ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»
Санкт-Петербургская школа физико-математических и компьютерных наук

Кафедра математических и информационных технологий

Абросимов Григорий Константинович

Изучение и реализация алгоритмов поиска условных зависимостей по включению

Бакалаврская работа

Допущена к защите.
Зав. кафедрой:
к. ф.-м. н., доцент Храбров А. И.

Научный руководитель:
Кандидат физико-математических наук, доцент, департамент информатики Мухин М. С.

Научный консультант:
Ассистент кафедры информационно-аналитических систем СПБГУ Чернышев Г. А.

Рецензент:
Разработчик ПО, ООО «Яндекс.Такси Технологии» Слободкин Е. С.

Санкт-Петербург
2025

Оглавление

Аннотация

В современном мире реляционные базы данных остаются одним из самых популярных методов хранения и обработки информации. Однако, с ежедневным ростом количества хранимых данных появляется необходимость в профилировании, или же извлечении метаинформации из данных. Это помогает как при проектировании новых баз, так и в поддержке и оптимизации запросов у старых. Одними из самых популярных зависимостей реляционных данных являются зависимости включения — они помогают искать внешние ключи в таблицах, а так же накладывать ограничения целостности , что повышает качество базы данных в целом. Не так давно были введены условные зависимости включения: такие зависимости работают только на тех данных, у которых некоторый набор атрибутов принимает определённые значения. С их помощью можно оптимизировать запросы в базу данных, искать повреждённые данные, а так же применять зависимости включения на определённом срезе данных. Данная работа посвящена исследованию условных зависимостей включения, а так же реализации алгоритмов их поиска в инструменте для профилирования данных Desbordante.

Ключевые слова: Реляционные БД, профилирование данных, зависимости данных, условные зависимости включения.

Abstract

In the modern world, relational databases remain one of the most popular methods for storing and processing information. However, with the daily growth in the amount of stored data, there arises a need for profiling, or extracting metadata from the data. This is helpful both in designing new databases and in maintaining and optimizing queries in existing ones. One of the most popular dependencies in relational data is inclusion dependencies—they assist in finding foreign keys in tables and imposing integrity constraints, which enhances the overall quality of the database. Recently, conditional inclusion dependencies have been introduced: these dependencies operate only on those data where a certain set of attributes takes specific values. They can be used to optimize queries in the database, search for corrupted data, and apply inclusion dependencies to a specific subset of data. This work is dedicated to the study of conditional inclusion dependencies and the implementation of algorithms for their search in the data profiling tool Desbordante.

Keywords: Relational Databases, Data Profiling, Data Dependencies, Conditional Inclusion Dependencies.

Введение

Несмотря на то, что в данный момент большую популярность имеют различные нереляционные базы данных, реляционные базы не теряют своей актуальности. Они удобны за счёт своей структуры — все данные хранятся в таблицах и состоят из атрибутов, на эти атрибуты могут быть наложены ограничения, а сами таблицы в свою очередь имеют множество взаимосвязей, которые описаны в схеме реляционной базе данных. Эти свойства позволяют удобно и быстро выполнять различные операции над данными как в одной, так и в нескольких таблицах.

Однако, если данных оказывается слишком много, то при работе с ними могут возникнуть проблемы — операция может выполняться слишком долго, а ошибочные записи могут встретиться с большей вероятностью. Чтобы упростить работу с большим объёмом реляционных данных, может оказаться полезным профилирование данных [1] — процесс анализа данных с целью извлечения закономерностей в них. Эти закономерности можно описать с помощью примитивов [11] — формально описанных наборов правил, которым подчиняются определённые части данных. Одними из самых популярных примитивов считаются зависимости включения, о которых пойдёт речь в данной работе.

Зависимость включения (INclusion Dependency, IND) [12] — неформально это такое отношение между двумя таблицами, при котором множество значений атрибута первой таблицы является подмножеством значений атрибута второй таблицы. Такие зависимости позволяют находить внешние ключи, а так же поддерживать ссылочную целостность [10]. Поиск таких зависимостей является NP-трудной задачей [7], поэтому искать их с помощью SQL запросов нерационально. Для решения данной задачи существует несколько алгоритмов [9, 12, 14, 15], которые работают намного быстрее запросов к базе данных.

Иногда такие зависимости могут выполняться не на всех данных, а только лишь на их подмножестве — в таком случае имеют место быть приближённые зависимости включения (Approximate INclusion Dependency, AIND) [13] — это такие зависимости, которые выполняются с некоторой ошибкой ε. Однако, такие зависимости дают ограниченную информацию о той части данных, на которой выполняется IND. Поэтому, было введено понятие условных зависимостей включения (Conditional INclusion Dependency, CIND) [6] — они задаются с помощью AIND, а так же набора правил, которые применяются ко всем оставшимся атрибутам первой таблицы. Эти правила задают область действия встроенной IND. С помощью CIND можно решать такие задачи, как:

- Выявление некачественных данных, в частности проблема пропавшей ссылочной целостности, описанная подробно в статье [6];

- Оптимизация запросов к реляционной СУБД, в частности оператора join;
- Анализ данных на основе полученных условий, выявление закономерностей;
- Использование `IND` на определённом срезе данных, заданным подмножеством условий CIND.

Алгоритмы поиска зависимостей включения реализованы [11] в различных популярных инструментах профилирования данных. Одним из таких инструментов является Desbordante[1] — это наукоёмкий инструмент профилирования данных с открытым исходным кодом [5]. Реализован Desbordante на языке программирования C++, благодаря чему реализации алгоритмов отличаются высокой производительностью. Вместе с тем, он предоставляет различные интерфейсы взаимодействия, а так же подробные примеры использования алгоритмов, что делает применение этих алгоритмов лёгким на практике. Одной из целей Desbordante является предоставление широкого набора алгоритмов для решения прикладных задач, и поэтому проект активно развивается и нуждается в расширении набора поддерживаемых примитивов.

Алгоритм поиска CIND [6] появился более 10 лет назад, однако его реализация в инструментах профилирования данных на данный момент отсутствует, он существует лишь в виде исходного кода[2]. В связи с этим его применение на практике представляет определённые трудности. Это послужило мотивацией данной работе, которая посвящена разработке промышленной реализации алгоритма поиска CIND и его интеграция в Desbordante.

[1]https://github.com/Desbordante/desbordante-core
[2]https://hpi.de/naumann/projects/repeatability/data-profiling/metanome-ind-algorithms.html

1. Цель и задачи

Данная работа ставит своей целью реализовать алгоритмы поиска CIND, а так же интегрировать его в инструмент профилировки данных Desbordante. В результате должен получиться алгоритм, который будет пригоден для промышленного использования.

Для достижения этой цели поставлены следующие задачи:

- Изучить различные виды зависимостей включения: IND, AIND, CIND, а так же алгоритмы для их поиска. Изучить существующие инструменты профилирования данных. Составить обзор предметной области;
- Реализовать алгоритмы поиска условий для CIND;
- Интегрировать полученные алгоритмы в Desbordante, реализовать пользовательский интерфейс для языка Python;
- Изучить производительность полученных алгоритмов, сравнить реализации по времени работы и потребляемой памяти.

Структура работы

В первой главе представлен обзор предметной области, включающий в себя обзор зависимостей включения и условных зависимостей включения в частности. Здесь же приведён обзор алгоритмов поиска IND и условий для CIND. Во второй главе описаны сами алгоритмы, а так же их реализация, включая примеры использования на языке Python. В третьей главе описаны эксперименты, в рамках которых приводится сравнение алгоритмов между собой на различных наборах данных с различными параметрами. На основе результатов подведены итоги.

2. Обзор предметной области

2.1. Обзор существующих инструментов

В настоящее время алгоритмы поиска зависимостей включения поддерживаются в двух инструментах для профилирования данных [15] — это Metanome [3] и Desbordante [5].

2.1.1. Metanome

Metanome[3] — инструмент для профилирования данных с открытым исходным кодом, написанный на языке Java. Он является библиотекой алгоритмов, которые нацелены на поиск и валидацию различных примитивов, в том числе зависимостей включения. Чтобы сделать запуск алгоритмов удобным, в Metanome существует свой графический интерфейс, где можно не только запустить алгоритм, но и посмотреть на визуализацию результатов.

Разрабатывает инструмент группа учёных из института Hasso-Plattner-Institut, который расположен в немецком городе Потсдам, и, зачастую, именно они являются авторами алгоритмов, которые в нём представлены. В частности, они ввели такой примитив, как CIND [6] и разработали алгоритмы их поиска. Тем не менее, данные алгоритмы не были интегрированы в Metanome и в данный момент инструмент не поддерживает данный примитив.

