Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 23.Б08-мм

Реализация алгоритма поиска условных функциональных зависимостей CFDFinder

Кожуков Иван Сергеевич

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
ассистент кафедры ИАС Г.А. Чернышев

Санкт-Петербург
2025

Оглавление

1. Введение **3**

2. Постановка задачи **4**

3. Условные функциональные зависимости **5**

3.1. Основные определения 5

3.2. Поиск CFD 6

3.3. Отсечение ненужных CFD 6

3.4. Типы условий для CFD 8

4. Алгоритм CFDFinder **10**

4.1. Предварительная обработка 10

4.2. Выборка кандидатов 12

4.3. Алгоритм генерации таблицы шаблонов 12

4.4. Отсечение шаблонов и кандидатов 13

4.5. Постобработка результатов 14

5. Реализация CFDFinder **15**

6. Эксперименты и сравнение с Metanome **18**

6.1. Сравнение результатов 18

6.2. Производительность 20

7. Заключение **22**

Список литературы **23**

1. Введение

Цель профилирования данных — изучить их структуру и содержание, а также выявить в них взаимосвязи. Важным аспектом этого процесса является поиск зависимостей между атрибутами, которые помогают находить скрытые связи и очищать данные от избыточности.

Хотя для выражения этих взаимосвязей обычно используются функциональные зависимости (functional dependencies, далее — FD), они часто оказываются слишком жёсткими для реальных данных. В больших наборах информация может быть противоречивой, и зависимости выполняются только при определённых условиях.

Поэтому научным сообществом были предложены условные функциональные зависимости (conditional functional dependencies, далее — CFD). Они позволяют задать ограничения, при которых правило зависимости работает, что значительно повышает гибкость и практическую ценность анализа, обеспечивая более точные результаты для неидеальных данных.

Desbordante [3] — высокопроизводительный инструмент для наукоёмкого профилирования данных, включающий в себя множество алгоритмов для нахождения и проверки различных примитивов. На данный момент в Desbordante уже реализованы алгоритмы поиска и проверки CFD, но имеется необходимость в расширении функциональности платформы. В результате прошлой учебной практики мною был проведен обзор [5] магистерской диссертации, в которой описывается алгоритм CFDFinder [4]. Этот алгоритм использует многоэтапный подход, который позволяет находить CFD с различными типами условий. CFDFinder был создан на языке Java для платформы Metanome [2], что значительно ограничивает его производительность. Поэтому в рамках продолжения прошлой учебной практики данный алгоритм было решено добавить в платформу Desbordante[1], реализовав его на языке C++.

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

2. Постановка задачи

Целью работы является реализация алгоритма поиска условных функциональных зависимостей CFDFinder. Для её достижения были поставлены следующие задачи:

1. Интегрировать алгоритм CFDFinder в платформу Desbordante;
2. Протестировать интегрированный алгоритм;
3. Сравнить производительность с реализацией в Metanome.

3. Условные функциональные зависимости

Здесь приведены сокращенные определения и принципы работы алгоритма, которые более подробно описаны в магистерской диссертации [4].

3.1. Основные определения

Пусть R — схема некоторого экземпляра отношения r.

Определение 1. FD $f\colon X \to A$, *где* $X \subseteq R$ *и* $A \in R$, допустима *над* r, *если* $\forall\ t_i, t_j \in r : t_i[X] = t_j[X] \Longrightarrow t_i[A] = t_j[A]$. *Будем называть* X — левой частью или LHS (left-hand side), *а* A — правой частью или RHS (right-hand side).

Определение 2. FD $f\colon X \to A$ *является* обобщением *другой FD* $g\colon Y \to A$, *если* $X \subset Y$. *В данной работе при упоминании обобщения будем считать, что* $|Y \setminus X| = 1$.

Определение 3. FD $f\colon X \to A$ *является* специализацией *другой FD* $g\colon Y \to A$, *если* $Y \subset X$.

Определение 4. CFD φ : — *это пара* $(f : X \to Y,\ T_p)$, *где* $f : X \to Y$ — *функциональная зависимость,* встроенная в CFD φ, *и* T_p — таблица шаблонов *(pattern tableau) из атрибутов* X *и* Y *и* $\forall A \in X \cup Y$ *и* $\forall t \in T_p : t[A] = a$, *где* $a \in dom(A)$, *или* $t[A] =$ "_", *где* "_" — переменная привязки *(variable binding, wildcard), которая обозначает, что для выполнения условия подходит любое значение из* $dom(A)$. *Кортеж* $t \in T_p$ *называется* шаблонным кортежем или шаблоном.

