внимание двум дополнительным параметрам, способным показать эффективность имплементации алгоритма:

1. максимальный размер левой части зависимости, при котором потребляемая алгоритмом память не превышает установленную;

2. количество выбросов при многократном запуске конкретной реализации алгоритма — показывает стабильность платформы.

Таким образом,

- Для сравнения времени исполнения на 10 последовательных запусках строились доверительные интервалы с уровнем доверия 95%.

- Для сравнения потребляемой памяти использовался инструмент massif из набора valgrind в случае C++ и jstat в случае Java.

- Для оценки максимального размера левой части были проведены запуски с разным значением этого параметра, и фиксировалось значение, при котором программа запрашивала больше допустимого объёма памяти. Для ограничения потребляемой памяти, соответственно, использовалась утилита ulimit и средства, встроенные в виртуальную машину Java.

4.4. Обеспечение стабильности экспериментов

На ранних стадиях экспериментирования было отмечено, что на некоторых таблицах замеряемое время исполнения могло различаться от запуска к запуску вплоть до 20 раз, что не могло быть объяснено особенностями Java или параллельным исполнением тяжеловесного процесса. Наличие подобных флуктуаций, разумеется, приводит к пересечению доверительных интервалов и делает невозможным сравнение двух платформ, поэтому было проведено дальнейшее исследование возможных причин и выявлены два фактора:

1. Во-первых, Pyro использует генератор случайных чисел для получения образцов данных, определяющих направление дальнейшего обхода пространства поиска и позволяющих избежать трудоёмкого вычисления ошибки на всей таблице. Таким образом, эффективность подхода Pyro напрямую зависит от качества отбираемых образцов данных, что происходит случайным образом. Соответственно, изменение поставляемой алгоритму последовательности псевдослучайных чисел может привести к изменению порядка обхода пространства и, в свою очередь, к изменению времени работы. Более того, таблицы, содержащие большое количество NULL-значений, влекут к выбору образца со столь же высокой долей NULL-значений, и эффективность Pyro снижается на порядки при неудачной последовательности случайных чисел, когда большинство выбранных образцов обладают этим свойством. Как итог, для стабильных и повторяемых экспериментов необходимо реализовать одинаковый генератор для обеих реализаций, и особенно это актуально для таблиц с высоким содержанием NULL-ов.

2. Во-вторых, было замечено, что время исполнения конкретной реализации алгоритма могло значительно меняться даже после описанных модификаций. Объяснением такого поведения оказался тот факт, что современные процессоры адаптируют свою тактовую частоту под уровень нагрузки, экономя таким образом энергию [4]. В итоге, при проведении экспериментов было решено фик-

сировать допустимую частоту для каждого ядра при помощи команды cpufreq-set.

5. Сравнение с Metanome

5.1. Сравнение производительности

Для сравнения производительности Metanome и Desbordante были развёрнуты без дополнительных модификаций, в поставляемом виде. Соответственно, для Metanome это сборка и исполнение через OpenJDK 11 с настройками по умолчанию, для Desbordante — сборка через gcc с уровнем оптимизаций "-O3". В каждом эксперименте использовался алгоритм Pyro со значением ошибки приближённой ФЗ 0.01, и измерялось время непосредственно работы алгоритма, от окончания обработки csv-файла до получения набора ФЗ.

Стоит заметить, что реализация на C++ имеет широкие возможности по ускорению работы, например, через настройку библиотечных или имплементацию собственных структур данных и использование альтернативных распределителей памяти. Известно, что подключение таких аллокаторов может заметно уменьшить время работы программы на C++ [20]. С другой стороны, хотя для Metanome также используются стандартные настройки среды исполнения, в ИВ2 было выявлено, что они являются наиболее оптимальными.

Платформа	adult	breast_cancer	CIPublicHighway	EpicMeds	EpicVitals	Iowa1KK	LegacyPayors	Neighbors100K
Desbordante	8381 ± 154	34 ± 0	166342 ± 156446	24599 ± 873	2580 ± 33	22854 ± 566	385 ± 21	35 ± 3
Metanome	8177 ± 176	117 ± 2	210615 ± 198689	55680 ± 4346	3383 ± 103	27228 ± 318	875 ± 44	122 ± 8

Таблица 2: ИВ1: сравнение времени исполнения (мс) Desbordante и Metanome

Полученные в результате замеров доверительные интервалы нанесены на Рис. 1. Чтобы отобразить результаты, сильно варьирующиеся от датасета к датасету, на одном графике, была использована логарифмическая шкала для оси ординат. По графику можно сделать вывод, что Desbordante превосходит Metanome на всех таблицах, кроме Adult и CIPublicHighway, где доверительные интервалы пересекаются. На остальных отношениях наблюдаемое ускорение лежит в пределах от 1.19 до 3.43, достигая в среднем 2.12. Точные значения времени исполнения занесены в таблицу 2.