2.1.2. Desbordante

Desbordante[4] — новый инструмент для профилирования данных с открытым исходным кодом, написанный на языке C++. Его разрабатывает группа студентов из Санкт-Петербургского Государственного Университета с 2019 года. Назначение у инструмента во многом совпадает с Metanome, но у него есть ряд отличительных особенностей:

1. Производительность. Поскольку Desbordante реализован на языке C++, он предлагает пользователям более быстрые версии алгоритмов. Это достигается не только за счёт того, что язык программирования C++ работает быстрее Java, но и за счёт низкоуровневых оптимизаций, которые на Java реализовать является проблематичным. В частности, используются SIMD расширения процессора для векторизации вычислений. В результате, Desbordante нередко оказывается более эффективным решением, чем Metanome [4].

[3] hpi.de/naumann/projects/data-profiling-and-analytics/metanome-data-profiling.html
[4] https://github.com/Desbordante/desbordante-core

2. Доступность. Metanome имеет сложную модульную структуру, и его установка может вызвать затруднения. Desbordante доступен пользователям в качестве pip пакета для языка Python, который можно установить одной командой. Также, он имеет CLI интерфейс для взаимодействия на случай, если Python окажется непригодным для выполнения задачи. Кроме того, он имеет онлайн демо версию[5], которая не требует установки и имеет схожий функционал с графическим интерфейсом Metanome.

3. Библиотека. Несмотря на то, что Desbordante более молодой проект и имеет меньше алгоритмов, чем Metanome, в нём реализовано большее количество примитивов, что делает его пригодным для выполнения большего числа задач.

Инструмент	FD	CFD	IND	UCC	OD	FAC	DD	AR
SAP IS	+	-	-	ч	-	-	-	-
Informatica DQ	+	-	+	+	-	-	-	-
IBM InfoSphere IS	-	-	+	+	-	-	-	+
Experian Pandora	+	-	-	+	-	-	-	-
DataCleaner	-	-	ч	ч	-	-	-	-
Ataccama ONE	+	-	+	+	-	-	-	-
Oracle WB	ч	-	-	-	-	-	-	-
Talend	+	-	+	+	-	-	-	-
HoloClean	+	+	-	-	-	-	-	-
Uni-Detect	+	-	-	-	-	-	-	-
Metanome	+	+	+	+	+	-	-	-
Desbordante	+	+	+	+	+	+	+	+

Рис. 1: Сравнение инструментов профилирования данных по набору поддерживаемых примитивов. Источник [11]

.

Вывод

Desbordante имеет ряд преимуществ перед Metanome, но имеет библиотеку алгоритмов гораздо меньшего размера. На данный момент он находится в стадии активной разработки и ставит перед собой задачу расширять список поддерживаемых примитивов и алгоритмов. В связи с этим, было принято решение реализовывать алгоритм поиска CIND в рамках данного инструмента.

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

2.2. Обзор условных зависимостей включения

Источником для данного раздела служит статья [6], описывающая CIND и алгоритмы поиска условий к ним.

2.2.1. Определение CIND

Для начала введём несколько формальных определений, связанных с областью зависимостей включения. Пусть R_1, R_2 — реляционные схемы над фиксированным набором атрибутов $A_1, A_2, \ldots, A_k$. Множество значений каждого атрибута обозначим как $dom(A)$. Пусть I_1, I_2 — экземпляры R_1, R_2 соответственно. Каждый экземпляр I состоит из набора кортежей t таких, что $t[A] \in dom(A)$ для каждого атрибута $A \in R$. Пусть X, X_P и Y, Y_P — множества атрибутов R_1, R_2 соответственно. Обозначим за $t[X]$ проекцию кортежа t на атрибуты X.

Замечание: если применять определения на практике, то можно считать, что I_1, I_2 — таблицы в реляционной БД, а t — строка таблицы.

Определение 1 (Зависимость включения)**.** *Зависимостью включения (INclusion Dependency,* IND*) называется отношение $R_1[X] \subseteq R_2[Y]$, которое выполняется на всей таблице. То есть, $\forall t_1 \in R_1\ \exists t_2 \in R_2 : t_1[X] = t_2[Y]$.*

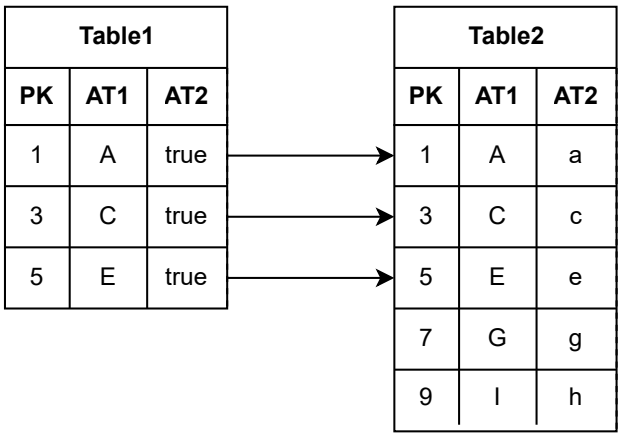

Рис. 2: *Пример IND: Table1.PK → Table2.PK*

Здесь мы говорим, что кортеж $t_1 \in R_1$ удовлетворяет IND, если существует ссылочный кортеж $t_2 \in R_2$, для которого верно $t_1[X] = t_2[Y]$. Атрибуты X и Y называются атрибутами включения.

Определение 2 (Приближённая зависимость включения)**.** *Приближённая зависимость включения (Approximate INclusion Dependency,* AIND*) — это такая* IND*, которая выполняется на непустом множестве кортежей из R_1. То есть, $\exists t_1 \in R_1, t_2 \in R_2 : t_1[X] = t_2[Y]$. Обозначается как $R_1[X] \subseteq' R_2[Y]$.*

Приближённая зависимость включения обладает своей метрикой ошибки, которая определяет долю кортежей, на которых зависимость не выполняется.

Определение 3 (Ошибка AIND)**.** *Ошибкой* AIND *называют число* $\varepsilon \in [0, 1]$*, которое показывает долю кортежей, на которых соответствующая* IND *не выполняется. Вычисляется по формуле:*

$$\varepsilon = 1 - \frac{|I_1[R_1[X] \subseteq' R_2[Y]]|}{|I_1|},$$

где $I_1[R_1[X] \subseteq' R_2[Y]]$ *— множество всех кортежей, для которых выполняется зависимость* $R_1[X] \subseteq R_2[Y]$*.*

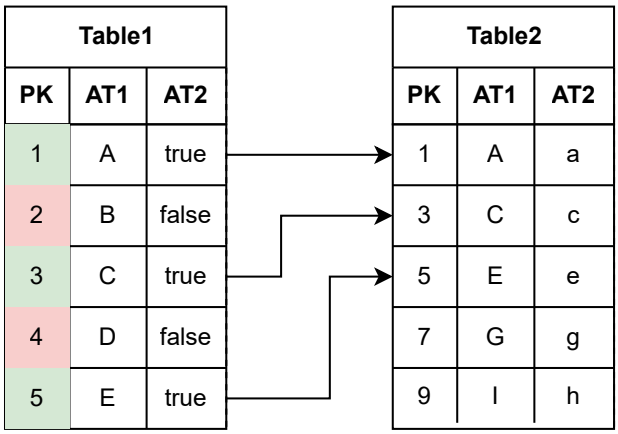

Table1		
PK	AT1	AT2
1	A	true
2	B	false
3	C	true
4	D	false
5	E	true

Table2		
PK	AT1	AT2
1	A	a
3	C	c
5	E	e
7	G	g
9	I	h

Рис. 3: *Пример AIND: Table1.PK* $\rightarrow$ *Table2.PK* с метрикой ошибки $\varepsilon = 0.4$

Определение 4 (Таблица условий)**.** *Таблица условий (Pattern Tableau)* T_P *состоит из кортежей над атрибутами* X_P *и* Y_P*. Для каждого такого кортежа* t_P *верно следующее:* $\forall A \in X_P \cup Y_P : t[A] = (a \in A \vee \text{‘—’})$*, где ‘—’ это специальное значение. Такие кортежи называются шаблонами, или условиями.*

Говорят, что кортеж $t_1 \in R_1$ удовлетворяет условию t_P, если $\forall A \in X_P : t_1[A] = t_P[A] \vee t_P[A] = \text{‘—’}$. Атрибуты X_P, Y_P называются атрибутами условия. Важно, что множества X, X_P и Y, Y_P не пересекаются.