Определение 5. *Кортеж* $t \in r$ соответствует *шаблону* $t_p \in T_p$ *на множестве атрибутов* $S \subset R$, *обозначается* $t \asymp t_p$, *если* $\forall B \in S : t_p =$ "_" *или* $t_p[B] = t[B]$.

Определение 6. CFD φ $(R : X \to Y, T_p)$ соблюдается *на отношении* r, *если* $\forall t_i, t_j \in r$ *и* $\forall t_p \in T_p : t_i[X] = t_j[X] \asymp t_p[X] \Longrightarrow t_i[Y] = t_j[Y] \asymp t_p[Y]$.

3.2. Поиск CFD

Мы рассматриваем случай, когда нам неизвестны допустимые FD для заданного набора данных перед началом поиска CFD.

Так как процесс обнаружения FD легче процесса обнаружения CFD [1], появляется возможность использовать поиск FD как предварительный этап для обнаружения CFD. Более того в [4] показывается, что CFD, которые не относятся к стандартным FD, могут быть сформированы только на основе недопустимых FD и предлагается новый подход извлечения множества наиболее специализированных недопустимых FD (MaxNonFD), который интегрируется в работу алгоритма HyFD [7].

Далее для полученных кандидатов CFD можно подобрать таблицу на основе заданных критериев. Для этого в CFDFinder был доработан и модифицирован жадный алгоритм, предложенный в работе [6], которые позволяет сгенерировать таблицу шаблонов для кандидата CFD.

3.3. Отсечение ненужных CFD

Не все соблюдающиеся CFD имеют практический интерес, поэтому в [4] предлагается несколько критериев интересности, которые позволяют уменьшить пространство поиска и улучшить качество конечного результирующего набора.

Определение 7. Покрытие (cover) шаблона p — *множество кортежей, соответствующих p.*

Определение 8. Локальная поддержка p (local support) — *отношение количества кортежей в покрытии шаблона к размеру экземпляра отношения.*

Определение 9. Глобальная поддержка T_p (global support) — *отношение количества кортежей в объединении покрытий по всем шаблонам из T_p к размеру экземпляра отношения.*

Будем называть *поддержкой CFD* φ глобальную поддержу таблицы шаблонов φ.

Определение 10. Хранители (keepers) шаблона p — *множество кортежей, которые покрываются p и не приводят к нарушению встроенной FD и нарушениям вида:* $\exists t \in r$ *и* $\exists t_p \in T_p$*:* $t[X] \asymp t_p[X]$*, но* $t[Y] \not\asymp t_p[Y]$.

Определение 11. Локальная уверенность *p (local confidence) — отношение количества кортежей в наборе хранителей шаблона p к размеру покрытия этого шаблона.*

Определение 12. Глобальная уверенность T_p (global confidence) — *отношение количества кортежей в объединении хранителей по всем шаблонам* T_p *к количеству кортежей в объединении покрытий по всем шаблонам* T_p.

Будем называть *уверенностью CFD* φ глобальную уверенность таблицы шаблонов φ.

Определение 13. Минимальный прирост поддержки *(support gain) — нижняя граница локальной поддержки шаблонов в таблице.*

Определение 14. Максимальное снижение поддержки *(support drop) — верхняя граница разницы поддержки текущего кандидата CFD и поддержки его обобщений.*

Определение 15. Метрика g_1 — *доля нарушающих пар записей для заданной FD.*

В работе [4] предлагается несколько стратегии отсечения. При использовании алгоритма CFDFinder можно выбрать одну из них.

1. Отсекать CFD, которые не соответствуют заданным минимальным порогам уверенности и поддержки. Данная стратегия отсечения является принципом работы алгоритма из [6];

2. Отсекать CFD, которые не соответствуют заданному минимальному порогу прироста поддержки, максимальному порогу снижения поддержки;

3. Отсекать CFD, для встроенной FD которых превышена верхняя граница метрики g_1. Данная стратегия позволяет получить набор *частичных* FD, которые эквивалентны CFD, покрывающим весь экземпляр отношения.

Также в реализованной версии алгоритма для каждой стратегии можно указать максимальный размер LHS, на основе которого будут отсекаться CFD, добавляемые в набор результат.

3.4. Типы условий для CFD

Определение 16. *Шаблон p_p , который содержит ровно на одну привязанную константу больше, чем шаблон p_c, называется* родительским *шаблоном для p_c. Аналогично, шаблон p_c, содержащий ровно на одну привязанную константу меньше, чем шаблон p_p, называется* дочерним *шаблоном для p_p.*

Определение 17. *Шаблон, который содержит только переменные привязки для всех атрибутов, называется* нулевым шаблоном.