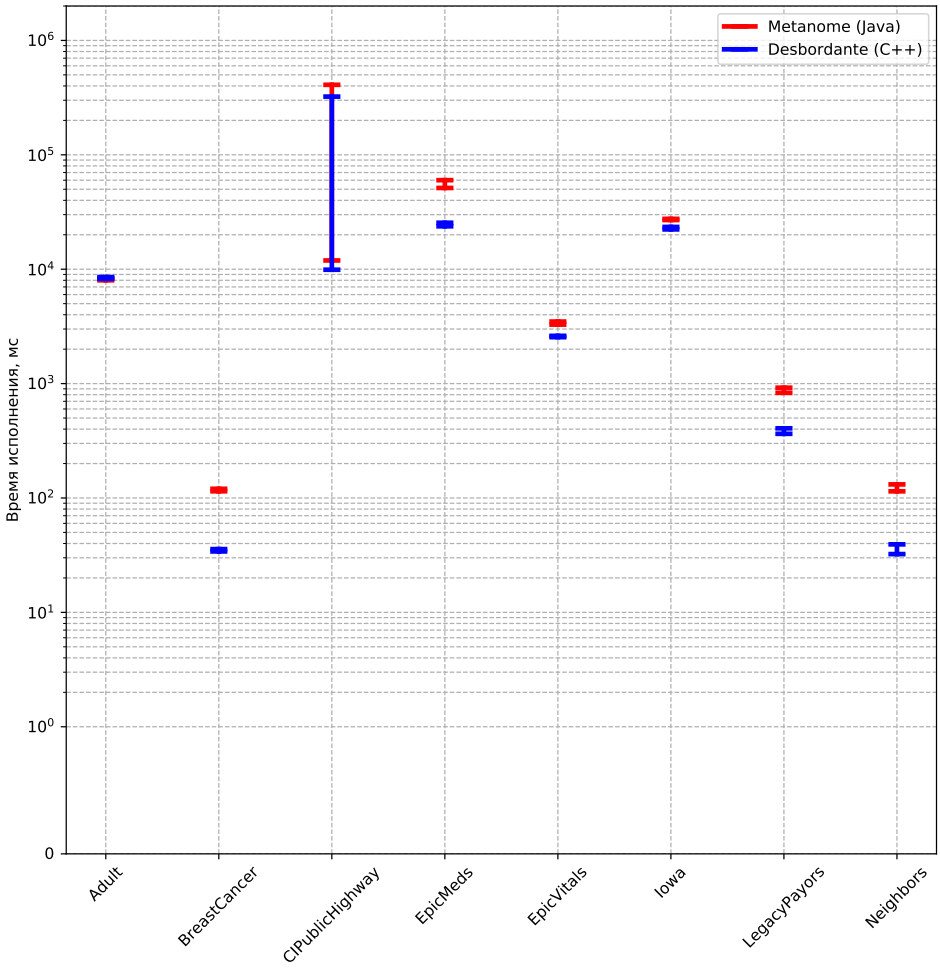

Рис. 1: ИВ1: сравнение производительности Desbordante и Metanome

Method	adult	breast_cancer	CIPublicHighway	EpicMeds	EpicVitals	Iowa1KK	LegacyPayors	Neighbors100K
Desbordante, AVG	174.03 MB	567.57 KB	858.37 MB	248.44 MB	133.63 MB	495.58 MB	88.80 MB	9.65 MB
Desbordante, MAX	311.93 MB	1.28 MB	1.40 GB	306.35 MB	181.89 MB	673.27 MB	177.00 MB	22.35 MB
Metanome, AVG	281.62 MB	16.18 MB	999.29 MB	638.68 MB	193.33 MB	725.60 MB	268.34 MB	55.68 MB
Metanome, MAX	575.57 MB	24.49 MB	1.89 GB	1.18 GB	426.83 MB	1.12 GB	493.61 MB	109.99 MB

Таблица 3: ИВ1: сравнение среднего объёма памяти, требуемого при работе Desbordante и Metanome

Что касается потребления памяти, был измерен средний и максимальный объём памяти, требуемый приложением во время работы — результаты представлены в таблице 3. Здесь также наблюдается превосходство реализации на C++: значение занимаемой в среднем памяти уменьшилось в среднем в 1.88 раза, варьируясь от 1.46 до 2.57,

причём максимальную требуемую память удалось сократить ещё сильнее. Заметим, что значения на таблице “breast_cancer” были приняты за выброс и проигнорированы.

5.2. Настройка Metanome

Программы, написанные на языке Java, исполняются в управляемой виртуальной машиной среде, поэтому на производительность можно значительно повлиять, задав определённые настройки виртуальной машины. Задачей данного исследовательского вопроса является поиск и определение настроек, позволивших бы увеличить производительность Metanome. В рамках сопутствующих экспериментов были опробованы следующие актуальные направления вариации параметров JVM:

1. Версии JDK: 11 и 15. Рассмотрены именно эти версии, поскольку на момент публикации данной работы они являются, соответственно последними долго- и краткосрочно поддерживаемыми версиями виртуальной машины.

