Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Струтовский Максим Андреевич

Реализация современных алгоритмов для поиска зависимостей в базах данных

Курсовая работа

Научный руководитель:
ассистент кафедры ИАС Чернышев Г.А.

Санкт-Петербург
2020

Оглавление

1. Введение

Функциональные зависимости — один из видов зависимостей, рассматриваемых при работе с реляционными базами данных. Владение подобной метаинформацией позволяет осуществлять очистку данных, проектирование схемы и оптимизацию запросов. Данная работа посвящена алгоритмам поиска функциональных зависимостей.

Алгоритмы поиска функциональных зависимостей обладают высокой стоимостью как в требуемой памяти, так и в затрачиваемом времени: зачастую минимальный набор функциональных зависимостей не может быть найден за полиномиальное от количества атрибутов время, и худшем случае сложность задачи можно оценить как $\mathcal{O}(N^2 \cdot 2^M)$, где N — количество строк в таблице, M — столбцов. Например, в таблице с 12 атрибутами всего 24564 возможных зависимостей, а количество кортежей может превосходить $1 \cdot 10^6$, вследствие чего, чтобы найти все функциональные зависимости по определению, то есть перебрав все возможные пары кортежей, необходимо совершить порядка 10^{16} операций. С другой стороны, количество зависимостей на практике не достигает приведённой оценки, в связи с чем существует множество алгоритмов, использующих всё более совершенные методы исследования пространства возможных функциональных зависимостей, оценка теоретического времени работы которых не даёт корректного представления об их производительности на реальных наборах данных. Таким образом, становится актуальной задача реализации современных алгоритмов поиска функциональных зависимостей на эффективном языке программирования, обладающем широким набором возможностей по оптимизации ресурсов, требуемых алгоритмами, а также позволяющем проводить точное измерение использования этих ресурсов. Существующее решение (Metanome) не решает эту задачу в полной мере из-за следующих недостатков языка Java:

- Часть ресурсов уходит на работу виртуальной машины Java, вследствие чего программы, написанные на этом языке, требуют ощутимо большего объёма памяти и исполняются медленнее при вы-

сокопроизводительных вычислениях, чем решения, основанные на более низкоуровневых технологиях.

- Используемые при работе кода, написанного на Java, технологии наподобие JIT компиляции и сборки мусора делают скорость работы программы менее стабильный, что усложняет задачу оценки производительности алгоритмов [7].

1.1. Постановка задачи

Целью данной работы является эффективная реализация современных алгоритмов поиска функциональных зависимостей на языке C++. Для достижения этой цели в данной работе были сформулированы следующие задачи:

- произвести обзор предметной области и приложений поиска функциональных зависимостей;
- произвести обзор существующего ПО для поиска функциональных зависимостей;
- реализовать алгоритм Pyro на языке программирования C++;
- сравнить производительность имплементации на C++ с существующим решением, написанным на Java.

2. Функциональные зависимости

2.1. Основные понятия

Определение 1. Функциональная зависимость определяет связь между атрибутами отношения. 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.

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.

Недостаток такого определения функциональной зависимости заключается в том, что артефакты в реальных данных, такие как опечатки, отсутствующие значения и шум, 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}, служащей формальным критерием того, что не совпадает только "небольшая" часть кортежей, и кандидат можно записать как приближённую ФЗ ($e(X \to Y) \leq e_{max}$).

Определение 3. Дано отношение r и потенциальная приближённая ФЗ $X \to Y$, тогда погрешность вычисляется как [9]:

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

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

Заметим, что при $e_{max} = 0$ поиск приближённых ФЗ сводится к поиску обычных ФЗ.

2.2. Поиск функциональных зависимостей

Знания о наличии функциональных зависимостей в таблице используются в следующих областях:

- нормализация схемы базы данных;
- очистка данных [1];
- оптимизация запросов [11];
- анализ данных;
- восстановление данных [13].

Таким образом, функциональные зависимости — важный инструмент при работе с большим количеством данных. Один из способов их поиска — привлечение эксперта в предметной области, знакомого с закономерностями и правилами, однако, если это невозможно, прибегают ко второму способу — использованию алгоритмов для обнаружения зависимостей [6]. Такие алгоритмы в основном используют одну из трёх стратегий.

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.

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.

3. *Гибридный подход* [10]. Современный гибридный подход был предложен, 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. Такой концепт показал свою состоятельность как на производительности, так и на количестве алгоритмов, основанных на нём. Подход был расширен для поиска приближённых зависимостей [9] и поиска зависимостей в условии динамически обновляющихся данных [5]. В первой приведённой работе был предложен алгоритм Pyro, который будет описан далее; во второй работе 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 [3, 8].