Стратегия расширения рассматривается как метод перечисления всех возможных шаблонов в наборе данных с помощью итеративной специализации нулевого шаблона, т.е. замены каждой переменной привязки в шаблоне на значение из домена этого атрибута. Вынесение логики расширения в отдельный модуль в алгоритме из [6] позволяет создавать CFD с различными типами условий. В CFDFinder реализуется три варианта:

Стандартные условия. Данный тип соответствует стандартному определению условия CFD.

Условия отрицания. Этот тип расширяет стандартные условия, позволяя вместе с условиями вида $A = 4$ использовать условия вида $A \neq 4$.

Условия с диапазоном. Здесь используется следующее представление переменной привязки. В таблице T для CFD φ: $(X \longrightarrow Y,\ T)$ для каждого шаблона $t_p \in T$ и для каждого атрибута $A \in X \cup Y$ $t_p[A] =$

$[a_l, a_r]$, где $a_l \in dom(A)$ — нижняя граница, а $a_r \in dom(A)$ — верхняя граница и $a_l < a_r$. Кортеж t соответствует этому условию, если $a_l \leq t[A] \leq a_r$. Таким образом, условия с диапазоном обобщают стандартные условия при $a_l = a_r = a \in dom(A)$.

4. Алгоритм CFDFinder

CFDFinder представлен как гибкий алгоритм поиска интересных CFD. При помощи поддержки нескольких стратегий отсечения и типов условий алгоритм позволяет находить соблюдающиеся CFD под разные задачи и наборы данных.

Алгоритм состоит из нескольких этапов, которые изображены на рисунке 1.

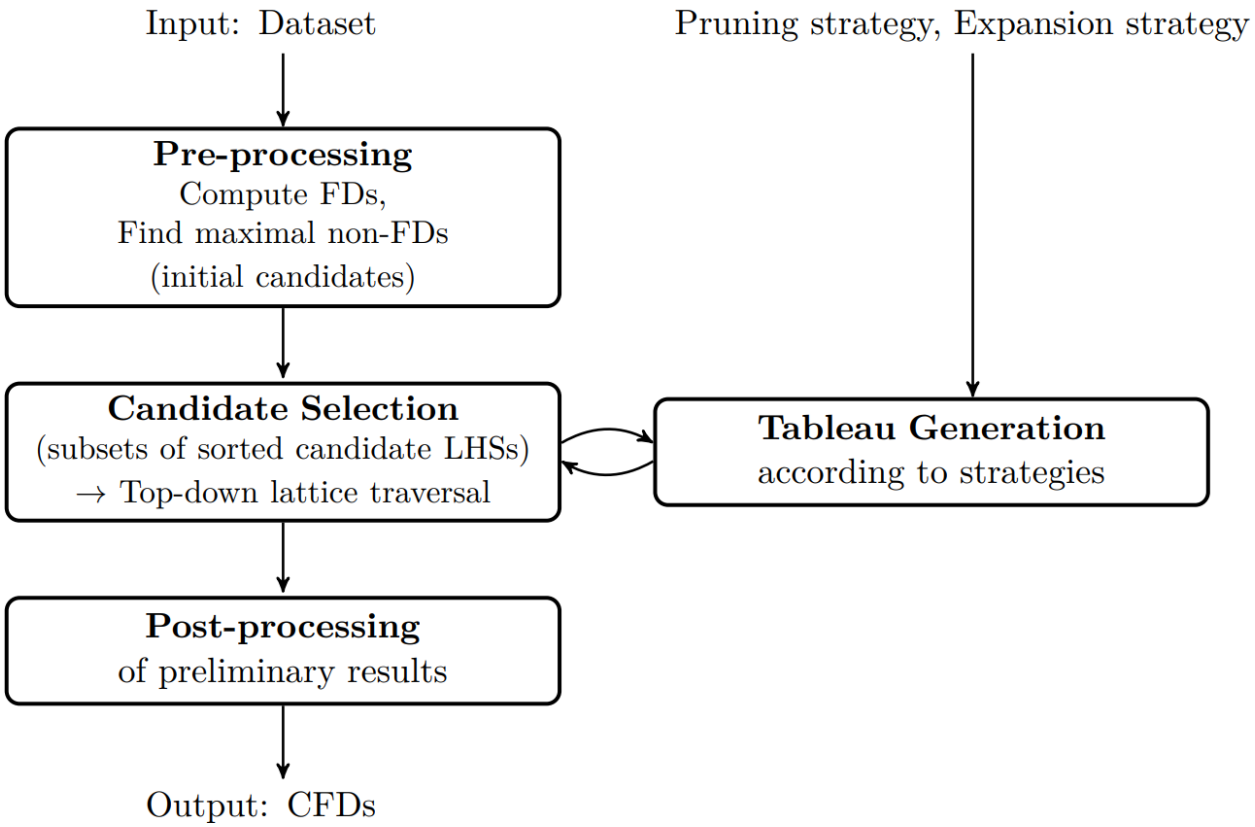

Рис. 1: Логика алгоритма CFDFinder [4].