2. Метод компиляции: по умолчанию в Java используется JIT (Just-In-Time) подход, но, используя виртуальную машину GraalVM, можно опробовать и AOT (Ahead-Of-Time) подход.

3. Тип JIT компилятора: клиентский (C1), серверный (C2) и многоуровневый (Tiered Compilation).

4. При использовании JIT компиляции есть возможность отрегулировать пороги компиляции, определяющие её агрессивность.

5. Сборщик мусора: G1, MarkSweep, Parallel, Serial.

6. Размер кучи, минимальный и максимальный.

Из результатов запуска Metanome с использованием различных версий JVM, см. таблицу 4, можно сделать следующие выводы:

Версия	adult	breast_cancer	CIPublicHighway	EpicMeds	EpicVitals	Iowa1KK	LegacyPayors	Neighbors100K
HotSpot JDK 11	7642 ± 64	115 ± 2	41178 ± 220	51930 ± 4029	3329 ± 144	27318 ± 228	809 ± 32	152 ± 6
HotSpot JDK 15	7886 ± 62	114 ± 4	57174 ± 527	67583 ± 3318	3413 ± 55	27545 ± 472	879 ± 36	151 ± 5
GraalVM JDK 11	8888 ± 36	125 ± 4	46828 ± 620	162917 ± 177	5836 ± 60	25968 ± 1156	1082 ± 21	153 ± 9

Таблица 4: ИВ2: время исполнения (мс) Metanome при использовании различных версий JDK (рассматривается JIT компиляция)

- В большинстве случаев, особенно на небольших таблицах, выбор версии виртуальной машины практически не влияет на производительность.
- Исключениями являются таблицы CIPublicHighway и EpicMeds, на которых измеренное время увеличивается на 50% и 150% соответственно. Анализ событий ядра с помощью perf показал, что количество инструкций процессора, генерируемых на упомянутых датасетах при использовании GraalVM, возрастает более чем в три раза. В то же время, количество таких событий, как промахи кэшей и буферов ассоциативной трансляциии, остаётся на том же уровне. Таким образом, можно объяснить наблюдаемое замедление неоптимизированной кодогенерацией со стороны GraalVM.

Подход	adult	breast_cancer	CIPublicHighway	EpicMeds	EpicVitals	Iowa1KK	LegacyPayors	Neighbors100K
AOT (GraalVM)	8525 ± 39	119 ± 6	54108 ± 1329	163724 ± 385	5761 ± 121	27417 ± 1864	1105 ± 51	147 ± 5
Client	8229 ± 116	282 ± 2	44681 ± 597	45006 ± 1686	3138 ± 136	26602 ± 469	842 ± 17	233 ± 9
Server	8205 ± 52	282 ± 3	44691 ± 552	46962 ± 3588	3160 ± 154	27033 ± 320	848 ± 23	252 ± 11
Tiered (default)	7703 ± 126	114 ± 2	41527 ± 243	51261 ± 3480	3312 ± 116	27098 ± 345	822 ± 20	149 ± 5
T3=2K, T4=15K	8029 ± 78	174 ± 2	42747 ± 231	47883 ± 5175	3331 ± 146	28335 ± 210	887 ± 27	159 ± 8
T3=0.5K, T4=5K	7845 ± 140	120 ± 2	41660 ± 252	50770 ± 4484	3401 ± 104	27802 ± 617	821 ± 32	153 ± 7
T3=6K, T4=30K	7953 ± 89	176 ± 2	42560 ± 339	47740 ± 5113	3389 ± 144	27859 ± 293	892 ± 26	162 ± 11

Таблица 5: ИВ2: время исполнения (мс) Metanome при изменении настроек компиляции

Следующим шагом было варьирование настроек компилятора, преобразующего байт-код в нативные инструкции. По результатам, представленным в таблице 5, невозможно выделить однозначно лучшую опцию: наибольшее ускорение составляет 10%, причём зачастую доверительные интервалы пересекаются. В рамках данной части эксперимента были использованы разные способы JIT компиляции, используемые в виртуальной машине: C1, C2 и многоуровневая компиляция. C1 и C2 представляют собой разные компиляторы [22], нацеленные на, со-

ответственно, быструю, но малоэффективную и медленную, но высокопроизводительную оптимизацию кода, в то время как многоуровневая компиляция позволяет комбинировать оба компилятора. Решение о том, какой из компиляторов использовать для конкретного метода, принимается на основе счётчика его вызовов, уменьшающегося со временем — такая эвристика приводит к оптимизации наиболее востребованных методов — и нескольких порогов, среди которых наиболее важными были выделены T3 и T4 ("-XX:Tier3InvocationThreshold" и "-XX:Tier4InvocationThreshold", соответственно). При достижении T3 используется компилятор C1, при достижении T4 — C2. Результаты, показанные в таблице 5, демонстрируют, что AOT компиляция позволяет достичь ускорения лишь на малых таблицах, впрочем уступая многоуровневой компиляции. Закономерным образом многоуровневая компиляция, используемая по умолчанию, превосходит каждый из компиляторов по отдельности, и значения порогов по умолчанию (T3=200, T4=5000) показывают наилучшие результаты.