Определение 5 (Условная зависимость включения)**.** *Условной зависимостью включения (Conditional INclusion Dependency,* CIND*) называется отношение вида:*

$$\varphi : (R_1[X; X_P] \subseteq R_2[Y; Y_P], T_P).$$

Оно состоит из IND $R_1[X] \subseteq R_2[Y]$ *и таблицы условий* T_P*, которая определяет подмножество кортежей, на которых выполняется эта* IND*.*

φ выполняется для экземпляров I_1, I_2, если выполнены следующие условия:

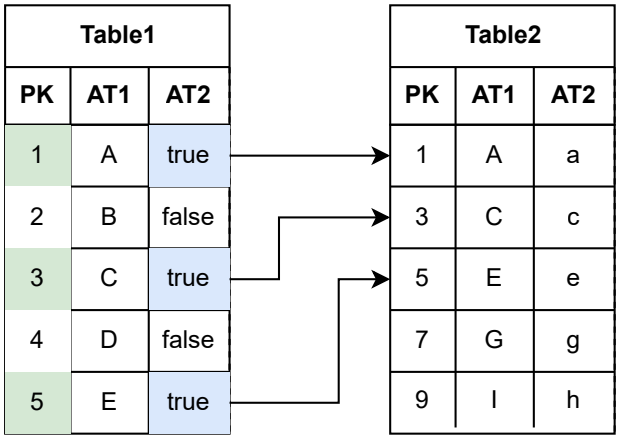

Рис. 4: *Пример CIND: Table1.PK* $\rightarrow$ *Table2.PK* с таблицей условий $T_P = \{(Table1.AT1 = \text{‘—’},\ \ Table1.AT2 = true)\}$

1. Условие выбора для I_1. Пусть $t_1 \in I_1$ удовлетворяет какому-либо условию $t_P \in T_P$. Тогда, t_1 должна удовлетворять и встроенной IND.

2. Условие требования для I_2. Пусть Пусть $t_1 \in I_1$ удовлетворяет какому-либо условию $t_P \in T_P$. Более того, t_1 удовлетворяет встроенной IND, то есть $\exists t_2 \in I_2 : t_1[X] = t_2[Y]$. Тогда t_2 так же должна удовлетворять t_P.

Заметим, что условие выбора для I_1 включает в себя условие требования для I_2. Более того, шаблон $t_P \mid \forall A \in Y_P : t_2[A] = \text{‘—’}$ является полностью валидным, то есть выполняется всегда, когда выполняется условие выбора. Поэтому, в дальнейшем, при поиске CIND будем учитывать только условие выбора.

Метрики условий для CIND

Теперь формально определим метрики качества условий: валидность (validity) и полноту (completeness). Пусть дана CIND ϕ и экземпляры I_1, I_2, которые ей удовлетворяют. Обозначим за I_φ множество кортежей из I_1, которые удовлетворяют встроенной IND. Так же, для ненормализованных таблиц нам интересны группы включённых кортежей, которые имеют одинаковые значения на атрибутах X. Пусть g_x — это группа кортежей из I_1, которая равна x на атрибутах зависимости. То есть: $g_x = \{t | t \in I_1 \wedge t[X] = x\}$. Такая группа называется включённой, если хотя бы один кортеж из неё является включённым, т. е. удовлетворяет IND. Аналогично, группа удовлетворяет условию, если хотя бы один кортеж из неё удовлетворяет условию. Тогда, пусть G_1 — множество групп I_1, а G_φ — множество включённых групп. И, наконец, обозначим за $I[t_P]$ множество всех кортежей из I, которые удовлетворяют условию t_P, а за $G[t_P]$ — множество всех таких групп.

Замечание: в дальнейшем, условия, метрики которых посчитаны для групп, будут называться ***групповыми*** *условиями. В свою очередь те условия, метрики ко-*

торых посчитаны для обычного представления таблицы, будут называться ***строковыми*** *условиями.*

Определение 6 (Валидное условие)**.** *Условие называется валидным, если все кортежи (группы), которые ему удовлетворяют, являются включёнными. Является аналогом точности (precision) классификатора из теории машинного обучения.*

Определение 7 (Полное условие)**.** *Условие называется полным, если ему удовлетворяют все включённые кортежи (группы). Для групп так же может обозначаться покрытием (covering). Является аналогом полноты (recall) классификатора из теории машинного обучения.*

В реальности получить полностью валидные или полные условия практически невозможно. Поэтому, определим метрики валидности и полноты как метрики, которые возможно посчитать.

Определение 8 (Валидность условия)**.** *Валидность условия* $valid(t_P) \in [0, 1]$ *вычисляется по формуле*

$$valid(t_P) = \frac{|I_\varphi[t_P]|}{|I_1[t_P]|}$$

для строковых условий, и по формуле

$$valid_g(t_P) = \frac{|G_\varphi[t_P]|}{|G_1[t_P]|}$$

для групповых. Условие t_P *называется* γ*-валидным, если его валидность больше* γ*.*

Определение 9 (Полнота условия)**.** *Полнота условия* $completeness(t_P) \in [0, 1]$ *вычисляется по формуле*

$$completeness(t_P) = \frac{|I_\varphi[t_P]|}{|I_\varphi|}$$

для строковых условий, и по формуле

$$completeness_g(t_P) = \frac{|G_\varphi[t_P]|}{|G_\varphi|}$$

для групповых. Условие t_P *называется* δ*-полным, если его полнота больше* δ*.*

Используя эти метрики, можно задать ограничения на найденные условия, и фильтровать как те, которые охватывают слишком значимое число невалидных кортежей, так и те, которые применяются на черезчур маленькой выборке данных.

Выводы

Подводя итоги этой секции, можно выделить 2 пункта, на которые стоит обратить внимание:

- CIND — составной примитив и состоит из 2-х частей: AIND и таблицы условий. Это значит, что при поиске CIND необходимо сначала искать AIND, а затем для каждой AIND отдельно искать интересующие нас условия. Поскольку алгоритмы поиска AIND уже описаны [9, 10] и реализованы в Desbordante, задача состоит лишь в реализации алгоритмов поиска условий. Так же, стоит задача в реализации алгоритма-фасада, который последовательно вызывает поиск AIND и поиск условий для них не обременяя этой задачей конечных пользователей.

- Условия обладают своими характеристиками качества — валидностью и полнотой. Поскольку полностью валидных и полных условий в таблице может не существовать, нас могут заинтересовать и такие условия, у которых валидность больше какого-то порогового значения γ, а полнота — выше пороговой δ. Следовательно, алгоритмы должны поддерживать эти пороговые значения как параметры. Так же, следует учесть, что данные метрики могут считаться для двух представлений таблицы: обычного и сгруппированного по значениям атрибутов включения. Соответственно, тип представления так же должен быть отражён в параметрах алгоритма.

2.3. Обзор алгоритмов

2.3.1. Алгоритмы поиска AIND

В настоящее время, в Desbordante реализовано 3 алгоритма для поиска `IND`: SPIDER [9], Faida [10] и MIND [14]. Реализация Faida не поддерживает поиск AIND, и потому не подходит под текущую задачу. MIND алгоритм позволяет искать AIND, но в Desbordante он был реализован сравнительно недавно, и не проводились эксперименты, доказывающие его эффективность в рамках данной реализации. Поэтому был выбран SPIDER — он является эффективным алгоритмом как для поиска IND, так и для поиска AIND.

2.3.2. Структура алгоритма поиска CIND

CIND впервые были представлены в 2012 году в статье [6]. Здесь приведено 2 различных алгоритма поиска условий — это CINDERELLA и PLI-based Algorithm (далее — PLI-CIND), которые основаны на разных принципах и имеют свои преимущества и недостатки. Оба алгоритма основаны на идее внешнего левого соединения

двух таблиц. Ключевое различие алгоритмов заключается в методе поиска кандидатов в условия (то есть корректных кортежей, которые могут выступать в качестве условий, но с неизвестными метриками).

С целью упрощения дальнейшего разбора алгоритмов, каждый алгоритм изначально описывается как версия для поиска групповых условий, а затем рассказывается как алгоритм следует изменить для поиска строковых условий.