Далее в этой главе будет представлен поэтапный разбор алгоритма, приближенный к реальной реализации, а не описанию из магистерской диссертации.

4.1. Предварительная обработка

Перед началом поиска условных зависимостей алгоритм преобразует исходные данные для создания начального набора кандидатов и генерации таблиц.

Сначала для каждой колонки создается *индекс отображения кластеров* (ClusterMaps), который представляет собой отображение строковых значений атрибутов в кластеры идентификаторов строк, содержащих эти значения. Его инвертированная версия используется для преобразования числовых идентификаторов кластеров обратно в читаемые строковые значения при получении результатов алгоритма, а также для задания порядка на кластерах в стратегии расширения с диапазоном.

Затем из ClusterMaps извлекаются *индексы списков позиций* (position list index, PLI) и создаются *сжатые записи* (compressed records). PLI для каждой колонки содержит списки идентификаторов строк, в которых встречается одно значение. Сжатые записи представлены как массив сжатых строк — массив идентификаторов значений. В этом индексе сопоставляется идентификатор записи и набор значений в этой записи.

После создания индексов происходит сбор начального набора кандидатов CFD. Сначала происходит поиск минимального набора FD с помощью алгоритма HyFD. Параллельно с поиском на этапах валидации (validation) и индукции (induction) HyFD собираются те недопустимые FD, которые не могут быть обобщены далее на данном этапе из-за уже построенного множества допустимых FD.

После завершения работы HyFD обнаруженные maxNonFD извлекаются из оставных компонентов, к ним добавляются обобщения обнаруженных допустимых FD, которые не являются специализацией кандидатов в наборе maxNonFD.

После получения набора кандидатов CFD алгоритм организует его в структуру уровней на основе количества атрибутов в LHS, формируя из них решетку (lattice).

В конце предварительной обработки алгоритм одновременно инвертирует индекс отображения кластеров, восстанавливает одиночные кластеры в PLI, а затем на их основе обновляет идентификаторы кластеров в сжатых записях, чтобы они соответствовали восстановленым PLI. Это необходимо для полного учета всех строк таблицы при составлении

покрытия на этапе расширения дочерних шаблонов.

4.2. Выборка кандидатов

Обход решетки происходит сверху вниз т.е. по уменьшению размера LHS. Начиная с последнего уровня алгоритм пытается создать допустимую таблицу шаблонов для каждого кандидата на основе заданных параметров с помощью функции `GenerateTableau`. Если создать таблицу получилось и она удовлетворяет установленным стратегией отсечения ограничениям, то кандидат добавляется в результирующий набор и на следующий уровень добавляются новые кандидаты CFD, сформированные из его обобщений. После завершения обработки уровня алгоритм переходит к следующему, пока не закончит обработку всей решетки.

4.3. Алгоритм генерации таблицы шаблонов

Сначала алгоритм формирует нулевой шаблон из атрибутов LHS кандидата CFD с помощью стратегии расширения. Все кортежи отношения добавляются в покрытие нулевого шаблона, так как по определению оно совпадает со всеми кортежами, после рассчитываются поддержка и уверенность и он добавляется в *границу* (frontier) F. Это множество шаблонов-кандидатов, отсортированное по дополнительной поддержке, которую этот шаблон приносит таблице.

Далее стратегия отсечения определяет, возможно ли улучшить таблицу T в соответствии с параметрами стратегии отсечения, чтобы при возможности закончить обработку раньше. Если улучшение возможно, алгоритм начинает обрабатывать шаблон с наибольшей дополнительной поддержкой, содержащийся в границе. Если шаблон удовлетворяет ограничениям стратегии отсечения, то он добавляется в таблицу, а его покрытие удаляется из покрытий всех остальных шаблонов в границе. Также для оставшихся в границе шаблонов стратегия отсечения проверяет, следует ли дальше рассматривать шаблон для добавления в таблицу на позднем этапе, если нет, то шаблон удаляется из неё.