Сборщик	adult	breast_cancer	CIPublicHighway	EpicMeds	EpicVitals	Iowa1KK	LegacyPayors	Neighbors100K
G1	7684 ± 114	113 ± 3	41635 ± 317	50468 ± 4065	3359 ± 126	26352 ± 485	818 ± 15	150 ± 9
MarkSweep	7717 ± 88	120 ± 4	42226 ± 504	66501 ± 2543	3513 ± 58	26447 ± 318	873 ± 8	143 ± 5
Parallel	7723 ± 89	114 ± 2	42448 ± 372	56927 ± 344	3291 ± 36	25829 ± 265	779 ± 17	139 ± 3
Serial	7911 ± 200	122 ± 23	43022 ± 922	55055 ± 2945	3390 ± 80	25814 ± 586	845 ± 15	148 ± 6

Таблица 6: ИВ2: время исполнения (мс) Metanome при использовании различных сборщиков мусора

Ещё одним фактором, способным повлиять на время работы программы, является сборщик мусора. Результаты, занесённые в таблицу 6, не могут говорить о превосходстве одного из сборщиков по причине частого пересечения доверительных интервалов, поэтому было принято решение использовать стандартный сборщик мусора Parallel.

Размер, МБ	adult	breast_cancer	CIPublicHighway	EpicMeds	EpicVitals	Iowa1KK	LegacyPayors	Neighbors100K
256	7972 ± 85	113 ± 3	ML	53301 ± 3989	3392 ± 87	ML	823 ± 27	150 ± 7
512	7502 ± 45	128 ± 4	ML	54054 ± 4278	3207 ± 94	27343 ± 272	818 ± 35	147 ± 7
1024	7483 ± 114	119 ± 3	41149 ± 436	53099 ± 4206	3265 ± 53	26026 ± 278	835 ± 40	139 ± 7
2048	7639 ± 115	120 ± 4	41022 ± 267	50358 ± 4390	3290 ± 124	24889 ± 191	830 ± 25	149 ± 4

Таблица 7: ИВ2: время исполнения (мс) Metanome при изменении начального размера кучи

Что касается начального размера кучи, доступного Java приложению, из данных таблицы 7 можно сделать вывод о том, что наличие дополнительной памяти в начале работы позволяет повысить производительность на больших таблицах.

В итоге, можно прийти к выводу, что изменение параметров или смена версии виртуальной машины не могут привести к убедительному приросту производительности на большинстве таблиц, и при исполнении Metanome стоит использовать JDK 11 с настройками по умолчанию.

5.3. Причины повышения производительности

Событие	P	breast_cancer	LegacyPayors	adult	EpicVitals	Iowa1KK	Neighbors100K	CIPublicHighway	EpicMeds
Ускорение	—	3.44	2.27	0.98	1.31	1.19	3.49	1.34	2.26
instructions	D	368.98M	7.59G	52.15G	15.05G	130.64G	1.18G	199.43G	48.67G
	M	4.80G	37.24G	106.32G	53.70G	275.52G	12.22G	489.63G	590.22G
L1-dcache-load-misses	D	1.75M	131.21M	716.73M	742.94M	3.56G	10.96M	2.65G	17.26G
	M	94.27M	468.57M	1.29G	975.52M	4.07G	155.90M	6.10G	17.24G
L1-icache-load-misses	D	8.72M	2.92M	73.72M	3.19M	49.65M	1.01M	34.61M	13.47M
	M	89.09M	113.13M	258.55M	131.81M	267.78M	89.57M	265.14M	186.29M
L2-misses	D	2.45M	165.54M	662.08M	786.95M	6.28G	14.41M	3.45G	9.37G
	M	139.69M	565.78M	1.42G	1.15G	7.43G	189.89M	6.38G	4.64G
LLC-load-misses	D	11.53K	13.12M	8.92M	26.77M	114.54M	761.36K	492.79M	77.43M
	M	1.97M	16.47M	32.46M	34.33M	217.43M	4.26M	879.14M	94.71M
LLC-store-misses	D	32.19K	28.14M	6.73M	10.62M	76.16M	746.36K	105.85M	19.45M
	M	2.66M	31.22M	65.53M	35.72M	207.50M	6.92M	263.19M	70.00M
cache-misses	D	203.48K	117.83M	105.77M	154.55M	877.17M	6.96M	1.95G	531.16M
	M	31.56M	255.98M	527.59M	332.53M	2.05G	64.09M	4.82G	918.86M
branch-instructions	D	79.17M	1.47G	10.77G	3.07G	26.09G	253.11M	40.97G	10.08G
	M	838.41M	7.13G	18.62G	11.99G	47.91G	2.24G	88.28G	170.86G
branch-misses	D	947.56K	8.64M	186.83M	29.47M	358.19M	1.60M	272.52M	143.19M
	M	37.00M	153.30M	362.65M	157.66M	832.07M	59.18M	493.29M	393.31M
context-switches	D	16.00	106.00	760.00	207.00	5.34K	3.00	243.00	3.17K
	M	2.50K	3.48K	4.23K	3.50K	11.36K	2.68K	13.90K	6.17K
cpu-migrations	D	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
	M	119.00	343.00	286.00	273.00	1.24K	217.00	1.46K	477.00
dTLB-load-misses	D	5.31K	5.75M	3.13M	16.45M	76.73M	328.80K	504.91M	36.52M
	M	827.63K	15.00M	9.44M	18.90M	91.24M	1.61M	839.22M	39.69M
iTLB-load-misses	D	3.05K	137.62K	673.54K	187.71K	1.46M	23.64K	1.44M	447.86K
	M	608.40K	1.76M	3.09M	2.09M	3.96M	790.17K	4.59M	2.62M
page-faults	D	508.00	65.19K	90.31K	67.32K	269.40K	7.60K	365.63K	123.22K
	M	17.89K	234.75K	302.52K	237.15K	379.81K	46.43K	550.19K	362.13K

