Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 23М04-мм

Реализация алгоритмов для поиска приближённых зависимостей включения в рамках платформы Desbordante

Чижов Антон Игоревич

Отчёт по учебной практике
в форме «Решение»

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

Санкт-Петербург
2024

Оглавление

Введение

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

Хотя ограничения целостности, такие как функциональные зависимости, успешно описывают семантику данных, некоторые данные в базе данных могут не соответствовать этим ограничениям. Одной из причин является тот факт, что данные могут поступать из различных независимых источников. В отличие от традиционных функциональных зависимостей, зависимости включения [8, 9] (IND) могут использовать разные отношения для проверки корректности данных.

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

Зависимости включения могут [2] использоваться для обнаружения несогласованности данных и восстановления целостности. На практике поиск IND может применяться для ограниченного набора задач. Аналогично функциональным зависимостям, для которых определено множество зависимостей, ключевой идеей которых является задание метрики для подсчёта ошибки, для зависимостей включения определяются приближённые IND. Приближённые зависимости можно [5, 11] применять для большего набора задач, например для обнаружения отсутствующих ограничений, проблем с целостностью данных.

Desbordante [6] представляет собой научно-ориентированный профи-

лировщик данных с открытым исходным кодом[1]. Создание Desbordante было мотивировано недостатками существующей платформы Metanome [4]. В Desbordante в качестве основного языка используется `C++`, а в Metanome — `Java`, из-за чего Metanome имеет не самую оптимальную производительность. Desbordante включает в себя алгоритмы для поиска различных примитивов и также предоставляет соответствующие интерфейсы для пользователей.

В данный момент Desbordante активно развивается и требует расширения набора поддерживаемых примитивов, в том числе включая поддержку IND и приближённых IND. Это стало мотивацией для задачи разработки алгоритмов поиска приближённых IND.

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

Постановка задачи

Целью данной работы является создание реализации алгоритма поиска приближённых зависимостей и его модификация с целью повышения производительности. Для достижения этой цели были поставлены следующие задачи:

- выполнить обзор предметной области;
- реализовать алгоритм поиска приближённых IND в рамках платформы Desbordante;
- исследовать возможные модификации реализованного алгоритма и модифицировать его наиболее эффективным образом;
- исследовать производительность алгоритма и выполнить тестирование.

1 Обзор предметной области

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

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

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

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

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

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

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

где пороговое значение ϵ задаёт максимальное значение ошибки для функции g_3', т.е. $g_3'(R_i(X) \subseteq R_j(Y)) \leq \epsilon$.

Пусть d это база данных над схемой базы данных R и r_i, $r_j \in d$ для соответствующих отношений R_i, $R_j \in \boldsymbol{R}$. Тогда функция g_3' вычисляет соотношение минимального количества кортежей, которые необходимо удалить из r, чтобы $R_i(X) \subseteq R_j(Y)$ выполнялась над r.

$$g_3'(R_i(X) \subseteq R_j(Y)) =$$
$$= 1 - \frac{max\{|\pi_X(r')| : r' \subseteq r_i\ u\ (d - \{r_i\}) \cup \{r'\} \models R_i(X) \subseteq R_j(Y)\}}{|\pi_X(r_i)|}$$

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

1.1 Алгоритмы поиска IND и AIND

В работе [10] проведён полноценный обзор алгоритмов поиска IND. Алгоритмы можно разделить на n-арные и унарные.

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

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

На данный момент существует только один алгоритм для поиска AIND Mind [12]. Mind — это n-арный алгоритм поиска IND, который использует поуровневый обход по решётке снизу вверх с отсечением кандидатов. Он может принимать в качестве входных данных посчитанные ранее унарные зависимости включения. При этом авторы данной статьи приводят информацию о том, как можно обобщить данный алгоритм для поиска AIND. Также алгоритм Mind использует СУБД для генерации кандидатов и их проверки.

Также Mind определяет свой алгоритм для поиска унарных зависимостей. Данный алгоритм похож на алгоритм Spider [7], но кандидаты-колонки группируются по типам. Поскольку Mind определяется в контексте СУБД, то информация о типах колонок известна в самом начале работы алгоритма. В случае, когда работа происходит вне СУБД, сначала необходимо вывести типы колонок. Но на практике колонки могут иметь смешанные типы, что может усложнить стадию валидации кандидатов.

Spider может быть обобщён для поиска AIND. Поэтому было решено в рамках данной работы реализовать алгоритм Spider, модифицировав его. После реализации Spider'а решено реализовать Mind, поскольку

данный алгоритм является единственным алгоритмом для поиска AIND.