Если обрабатываемый шаблон не подходит для добавления, то для него вычисляются все дочерние шаблоны в соответствии со стратегией расширения, для каждого из вычисляются покрытие, поддержка и уверенность, и, если шаблон подходит для рассмотрения по ограничениям стратегии отсечения, он добавляется в границу. Покрытие дочерних шаблонов вычисляется из покрытия их родительского шаблона в методе DetermineCover, чтобы снизить накладные расходы на вычисление покрытия по всему отношению. Так как дочерние шаблоны могут иметь много родителей, то стратегией отсечения гарантируется, что все родительские шаблоны будут рассмотрены до добавления дочернего в границу.

В конечном итоге алгоритм либо возвращает таблицу шаблонов для кандидата, либо завершает выполнение без результата, если глобальный порог поддержки не удаётся достичь до того, как все шаблоны-кандидаты будут использованы.

4.4. Отсечение шаблонов и кандидатов

CFDFinder реализуется несколько стратегий отсечения:

1. *Минимальный порог уверенности и поддержки.* Генерация таблицы продолжается до тех пор, пока в границе не закончатся шаблоны или пока не наберется минимальная поддержка для таблицы шаблонов. В таблицу добавляются только те шаблоны, которые имеют локальную уверенность больше или равную минимально установленной. Рассматриваются только шаблоны, имеющие ненулевую поддержку. Кандидат CFD добавляются в результирующий набор в случае, если таблица удовлетворяет заданным минимальным порогам;

2. *Минимальный порог прироста и максимальный порог снижения поддержки.* Алгоритм продолжает работу, пока в границе не закончатся шаблоны, или не наберется максимально установленное количество шаблонов в таблице, или пока не закончатся шаблоны,

имеющие поддержку выше и равную минимальному порогу прироста поддержки. Рассматриваются только шаблоны, имеющие прирост поддержки выше порога. Добавление в таблицу происходит на основе прироста поддержки и минимального порога уверенности. Кандидат CFD добавляется в результат, если снижение поддержки для него не превышает заданный порог;

3. *Метрика* g_1. Данная стратегия не рассматривает никакой другой шаблон кроме нулевого. Если на этапе проверки для добавления в таблицу нулевой шаблон не удовлетворяет метрике, то таблица остается пустой, иначе алгоритм добавляет в набор результатов частичную FD.

4.5. Постобработка результатов

В реализованной версии алгоритма в качестве постобработки выступает возможность фильтрации результатов по атрибуту RHS. Для этого создано расширение стратегии, основанной на минимальном приросте и максимальном снижении поддержки, которое перед добавлением в результирующий набор кандидата, проверяет наличие RHS в заданном списке и затем продолжает обработку наследованной стратегии. Данное расширение также интегрируется в Desbordante.

5. Реализация CFDFinder

Составные части алгоритма изображены на рисунке 2.