Таблица 8: ИВ3: измерение количества различных событий в процессе работы Desbordante(D) и Metanome(M)

Целью данного вопроса является исследование причин, по которым Desbordante оказывается быстрее Metanome. Во-первых, как было показано в ИВ1, Desbordante требует меньше памяти, и, возможно, тратит меньше времени на операции аллокации. Во-вторых, было изучено поведение обеих платформ с точки зрения событий аппаратного и

программного обеспечения, таких как промахи кэшей. Результаты соответствующих замеров, проведённых с помощью perf, представлены в таблице 8. Первая строка тела таблицы показывает ускорение, достигнутое на каждом датасете, взятое из таблицы 2, после чего перечислены измеряемые статистики и их значения для каждой из платформ: Desbordante (D) и Metanome (M).

По достигнутому ускорению Desbordante относительно Metanome можно разбить датасеты на две группы: высокое и низкое ускорение. К первой группе относятся EpicMeds, LegacyPayors, Neighbors и breast_cancer, ко второй — Adult, Iowa1KK и EpicVitals. Заметим, что среди событий, представленных в таблицы, нет единственного, чьи значения напрямую бы коррелировали со временем исполнения. Соответственно, нельзя сделать вывод о том, что код, написанный на C++, оказался быстрее в результате оптимизации какого-то одного фактора, например, промахов кэшей. Следовательно, необходимо выделить несколько статистик, влияющих на производительность:

1. В первую очередь было выделено количество инструкций. Данная статистика уже показала свою значимость при исследовании Metanome в ИВ2, хотя в данном случае количество инструкций не определяет однозначно время исполнения. К примеру, отношения Adult и Iowa1KK демонстрируют примерно одинаковые пропорции относительно этой метрики, однако значения ускорения, достигнутого на каждом из них, заметно отличаются. Тем не менее, можно заметить, что в случаях, когда количество инструкций различается не менее чем в пять раз, наблюдается ощутимый прирост в производительности.

2. Следующим фактором выступают промахи как кэшей данных, так и кэшей инструкций разного уровня (L1/L2/L3 data/instruction cache misses). Среди рассматриваемых таблиц, именно на Adult и Iowa1KK достигается наихудшее значение соотношения количества промахов кэшей данных первого уровня (L1). Однако, Iowa1KK показывает более низкие пропорции, чем Adult, хотя достигнутое

на Iowa1KK ускорение превышает ускорение на Adult. Таким образом, вероятно наличие третьего фактора, определяющего производительность.

3. Наконец, стоит отметить количество инструкций ветвления и соответствующих промахов (branch instructions, branch misses). Во-первых, доля инструкций ветвления составляет приблизительно 20% от всех инструкций для каждого датасета и платформа, из чего следует, что для данного фактора справедливы те же соображения, что были приведены для количества всех инструкций.

   Во-вторых, значимость промахов ветвления подтверждается тем фактом, что это единственная статистика, по которой Iowa1KK превосходит Adult. Соответственно, именно уменьшение количества промахов ветвления позволило достичь ускорения на Iowa1KK.