В связи с необходимостью получения знаний о функциональных зависимостях при анализе данных, существует множество алгоритмов и некоторое количество поддерживающих их платформ, предназначенных для их обнаружения. В данной работе будет рассмотрен один подобный проект, занимающий лидирующие позиции по современности и количеству реализованных алгоритмов.

2.3. Metanome

Metanome — открытая платформа, на уровне бэкэнда реализованная на языке программирования Java, основной целью которой является анализ метаданных [4]. 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, приближённых, порядковых зависимостей, зависимостей включения, уникальных сочетаний колонок и т.д. Кроме того, Metanome содержит реализацию многих структур данных, используемых в поиске, таких как: партиции, наборы разности и согласованности и различные графы 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. Pyro

Pyro — современный алгоритм для поиска приближённых ФЗ, основанный на гибридном подходе. Как было сказано выше, такой подход используют как множества согласованности, так и партиций в качестве ключевых структур данных. В реализации они представлены классами *AgreeSetSample* и *PLICache*. Данные сущности являются основой для оценки погрешности зависимостей-кандидатов и их последующей валидации, соответственно.

Отметим, что кэш партиций (*PLICache*) позволяет ускорить любой алгоритм, не только Pyro, использующий пересечение партиций в качестве основной операции при проверке функциональных зависимостей. Поскольку при верификации кандидатов одни и те же наборы атрибутов встречаются неоднократно, например, для проверки $X, Y \to W$ и $X, Y \to Z$ необходимо вычислить пересечение партиций $\pi(XY)$, трудоёмкая операция пересечения партиция будет вызвана несколько раз, с одним и тем же результатом. Чтобы избежать этого, используется префиксное дерево, содержащее результаты пересечений и проверяемое перед каждым пересечением на наличие уже вычисленной партиции. После выполнения пересечения результат сохраняется в дерево с вероятностью 0.5 — сохранение каждого результата привело бы к быстрому росту используемой памяти. В [2] были предложены более эффективные методы кэширования, но в данной работе ограничимся простым кэшированием, представленным в оригинальной статье.

Основную логику работы Pyro можно разделить на две фазы:

1. *оценка погрешности* для генерирования кандидатов на роль приближённых ФЗ;

2. основанный на сэмплировании *обход пространства кандидатов* в глубину.

Оценка погрешностей кандидатов — одна из самых трудоёмких частей алгоритмов поиска приближённых ФЗ. Pyro облегчает эту задачу следующим образом: вместо точного вычисления погрешности каждого

кандидата через пересечение партиций Pyro приближённо оценивает её на выборке кортежей, прибегая к упомянутому вычислению только в том случае, если полученная оценка не превосходит заданного гиперпараметра e_{max}. Непосредственно оценка основана на сравнении подмножеств кортежей, то есть использует методы, похожие на лежащие в основе вышепредставленного класса алгоритмов "обработка множеств согласованности и разности кортежей". В Pyro используется сфокусированное сэмплирование для достижения более точной оценки и меньшей зависимости от случайно выбранного подмножества кортежей. Говоря более подробно, сфокусированное сэмплирование отличается от обычного двумя условиями:

1. отбираемые множества согласованности обязаны быть надмножествами некоторого заданного набора атрибутов X, и
2. отбираемые кортежи должны иметь одинаковые значения при проекции на X.

Подобно пересечённым партициям, выборки наборов согласованности хранятся в собственном кэше, используемом для быстрого получения подходящих наборов при оценке погрешности на определённых наборах атрибутов.

Как и в большинстве алгоритмов, в Pyro присутствует пространство поиска — пространство всевозможных зависимостей, которое необходимо обойти, чтобы проверить каждую комбинацию левых и правых частей. Pyro представляет это пространство уникальным образом, следуя стратегии "разделяй и властвуй": для каждого атрибута X создаётся собственное пространство, которое можно рассматривать как частично упорядоченное (по включению левой части) множество зависимостей вида $Y \to X$. Каждый обход пространства начинается из *стартовой площадки* — одиночного атрибута в нижней части множества с наименьшей погрешностью соответствующей ФЗ. Проверяя прямые надмножества текущей вершины и оценивая их погрешности, алгоритм *восходит* по множеству, пока не встретит минимальную ФЗ. После верификации

найденной минимальной зависимости, пусть $Y_1, Y_2, Y_3, \ldots, Y_N \to X$, Pyro *стекает* к нижним уровням множества, оценивая погрешности обобщений этой зависимости: $Y_2, Y_3, \ldots, Y_N \to X$, $Y_3, \ldots, Y_N \to X$ и т.д. Каждое обобщение становится кандидатом на минимальную ФЗ, и Pyro рекурсивно *стекает* по ним, пока погрешности не превзойдут заданный порог. Наконец, после проверки новых кандидатов и их дополнений Pyro повторяет описанный процесс для непосещённых областей пространства поиска.