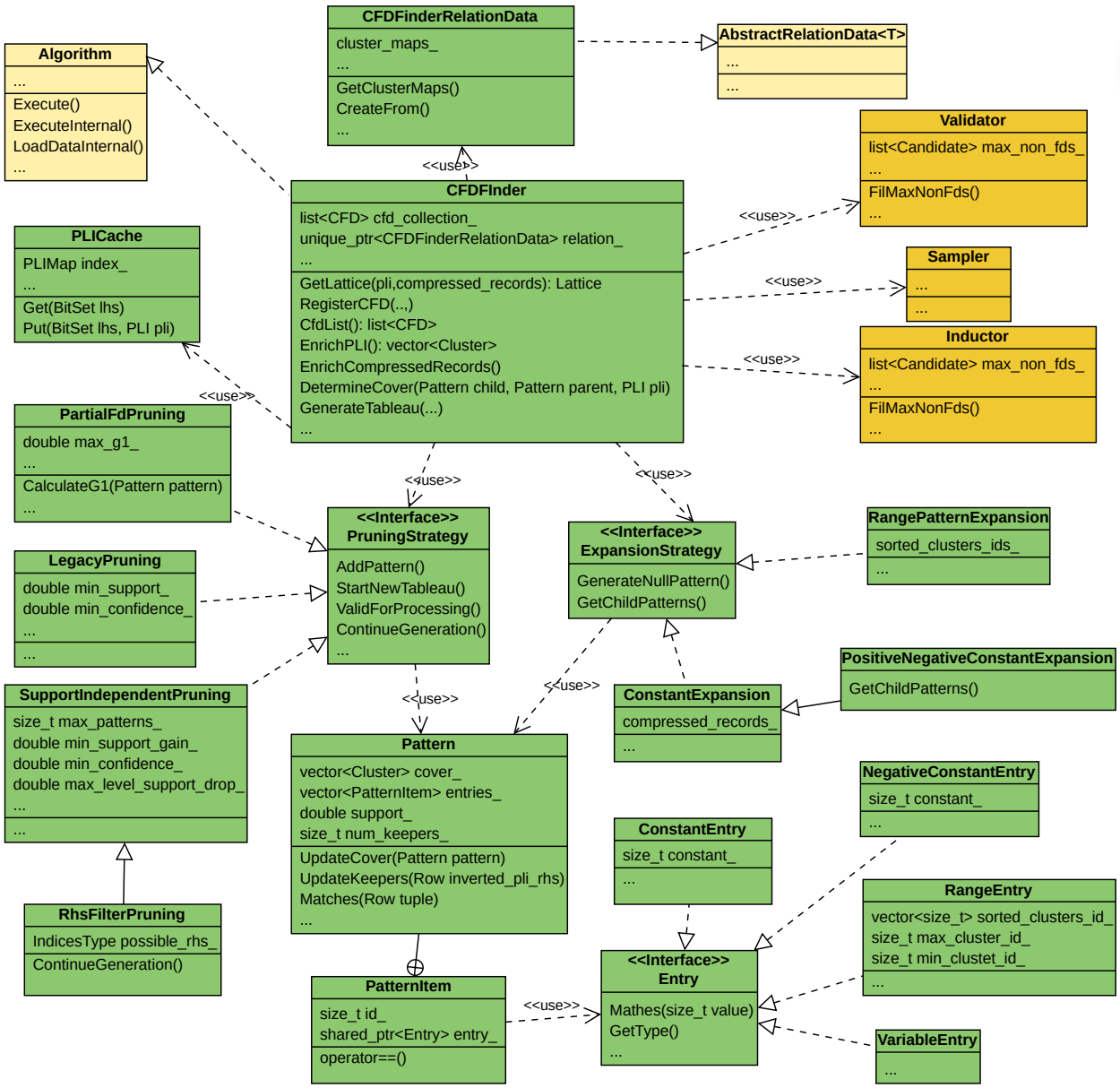

Рис. 2: Схема некоторых классов CFDFinder в Desbordante. Зеленым отмечены с нуля реализованные автором классы, темно-желтым — то, что уже имелось и переиспользовалось, светло-желтым — то, что уже имелось и не относится напрямую к алгоритму.

Класс `Algorithm` является базовым классом для всех алгоритмов в Desbordante, который предоставляет базовый интерфейс, необходимый для работы с примитивами. От него унаследован класс алгоритма

CFDFinder.

Для создания класса представления данных CFDFinderRelationData используется шаблонный класс AbstractRelationData, который в качестве параметра имеет класс представления данных в одной колонке.

Для чтения данных используются уже реализованные средства платформы Desbordante. Создание объекта класса представления, индексов списков позиций и индекса отображения кластеров происходит в методе LoadDataInternal.

Основная работа выполняется в методе ExecuteInternal. Сначала из класса представления извлекается объекты PLIs, которые затем инвертируются отдельную структуру, а также создаются сжатые данные. С помощью этих структур инициализируются объекты классов Validator, Sampler и Inductor, которые являются составными компонентами алгоритма HyFD. Так как алгоритм HyFD уже реализован в платформе, данные классы переиспользуются с тем изменением, что в классы Validator и Inductor добавлена логика сбора MaxNonFD и их извлечения.

Метод EnrichPLI восстанавливает одиночные кластеры, которые не вошли в объект PLI и используется для построения покрытия нулевого шаблона.

Решетка представлена как std::vector<Level>, где Level — std::unordered_set<Candidate>.

Также создается объект класса PLICache, который используется для сокращения количества пересечений PLI, необходимых при изучении нескольких кандидатов с похожими LHS.

После подготовки всех структур данных алгоритм инициализирует экземпляры классов стратегий отсечения и расширения в зависимости от выбранной конфигурации. Классы PruningStrategy и ExpansionStrategy предоставляют базовый интерфейс стратегий.

Генерация таблицы происходит в методе GenerateTableau. Создание объекта нулевого шаблона делегировано стратегии расширения. Для реализации границы шабло-

нов используются два синхронизированных индекса: `std::priority_queue<Pattern>` для сортировки по дополнительной поддержке и `std::unordered_set<std::vector<PatternItem>>` для быстрого поиска шаблона в границе.