В заключение рассмотрим остальные приведённые типы событий:

1. Миграции процессора и переключения контекста (CPU migrations, context switches). Заметим, что у Desbordante данные показатели значительно ниже, причём количество миграций процессора в основном остаётся на нулевом значении. Такое поведение объясняется многопоточной природой исполняемых Java приложений, и данные дорогостоящие события можно было бы минимизировать, привязав виртуальную машину к единственному ядру. С другой стороны, это скорее всего привело бы к замедлению работы программы, поскольку процессорное время стало бы уходить на сборку мусора и JIT компиляцию, исполняющиеся обычно в отдельных потоках.

2. Промахи буферов ассоциативной трансляции (tlb misses). Реализация на C++ показывает более низкие значения данной статистики, но не наблюдается корреляции со временем исполнения, см. соответствующие значения на Adult и Iowa1KK.

3. Отказы страниц (page faults). Хотя количество отказов страниц также ниже у Desbordante, на данном этапе этот тип событий

несёт меньший вклад, чем промахи кэшей и буферов ассоциативной трансляции, поскольку код Desbordante далёк от оптимизированного.

Уровень	adult	breast_cancer	CIPublicHighway	EpicMeds	EpicVitals	Iowa1KK	LegacyPayors	Neighbors100K
O0	73702 ± 177	258 ± 2	420799 ± 672	73522 ± 99	14214 ± 21	220619 ± 85	2483 ± 8	589 ± 0
O1	8805 ± 7	37 ± 0	47053 ± 96	25500 ± 70	2758 ± 13	25439 ± 65	397 ± 0	54 ± 0
O2	8319 ± 9	37 ± 0	42809 ± 61	24952 ± 44	2565 ± 8	23755 ± 42	364 ± 2	48 ± 0
O3	8209 ± 28	36 ± 0	42897 ± 71	24799 ± 101	2567 ± 8	23169 ± 43	363 ± 1	47 ± 0

Таблица 9: ИВ3: время исполнения (мс) при различных уровнях оптимизации GCC

Наконец, возможно, компилятор C++ может использовать более дорогостоящие и эффективные оптимизации, чем JIT компилятор Java, поскольку от него не требуется быстрая, укладывающаяся во время исполнения программы генерация оптимизированного кода. Чтобы проверить данную гипотезу, были проведены замеры времени исполнения Desbordante с четырьмя доступными уровнями оптимизации, использующими всё более трудоёмкие алгоритмы. Таблица 9, куда были занесены результаты, показывает, что уже уровень “-O1” позволяет добиться конкурентоспособной производительности, тогда как дальнейшее повышение уровня не приводит к значительному ускорению.

6. Уменьшение трудоёмкости задачи

maxLHS	adult	CIPublicHighway	EpicVitals	Iowa1KK	LegacyPayors	SG_Bioentry
1	119	762	1458	4265	417	61
2	740	5789	2554	19073	550	171
3	2587	ML	2270	29707	372	216
4	5694	ML	2672	28802	372	228
5	9876	ML	2631	25629	369	100
10	8368	ML	2671	23498	369	100
100	8331	ML	2676	23678	370	99

Таблица 10: ИВ4: время исполнения (мс) Desbordante при изменении параметра maxLHS

maxLHS	adult	CIPublicHighway	EpicVitals	Iowa1KK	LegacyPayors	SG_Bioentry
1	194	737	1652	4841	617	203
2	923	6276	3363	21875	960	344
3	2300	ML	3051	36150	768	507
4	4990	ML	2979	ML	936	419
5	9243	ML	3078	31186	848	253
10	8541	ML	3023	28998	785	301
100	8257	ML	2939	28713	795	252

Таблица 11: ИВ4: время исполнения (мс) Metanome при изменении параметра maxLHS

Поскольку поиск всех ФЗ — трудоёмкая операция, и зачастую пользователя интересуют зависимости с небольшой левой частью, состоящей, например, из двух-трёх атрибутов, многие алгоритмы имеют возможность ограничить максимальный размер левой части. Таким образом, уменьшается пространство поиска, которое обходит алгоритм, и использование алгоритма становится доступным даже на маломощных системах.

В данном разделе было изучено поведение Pyro в обеих платформах. Было установлено ограничение по памяти в 1ГБ, и проведена серия экспериментов по измерению времени исполнения при разном максимальном размере левой части. Результаты занесены в таблицы 10 и 11. Из приведённых данных можно сделать следующие выводы:

- CIPublicHighway демонстрирует, что уменьшение максимально воз-

можной левой части позволяет уложиться в ограничение по памяти.

- С точки зрения скорости исполнения программы, при ограничении количества атрибутов в левой части уже двумя только три таблицы удаётся обработать быстрее, чем без подобного ограничения: Adult, Iowa, and CIPublicHighway. Более того, зачастую ограничение на левую часть приводит к замедлению работы программы.

- Тот факт, что Desbordante потребляет меньше памяти, позволяет успешно завершить работу в некоторых случаях (IOWA1KK, maxLHS=4).

