Санкт-Петербургский государственный университет

Полынцов Михаил Александрович

Выпускная квалификационная работа

Реализация алгоритма поиска UCC 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 программист ООО «МАЙ.ГЕЙМЗ ДЕВЕЛОПМЕНТ» Е. С. Ключиков

Санкт-Петербург
2023

Saint Petersburg State University

Mikhail Polyntsov

Bachelor's Thesis

Implementing UCC 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 developer at «MY.GAMES DEVELOPMENT» E. S. Klyuchikov

Saint Petersburg
2023

Оглавление

Введение

Профилирование данных — это процесс анализа данных, направленный на извлечение метаданных [1]. Его можно разделить на два типа: ненаукоёмкое и наукоёмкое [8]. Профилирование первого типа включает в себя поиск таких метаданных как, например, время создания данных и их авторство. Второй тип направлен на выявление такой гораздо более содержательной информации, как закономерности в данных. Анализируемые данные можно классифицировать на основе логической модели данных, к которой они относятся: графовые, табличные, транзакционные, пространственные и т. д. [3]. Модель данных определяет подходы, которые используются в алгоритмах поиска закономерностей. В данной работе будем рассматривать только табличные данные.

Существует множество способов формализации понятия закономерностей в табличных данных [4,5]. Одним из примеров такой формализации являются хорошо известные функциональные зависимости [11]. Такие формальные описания будем называть примитивами или просто зависимостями. Информация о закономерностях в данных используется для решения таких прикладных задач, как очистка данных (data cleaning), исследование данных (data exploration), реверс-инжиниринг баз данных (database reverse engineering) и оптимизация запросов в системах управления базами данных [1, 15]. Поэтому были разработаны и продолжают разрабатываться алгоритмы автоматического поиска зависимостей.

Другим примером закономерности в табличных данных является уникальная комбинация колонок (Unique Column Combination, UCC) — набор атрибутов таблицы, проекция по которым не имеет одинаковых кортежей. UCC называется минимальной, если никакое подмножество ее атрибутов не является UCC. Важным свойством UCC является то, что уникальность комбинации колонок сохраняется при добавлении в неё любых других колонок. Это означает, что при автоматическим поиске UCC, достаточно находить только минимальные. Информация о том, какие комбинации колонок являются уникальными, необходима для решения таких задач, как сопоставление схем (schema matching) [19] или

интеграция данных (data integration) [22]. Однако такая информация не всегда доступна, поэтому возможность поиска минимальных UCC над заданной таблицей является важной функциональностью любого профилировщика данных.

Платформа Desbordante [7] является наукоёмким профилировщиком данных с открытым исходным кодом[1]. Она содержит алгоритмы поиска различных примитивов и предоставляет интерфейсы к ним, а также реализует сценарии решения прикладных задач анализа данных. Начальной мотивацией для создания Desbordante послужили недостатки существующей платформы Metanome [6], описанные в работе [8]. К ним относятся, например, не самая оптимальная производительность из-за использования Java в качестве основного языка программирования и отсутствие дружественных к пользователю интерфейсов. Более того, в Metanome алгоритмы поиска примитивов изолированы от реальных задач, то есть платформа не предоставляет ничего кроме самих алгоритмов, что уменьшает ее эффективность в применении к индустриальным задачам. Поэтому одной из целей Desbordante является предоставление возможностей по решению прикладных задач с помощью этих алгоритмов. В данный момент Desbordante активно развивается и нуждается в расширении набора поддерживаемых примитивов, в том числе в поддержке UCC. Это послужило мотивацией для задачи разработки высокопроизводительного алгоритма поиска уникальных комбинаций колонок, поставленной перед автором данной работы.

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

1 Постановка задачи

Целью данной ВКР является реализация алгоритма поиска уникальных комбинаций колонок в Desbordante и его модификация с целью повышения производительности. Для достижения этой цели были поставлены следующие задачи.

1. Выполнить обзор предметной области.
2. Реализовать существующий алгоритма поиска UCC в платформе Desbordante.
3. Исследовать возможности модификации реализованного алгоритма, модифицировать его наиболее эффективным образом и получить ускорение.
4. Провести экспериментальное исследование реализаций.

2 Обзор

2.1 Поиск UCC

Поиск уникальных комбинаций колонок — алгоритмически сложная задача и принадлежит к классу NP-трудных [9], при этом размер результата растёт экспоненциально [20]. В исследовательской литературе представлено некоторое количество алгоритмов, решающих её. Поиск UCC реализуется с помощью одного из двух подходов: *основанном на колонках* (column-based) или *основанном на строках* (row-based).

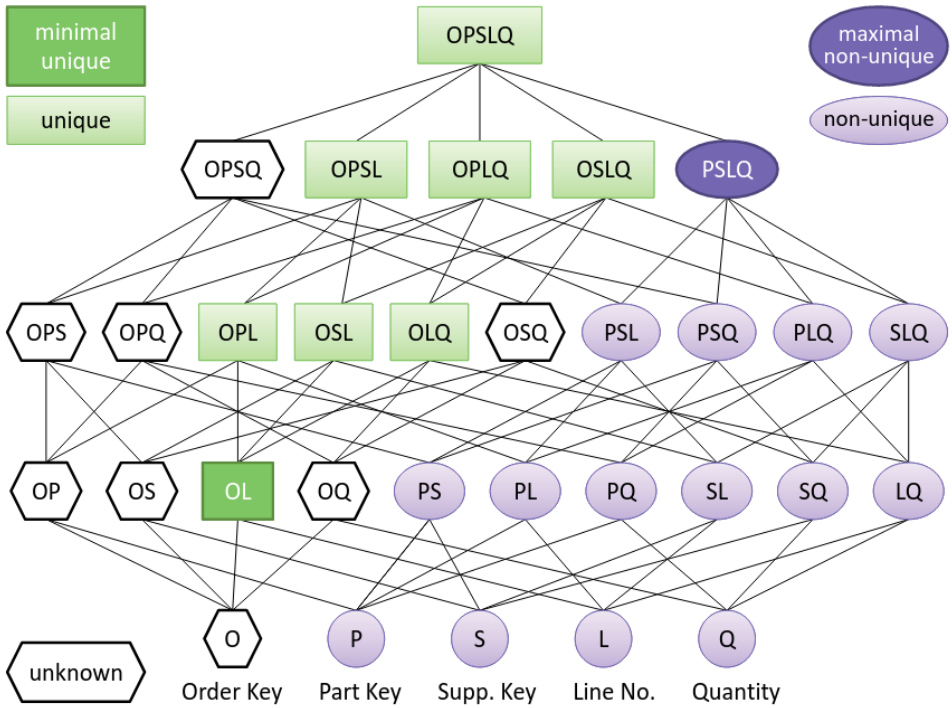

Рис. 1: Решётка атрибутов, источник [20].

Основанный на колонках подход заключается в генерации и обходе решётки, описывающей множество всех подмножеств атрибутов таблицы. Её пример можно видеть на Рис. 1. Во время обхода решетки наборы атрибутов в узлах идентифицируются как не-UCC (отмечены розовым цветом), неминимальные UCC (отмечены салатовым) и минимальные UCC (отмечены зеленым, соответственно результат работы алгоритма). То есть наивный алгоритм поиска UCC можно реализовать как обход решетки-графа до тех пор, пока все узлы не будут посещены и идентифи-

цированы. Однако такой полный перебор очень неэффективен, поэтому в алгоритмах большое внимание уделяется оптимизациям, позволяющим не рассматривать заранее какие-то пути в графе.

Основанный на строках подход перебирает пары кортежей и составляет список наборов колонок, которые не являются UCC. Такой список ещё называют негативным покрытием (negative cover). Если какие-то два кортежа совпадают в наборе колонок, то этот набор точно не может являться UCC для всей таблицы и соответственно попадает в негативное покрытие. Следующим шагом алгоритм преобразует список не-UCC в список минимальных UCC или положительное покрытие (positive cover). Такое преобразование может быть сделано двумя способами: построением дополнения к не-UCC [16] или постепенным уточнением положительного покрытия [10]. Полный перебор всех пар кортежей тоже довольно неэффективен и потому нуждается в оптимизации.

Далее рассмотрим самые значимые научные работы, посвященные алгоритмам поиска уникальных комбинаций колонок.

В исследовании [13] авторы описывают и анализируют первые наивные алгоритмы поиска UCC, реализующие обход решетки. В работе присутствует только алгоритмический анализ без результатов каких-либо экспериментов и соответственно только теоретические выводы.

В исследовании [12] авторы представляют первый применимый на практике алгоритм Gordian для поиска UCC. Алгоритм реализует основанный на строках подход. Преобразование негативного покрытия в положительное производится посредством построения дополнения. Результаты экспериментов показывают, что Gordian ищет все составные ключи за то же время, за которое полный перебор ищет ключи состоящие из одного атрибута. При этом алгоритмы потребляют примерно одинаковое количество памяти.