В случае удачной генерации таблицы кандидат добавляется в коллекцию результатов в помощью метода `RegisterCFD`. Переменные шаблонов, которые представлены как значения из сжатых данных, при регистрации восстанавливаются в строковые значения с помощью `InvertedClusterMaps`.

После завершения работы алгоритма полученные результаты можно получить извне с помощью метода `CFDList`.

6. Эксперименты и сравнение с Metanome

6.1. Сравнение результатов

Чтобы проверить корректность реализации C++ необходимо сравнить результаты работы оригинального алгоритма CFDFinder и его интегрированной версии. С этой целью были написаны модульные тесты для интегрированного кода в различных конфигурациях на различных наборах данных.

В ходе эксперимента было выявлено, что реализованные версии алгоритма не являются эквивалентными. Граница, которая используется на этапе генерации таблицы, на обоих языках реализована как очередь с приоритетами. Различные с точки зрения привязанных констант шаблоны в границе могут иметь одинаковый приоритет на определенном этапе цикла, так как приоритет задается показателем поддержки, а в случае равенства — уверенностью. В силу того, что порядок получения элементов с одинаковым приоритетом из очереди не гарантирован, на одном шаге цикла перебора шаблонов разные версии алгоритма могут получить разные по привязкам шаблоны. Из-за того что при добавлении шаблона в таблицу, мы перестраиваем очередь заново, обновляя элементы покрытия оставшихся шаблонов относительно добавленного, конечные таблицы у двух кандидатов после завершения этапа генерации могут сильно отличаться.

Для обеспечения эквивалентности реализаций в оригинальный и адаптированный алгоритм было добавлено собственное сравнение на основе уникального порядкового номера создания шаблона в случае одинаковых показателей поддержки и уверенности шаблона, а также сортировка кластеров в покрытии, чтобы на этапе расширения шаблоны с одинаковыми привязками создавались с одинаковым номером. Внесенные изменения не включены в работу конечного алгоритма, а используются только для сравнения результатов и экспериментов.

В качестве ожидаемых значений были взяты результаты работы модифицированного алгоритма Metanome при тех же стратегиях и при

тех же параметрах.

Также стоит отметить, что при проведении экспериментов были обнаружены некоторые особенности Java-версии:

Проблема 1. При создании объекта стратегии расширения `RangePatternExpansionStrategy` для задания порядка извлекаются и сортируются элементы `InvertedClusterMap`, которые представлены как пара `<Integer, String>`. При сортировке по полю `String` не проверяется, что значения могут быть `null` (которые могут появиться, если в данных был `null`), из-за чего возможна ошибка времени выполнения.

Проблема 2. В конфигурации

- **Данные:** первые 4 записи mushroom[2]
- **Стратегия отсечения:** `LegacyPruning`
- **Минимальная поддержка:** 0.8
- **Минимальная уверенность:** 1
- **Стратегия расширения:** `ConstantPatternExpansionStrategy`

замечено неопределенное поведение при расширении шаблона с номером `"503338"`. После расширения этого шаблона в методе `validForProcessing` проверяется, что рассмотрены все родители шаблона с номером `"505930"`. Для этого используется метод `Java.util.Set.containsAll` для множества рассмотренных шаблонов, и он возвращает `false`, хотя по анализу дампа должен вернуть `true`. При переписывании на ручной обход и проверку в множестве рассмотренных шаблонов `validForProcessing` возвращает `true`. Возможно, причина такого поведения заключается в хэш-функции класса `Pattern`, так как после изменения способа хеширования всё работает правильно.

[2] https://www.kaggle.com/datasets/uciml/mushroom-classification?resource=download

6.2. Производительность

В таблице 1 приведены экспериментальные наборы данных. Большинство из них были взяты из работы [4].

Таблица 1: Экспериментальные наборы данных [4].

Набор	**Столбцы**	**Строки**	**Размер (КБ)**
abalone	9	4,177	187
chess	7	28,056	531
nursery	9	12,960	1024
Wisconsin Breast Cancer	11	699	20
Bridges	13	108	6
Echocardiogram	13	132	6

Характеристики системы: Intel(R) Core(TM) i7-12700H CPU @ 3.5GHz (400–4700GHz, 14 core 20 threads), 15 GiB RAM, Ubuntu 24.04.2 LTS, Kernel 6.14.0-32-generic, gcc (Ubuntu 13.3.0-6ubuntu2 24.04) 13.3.0. Metanome запускался на OpenJDK 21 с максимальным размером кучи 8 ГБ и остальными настройками по умолчанию. Desbordante компилировался с флагом оптимизаций -O3.