2.3.3. Алгоритм CINDERELLA

Алгоритм Conditional INclusion DEpendency REcognition Leveraging deLimited Apriori (сокращённо — CINDERELLA) основан на поиске ассоциативных правил. Неформально, ассоциативные правила имеют такой смысл: «Кто покупает X и Y, так же зачастую покупает Z». В CINDERELLA эти правила имеют следующую интерпретацию: «Строки таблицы, чьи значения атрибутов равны кортежу x, так же обычно являются включёнными». Для поиска ассоциативных правил используется алгоритм Apriori [2], который представляет таблицу в виде контейнеров (ориг. basket): в такие контейнеры кладутся все пары вида (conditional_attribute:value), которые соответствуют одной группе или строке (в зависимости от типа условия), а так же индикатор включения данного контейнера.

Apriori алгоритм использует метрики support и confidence для сужения области поиска и вывода нужных правил. В нашем случае этими метриками выступают валидность и полнота. Область поиска можно сократить по метрике полноты пользуясь следующим инвариантом: с ростом количества атрибутов, для которых определено конкретное значение метрика полноты условия не возрастает. Тогда, мы получим все наборы кандидатов в условия, которые имеют необходимую полноту. После фильтрации по валидности, мы получим полный список «хороших» условий.

Алгоритм CINDERELLA основан на одной из вариаций Apriori — алгоритме MultipleJoins [16]. Отличается он тем, что использует следующую оптимизацию: нас интересуют только те правила, у которых атрибут включения имеет положительное значение (другими словами, нас интересуют только условия для включённых групп). Все остальные элементы правила формируют нужное нам условие. Благодаря этому, CINDERELLA при поиске кандидатов может использовать один join вместо трёх, как в MultipleJoins, за счёт чего сильно сужается область поиска кандидатов и повышается производительность.

Чтобы описать поведение алгоритма, введём следующие понятия: пусть L_k — множество всех часто встречающихся наборов элементов длины k, то есть множество кандидатов в условия длины k, которые имеют нужную нам полноту (при этом они могут иметь недостаточную валидность). Затем, обозначим за C_k — множество кандидатов в L_k.

Для начала, получим L_1 за один проход по всем контейнерам. Затем, формируем L_2, добавляя индикатор включения в начало каждого набора элементов. Затем в цикле ищем L_k. Сначала формируем C_k из L_{k-1} используя следующий приём: кандидат формируется из двух наборов, в которых первые $k-2$ элемента одинаковые, а последний элемент первого набора меньше последнего элемента второго. Затем, к первому набору в конец добавляется последний элемент второго. Таким образом, мы поддерживаем отсортированный порядок элементов в наборе. Затем нужно определить, действительно ли получившийся набор элементов является правильным: может оказаться так, что ни одна строка получившемуся условию не соответствует. Для этого пробегаемся по всем подмножествам длины $k-1$, которые содержат индикатор включения и проверяем, что такие подмножества лежат в L_{k-1}. У получившихся кандидатов заново считаем полноту, и фильтруя по ней, получаем L_k.

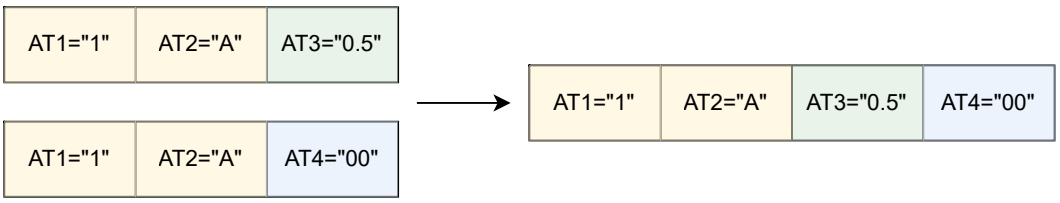

Рис. 5: *Визуализация отбора кандидатов алгоритмом CINDERELLA*

Получив все возможные наборы элементов с нужной полнотой, остаётся лишь выбрать из них те, что имеют нужную валидность. Это можно делать в процессе вычисления полноты каждого набора элементов. А чтобы применять такой алгоритм для поиска строковых условий, нужно формировать по контейнеру на каждую отдельную строку таблицы.

2.3.4. Алгоритм PLI-CIND

Второй алгоритм основан на идее пересечения позиционных списков атрибутов (Positions Lists Intersection, PLI), поэтому он называется PLI-based Algorithm или же PLI-CIND. Позиционный список атрибута — это преобразованное представление атрибута таблицы, в котором каждому значению атрибута A в строке t сопоставляется набор индексов в таблице. Этими индексами могут выступать как индексы строк для поиска строковых условий, так и индексы групп для поиска групповых условий. Эти позиционные списки можно объединять пересекая их между собой, формируя позиционные списки значений комбинации атрибутов. Данное представление таблицы используется несколькими алгоритмами, например TANE [17] и DynFD [8], которые ищут функциональные зависимости.

PLI-CIND, в отличие от CINDERELLA не пытается на каждой итерации искать все условия одинаковой длины. Напротив, он пытается расширить текущее условие до тех пор, пока это возможно. Если говорить, что CINDERELLA ищет кандидатов

поиском «в ширину», то здесь применяется подход поиска «в глубину». Алгоритм разбит на 2 фазы: сначала мы ищем позиционные списки для каждого отдельного атрибута. Затем, рекурсивно пытаемся расширить каждый список, пересекая его со списком для атрибута, чей порядковый номер больше.

Первая фаза алгоритма делается за один проход по таблице: нам нужно извлечь информацию по каждому столбцу ровно один раз. Нужно учитывать, что нас интересуют только те индексы, которые соответствуют включённым группам, поэтому в каждый список заносим только эти индексы. Здесь же проверяем каждое значение атрибута на полноту, как если бы это значение формировало условие, которое мы ищем. В итоге оставляем только те списки, которые имеют нужную нам полноту.

Вторая фаза состоит из рекурсивного поиска условий. Рекурсию запускаем из каждого атрибута отдельно. Затем, пока текущий позиционный список не пуст, пытаемся расширить его добавлением нового атрибута. Делается это пересечением текущего списка со списком позиций этого атрибута. В процессе пересчитываем полноту и валидность, полные списки отправляются по рекурсии дальше, а валидные — заносятся в ответ.

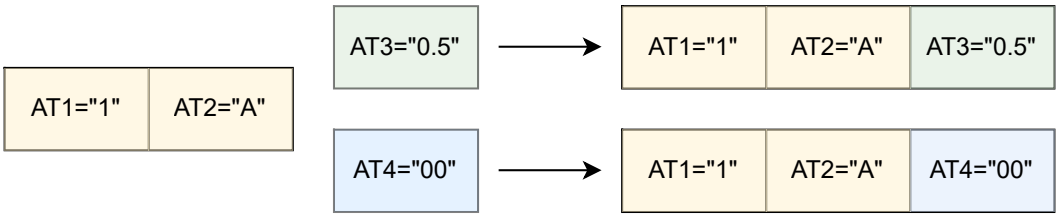

Рис. 6: *Визуализация отбора кандидатов алгоритмом PLI-CIND*

Реализации для различных типов условий различаются лишь типом индексов, которые используются на первой фазе при формировании позиционных списков.

Выводы

Учитывая вышесказанное, можно сказать, что объективно лучшего алгоритма не существует. CINDERELLA ищет кандидатов «в ширину», что позволяет существенно снизить область поиска условий. Вместе с тем, этот алгоритм должен хранить намного больший объём данных в процессе работы: для проверки правильности набора необходимо хранить кандидатов всех размеров в виде хэш-таблицы, что хоть и позволяет достичь высокой скорости работы, очень сильно повышает потребление памяти. PLI-CIND, напротив, потребляет память только на поддержание текущего кандидата, что делает совокупное потребление памяти не зависящим от числа кандидатов. Однако, поиск «в глубину» никак не ограничивает область поиска для расширения кандидата, за счёт чего алгоритм не так эффективно по времени это делает. В дальнейшем эта разница будет показана в экспериментах.