Авторы работы [2] представляют новый алгоритм поиска уникальных комбинаций колонок HCA и его гибридную версию HCA-Gordian. Алгоритм реализует основанный на колонках подход. В экспериментальном исследовании авторы показывают, что HCA и HCA-Gordian работают на порядок быстрее рассмотренных выше Gordian и наивных алгоритмов.

В работе [20] авторы представляют новый алгоритм DUCC. Алгоритм реализует основанный на колонках подход. Эффективность достигается за счет агрессивного отсечения кандидатов (не всегда корректного), особого гибридного подхода к обходу решетки и дополнительного шага проверки кандидатов. По результатам экспериментов авторы приходят к выводу, что DUCC на порядок эффективнее своих предшественников Gordian и HCA.

В статье [18] представлен алгоритм HyUCC. Авторы данной работы предлагают новый гибридный алгоритм, который реализует и основанный на колонках, и основанный на строках подходы и переключается между ними, когда текущий подход оказывается неэффективным. Авторы приводят результаты экспериментов, которые показывают, что HyUCC эффективнее предыдущего самого эффективного алгоритма DUCC на порядки.

В работе [14] авторы представляют алгоритм HPIValid. Алгоритм использует новый подход, который моделирует поиск UCC как задачу об обходе минимальных вершинных покрытий гиперграфа (hitting set enumeration problem in hypergraphs). HPIValid имеет схожую с HyUCC структуру: переключается между сэмплированием и валидацией. Однако и алгоритм сэмплирования, и алгоритм валидации сильно отличаются от использованных в HyUCC. Авторы проводят эксперименты и по их результатам утверждают, что HPIValid более эффективен, чем HyUCC, как по памяти, так и по времени выполнения.

Таблица 1: Алгоритмы поиска UCC.

Алгоритм	Год публикации	Основанный на колонках	Основанный на строках	Эффективнее предыдущих
Apriori	1999	+	−	
Gordian	2006	−	+	+
HCA	2011	+	−	+
DUCC	2013	+	−	+
HyUCC	2017	+	+	+
HPIValid	2020	−	−	?

Все рассмотренные алгоритмы и их характеристики отображены в

таблице 1. Авторы HPIValid в работе [14] утверждают, что их алгоритм эффективнее HyUCC. Однако в своих экспериментах они сравнивали HPIValid, реализованный на C++, с HyUCC, реализованном на Java. Различие в технологиях может давать разницу на порядки в производительности. Также HyUCC запускался в однопоточном режиме, хотя эксперименты в оригинальный статье, описывающей HyUCC, показывают, что параллельная версия может давать более чем тридцатикратное ускорение в зависимости от входных данных. Таким образом, было принято решение реализовать HyUCC в качестве первого алгоритма поиска UCC в Desbordante, поскольку его эффективность точно известна и в Desbordante уже реализован алгоритм поиска функциональных зависимостей того же класса HyFD.

2.2 Алгоритм HyUCC

Основные модули алгоритма HyUCC и потоки данных можно увидеть на Рис. 2.

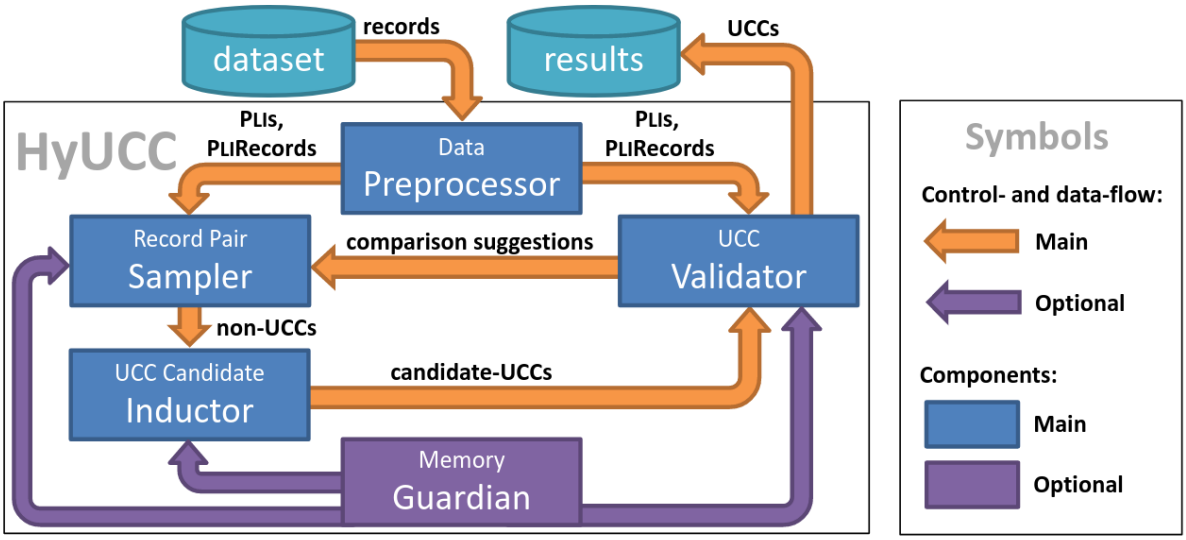

Рис. 2: Модули алгоритма HyUCC, источник [18].

Preprocessor. Получая входные данные, HyUCC сначала передаёт их в модуль Preprocessor, вычисляющий несколько промежуточных представлений таблицы, которыми пользуются другие модули. Чтобы найти уникальные комбинации колонок, значения элементов строк знать

не обязательно, достаточно знать позиции одинаковых значений для каждого атрибута. Поэтому вместо хранения входной таблицы целиком, Preprocessor преобразует её колонки в партиции (Position List Indexes, PLI или partitions [21]). Далее модуль строит ещё две вспомогательные структуры, позволяющие по элементу таблицы быстро найти соответствующие кластеры в партициях.

Sampler. Данный компонент реализует первую часть поиска UCC основанного на строках подхода. Алгоритм для каждой пары строк ищет колонки, в которых значения элементов строк совпадают. Такие колонки не являются UCC для всей таблицы и попадают в негативное покрытие. Нахождение всех не-UCC требует попарного сравнения всех записей, то есть обладает квадратичной сложностью и плохо масштабируется с увеличением длины таблицы. Также разные пары кортежей могут совпадать в одних и тех же колонках, порождая уже известные non-UCC. Чтобы избежать квадратичной сложности, алгоритм сравнивает только часть кортежей, а наиболее выгодные пары для сравнения (порождающие новые non-UCC) выбираются с помощью метода скользящего окна на кластере (cluster windowing).

Inductor. Данный компонент реализует вторую часть поиска UCC основанного на строках подхода. Inductor принимает в качестве входных данных негативное покрытие от Sampler и генерирует из него наборы колонок, которые являются кандидатами на UCC (UCC-кандидаты). UCC-кандидат — это минимальный уникальный набор колонок над той выборкой из таблицы, что рассматривал Sampler на текущем шаге. Является ли кандидат настоящим UCC для всей таблицы проверяется в компоненте Validator. Алгоритм генерации кандидатов из негативного покрытия такой же, как для функциональных зависимостей в алгоритме FDep [10]. Inductor хранит префиксное дерево наборов колонок и уточняет его используя не-UCC. Для каждого не-UCC алгоритм удаляет из дерева все обобщения (generalizations), потому что они не могут быть UCC и добавляет в дерево все специализации (specializations). При следующем вызове Inductor продолжает уточнять уже имеющиеся UCC в дереве.

Validator. Данный компонент реализует поиск UCC, основанный на колонках. Validator получает на вход кандидатов на UCC в виде префиксного дерева от Inductor и проверяет, являются ли они UCC для всей таблицы, представленной как набор партиций. Данный компонент обходит префиксное дерево поуровнево снизу вверх, используя поиск в ширину. Каждый листовой узел дерева соответствует кандидату на UCC, который вычисляется как путь до листа. Если валидация вернула положительный результат, то кандидат является UCC и остается в дереве. В противном случае кандидат удаляется из дерева и вместо него добавляются новые наборы колонок, содержащие удаленного кандидата как подмножество и имеющие на одну колонку больше. Валидация кандидатов выполняется не дорогим пересечением партиций, а проверкой напрямую, что в проекции по колонкам кандидата существует две строки, лежащие в одних и тех же кластерах, то есть равные. В конце данной фазы префиксное дерево содержит UCC над всей таблицей, но возможно не все, если валидация была прервана раньше из-за неэффективности.

Guardian. Данный модуль следит за потреблением памяти и не является обязательным для правильной работы алгоритма. Если в какой-то момент используемая память подходит к допустимому максимуму, Guardian уменьшает максимальный размер UCC, которые будут найдены, а все большие уже найденные удаляет из дерева в Inductor.