1.2 Алгоритм Spider

Алгоритм поиска унарных IND Spider состоит из трёх стадий: препроцессинг, генерация кандидатов и валидация кандидатов.

1. Препроцессинг.

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

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

2. Генерация кандидатов.

Для каждой колонки создаётся объект атрибута. Он содержит:

- итератор по отсортированному домену значений колонки (т.е. итерация по строкам колоночного файла);
- два списка атрибутов — список исходных и список зависимых атрибутов.

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

На данной стадии, Spider инициализирует все объекты атрибутов с итераторами, указывающими на первые значения соответствующих

колоночных файлов. Оба списка кандидатов заполняются всеми атрибутами за исключением текущего. Более продвинутая версия алгоритма может уточнять данные списки кандидатов, используя информацию о типах колонок (например, нет смысла проверять зависимость между числовым типом с плавающей запятой и целочисленным типом). Базовая версия работает со всеми значениями как со строками.

3. Валидация кандидатов

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

- Из вершины кучи достаётся набор объектов атрибутов, имеющих одинаковое значение.
- Для каждого объекта атрибута, принадлежащего набору, его список зависимых атрибутов пересекается с набором. Другими словами, алгоритм удаляет атрибуты, которых нет в наборе из списка зависимых атрибутов. Соответствующие исходные атрибуты удаляются аналогичным образом (если для объекта атрибута A из списка кандидатов удаляется зависимый кандидат B, тогда для B удаляется исходный кандидат A).
- Для каждого необработанного объекта атрибута из набора, алгоритм продвигает итератор и вставлять объект атрибута в кучу.

2 Модификация Spider

1. Стадия препроцессинга.

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

На практике, создание колоночных файлов можно выполнять одновременно, за один проход по входной таблице. При этом заполнение структур данных, в которых хранятся данные для колоночных файлов может выполняться параллельно. Детали реализации рассматриваются в главе 3.

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

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

2. Генерация кандидатов.

В главе 1 упоминалось, что список кандидатов может быть уточнён благодаря информации о типах. Однако для того, чтобы использовать информацию о типах необходимо сделать дополнительный проход по значениям колонок, чтобы определить её тип.

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

3. Поиск приближённых IND.