3. Реализация алгоритмов

Каждая секция данного раздела посвящена отдельной задаче, которая возникала в ходе реализации. Схема ниже показывает реализованные по итогу классы и их взаимосвязи. Зелёным обозначено то, что реализовано в рамках данной работы, а синим — то, что уже было реализовано в Desbordante

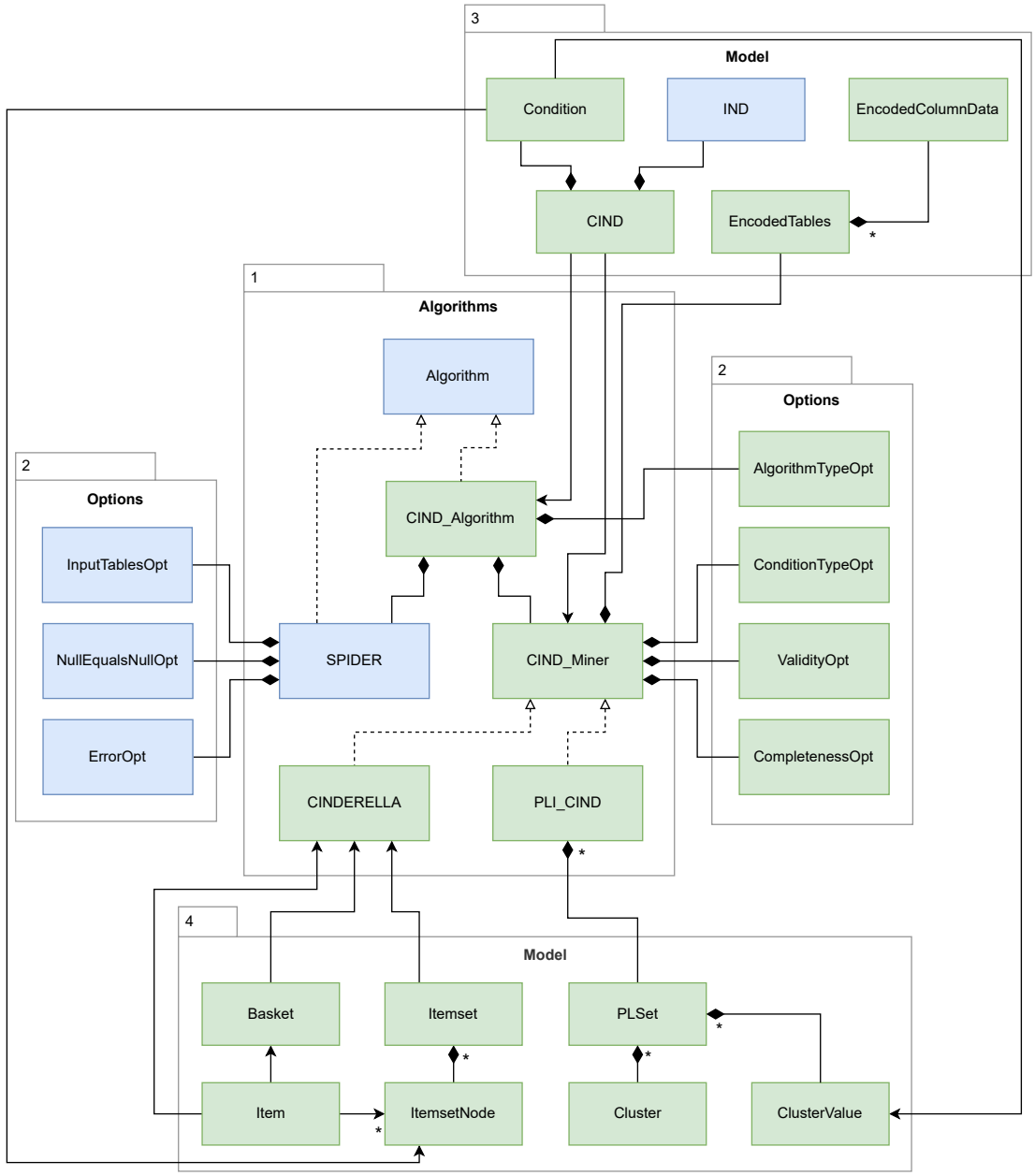

Рис. 7: *Схема иерархии классов алгоритмов поиска CIND.*

Последующие разделы будут затрагивать отдельные пакеты, выделенные на схеме, нумерация разделов совпадает с нумерацией пакетов.

3.1. Реализованные алгоритмы

Для того, чтобы в будущем не переписывать готовые алгоритмы, пытаясь интегрировать их в DESBORDANTE, было принято решение реализовывать алгоритмы сразу внутри этой платформы. В DESBORDANTE все алгоритмы, с которыми может взаимодействовать внешний пользователь, наследуются от абстрактного класса `Algorithm`. Этот класс предоставляет интерфейс, который является единым для всех алгоритмов поиска примитивов. В частности, в нём происходит разделение этапов подготовки данных (метод `LoadData`) и запуска алгоритма на них (метод `Execute`). Так же, в нём реализована общая функциональность: взаимодействие с индикатором прогресса, управление стадиями выполнения алгоритма, управление пользовательскими параметрами. По этой причине класс `CIND_Algorithm`, являющийся алгоритмом поиска CIND был унаследован от него.

`CIND_Miner` — это абстрактный класс семейства алгоритмов поиска условий для CIND. Поскольку не предполагается взаимодействие с ним пользователя напрямую, он не наследуется от `Algorithm` и имеет свой интерфейс взаимодействия. Это сделано для более удобной интеграции выбора алгоритма поиска условий, который задаётся пользовательским параметром и происходит на этапе подготовки данных. Во время каждого запуска алгоритма, он принимает три параметра от пользователя: тип представления таблицы, на которых считаются метрики условия, а так же пороговые значения этих метрик.

В качестве алгоритма поиска AIND был выбран `SPIDER`, поскольку он является одним из самых быстрых алгоритмов, позволяющим искать AIND, который при этом простой в интеграции. Во время этапа подготовки данных ему на вход в качестве параметров подаются входные данные вместе с индикатором наличия NULL значений в таблице. Так же, чтобы искать AIND, ему на вход передаётся параметр, который задаёт минимальную долю данных, которые должны покрывать найденные AIND.

Внутри `CIND_Algorithm` имеет пайплайн из двух алгоритмов: `SPIDER` и `CIND_Miner`. `SPIDER` возвращает список всех AIND, которые удалось найти, а `CIND_Miner` для каждого из них по отдельности ищет набор условий, создавая список CIND. При этом, во время этапа подготовки данных он извлекает список таблиц из `SPIDER` и передаёт их `CIND_Miner` для дальнейшей обработки.

3.2. Пользовательские параметры

Параметры метода `LoadData`

Эти параметры задаются один раз для каждого набора данных, на этапе их подготовки.

- `AlgorithmTypeOpt` — тип используемого алгоритма для поиска условий, прини-

мает значения `"cinderella"` и `"pli_cind"`. Значение хранится в `CIND_Algorithm`;

- `InputTablesOpt` — список названий таблиц, на которых хотим запустить алгоритм. Во время инициализации загружает таблицы в память и приводит их к готовому для работы состоянию. Значение хранится в `SPIDER`;
- `NullEqNullOpt` — Определяет, является ли NULL валидным значением какого-либо атрибута в таблице. Значение хранится в `SPIDER`.

Параметры метода `Execute`

Эти параметры задаются каждый раз для каждого запуска алгоритма.

- `ErrorOpt` — порог ошибки для AIND, значение хранится в `SPIDER`;
- `ConditionTypeOpt` — тип условий которые будут найдены алгоритмом, принимает значения `"row"` и `"group"`. Значение хранится в `CIND_Miner`;
- `ValidityOpt` — порог валидности условий, значение хранится в `CIND_Miner`;
- `CompletenessOpt` — порог полноты условий, значение хранится в `CIND_Miner`.

3.3. Хранение таблицы и примитивов

Обычно, в Desbordante таблицы хранятся в виде итератора, идущего по строкам. Это может быть удобно для большинства алгоритмов, но в данном случае, алгоритмы CINDERELLA и PLI-CIND работают с отдельными атрибутами таблицы. Поэтому есть необходимость преобразовать строковое представление таблицы в колоночное. Конечно, в Desbordante существуют несколько структур данных, которые работают с колоночным представлением таблицы, но они не поддерживают необходимый для CINDERELLA и PLI-CIND функционал — например, список уникальных значений в каждом столбце, или же $dom(A)$. Более того, в этих структурах данных значения атрибутов представлены в строковом виде, что значительно увеличивает потребление памяти как при хранении самой таблицы, так и при работе алгоритмов. Это послужило причиной реализации классов `EncodedColumnData` и `EncodedTables`.

`EncodedColumnData` — класс для хранения атрибута таблицы в преобразованном виде. Внутри все реальные значения сопоставлены с уникальным целочисленным порядковым номером. Это позволяет значительно экономить память при работе с таблицами, у которых атрибуты принимают большие значения в строковом представлении или отдельные значения встречаются слишком часто. Более того, работа с целочисленными индексами значений выполняется существенно быстрее, чем со строками, что так же даёт бонус к производительности алгоритмов. Этап преобразования происходит во время исполнения `LoadData`, в то время как извлечение исходного значения

атрибута происходит при добавлении условия в конечный ответ, что означает, что без необходимости значения арибутов не извлекаются.