HyUCC начинается с основанной на строках фазы, то есть с модуля Sampler. Когда данный подход становится неэффективным, алгоритм переключается на основанную на колонках фазу, то есть выполняет переход к модулю Validator, во время которого уточняет уже найденные UCC в Inductor. Эффективность первой фазы определяется как число найденных не-UCC на одно сравнение. Когда эта эффективность становится ниже определенного порогового значения, происходит переход ко второй фазе. Validator в свою очередь считает эффективность валидации, которая вычисляется как количество кандидатов на UCC проверенных, как неверные, деленное на количество кандидатов, проверенных как верные. Как только это значение становится меньше того же порогового значения, что и в Sampler, происходит переключение на

основанную на строках фазу. При этом Validator передает Sampler пары строк, которые необходимо сравнить в первую очередь, так как они с большей вероятностью породят еще неизвестные не-UCC. Выполнение алгоритма всегда заканчивается на фазе валидации.

2.3 Платформа Desbordante

Desbordante — высокопроизводительный профилировщик данных, нацеленный на наукоемкое профилирование. Основными идеями, с которыми платформа разрабатывается, являются: высокая производительность, поэтому все алгоритмы реализованы на языке C++; доступность для пользователей, поэтому Desbordante обладает несколькими пользовательскими интерфейсами, в том числе веб-интерфейсом; нацеленность на решение прикладных задач с помощью поиска примитивов.

Принцип использования платформы и в частности выполнение алгоритмов поиска примитивов предельно прост. Для примера возьмем консольный интерфейс. Пользователю необходимо запустить исполняемый файл Desbordante и указать в параметрах командной строки набор данных, используемый алгоритм и параметры его конфигурации. Далее алгоритм начнет работу и по окончании на экране будут выведены все найденные алгоритмом зависимости.

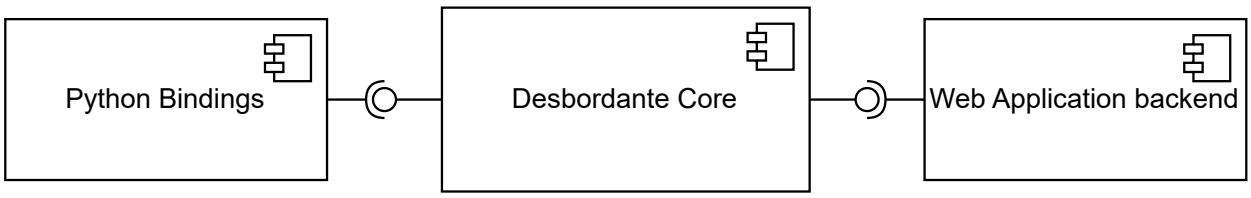

Рис. 3: Компоненты платформы, взаимодействующие с ядром Desbordante.

Ядром Desbordante (Desbordante Core) является библиотека на языке C++, в которой реализованы алгоритмы поиска зависимостей, вся необходимая для них инфраструктура и структуры данных. Библиотека предоставляет API для конфигурации алгоритмов, вызова алгоритмов над конкретным набором данных и получения результатов. Этим API пользуется библиотека на языке Python, которая является обёрткой над

ядром Desbordante и делает доступными все возможности платформы для программ на Python. Также API использует бэкенд веб-приложения[2], реализующего веб-интерфейс для платформы. Данное взаимодействие показано на Рис. 3. У Desbordante также есть консольный интерфейс, однако в данный момент он является частью библиотеки-ядра, а в будущем будет реализован на языке Python через взаимодействие с Python-оберткой.

В данной работе сосредоточимся только на ядре Desbordante. На Рис. 4 представлены его основные компоненты. Компонент `Algorithms` является в некотором смысле основным, поскольку содержит реализации самих алгоритмов и вспомогательные структуры, уникальные для конкретного алгоритма. `Algorithms` использует компоненты `Utility`, `Data Model` и `Options`. `Utility` содержит вспомогательные модули, потенциально полезные для всех алгоритмов, например, класс, реализующий индикатор прогресса алгоритма, функции, вычисляющие различные метрики от строк, вспомогательные функции для работы с битсетами и т. д. `Data Model` реализует классы, представляющие данные. Сюда входят классы для представления таблицы, таблицы с типизированными колонками, транзакционных данных, графовых данных. Компонент `Options` содержит всю логику, связанную с опциями алгоритмов. Алгоритмы могут иметь сложную логику конфигурирования. Например, некоторые опции могут становиться доступными только при особых значениях других опций. Также каждую опцию необходимо проверять на корректность. У опций может быть (или отсутствовать) значение по умолчанию или логика алгоритма может требовать особого вида значения опции, отличающегося от того, в котором её задает пользователь. Например, пользователь задает опцию в виде строки, а в алгоритме она преобразовывается в значение перечисления. Для того чтобы избежать повторения кода работы с опциями в разных алгоритмах, вся общая логика вынесена в отдельный компонент. `Algorithm Factory` создает и конфигурирует объекты алгоритмов. Компонент `Python Bindings` реализует обертку над C++ библиотекой и предоставляет интерфейс для

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

Python. В компоненте `Tests` реализованы тесты алгоритмов и некоторых вспомогательных модулей, а также необходимая для тестирования инфраструктура.

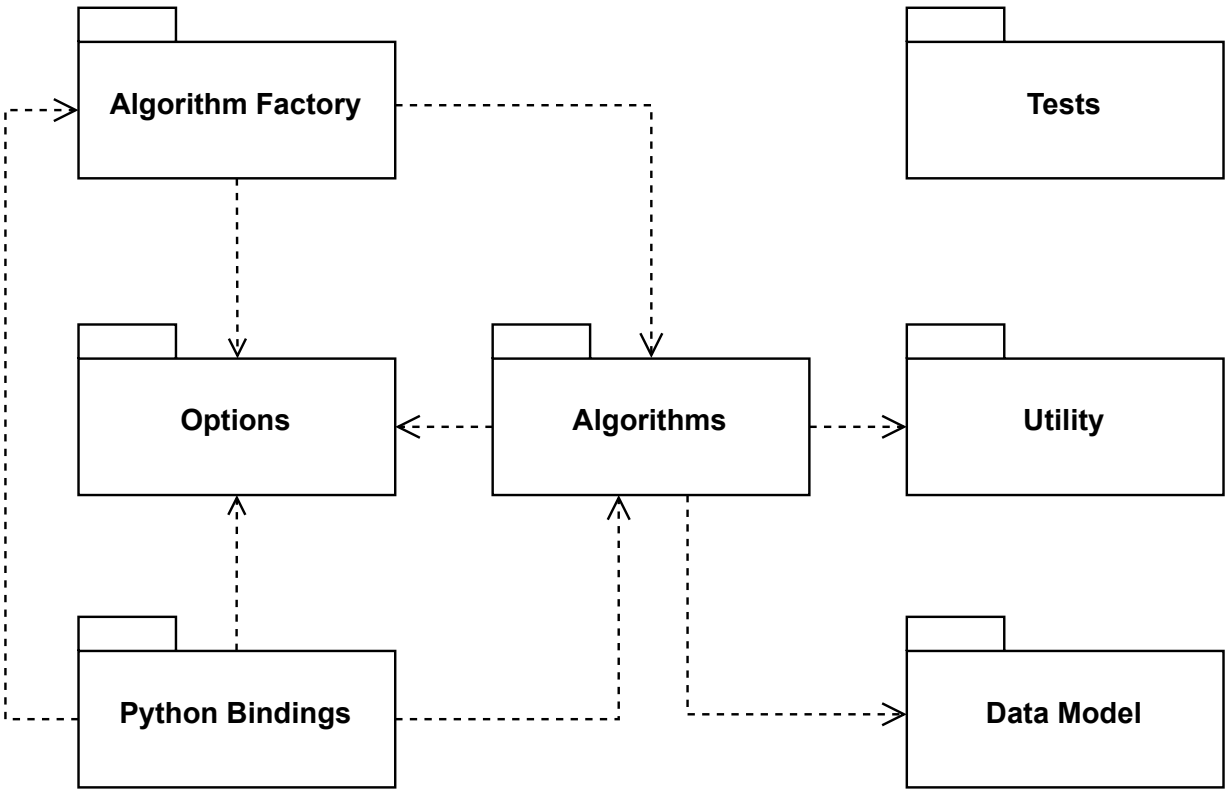

Рис. 4: Компоненты ядра Desbordante.

3 Реализация

3.1 Обычная версия HyUCC

При добавлении нового алгоритма в Desbordante, необходимо реализовать класс, который является наследником абстрактного класса `Algorithm`. `Algorithm` предоставляет интерфейс, общий для всех алгоритмов поиска примитивов. Также в нем реализована такая общая функциональность, как взаимодействие с индикатором прогресса, взаимодействие с объектом конфигурации, управление стадиями выполнения алгоритма.