Для минимизации влияния сторонних факторов на производительность кеши ОС сбрасывались между каждым запуском с помощью записи “3” в /proc/sys/vm/drop_caches, на время экспериментов выключался файл подкачки с помощью команды swapoff -a, частота процессора фиксировалась и выставлялась в максимально возможное значение с помощью cpupower, все тяжеловесные процессы были отключены.

Для экспериментов была выбрана конфигурация с использованием стратегии отсечения `SupportGainPruning` по аналогии с экспериментами в [4]. Минимальный порог прироста поддержки устанавливался как 5% от количества строк набора, максимальный порог падения — 10%, максимальное количество шаблонов в таблице — 2000, минимальная поддержка — 1. Каждое измерение производилось 10 раз на эквивалентных версиях алгоритма, в таблице 2 приведено среднее значение в секундах. На всех наборах NULL-значения считались равными. Для вывода результатов в Metanome использовалась стратегия вывода

`FileStrategy`.

Таблица 2: Результаты экспериментов.

Набор данных	**C++**	**Java**	**Ускорение**
abalone	96,078 с	130,291 с	1,36x
chess	4,120 с	48,169 с	11,69x
nursery	10,188 с	102,215 с	10,03x
Wisconsin Breast Cancer	142,910 с	168,620 с	1,18x
Bridges	35,020 с	42,047 с	1,20x
Echocardiogram	17,482 с	53,935 с	3,08x

В случае наборов abalone, Bridges и Wisconsin Breast Cancer около 60–70% времени выполнения уходит на вычисление покрытия в `DetermineCover`, а именно на функцию `Matches` класса `Pattern`. Возможно, маленькая разница в производительности на этих наборах объясняется тем, что JIT компилятор Java хорошо справляется с оптимизацией горячих участков кода и девиртуализацией.

Также профилирование кода на этих наборах показало, что последовательная обработка кандидатов на уровне является узким местом, так как алгоритм до тех пор, пока не обработает всех кандидатов на уровне, не переходит к следующему, а суммарное время обработки кандидатов может быть очень велико, особенно на низких уровнях решетки.

Приведенные эксперименты являются предварительными. В дальнейшем планируется выполнить:

- Изучение производительности интегрированной версии без сортировки кластеров в покрытии;
- Сравнение двух версий алгоритмов по количеству потребляемой памяти;
- Модификацию алгоритма путем добавления параллельной обработки кандидатов в решетке, а затем изучение ее производительности.

7. Заключение

По итогам работы были достигнуты следующие результаты:

1. Алгоритм CFDFinder интегрирован в платформу Desbordante на языке C++. При реализации были переиспользованы общие части из HyFD;
2. При экспериментах обнаружены ошибка времени выполнения и неопределенное поведение в Java версии алгоритма;
3. С помощью профилирования найдены узкие места и предложена модификация алгоритма с использованием многопоточности;
4. Проведено сравнение производительности с реализацией в Metanome. Ускорение удалось добиться на всех экспериментальных наборах, в среднем версия Desbordante быстрее в 4,16 раз;

Ссылка на pull request: https://github.com/Desbordante/desbordante-core/pull/601.

Список литературы

[1] Conditional Functional Dependencies for Data Cleaning / Philip Bohannon, Wenfei Fan, Floris Geerts et al. // 2007 IEEE 23rd International Conference on Data Engineering. — 2007. — P. 746–755.

[2] 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.

[3] Desbordante: from benchmarking suite to high-performance science-intensive data profiler / George Chernishev, Michael Polyntsov, Anton Chizhov et al. // Proceedings of the 8th International Conference on Data Science and Management of Data (12th ACM IKDD CODS and 30th COMAD). — CODS-COMAD '24. — New York, NY, USA : Association for Computing Machinery, 2025. — P. 234–243. — URL: https://doi.org/10.1145/3703323.3703725.

[4] Grundke Maximilian. Discovering Interesting Conditional Functional Dependencies : Master's thesis / Maximilian Grundke ; Hasso-Plattner-Institute, University of Potsdam. — Potsdam, Germany, 2018.

[5] Ivan Kozhukov. Review of the CFDFinder conditional functional dependency search algorithm. — 2025. — URL: https://github.com/Desbordante/desbordante-core/pull/615/files/40eb532ee33a424ba3a13527db624e6376ed1a42.

[6] On generating near-optimal tableaux for conditional functional dependencies / Lukasz Golab, Howard Karloff, Flip Korn et al. // Proc. VLDB Endow. — 2008. — Aug. — Vol. 1, no. 1. — P. 376–390.

[7] 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.