4. Реализация Pyro

Для реализации алгоритма Pyro были использованы наработки, полученные в рамках переписывания алгоритма TANE [12] на C++, — консольное приложение, реализованное на языке C++17 с применением библиотек boost и googletest. Приложение можно собрать под операционные системы Windows и Ubuntu — такая возможность была достигнута с помощью системы сборки CMake. Исходный код находится в свободном доступе на GitHub [1].

При реализации алгоритма было выделено несколько ключевых сущностей:

1. AgreeSetSample — класс, необходимый для сэмплирования множеств согласованности.

2. VerticalMap — шаблонный класс, реализованный как префиксное дерево `Vertical→V`, используемый как для быстрого получения информации типа V о единичной вершине, так и для обслуживания более сложных запросов, например, с целью получения множества вершин, удовлетворяющих некоторому условию.

3. PLICache — кэш партиций, реализованный как `VerticalMap<PLI>`.

4. SearchSpace — класс, реализующий логику обхода пространства поиска.

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

5. Эксперименты

Таблица 1: Сравнение производительности двух реализаций алгоритма Pyro

Реализация	Adult	BreastCancer	CIPublicHighway	EpicMeds	EpicVitals	Iowa	LegacyPayors	Neighbors
Java	26555 ± 341	26 ± 11	252239 ± 246135	247015 ± 800	10109 ± 208	138289 ± 2634	3159 ± 125	310 ± 44
C++	11895 ± 226	20 ± 0	57637 ± 29005	21216 ± 169	8133 ± 71	89220 ± 3925	585 ± 13	103 ± 0
Ускорение	2.2	—	—	11.6	1.2	1.5	5.4	3.0
Размерность	$15 \times 100K$	30×500	$18 \times 400K$	$10 \times 1.3M$	$7 \times 1.2M$	$24 \times 9M$	$4 \times 1.4M$	$7 \times 100K$

В таблице 1 приведены результаты замеров времени работы алгоритма Pyro на системе Ubuntu 18.04, Core i7 4700HQ 2.4–3.4ГГц, 12ГБ; gcc 10.2.0, OpenJDK 64-bit Server VM 11.0.9.1. Эксперименты проводились следующим образом: обе программы были запущены 10 раз на каждом из наборов данных, и в процессе исполнения замерялось время работы алгоритма в миллисекундах, без учёта времени, необходимого для загрузки данных в память и их преобразования во внутреннее представление. Для полученных результатов были построены доверительные интервалы с доверительной вероятностью 95% и занесены в таблицу. Для таблиц, на которых доверительные интервалы не пересеклись, также приведено полученное ускорение. При анализе представленных результатов можно прийти к следующим выводам:

- На большинстве таблиц удалось достичь существенного выигрыша по времени работы программы. Ускорение варьируется от 1.2 на EpicVitals до 11.6 на EpicMeds, достигая в среднем значения 4.15.

- Размах доверительных интервалов, полученных при измерении времени работы кода на C++, меньше размаха на Java на всех отношениях, кроме Iowa, что свидетельствует о меньшем разбросе времени работы программы от запуска к запуску.

- Пересечение интервалов на таблице BreastCancer можно объяснить её небольшим размером.

- На отношении CIPublicHighway время исполнения сильно варьируется, что нельзя объяснить ни работой сборщика мусора на очередной итерации, так как тогда это не наблюдалось бы в случае с C++, ни прерываниями ОС, так как система обладает многоядерным процессором, и в период проведения экспериментов не были запущены иные тяжеловесные процессы.

6. Заключение

В ходе данной работы были достигнуты следующие результаты:

- произведён краткий обзор предметной области;
- выделены слабые стороны существующих решений;
- разобран и реализован на языке C++ алгоритм поиска приближённых функциональных зависимостей Pyro;
- произведено сравнение реализации на C++ с реализацией на Java, показавшее более эффективную и стабильную работу первой на большинстве таблиц.

6.1. Направления продолжения работы

- исследование опций виртуальной машины Java, влияющих на производительность кода;
- анализ причин чрезмерной флуктуации времени работы на некоторых таблицах;
- сравнение требуемого различными реализациями объёма памяти;
- анализ причин превосходства с точки зрения производительности предложенной реализации.

Список литературы

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

[2] Birillo Anastasia, Bobrov Nikita. Smart Caching for Efficient Functional Dependency Discovery // New Trends in Databases and Information Systems / Ed. by Tatjana Welzer, Johann Eder, Vili Podgorelec et al. — Cham : Springer International Publishing, 2019. — P. 52–59.

[3] 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.

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

[5] 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.

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

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

[8] 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.

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

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

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

[12] 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.

[13] UGuide: User-Guided Discovery of FD-Detectable Errors / Saravanan Thirumuruganathan, Laure Berti-Equille, Mourad Ouzzani et al. — 2017. — 05. — P. 1385–1397.