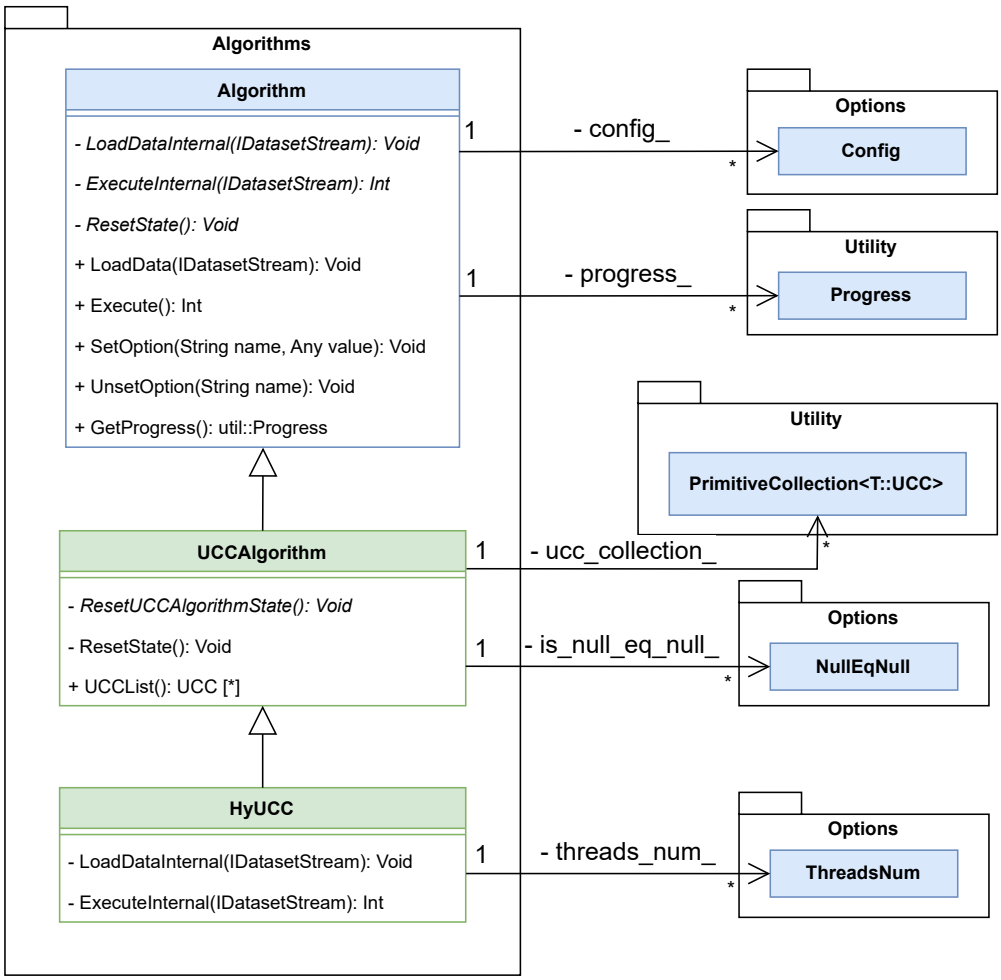

Рис. 5: Иерархия классов алгоритмов поиска примитивов.

Диаграмма, показывающая место HyUCC в иерархии наследования классов алгоритмов представлена на Рис. 5. Зеленым цветом обозначены классы, реализованные автором в ходе данной работы, си-

ним — все остальные. Класс `Algorithm` имеет три абстрактных метода. `LoadDataInternal` должна реализовать логической шаг первичной обработки входной таблицы, в него всегда входит парсинг. Также частью этого шага может быть построение нужных алгоритмов особых представлений таблицы. В HyUCC в него входит построение партиций. Обработка таблицы выделена в отдельный шаг, чтобы позволить выполнять один и тот же алгоритм с разными параметрами над одной и той же таблицей без её повторной обработки, тем самым повысив производительность. Функция `ResetState` выполняет сброс промежуточных структур алгоритма (кроме построенных на шаге загрузки данных), чтобы повторный запуск работал корректно. В `ExecuteInternal` должна быть реализована сама логика поиска. Функция возвращает время работы алгоритма в миллисекундах.

Абстрактный класс `UCCAlgorithm` предоставляет интерфейс, общий для всех алгоритмов поиска UCC. В данный момент это только HyUCC, но разрабатываются и другие. В интерфейс по сути входит только метод получения результатов алгоритма, то есть списка найденных уникальных комбинаций колонок. Также данный класс хранит опцию, характеризующую семантику сравнения NULL-значений, общую для всех алгоритмов поиска UCC. Реализует виртуальный `ResetState`, который не делает ничего кроме очищения списка найденных комбинаций колонок на прошлом запуске и вызова абстрактной функции `ResetUCCAlgorithmState`. В функции `ResetUCCAlgorithmState` класс-наследник `UCCAlgorithm` должен реализовать сброс своих промежуточных структур. В HyUCC она определяется как пустая, поскольку все промежуточные структуры разрушаются с выходом из функции `ExecuteInternal`.

В работе [18], описывающей HyUCC, сказано, что HyUCC и HyFD очень похожи архитектурно, однако не делается почти никаких утверждений о том, какие модули алгоритмов могут быть переиспользованы. Более того в реализации HyUCC в Metanome[3] код модулей, совпадающих с HyFD, просто скопирован в HyUCC, что неприемлемо для индустри-

[3]https://github.com/HPI-Information-Systems/metanome-algorithms/blob/master/HyUCC

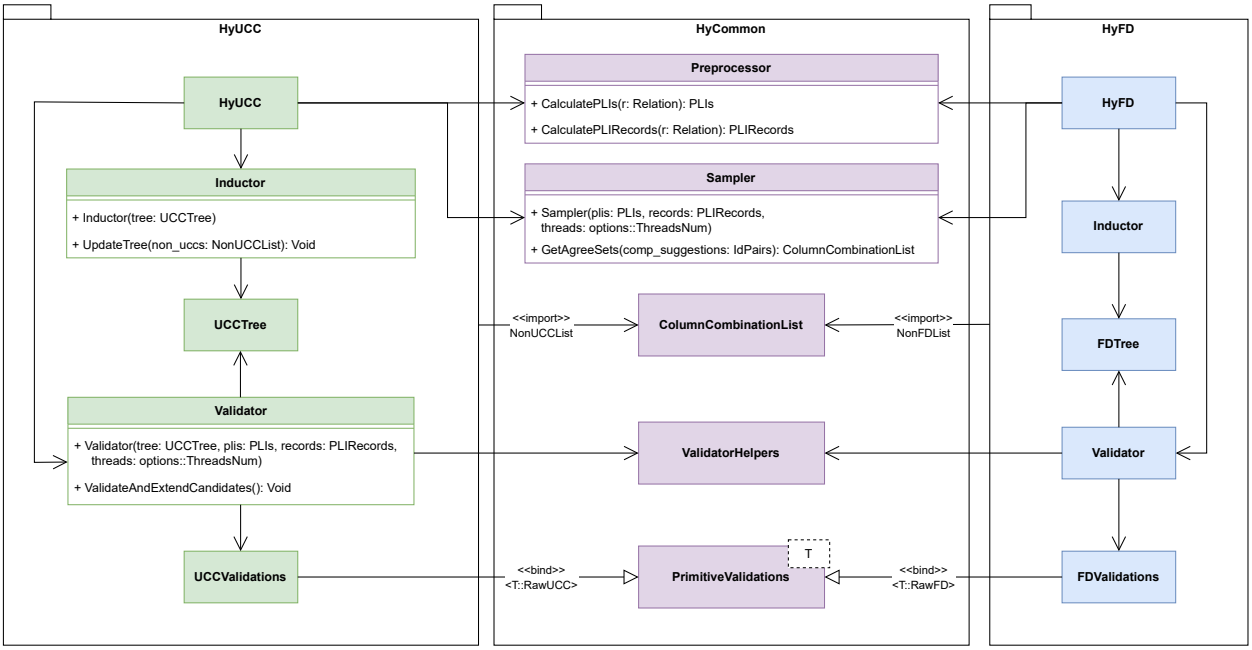

Рис. 6: Иерархия классов HyUCC и HyFD.

ального решения. HyFD уже был реализован в Desbordante, поэтому при разработке HyUCC необходимо было выделить общие модули и переиспользовать их. Диаграмма классов алгоритмов HyUCC и HyFD, а также их основных модулей изображена на Рис. 6. Зеленым цветом изображены классы, реализованные автором в ходе данной работы, фиолетовым — общие модули HyUCC и HyFD, вынесенные из HyFD и обобщенные для использования обоими алгоритмами, синим — классы, реализованные другими людьми.

Компонент `Preprocessor` в HyUCC полностью аналогичен компоненту в HyFD.

Компонент `Sampler` также идентичен в обоих алгоритмах, однако наборы колонок, который он возвращает в качестве результата имеет разную семантику. В HyUCC набор колонок целиком означает не-UCC, а в HyFD он описывает не-FD. Все колонки кроме последней считаются левой частью функциональной зависимости, а последняя колонка — правой. Это семантическое значение в коде задается при помощи классов-оберток над `Sampler` (не изображенных на диаграмме для простоты), которые реализуют методы `GetNonUCCs` и `GetNonFDs` соответственно. В качестве возвращаемых типов и тем самым типов для представле-

ния списков не-UCC и не-FD используются псевдонимы `NonUCCList` и `NonFDList`.

`ValidatorHelpers` реализует общую функциональность для модулей `Validator`. А именно функцию логгирования текущего уровня решетки, построение идентификатора строки, построение следующего уровня обхода решетки атрибутов.