7. Выводы

На основе проделанной работы можно сделать следующие выводы:

1. При проведении обзора существующих решений удалось установить, что единственный инструмент для реализации и исследования алгоритмов поиска ФЗ — Metanome. Остальные решения либо предлагают упрощённую функциональность, либо решают более узкую задачу, либо не решают задачу поиска набора всех ФЗ.

2. Экспериментальное сравнение предлагаемого решения с Metanome показало, что Desbordante требует в среднем практически в два раза меньше памяти и завершает работу в два раза быстрее. Была исследована возможность настройки среды исполнения Metanome с целью сокращения данного разрыва, но настройки по умолчанию оказались наиболее эффективными. Наконец, более высокую производительность Desbordante можно объяснить оптимизацией аппаратных событий, прежде всего: количества инструкций, промахов кэшей и промахов инструкций ветвления.

3. С помощью Desbordante была исследована возможность уменьшения трудоёмкости алгоритмов засчёт ограничения максимального размера левой части рассматриваемых ФЗ. Эксперименты показали, что такой метод способен значительно сократить время исполнения на большинстве таблиц при maxLHS=1–2, однако при бóльших значениях параметра производительность оказывается хуже, чем при неограниченной левой части.

8. Заключение

В рамках данной работы были достигнуты следующие результаты:

1. Выполнен обзор алгоритмов и инструментов для поиска функциональных зависимостей.

2. Спроектирована и реализована платформа Desbordante, позволяющая использовать алгоритмы TANE, Pyro и предоставляющая возможности для добавления новых алгоритмов.

3. Проведено сравнение производительности Desbordante и существующего решения Metanome.

   - сформулированы метрики для сравнения и проведены эксперименты, показавшие превосходство Desbordante;
   - проведены эксперименты, показавшие, что невозможно достичь аналогичного уровня производительности с помощью настройки Metanome;
   - сформулированы и экспериментально проверены гипотезы, объясняющие причины повышенного уровня производительности.

4. Исследована возможность уменьшения трудоёмкости алгоритмов поиска ФЗ через ограничение максимального размера левой части.

Кроме того, по теме данной работы была написана статья "Desbordante: a Framework for Exploring Limits of Dependency Discovery Algorithms" [9], представленная на конференции FRUCT'29 и принятая к публикации в сборнике трудов.

9. Благодарности

Автор выражает отдельную признательность коллегам, помогавшим ему при создании данной работы:

- Георгию Чернышеву — за первоначальную идею и видение проекта, организацию работы и колоссальную помощь и курирование при выполнении всех аспектов работы, от постановки вопросов исследований до подготовки текстов и презентаций.
- Никите Боброву — за регулярную и неоценимую помощь в изучении предметной области;
- Кириллу Смирнову — за множественные советы и рекомендации по техническим вопросам;
- Илье Вологину — за реализацию базовых классов и структур на начальном этапе проекта.

Список литературы

[1] Abedjan Ziawasch, Golab Lukasz, Naumann Felix. Profiling relational data: a survey // The VLDB Journal. — 2015. — Vol. 24. — P. 557–581.

[2] Armstrong W. W. Dependency Structures of Data Base Relationships // IFIP Congress. — 1974.

[3] BigDansing: A System for Big Data Cleansing / Zuhair Khayyat, Ihab Ilyas, Alekh Jindal et al. — 2015. — 06.

[4] CPU frequency scaling. — 2021. — Access mode: `https://wiki.archlinux.org/index.php/CPU_frequency_scaling`.

[5] Caruccio Loredana, Cirillo Stefano. Incremental Discovery of Imprecise Functional Dependencies // J. Data and Information Quality. — 2020. — Oct. — Vol. 12, no. 4. — Access mode: `https://doi.org/10.1145/3397462`.

[6] Chu Xu, Ilyas Ihab F., Papotti Paolo. Discovering Denial Constraints // Proc. VLDB Endow. — 2013. — Aug. — Vol. 6, no. 13. — P. 1498–1509. — Access mode: `https://doi.org/10.14778/2536258.2536262`.

[7] Data Auditor / Lukasz Golab, H. Karloff, Flip R. Korn, D. Srivastava // Proceedings of the VLDB Endowment. — 2010. — Vol. 3. — P. 1641 – 1644.

[8] Data profiling with metanome / Thorsten Papenbrock, Tanja Bergmann, Moritz Finke et al. // Proceedings of the VLDB Endowment. — 2015. — 08. — Vol. 8. — P. 1860–1863.

[9] Desbordante: a Framework for Exploring Limits of Dependency Discovery Algorithms (paper accepted) / Maxim Strutovskiy, Nikita Bobrov, Kirill Smirnov, George Chernishev // FRUCT'29: Proceedings of the 29th Conference of Open Innovations Association FRUCT. — fruct.org, 2021.

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

[11] FDTool: a Python application to mine for functional dependencies and candidate keys in tabular data / Matt Buranosky, Elmar Stellnberger, Emily Pfaff et al. // F1000Research. — 2019. — 06. — Vol. 7. — P. 1667.