Алгоритм Spider может быть обобщён для того, чтобы искать не

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

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

3 Реализация

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

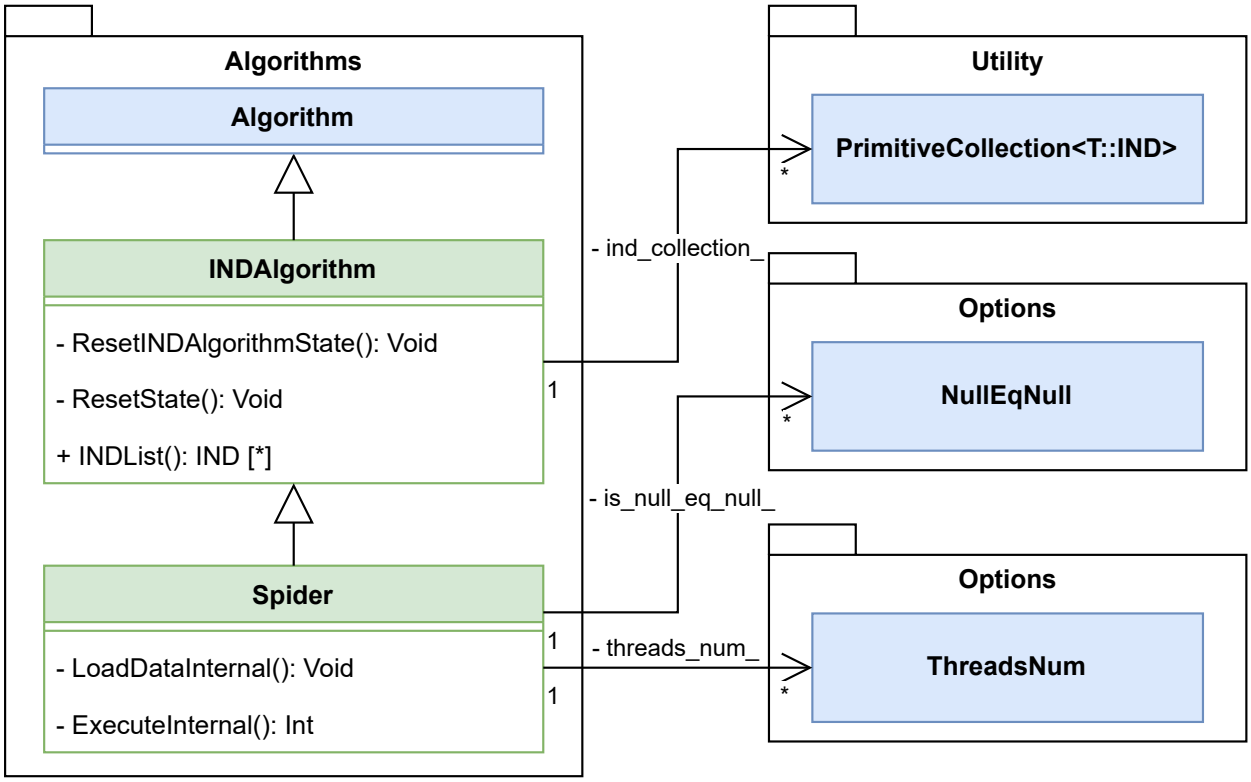

Рис. 1: Иерархия классов алгоритмов поиска IND.

На Рис. 1 представлена диаграмма, показывающая иерархию наследование классов алгоритмов поиска зависимостей включения. Зелёным цветом обозначены классы, реализованные в рамках данной работы. Наиболее важными методами при реализации нового алгоритма являются `LoadDataInternal`, `ExecuteInternal` и `ResetState`. Метод `LoadDataInternal` необходим для загрузки данных, с которыми происходит дальнейшая работа. На этом шаге происходит обработка входных таблиц, по итогам которой создаются отсортированные домены для всех колонок в случае алгоритма Spider. Также поскольку загрузка данных выполняется в отдельном шаге, ещё до начала выполнения, впоследствии можно запускать один и тот же алгоритм, выставляя новые опции. При

этом нет необходимости выполнять наиболее ресурсозатратный шаг — загрузку данных. В функции `ExecuteInternal` содержится основная логика самого алгоритма, в случае алгоритма Spider это генерация и валидация кандидатов. Функция `ResetState` необходима для выполнения сброса состояния алгоритма, после выполнения которой алгоритм должен вернуться к состоянию когда данные были загружены. Это может быть полезно при необходимости повторного выполнения алгоритма, но с другими конфигурационными параметрами.

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

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

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

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

Для подсчёта количества используемой памяти выполняется проход

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

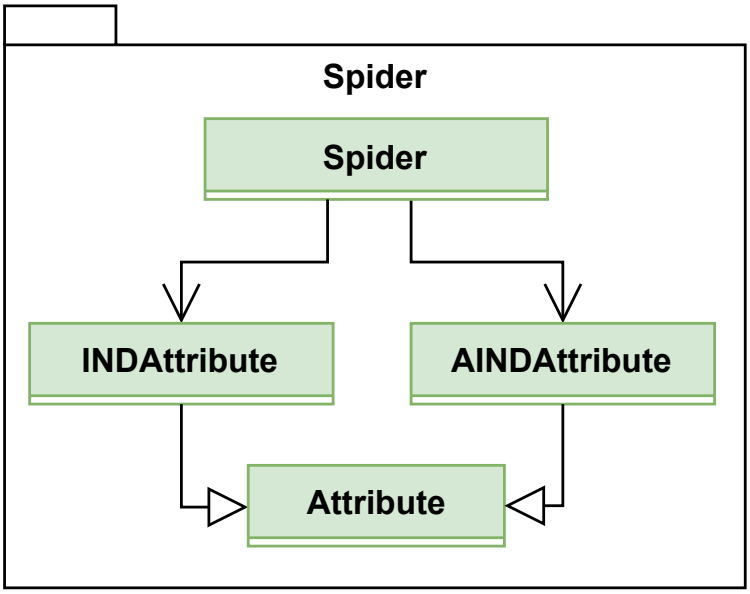

Рис. 2: Иерархия классов алгоритма Spider.

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

Класс `INDAttribute` реализует логику объекта атрибута, представленного в главе 1.2. То есть данный объект хранит два списка индексов кандидатов и отсекает кандидатов по мере обработки атрибута. А класс `AINDAttribute` содержит только вектор счётчиков, которые подсчитывают количество совпадающих значений с каждым из кандидатов. По итогам обработки всех объектов атрибута не происходит отсечения кан-

дидатов, а валидность кандидатов определяется после обработки всех доменов (кандидат валиден, если ошибка меньше порога возможной ошибки).