`PrimitiveValidations` хранит кандидатов, проверенных как некорректные при валидации, статистику проверенных кандидатов и пары строк, которые следует проверить в первую очередь при следующем вызове `Sampler` (*comparison suggestions*).

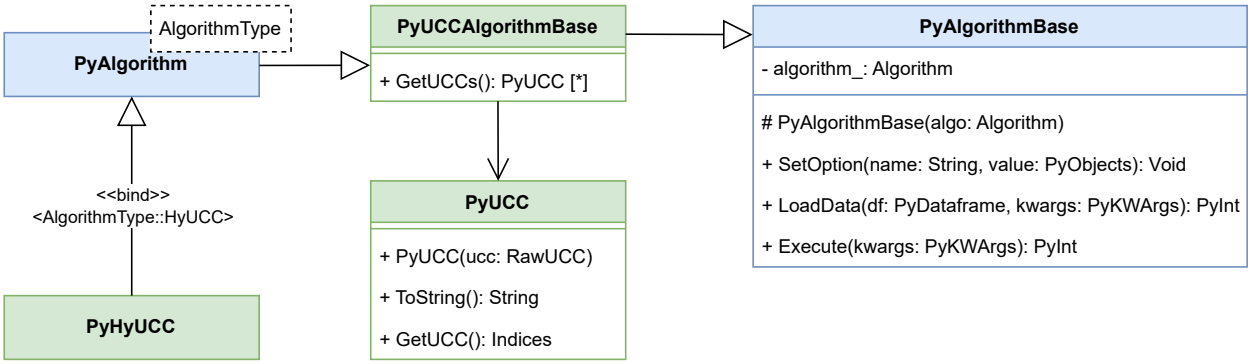

Рис. 7: Иерархия классов, предоставляющих Python-интерфейс к HyUCC.

Для того чтобы HyUCC был доступен из программ на Python, необходимо реализовать соответствующие классы-обертки и зарегистрировать их в модуле `Python Bindings`. На Рис. 7 представлены необходимые классы. `PyUCCAlgorithmBase` является базовым для всех алгоритмов поиска UCC. Он предоставляет общий интерфейс, а именно функцию получения найденных UCC. Для UCC также был реализован класс-обертка `PyUCC`. Класс `PyHyUCC` является оберткой для самого HyUCC и реализуется через уже существующий шаблон `PyAlgorithm`.

Также был разработан общий код для тестирования алгоритмов поиска UCC. При реализации нового алгоритма достаточно добавить его класс в список тестируемых алгоритмов, чтобы на нем запускалось тестирование. Тесты проверяют, что тестируемый алгоритм на фиксированных наборах данных возвращает правильный список уникальных комбинаций колонок. Сравниваются хеши от списков, а не

списки напрямую.

3.2 Модифицированная версия HyUCC

Поскольку каждому компоненту HyUCC для того чтобы начать работу необходимо получить все данные от предыдущего компонента, то никакая межкомпонентная параллельность не представляется возможной. Поэтому сосредоточимся на поиске узких мест внутри самих компонентов.

В работе [17], описывающей HyFD, авторы утверждают, что проверки отдельных кандидатов на UCC в `Validator` не связаны между собой так же, как и сравнения строк в `Sampler`, а значит могут выполняться параллельно. Однако в реализациях HyFD и HyUCC, представленных в Metanome, распараллелен только `Validator`. В работе [18], описывающей HyUCC, авторы уже говорят только про распараллеливание `Validator`. Поэтому автором данной работы сначала была реализована параллельная версия `Validator`, проверяющая разных кандидатов текущего уровня решетки параллельно. При реализации использовался пул потоков `boost::asio::thread_pool` из библиотеки `Boost.Asio`.

Профилирование кода показало, что на всех наборах данных большую часть времени занимает либо `Sampler`, либо `Validator`. На `Inductor` в среднем уходило не больше 5% времени выполнения, поэтому его не имеет смысла ускорять. На наборах данных, где большую часть времени занимало выполнение `Sampler`, около 80% времени от работы компонента уходило на функцию инициализации. Она выполняется один раз при первом вызове `Sampler`. Таким образом, нет смысла выполнять в разных потоках сравнение строк, то есть построение не-UCC из пары строк, вне функции инициализации. Более того, эксперименты показали, что распараллеливание сравнений строк внутри одной партиции наоборот заметно ухудшает производительности алгоритма, вероятно из-за слишком больших накладных расходов на управление потоками.

Функция инициализации сортирует кластеры во входных партициях и инициализирует очередь из объектов оценки эффективности сэмпли-

рования по отдельному атрибуту. Сортировка каждого кластера независима от каждого другого. Однако каждая сортировка по-отдельности вносит малый вклад в общее время выполнения, поэтому выделение каждой сортировки в отдельную задачу для пула потоков является не самым эффективным решением. Посредством экспериментов было выяснено, что выделять сортировки кластеров индивидуального атрибута и тем самым одной партиции в отдельную задачу гораздо более эффективно. В качестве пула потоков также использовался `boost::asio::thread_pool`.

Инициализация очереди из объектов оценки эффективности сэмплирования состоит из итерации по атрибутам таблицы и вычисления не-UCC для каждого атрибута со скользящим окном размера один. Вычисления не-UCC для разных атрибутов не связаны между собой и могут быть выполнены параллельно. Эксперименты с реализациями показали, что объединять несколько атрибутов в группу для обработки потоком немного более эффективно. В данной функции большую часть времени занимает сравнение двух строк, то есть вычисление набора атрибутов, в которых данные строки совпадают. Поскольку все строки одинаковой длины, это сравнение удобно векторизуется с помощью SIMD инструкций. Однако эксперименты показали, что ручная векторизация дает прирост в производительности только на очень широких таблицах (> 1000 колонок), а на остальных либо не дает никакого прироста, либо даже немного ухудшает производительность. Поэтому векторизованную версию было решено не использовать на недостаточно широких таблицах, то есть на абсолютном большинстве.

4 Эксперименты

4.1 Методология

В экспериментальном исследовании сравнивается один и тот же алгоритм HyUCC, реализованный в разных системах. При этом алгоритм возвращает полные и корректные результаты, если памяти достаточно, поэтому метрикой оценки производительности были выбраны время выполнения алгоритма и максимальный размер используемой памяти.

Поскольку не существует известного эталонного тестового набора данных для алгоритмов поиска уникальных комбинаций колонок, то экспериментальные данные были взяты из работ [17, 18], так как именно с реализацией алгоритма HyUCC, выполненной авторами этих работ, производилось сравнение. Используемые наборы данных и их характеристики представлены в Таблице 2.

Таблица 2: Экспериментальные наборы данных.

Имя	Тип	Количество строк	Количество колонок	Размер	Количество UCC
iris	Реальный	150	5	4.44КБ	0
balance-scale	Реальный	625	5	6.10КБ	1
chess	Реальный	28.05k	7	519.34КБ	1
abalone	Реальный	4.17k	9	187.38КБ	29
nursery	Реальный	12.96k	9	1.01МБ	1
breast-cancer-wisconsin	Реальный	699	11	19.42КБ	0
bridges	Реальный	108	13	5.98КБ	5
echocardiogram	Реальный	132	13	5.96КБ	72
adult	Реальный	32.56k	15	3.44МБ	0
letter	Реальный	20k	17	695.86КБ	0
ncvoter	Реальный	1k	19	150.71КБ	69
hepatitis	Реальный	155	20	7.37КБ	348
horse	Реальный	300	29	24.86КБ	253
EpicVitals	Реальный	1.24M	7	32.61МБ	0
EpicMeds	Реальный	1.28M	10	54.15МБ	1
plista	Реальный	1k	63	575.05КБ	0
flight	Реальный	1k	109	568.57КБ	26652
fd-reduced-30	Синтетический	250k	30	67.95МБ	3564
iowa1kk	Реальный	1M	24	209.55МБ	1

Для того чтобы минимизировать влияние сторонних факторов на производительность алгоритмов были выполнены следующие действия. Кеши ОС сбрасывались между каждым запуском с помощью записи “3” в `/proc/sys/vm/drop_caches`, на все время экспериментов выключался

файл подкачки командой `swapoff -a`, частота процессора фиксировалась командой `cpufreq-set` и выставлялась в наибольшее возможное значение. Все тяжеловесные процессы были отключены на экспериментальной машине.

Характеристики системы, на которой проводились эксперименты: Intel Core i5-7200U CPU @ 2.50GHz (2 cores, 4 threads), 12 GiB RAM, 240GB KINGSTON SA400S3, Ubuntu 20.04.5 LTS, Kernel 5.11.0-71-generic, gcc-11 (Ubuntu 11.1.0-1ubuntu1 20.04) 11.1.0. Metanome запускался на OpenJDK версии 11.0.11 (LTS) с настройками по умолчанию (кроме размера кучи), поскольку такая конфигурация показывает наилучшие результаты [7]. Максимальный размер кучи виртуальной машины Java был выставлен в десять гигабайт. Desbordante компилировался с флагом оптимизаций `-O3`.