Класс `EncodedTables` хранит внутри себя набор всех атрибутов всех таблиц, что удобно в случае, когда необходимо работать с атрибутами сразу нескольких таблиц — ведь они лежат в едином контейнере. Это применяется как минимум во время классификации атрибутов на атрибуты включения и атрибуты условия, что происходит внутри `CIND_Miner`. Класс имеет конструктор от `InputTables` — стандартного представления таблиц, которое является типом параметра `InputTablesOpt`, и строит всё представление за один проход по таблицам, что достаточно эффективно. Следует учесть, что в памяти хранятся сразу оба представления входных данных, поскольку `SPIDER` работает только с `InputTables`.

Далее, опишем что из себя представляет класс для хранения `CIND`. Внутри себя он хранит `IND` класс, в котором хранится приближённая зависимость включения с действительной метрикой ошибки этой самой зависимости. Так же, внутри него хранится список условий, имеющих тип `Condition`. В этом классе представлены как значения атрибутов условия, так и действительная валидность и полнота. `Condition` имеет конструкторы от классов `ItemsetNode` и `ClusterValue`, которые используются для хранения кандидатов алгоритмами CINDERELLA и PLI-CIND соответственно и описаны в следующем разделе.

3.4. Поиск и хранение кандидатов

3.4.1. CINDERELLA

Несмотря на то, что алгоритм Apriori уже реализован в Desbordante, структуры данных оказались неприменимы для решения данной задачи, поэтому были реализованы свои классы для хранения кандидатов условия. Для быстрой валидации кандидатов, они хранятся в виде дерева, которое реализовано классом `Itemset`. Дерево хранит внутри себя вершины, представленные классом `ItemsetNode`, а значения этих вершин, которые представляют из себя атрибут и его значение, лежат в классе `Item`. Вершина добавляется в дерево только в том случае, если соответствующее ей условие (которое однозначно восстанавливается в виде пути от корня до неё) является достаточно полным. Чтобы ускорить поиск условий длины k, все вершины на уровне $k-1$, у которых нет детей, автоматически удаляются из дерева и пытаются запустить этот процесс у родителей. Внутри `Itemset` хранится список вершин на двух последних уровнях для быстрого доступа к вершинам, которые необходимы при формировании нового уровня, а так же при удалении листов, которое описано выше.

При реализации кандидатов в условия и контейнеров, была сделана следующая оптимизация. Заметим, что нам не нужно в условие добавлять индикатор включения: достаточно просто знать индикатор включения самого контейнера. Более того, чтобы

каждый раз для каждого условия не пробегаться по всем контейнерам (которых в строковом представлении таблицы может быть очень много), для каждого условия в `ItemsetNode` хранится информация о контейнерах, которые удовлетворяют данному условию. Информация представляет из себя порядковый номер группы, список индексов строк в таблице, которые удовлетворяют условию, а так же индикатор включения этого контейнера. Для L_1 эта информация высчитывается во время инициализации. При образовании нового кандидата, информация о контейнерах его «родителей» пересекается, что происходит эффективно за счёт поддерживания этой информации в отсортированном виде. Это позволяет существенно снизить время вычисления метрик для каждого условия. Сами же метрики вычисляются в конструкторе `ItemsetNode`: достаточно иметь информацию о контейнерах для данного кандидата, а так же общее количество включённых контейнеров, которое является константным во время работы алгоритма. Если новая вершина не является достаточно полной, она не будет добавлена в дерево кандидатов. Это позволило пропустить этап формирования L_k из C_k, что так же положительно повлияло на производительность.

3.4.2. PLI-CIND

В Desbordante уже существуют классы для работы с `PositionListsIndices`, однако он не поддерживает ключевую функцию, которая необходима алгоритму PLI-CIND — хранение значений позиционного списка (кластера). Более того, он был реализован специфично для определённого алгоритма и высчитывал множество ненужных метрик, что замедлило бы работу. Поэтому, на его основе был реализован класс `PositionListsSet` (сокращённо `PLSet`), который хранит значения для каждого кластера, и выполняет только необходимую работу.

Во время работы PLI-CIND инициализирует `PLSet` для каждого атрибута условия, а затем рекурсивно пытается расширить эти списки для поиска следующих кандидатов. Условием при таком подходе является каждый отдельный кластер. Для вычисления метрик нужно лишь пересечь список порядковых номеров групп (строк), хранящихся в кластере, и список индексов включённых групп (строк), что имеет линейную асимптотику, поскольку оба списка поддерживаются в отсортированном состоянии.

Стоит отметить, что при поиске групповых условий в позиционных списках могут появляться дубликаты. Однако такое поведение существенно усложняет и замедляет пересечение двух `PLSet`, в котором используется список соответствий `<индекс_строки>` → `<номер_кластера>`. Следовало оставить структуру данных в похожем на реализацию `PositionListsIndices` виде, чтобы она была простой для понимания другими разработчиками Desbordante. В связи с этим было принято решение хранить отдельно список соответствий `<индекс_строки>` → `<индекс_группы>` для вычисления метрик групповых условий.

4. Интеграция в Desbordante

Поскольку интеграция в Desbordante является одной из целей данной работы, все алгоритмы изначально реализовывались в рамках данного инструмента. Пользовательский интерфейс реализован для использования на языке Python с помощью привязок, реализованных с использованием библиотеки PyBind11[6].

Ниже приводится пример, в котором показано, как запустить алгоритм и вывести результаты в консоль:

```
import desbordante

algo = desbordante.cind.algorithms.Default()
TABLES = [(f'{table_name}.csv', ',', True) for table_name in ["cind_test_de",
    "cind_test_en"]]
algo.load_data(tables=TABLES, algo_type="cinderella")
algo.execute(error=0.5, validity=0.75, completeness=0.25, condition_type="row"
    )
for cind in algo.get_cinds():
    print(cind)
```

Listing 1: Пример использования алгоритма поиска CIND

Отрывок вывода данного кода показан ниже. Выводится как информация о самой AIND, так и об условиях: их количество, названия атрибутов, значения метрик.

```
(cind_test_de.csv, [pid]) -> (cind_test_en.csv, [pid]) with error threshold =
    0.5
Possible conditions number: 19
Possible conditions:
    (cind_test_de.csv.cent, cind_test_de.csv.birthplace, cind_test_de.csv.
    deathplace, cind_test_de.csv.desc);
    ("18", "-", "-", "-", validity = 0.777778, completeness = 1.000000);
    ("-", "-", "USA", "-", validity = 1.000000, completeness = 0.285714);
    ("-", "Kap", "-", "-", validity = 1.000000, completeness = 0.428571);
    ("-", "Sud", "-", "-", validity = 1.000000, completeness = 0.428571);
    ...
```

Listing 2: Пример вывода алгоритма поиска CIND

Ниже приведены таблицы, на которых запускался данный пример:

[6]https://github.com/pybind/pybind11

Таблица 1: cind_test_en.csv

pid	cent	birthplace	deathplace	desc
Cecil Kellaway	18	SA	USA	Actor
Mel Sheppard	18	USA	USA	Athlette
Buddy Roosevelt	18	CO	CO	Stunt
Sante Gaiardoni	19	Null	Null	Olympic

Таблица 2: cind_test_de.csv

pid	cent	birthplace	deathplace	desc
Cecil Kellaway	18	Kap	LA	Schauspieler
Cecil Kellaway	18	Kap	Cal	Schauspieler
Cecil Kellaway	18	Kap	USA	Schauspieler
Cecil Kellaway	18	Sud	LA	Schauspieler
Cecil Kellaway	18	Sud	Cal	Schauspieler
Cecil Kellaway	18	Sud	USA	Schauspieler
Sam Sheppard	19	Null	Null	Mediziner
Mel Sheppard	18	Almonesson Lake	Queens	Leichtathlet
Isobel Elsom	18	Cambridge	LA	Schauspielerin
Isobel Elsom	18	Cambridge	Cal	Schauspielerin

Так же реализованы привязки для классов примитивов `CIND` и `Conditions`. Для работы с этими классами было реализовано следующее:

- Экземпляры примитивов `CIND` и `Condition` можно получать в Python;
- Экземпляры можно получать по одному, итерируясь по ним, и знать их общее количество;
- Информацию о каждом экземпляре можно выводить в строковом представлении (реализован метод `__str__()`);
- Экземпляры можно сравнивать (реализован метод `__eq__()`);
- Экземпляры можно помещать в структуры данных `set()` и `dict()` (реализован метод `__hash__()`).
- Каждый экземпляр можно обрабатывать по частям: у `CIND` можно получать количество условий, их список, а так же атрибуты условия, а у `Condition` можно получать его значение, валидность и полноту.