В главе 2 упоминалась возможная модификация стадии генерации кандидатов с их отсечением на основе максимального и минимального значения. Данная оптимизация в среднем даёт хорошее ускорение стадии валидации кандидатов, однако практически не влияет на общее время работы алгоритма. Это связано с тем, что стадия валидации кандидатов занимает в среднем около $\frac{1}{10}$ от времени выполнения стадии препроцессинга. Поэтому данная модификация не использовалась в итоговом решении.

4 Эксперименты

4.1 Методология

В экспериментальном исследовании сравнивается модифицированная реализация алгоритма Spider и тот же алгоритм, реализованный в Metanome. Время выполнения алгоритма является основной метрикой для измерения производительности алгоритмов. В экспериментах рассматривается среднее время выполнения за 5 запусков. Для очистки кэша файловой системы перед новым запуском алгоритма выполнялась команда `/proc/sys/vm/drop_caches`. Во время выполнения выполнения экспериментов все тяжеловесные процессы были отключены и фиксировалась частота процессора.

Характеристики системы. Для экспериментов использовалась система со следующими характеристиками: AMD Ryzen 7 4800H CPU @2.90GHz x 8 cores (16 threads), 32 GiB RAM, SSD A-Data S11 Pro AGAMMIXS11P512GT-C 512GB, running Ubuntu 22.04.

Конфигурация Java, используемая Metanome: openjdk 19.0.1 2022-10-18 OpenJDK Runtime Environment, OpenJDK 64-Bit Server VM (build 19.0.1+10-Ubuntu-1ubuntu122.04, mixed mode, sharing). Для Desbordante, использовалось: gcc(Ubuntu 11.3.0-1ubuntu1 22.04) 11.3.0, ldd (Ubuntu GLIBC 2.35-0ubuntu3.1) 2.35. Компиляция Desbordante происходила с опцией `-O3`.

Датасеты.

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

Таблица 1: Информация о наборах данных для экспериментов

В экспериментах используется 3 реальных датасета из различных областей и 3 синтетических, информация о которых представлена в таблице 1. Главной целью при выборе датасетов было собрать набор

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

4.2 Экспериментальное исследование

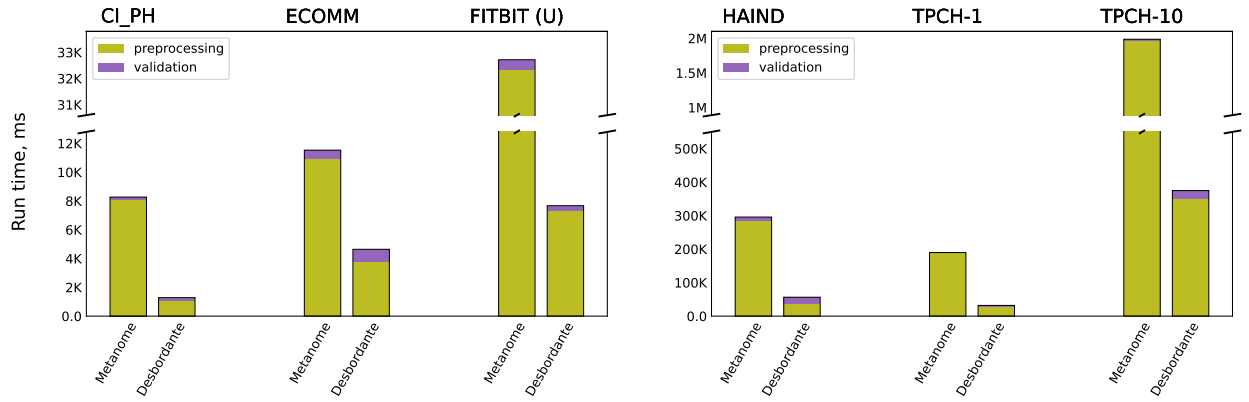

Рис. 3: Время выполнения Spider'а на различных датасетах (Desbordante и Metanome).

На Рис. 3 представлена диаграмма со сравнением времени выполнения между модифицированной версией Spider из Desbordante, запущенной в 8 потоков и однопоточной версией от Metanome.

	CI_PH	ECOMM	FITBIT (U)	HAIND	TPCH-1	TPCH-10
Parallel(8)	x6.45	x2.49	x4.27	x5.24	x5.93	x5.30

Таблица 2: Spider — Среднее ускорение

В таблице 2 представлено соотношение времени выполнения между реализациями. Среднее ускорение составило примерно 4.95. Для большинства датасетов ускорение составляет порядка 5-6 раз. Наибольшее

ускорение даёт модификация, связанная с параллельным заполнением структур доменов, поскольку эта стадия занимает наибольшее количество времени. Также заметный прирост скорости даёт отсутствие принудительного свопинга колоночных файлов после обработки.