И Metanome, и Desbordante замеряют время выполнения алгоритма самостоятельно средствами языка программирования и выводят в консоль. Это значение и было взято в качестве времени выполнения алгоритма. Используемая алгоритмом память измерялась с помощью команды `time -f '%M'`. На каждом наборе данных алгоритм запускался десять раз, в качестве времени выполнения бралось среднее значение и строились доверительные интервалы с уровнем доверия 95%. На всех наборах данных NULL-значения считались равными.

Были проведены следующие эксперименты.

1. Сравнение однопоточной версии HyUCC без дополнительных оптимизаций, реализованной в Desbordante, с реализацией в Metanome.

2. Сравнение многопоточной версии HyUCC без дополнительных оптимизаций, реализованной в Desbordante, с такой же реализацией в Metanome.

3. Сравнение многопоточной версии HyUCC с дополнительными оптимизациями, предложенными автором данной работы и реализованной в Desbordante, с многопоточной версией без оптимизаций в Desbordante и с многопоточной версией в Metanome.

Однопоточная версия HyUCC, реализованная в Desbordante, на графиках и в таблицах обозначается как “Desbordante”, многопоточная версия только с распараллеленым `Validator` как “Desbordante Parallel”, многопоточная версия с распараллеленым `Sampler` и `Validator` как “Desbordante Optimized”.

4.2 Экспериментальное исследование реализаций

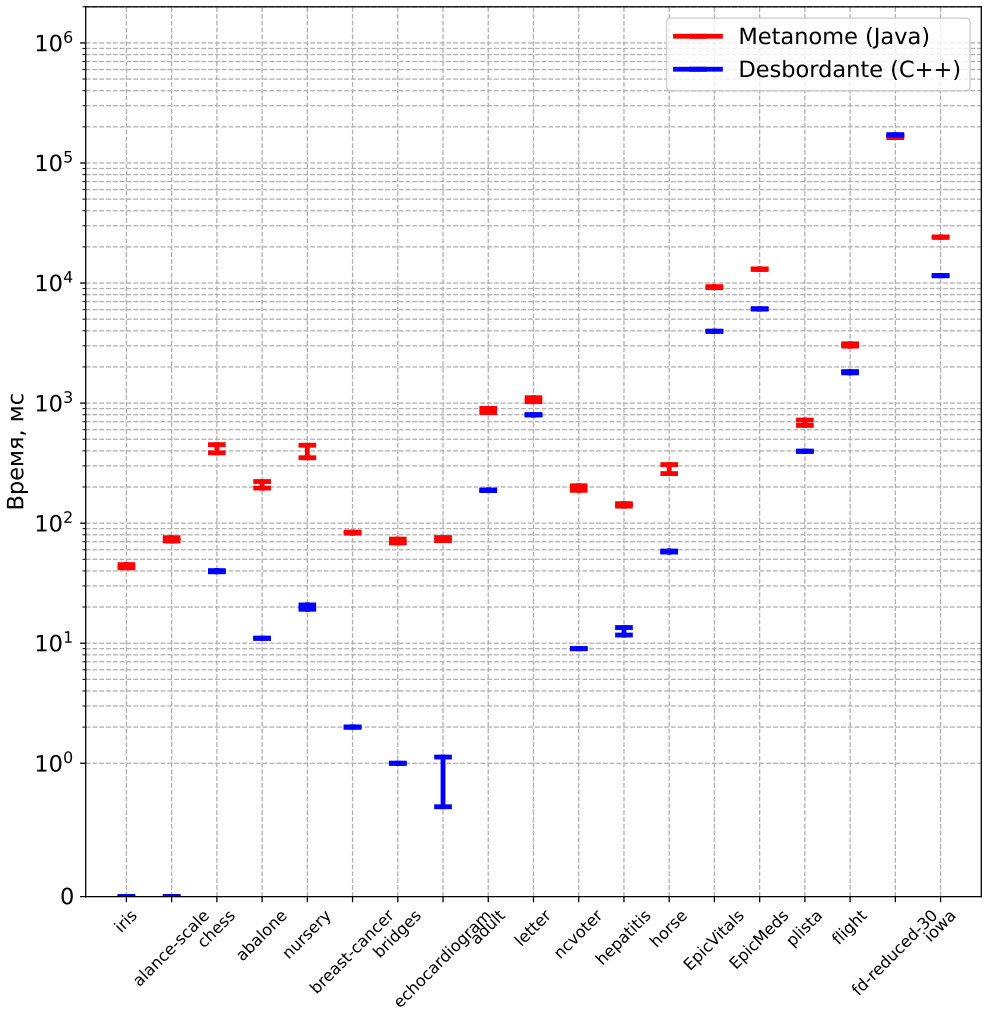

Рис. 8: Время выполнения однопоточных версий HyUCC.

Результаты первого эксперимента представлены на Рис. 8. Из них можно сделать вывод, что однопоточная реализация HyUCC в среднем на порядок эффективнее однопоточной реализации в Metanome. Реализация в Desbordante проигрывает в производительности Metanome только на наборе данных `fd-reduced-30` на 3–4%. Это может объясняться тем,

что алгоритм тратит 90% времени в функции `Validator`, проверяющей уникальность кандидата. То есть это случай, когда JIT компилятор виртуальной машины Java работает максимально эффективно. К тому же во время валидации алгоритм строит хеш-таблицу с массивом чисел в качестве ключа и числом в качестве значения. Реализация Metanome использует хеш-таблицу `Object2IntOpenHashMap` из библиотеки `fastutil`, специально оптимизированную под случай хранения `int` в качестве значений. Такой хеш-таблицы нет в наборе библиотек `Boost`, который используется в Desbordante, а искать отдельную реализацию нецелесообразно, поскольку проигрыш слишком мал.

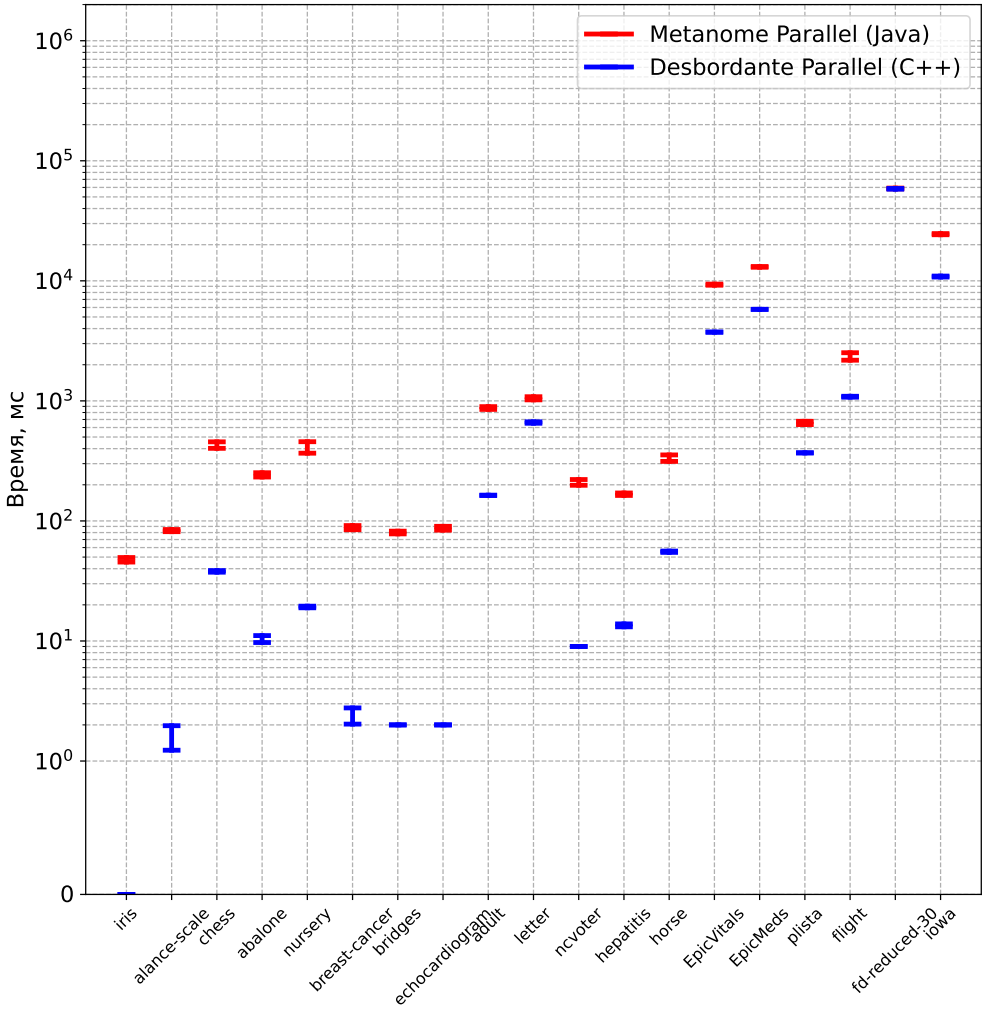

Рис. 9: Время выполнения многопоточных версий HyUCC, без распараллеленого `Sampler` в Desbordante.