[12] Flach Peter A., Savnik Iztok. Database Dependency Discovery: A Machine Learning Approach // AI Commun. — 1999. — Aug. — Vol. 12, no. 3. — P. 139–160.

[13] Functional dependency discovery: An experimental evaluation of seven algorithms / Thorsten Papenbrock, J. Ehrlich, J. Marten et al. — 2015. — 01. — P. 1082–1093.

[14] Georges Andy, Buytaert Dries, Eeckhout Lieven. Statistically Rigorous Java Performance Evaluation. — Vol. 42. — 2007. — 10.

[15] Incremental Discovery of Functional Dependencies with a Bit-vector Algorithm / Loredana Caruccio, Stefano Cirillo, Vincenzo Deufemia, Giuseppe Polese // Proceedings of the 27th Italian Symposium on Advanced Database Systems, Castiglione della Pescaia (Grosseto), Italy, June 16-19, 2019 / Ed. by Massimo Mecella, Giuseppe Amato, Claudio Gennaro. — Vol. 2400 of CEUR Workshop Proceedings. — CEUR-WS.org, 2019. — Access mode: http://ceur-ws.org/Vol-2400/paper-21.pdf.

[16] Informatica Data Quality Data Discovery Guide. — 2020. — Access mode: https://docs.informatica.com/data-quality-and-governance/data-quality/10-4-0/data-discovery-guide/

data-discovery-with-informatica-developer/
data-object-profiles/functional-dependency-discovery.html.

[17] Kruse Sebastian, Naumann Felix. Efficient Discovery of Approximate Dependencies // Proceedings of the VLDB Endowment. — 2018. — 03. — Vol. 11.

[18] Lopes Stéphane, Petit Jean-Marc, Lakhal Lotfi. Efficient Discovery of Functional Dependencies and Armstrong Relations. — Vol. 1777. — 2000. — 03. — P. 350–364.

[19] Maxino T. Revisiting Fletcher and Adler Checksums. — 2006.

[20] "No Bugs" Hare. Testing Memory Allocators: ptmalloc2 vs tcmalloc vs hoard vs jemalloc While Trying to Simulate Real-World Loads. — 2018. — http://ithare.com/testing-memory-allocators-ptmalloc2-tcmalloc-hoard-jemalloc-while-trying-to-simulate-real-world-loads/ .

[21] Novelli Noel, Cicchetti R. FUN: An Efficient Algorithm for Mining Functional and Embedded Dependencies // ICDT. — 2001.

[22] Oaks Scott. Java Performance: The Definitive Guide: Getting the Most Out of Your Code. — 1st edition edition. — Sebastopol, CA : O'Reilly Media, 2014. — May. — ISBN: 978-1-4493-5845-7.

[23] Oracle Warehouse Builder Data Modeling, ETL, and Data Quality Guide. — 2010. — Access mode: https://docs.oracle.com/cd/E18283_01/owb.112/e10935/data_profiling.htm#BABHBGIB.

[24] Papenbrock Thorsten, Naumann Felix. A Hybrid Approach to Functional Dependency Discovery. — 2016. — 06. — P. 821–833.

[25] Papenbrock Thorsten, Naumann Felix. Data-driven Schema Normalization // EDBT. — 2017.

[26] Paulley Glenn. Exploiting Functional Dependence in Query Optimization. — 2000. — 01.

[27] RuleMiner: Data quality rules discovery / Xu Chu, Ihab Ilyas, Paolo Papotti, Yin Ye. — 2014. — 03. — P. 1222–1225.

[28] SAP Information Steward User Guide. — 2017. — Access mode: https://help.sap.com/doc/PRODUCTION/18ec99b2d06449c5b8b79c784d5a3af9/4.2.8/en-US/is_42_user_en.pdf.

[29] SQL Server Functional Dependency Profile Request Options (Data Profiling Task). — 2017. — https://docs.microsoft.com/en-us/sql/integration-services/control-flow/functional-dependency-profile-request-options-data-profiling-task?view=sql-server-ver15 .

[30] TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. / Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, Hannu Toivonen // Comput. J. — 1999. — 02. — Vol. 42. — P. 100–111.

[31] Talend Open Studio for Data Quality User Guide. — 2021. — Access mode: https://help.talend.com/r/Se88elCF0HkomGAlfWThWA/UYCDlDPlZD6wst0iRQvZYQ.

[32] Wyss Catharine, Giannella Chris, Robertson Edward. FastFDs: A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies from Relation Instances Extended Abstract. — 2001. — 01. — P. 101–110.

[33] Yao Hong, Hamilton Howard J., Butz Cory J. FD_Mine: Discovering Functional Dependencies in a Database Using Equivalences // Proceedings of the 2002 IEEE International Conference on Data Mining. — ICDM '02. — USA : IEEE Computer Society, 2002. — P. 729.