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

В рамках данной работы проведён дополнительный эксперимент со сравнением времени выполнения алгоритма Spider для различных значений `scale factor` (от 1 до 15) синтетического датасета `TPCH`.

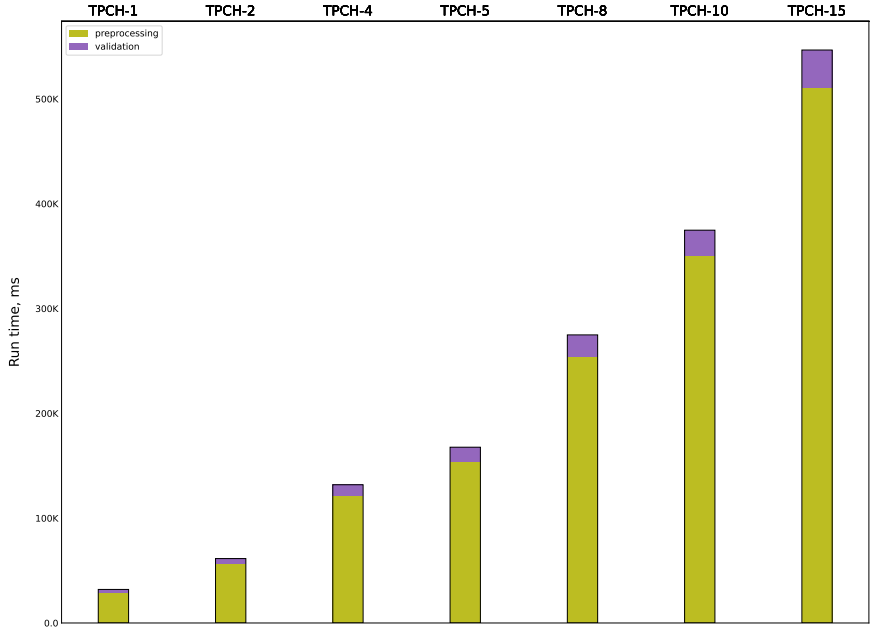

Рис. 4: Время выполнения Spider'а на датасете TPCH (Desbordante).

Из Рис. 4 видно, что время выполнения алгоритма пропорционально значению `scale factor`. Однако в данном даже для наибольшего

значения `scale factor` не происходит свопинга на диск. Например, для `TPCH-10` максимальное значение потребляемой памяти составляет порядка 6ГБ. Более важным является вопрос, касающийся производительности при свопинге промежуточных данных на диск. Однако в рамках данной работы данные эксперименты не проводились из-за отсутствия технической возможности.

Заключение

По итогам работы были получены следующие результаты:

- выполнен обзор предметной области и выбраны алгоритмы для реализации (Spider и Mind);
- реализован алгоритм поиска унарных IND и обобщён для поиска приближённых зависимостей;
- модифицирован алгоритм для повышения скорости его выполнения;
- выполнено сравнение производительности реализации с Metanome.

Результаты данной работы частично легли в основу статьи, которая была принята на конференцию FRUCT'33 (индексируется в Scopus). Код доступен на Github[3].

[3] https://github.com/Mstrutov/Desbordante/pull/304

Список литературы

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

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

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

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

[5] De Marchi Fabien, Petit Jean-Marc. Approximating a Set of Approximate Inclusion Dependencies. — 2005. — 01. — P. 633–640.

[6] Desbordante: a Framework for Exploring Limits of Dependency Discovery Algorithms / Maxim Strutovskiy, Nikita Bobrov, Kirill Smirnov, George Chernishev // 2021 29th Conference of Open Innovations Association (FRUCT). — IEEE, 2021. — may. — URL: http://dx.doi.org/10.23919/FRUCT52173.2021.9435469.

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

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

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

[10] Inclusion Dependency Discovery: An Experimental Evaluation of Thirteen Algorithms / Falco Dürsch, Axel Stebner, Fabian Windheuser et al. — 2019. — 11.

[11] Lopes Stéphane, Petit Jean-Marc, Toumani Farouk. Discovering Interesting Inclusion Dependencies: Application to Logical Database Tuning // Inf. Syst. — 2002. — mar. — Vol. 27, no. 1. — P. 1–19. — URL: https://doi.org/10.1016/S0306-4379(01)00027-8.

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