На Рис. 9 изображены результаты второго эксперимента. Они показывают, что многопоточная версия Desbordante без дополнительных

оптимизаций, то есть только с распараллеленым `Validator` в среднем на порядок эффективнее многопоточной реализации в Metanome. В том числе на таблице `fd-reduced-30` реализация в Desbordante выполняется статистически за то же время, что в Metanome.

Результаты третьего эксперимента изображены на Рис. 10. Они также отображены в таблице 3.

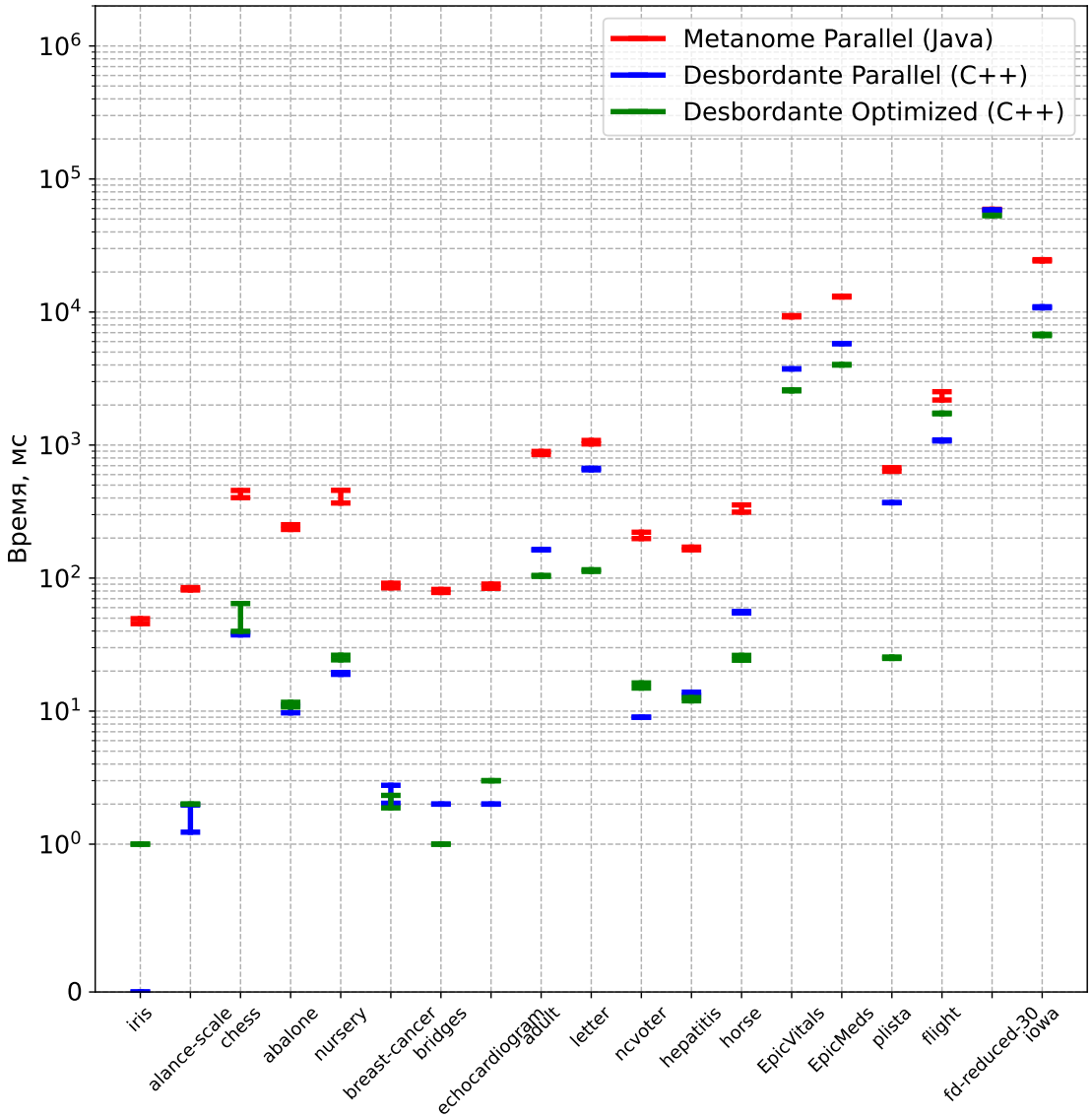

Рис. 10: Время выполнения многопоточных версий HyUCC, с распараллеленым `Sampler` в Desbordante.

По результатам можно сделать вывод, что модифицированная версия HyUCC с распараллеленным `Sampler`, реализованная в Desbordante в среднем в 1.3 раза эффективнее многопоточной версии Desbordante без распараллеленного `Sampler` и в 16.6 раз эффективнее параллельной

версии в Metanome. На некоторых таблицах модифицированная версия работает медленнее немодифицированной. Однако на всех из них кроме `flight` даже однопоточная версия Desbordante выполняется быстрее, чем за сто миллисекунд. То есть на этих таблицах выигрыш от распараллеливания перевешивается накладными расходами на управление дополнительными потоками. И поскольку проигрыш модифицированной версии находится в пределах десяти миллисекунд, это совсем незаметно для пользователя. Если же убрать таблицы, на которых время выполнения меньше ста миллисекунд, то модифицированная версия будет в среднем в 1.8 раз быстрее обычной многопоточной.

Таблица 3: Производительность оптимизированной версии HyUCC.

Набор данных	Metanome Parallel	Desbordante Parallel	Desbordante Optimized	DP/DO, ускорение	MP/DO, ускорение
EpicMeds	13066±92	5788±6	4212±11	1.37	3.10
EpicVitals	9293 ± 98	3738 ± 4	2584 ± 20	1.45	3.60
abalone	242 ± 9	10 ± 0	12 ± 0	0.83	20.17
adult	872 ± 23	163 ± 0	112 ± 0	1.46	7.79
balance-scale	83 ± 1	1 ± 0	2 ± 0	0.50	41.50
breast-cancer-wisconsin	88 ± 3	2 ± 0	2 ± 0	1.00	44.00
bridges	80 ± 2	2 ± 0	2 ± 0	1.00	40.00
chess	429 ± 26	38 ± 0	45 ± 1	0.84	9.53
echocardiogram	87 ± 3	2 ± 0	3 ± 0	0.67	29.00
fd-reduced-30	58802 ± 418	58380 ± 384	54679 ± 477	1.07	1.08
flight	2350 ± 168	1082 ± 9	1383 ± 135	0.78	1.70
hepatitis	167 ± 3	13 ± 0	15 ± 0	0.87	11.13
horse	334 ± 20	55 ± 0	49 ± 4	1.12	6.82
iowa1kk	24476 ± 271	10855 ± 101	6871 ± 79	1.58	3.56
iris	47 ± 2	0 ± 0	0 ± 0	1.00	47.00
letter	1051 ± 31	660 ± 9	117 ± 1	5.64	8.98
ncvoter	209 ± 11	9 ± 0	11 ± 1	0.82	19.00
nursery	411 ± 45	19 ± 0	25 ± 1	0.76	16.44
plista	655 ± 20	369 ± 0	337 ± 13	1.09	1.94

На Рис. 11 изображено сравнение модифицированной многопоточной версии HyUCC, реализованной в Desbordante, с многопоточной версией в Metanome по потребляемой памяти. Можно видеть, что модифицированная реализация HyUCC в Desbordante потребляет минимум в десять раз меньше памяти, чем многопоточная реализация в Metanome.

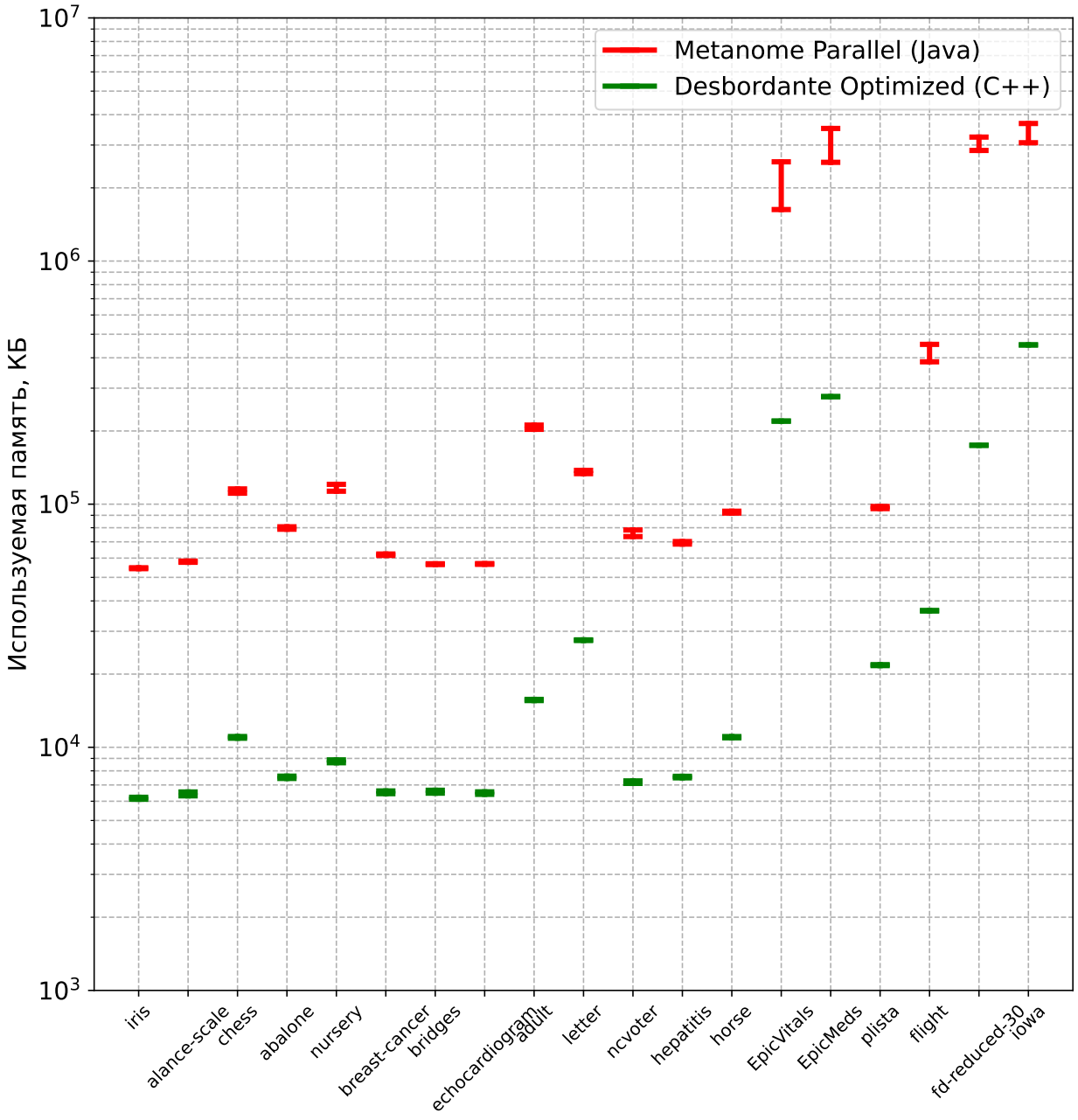

Рис. 11: Используемая память многопоточных версий HyUCC, с распараллеленым `Sampler` в Desbordante.

Заключение

В ходе работы были достигнуты следующие результаты.

1. Выполнен обзор существующих подходов к автоматическому поиску UCC и выбран один из наиболее эффективных алгоритмов поиска — HyUCC. С целью переиспользования имеющегося кода изучен алгоритм поиска функциональных зависимостей HyFD, как алгоритм того же класса, что и HyUCC, и его реализация в Desbordante.

2. Алгоритм HyUCC реализован для платформы Desbordante (C++). Созданы интерфейсы, общие для любых алгоритмов поиска UCC, модуль тестирования и интерфейс к HyUCC для взаимодействия через Python. При реализации были выделены в отдельные модули и переиспользованы общие части HyUCC и HyFD.

3. С помощью профилирования найдены узкие места и реализована модифицированная версия алгоритма с использованием многопоточности.

4. В рамках экспериментов использованы синтетические и реальные наборы данных, применявшиеся авторами Metanome. Были выявлены следующие факты:

   - обычная версия HyUCC, реализованная в платформе Desbordante, в среднем в 21 раз эффективнее реализации в Metanome;
   - модифицированная версия алгоритма в среднем в 2.5 раз эффективнее обычной реализации в платформе Desbordante, и в 17 раз — параллельной версии в Metanome;
   - модифицированная версия алгоритма потребляет в десять раз меньше памяти, чем многопоточная реализация в Metanome.

Код реализации открыт и доступен в репозитории[4].

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

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] Abedjan Ziawasch, Naumann Felix. Advancing the Discovery of Unique Column Combinations // Proceedings of the 20th ACM International Conference on Information and Knowledge Management. — CIKM '11. — New York, NY, USA : Association for Computing Machinery, 2011. — P. 1565–1570. — URL: https://doi.org/10.1145/2063576.2063801.

[3] Aggarwal Charu C. Data Mining: The Textbook. — Springer Publishing Company, Incorporated, 2015. — ISBN: 3319141414.

[4] Caruccio Loredana, Deufemia Vincenzo, Polese Giuseppe. Relaxed Functional Dependencies—A Survey of Approaches // IEEE Transactions on Knowledge and Data Engineering. — 2016. — Vol. 28, no. 1. — P. 147–165.

[5] Data Dependencies Extended for Variety and Veracity: A Family Tree / Shaoxu Song, Fei Gao, Ruihong Huang, Chaokun Wang // IEEE Transactions on Knowledge and Data Engineering. — 2022. — Vol. 34, no. 10. — P. 4717–4736.

[6] 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: https://doi.org/10.14778/2824032.2824086.

[7] Desbordante: a Framework for Exploring Limits of Dependency Discovery Algorithms / Maxim Strutovskiy, Nikita Bobrov, Kirill Smirnov, George A. Chernishev // 29th Conference of Open Innovations Association, FRUCT 2021, Tampere, Finland, May 12-14, 2021. —

IEEE, 2021. — P. 344–354. — URL: https://doi.org/10.23919/FRUCT52173.2021.9435469.

[8] Chernishev George, Polyntsov Michael, Chizhov Anton et al. Desbordante: from benchmarking suite to high-performance science-intensive data profiler (preprint). — 2023. — 2301.05965.

[9] Discovering All Most Specific Sentences / Dimitrios Gunopulos, Roni Khardon, Heikki Mannila et al. // ACM Trans. Database Syst. — 2003. — jun. — Vol. 28, no. 2. — P. 140–174. — URL: https://doi.org/10.1145/777943.777945.

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

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

[12] GORDIAN: Efficient and Scalable Discovery of Composite Keys / Yannis Sismanis, Paul Brown, Peter J. Haas, Berthold Reinwald // Proceedings of the 32nd International Conference on Very Large Data Bases. — VLDB '06. — VLDB Endowment, 2006. — P. 691–702.

[13] Giannella C., Wyss C.M. Finding Minimal Keys in a Relation Instance. — 1999.

[14] Hitting Set Enumeration with Partial Information for Unique Column Combination Discovery / Johann Birnick, Thomas Bläsius, Tobias Friedrich et al. // Proc. VLDB Endow. — 2020. — jul. — Vol. 13, no. 12. — P. 2270–2283. — URL: https://doi.org/10.14778/3407790.3407824.

[15] Ilyas Ihab F., Chu Xu. Data Cleaning. — New York, NY, USA : Association for Computing Machinery, 2019. — ISBN: 978-1-4503-7152-0.

[16] Lopes Stéphane, Petit Jean-Marc, Lakhal Lotfi. Efficient Discovery of Functional Dependencies and Armstrong Relations // Proceedings of the 7th International Conference on Extending Database Technology: Advances in Database Technology. — EDBT '00. — Berlin, Heidelberg : Springer-Verlag, 2000. — P. 350–364.

[17] Papenbrock Thorsten, Naumann Felix. A Hybrid Approach to Functional Dependency Discovery // Proceedings of the 2016 International Conference on Management of Data. — SIGMOD '16. — New York, NY, USA : Association for Computing Machinery, 2016. — P. 821–833. — URL: https://doi.org/10.1145/2882903.2915203.

[18] Papenbrock Thorsten, Naumann Felix. A Hybrid Approach for Efficient Unique Column Combination Discovery // Datenbanksysteme für Business, Technologie und Web (BTW 2017) / Ed. by Bernhard Mitschang, Daniela Nicklas, Frank Leymann et al. — Gesellschaft für Informatik, Bonn, 2017. — P. 195–204.

[19] Rahm Erhard, Bernstein Philip A. A Survey of Approaches to Automatic Schema Matching // The VLDB Journal. — 2001. — dec. — Vol. 10, no. 4. — P. 334–350. — URL: https://doi.org/10.1007/s007780100057.

[20] Scalable Discovery of Unique Column Combinations / Arvid Heise, Jorge-Arnulfo Quiané-Ruiz, Ziawasch Abedjan et al. // Proc. VLDB Endow. — 2013. — dec. — Vol. 7, no. 4. — P. 301–312. — URL: https://doi.org/10.14778/2732240.2732248.

[21] TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies / Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, Hannu Toivonen // Comput. J. — 1999. — Vol. 42, no. 2. — P. 100–111. — URL: https://doi.org/10.1093/comjnl/42.2.100.

[22] Toward Automated Large-Scale Information Integration and Discovery / Paul Brown, Peter Haas, Jussi Myllymaki et al. // Data Management in a Connected World: Essays Dedicated to Hartmut Wedekind on the Occasion of His 70th Birthday. — Berlin, Heidelberg : Springer-Verlag, 2005. — P. 161–180. — ISBN: 3540262954.