Ниже приведено продолжение примера выше, где показывается работа с найденными примитивами:

```
cind = algo.get_cinds()[0]
print(cind.conditions_number())
```

```
print(cind == algo.get_cinds()[0])
print(cind.get_condition_attributes())
print()
condition = cind.get_conditions()[2]
print(condition.data())
print(condition.validity(), condition.precision()) # validity has a synonymous
    method precision
print(condition.completeness(), condition.recall()) # completeness has a
   synonymous method recall
```

Listing 3: Пример использования найденных примитивов

```
19
True
('cind_test_de.csv.cent', 'cind_test_de.csv.birthplace', 'cind_test_de.csv.
   deathplace', 'cind_test_de.csv.desc')

('-', 'Kap', '-', '-')
1.0 1.0
0.42857142857142855 0.42857142857142855
```

Listing 4: Вывод примера использования найденных примитивов

Данные привязки позволяют использовать полученные алгоритмы в прикладных задачах. Используя библиотеку Desbordante вместе с другими библиотеками Python у пользователей появляется возможность разрабатывать специальные решения для дедупликации данных, очистки данных, восстановления схем и решения других проблем качества данных. Полученные примитивы можно использовать вместе со стандартными инструментами для работы с данными, такими как Pandas[7]. За счёт этого, Desbordante можно использовать для редактирования таблиц внутри Python программы, а так же в конвейерах машинного обучения.

[7]https://pandas.pydata.org

5. Эксперименты

5.1. Подготовка

5.1.1. Исследовательские вопросы и метрики

Чтобы проверить работоспособность алгоритмов, а так же их эффективность, был поставлен ряд экспериментов. В качестве основных метрик будут приводиться время работы и потребляемая память. Целью данного раздела являются ответы на следующие исследовательские вопросы:

- **RQ1:** *Достоверна ли выдача реализованных алгоритмов?*
- **RQ2:** *Как соотносятся результаты, полученные в статье [6], с реализацией в* Desbordante?
- **RQ3:** *Какой из алгоритмов поиска условий наиболее применим на практике?*

5.1.2. Наборы данных

Для проведения экспериментов было отобрано 5 наборов данных, каждый из которых имеет свою особенность. Ниже представлена таблица с характеристиками этих наборов данных:

Таблица 3: Описание экспериментальных наборов данных

Название	Размер	#Таблиц	#Атрибутов	#Строк	#AIND, $\varepsilon = 0.5$
Brazilian E-Commerce	126.3 MB	9	52	1.6 M	54
CIND DbPedia Example	600 B	2	10	14	5
DbPedia En/De Persons	146.9 MB	2	21	1.46M	22
Cartoon/Comics Titles	11 MB	2	18	90 K	3
US Baby Names	85 MB	2	10	3.6M	4

Набор Brazilian E-Commerce[8] — набор реальных данных, в котором присутствуют около 10 таблиц и совокупно более 50 атрибутов.

Набор CIND DbPedia Example является точной копией набора данных из примеров, которые приводятся в этой статье [6]; также он приводился в качестве набора данных для примеров использования в разделе 4. Этот набор данных используется для проверки корректности алгоритмов и найденных примитивов.

DbPedia En/De Persons[9] — набор данных аналогичный тому, который использовался для изучения CIND в статье [6]. Оригинальный набор данных на данный момент

[8]https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce
[9]https://downloads.dbpedia.org/wiki-archive/data-set-20.html

недоступен — на сайте проекта DbPedia доступны для загрузки наборы данных старше 2014 года. Поэтому, была выбрана версия от 2015 года, которая имеет больший размер данных.

Cartoon/Comics Titles[10] — относительно небольшой набор данных, обработка которого не должна занимать много времени. За счёт этого, на нём удобно изучать поведение алгоритмов при различных значениях параметров.

US Baby Names[11] — набор, состоящий из 2 таблиц, каждая из которых имеет по 5 атрибутов. Его отличительной особенностью является размер таблиц: в каждой из них более миллиона записей. Он позволяет изучить поведение алгоритмов в случаях, когда записей в базе данных огромное количество, а зависимости включения так или иначе покрывают всю таблицу.

5.1.3. Характеристики тестового стенда

Эксперименты проводились на тестовом стенде, которая имеет следующие характеристики:

- ОС: macOS Sequoia 15.4.1;
- Процессор: Apple M1 pro, 3.2 Ghz;
- Оперативная память: LPDDR5 SDRAM 6400 MT/s, 16 GB, пропускная способность: 204 GB/s;
- Архитектура: ARM;
- Хранилище: APPLE SSD AP0512R, 512GB;
- Конфигурация C++: Apple clang 17.0.0, boost: stable 1.88.0.

5.2. Эксперимент

Во время исследования производительности алгоритмов, каждый алгоритм при каждой конфигурации был запущен 10 раз, после чего в качестве результатов выбиралось среднее арифметическое этих запусков. Во всех экспериментах исследовались общие потребляемые ресурсы для всех существующих CIND. Следует отметить, что максимальная разница в производительности не достигала 2%.

[10]https://www.kaggle.com/datasets/nikhil1e9/myanimelist-anime-and-manga
[11]https://www.kaggle.com/datasets/robikscube/us-baby-name-popularity

5.2.1. Достоверность выдачи алгоритмов

Чтобы убедиться, что реализация работает корректно и выдаёт достоверные примитивы, алгоритмы были запущены на наборах данных CIND DbPedia Example и Cartoon/Comics Titles, найденные зависимости проверялись на достоверность. В результате:

- Все найденные зависимости действительно выполнялись с описываемой метрикой ошибки;
- Все найденные условия оказались корректными: каждому из них удовлетворял некоторый набор записей;
- Все найденные метрики условий оказались точными и соответствовали действительности.

Полученные результаты свидетельствуют в пользу того, что алгоритмы, описанные в статье [6] были реализованы в DESBORDANTE без ошибок.

5.2.2. Зависимость производительности от метрик

Для детального изучения производительности при различных параметрах наиболее пригодным набором данных оказался Cartoon/Comics Titles за счёт небольшого размера и небольшого количества зависимостей.

Ниже приводятся графики зависимостей времени работы и потребляемой памяти от метрики полноты. Значение метрики ошибки AIND $\varepsilon = 0.5$, пороговое значение валидности $\gamma = 0.9$

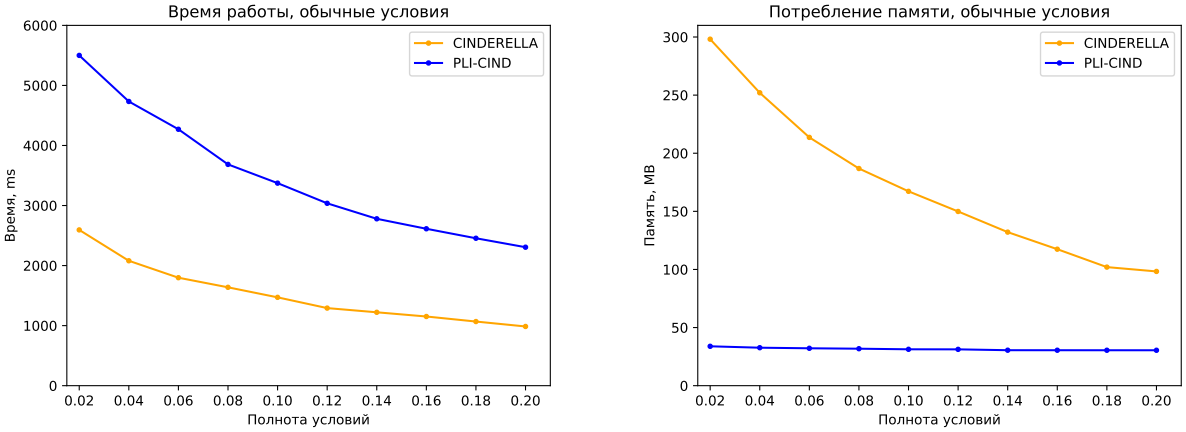

Рис. 8: Производительность поиска строковых условий

Из полученных результатов были сделаны следующие выводы:

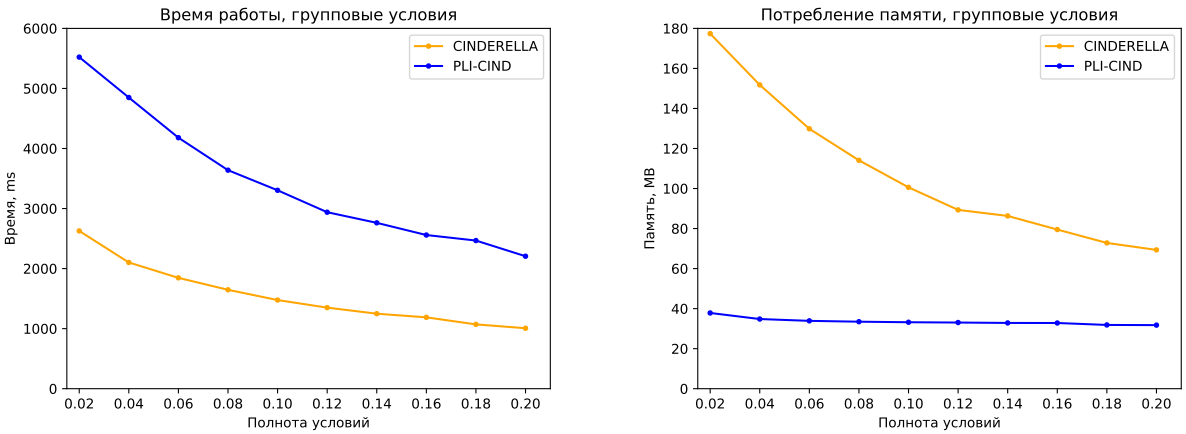

Рис. 9: Производительность поиска групповых условий

1. Производительность алгоритмов прямопропорционально зависит от порогового значения метрики полноты: чем оно выше, тем меньше время работы алгоритмов;

2. Потребление памяти CINDERELLA обратнопропорционально пороговому значению метрики полноты, в то время как потребление памяти PLI-CIND почти не зависит от него;

3. Время работы CINDERELLA действительно меньше времени работы PLI-CIND, в среднем в 2 раза;

4. PLI-CIND потребляет примерно одинаковое количество памяти независимо от метрики полноты;

Следует отметить, что производительность алгоритмов поиска условий зависит исключительно от метрики полноты: валидность используется только тогда, когда принимается решение о включении условия в ответ. Следовательно, она влияет только на размер итоговой таблицы условий и время её создания, что не оказывает значимого влияния на производительность CINDERELLA и PLI-CIND.

Сравнивая полученные данные с результатами, полученными в статье [6], можно утверждать, что зависимость производительности алгоритмов от метрики полноты сохраняется. Так же, алгоритм CINDERELLA работает быстрее, но потребляет заметно больше памяти, чем PLI-CIND, что так же соотносится с результатами в статье.

5.3. Производительность на различных наборах данных

Далее, была изучена производительность алгоритмов CINDERELLA и PLI-CIND на различных наборах данных. Диаграмма, показывающая результаты данного эксперимента, приведена на Рис. 10.

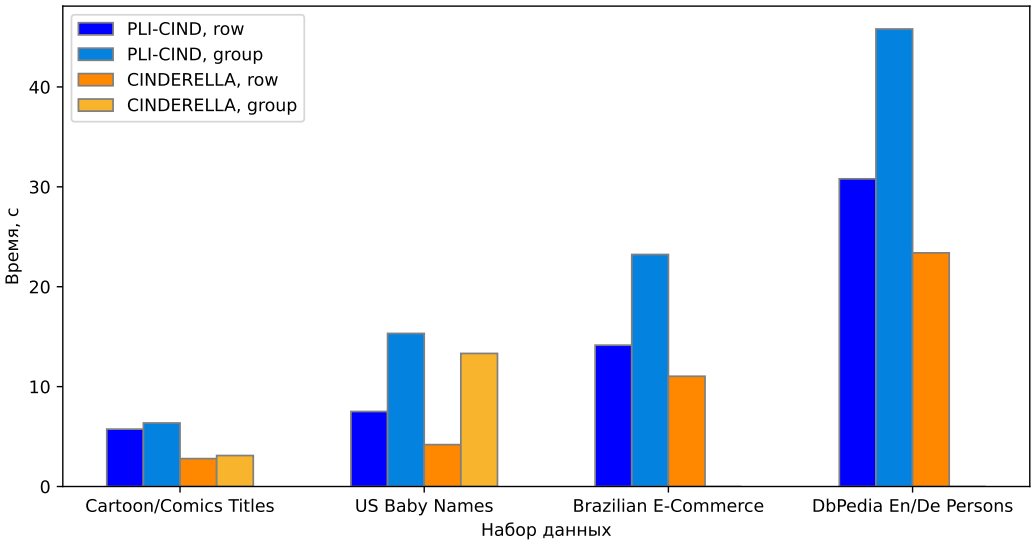

Рис. 10: Сравнение времени работы алгоритмов на различных наборах данных

В результате данного эксперимента были получены следующие заключения:

1. Из-за того, что количество найденных групповых условий в наборах Brazilian E-Commerce и DbPedia En/De Persons превышает 100 миллионов, CINDERELLA превышал допустимое потребление памяти и замерить его производительность не представляется возможным. Тем не менее, на тех наборах данных, на которых алгоритм завершался успешно, CINDERELLA всё так же показывает лучшее время, чем PLI-CIND, что подтверждает вывод, сделанный в рамках предыдущего эксперимента.

2. Несмотря на то, что при поиске строковых условий CINDERELLA показывает лучшее время, на больших наборах данных данная разница не превышает 35%. Более того, поиск групповых условий на больших наборах и вовсе невозможен. Из чего следует, что наиболее оптимальным на практике является алгоритм поиска условий PLI-CIND.

5.4. Выводы

Проведя ряд экспериментов, были получены следующие ответы на исследовательские вопросы:

- **RQ1:** *Достоверна ли выдача реализованных алгоритмов?*

  Реализованные алгоритмы на выходе выдают достоверные результаты: они находят CIND, метрики которых полностью соответствуют действительности.

- **RQ2:** *Как соотносятся результаты, полученные в статье [6], с реализацией в* DESBORDANTE*?*

  Результаты, полученные в ходе экспериментов соответствуют выводам, полученным в статье, которые описывают производительность двух алгоритмов поиска условий. Производительность алгоритмов прямопропорционально зависит от порогового значения метрики полноты. Так же показано, что CINDERELLA является более быстрым алгоритмом, а PLI-CIND — менее требовательным к памяти.

- **RQ3:** *Какой из алгоритмов поиска условий наиболее применим на практике?*

  По результатам проведённых экспериментов, можно утверждать, что на практике PLI-CIND является оптимальным выбором: несмотря на то, что он медленнее CINDERELLA на 30-35%, он является более экономичным по памяти алгоритмом, и может работать на больших наборах данных.

Заключение

В результате проделанной работы были выполнены следующие задачи:

1. Произведено знакомство с предметной областью поиска условных зависимостей включения. Написан обзор, выбраны алгоритмы SPIDER для поиска приближённых зависимостей, а так же CINDERELLA и PLI-CIND для поиска условий для них, данные выборы обоснованы. Так же был проведён анализ существующих инструментов для профилирования данных и был выбран Desbordante, данный выбор так же обоснован.

2. Реализованы алгоритмы CIND_Algorithm, CINDERELLA и PLI-CIND, а так же все необходимые структуры данных для них. Так же к ним были применены архитектурные оптимизации, которые влияют как на время работы алгоритмов, так и на потребляемую память.

3. Алгоритмы были внедрены в Desbordante, был реализован пользовательский интерфейс для взаимодействия как с самим алгоритмом, так и для взаимодействия с найденными примитивами.

4. Был проведён ряд экспериментов, в результате которых было выяснено, что алгоритмы выдают достоверный результат, исследована разница в производительности между CINDERELLA и PLI-CIND, а так же показано, что PLI-CIND — наиболее применимый на практике алгоритм.

Результаты работы доступны для изучения в репозитории Desbordante (pull-request 559)[12].

[12]https://github.com/Desbordante/desbordante-core/pull/559

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] Agrawal Rakesh, Srikant Ramakrishnan. Fast Algorithms for Mining Association Rules in Large Databases // Proceedings of the 20th International Conference on Very Large Data Bases. — VLDB '94. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1994. — P. 487–499.

[3] 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.

[4] 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). — 2021. — P. 344–354.

[5] Chernishev George, Polyntsov Michael, Chizhov Anton et al. Desbordante: from benchmarking suite to high-performance science-intensive data profiler (preprint). — 2023. — URL: https://arxiv.org/abs/2301.05965.

[6] Discovering conditional inclusion dependencies / Jana Bauckmann, Ziawasch Abedjan, Ulf Leser et al. // Proceedings of the 21st ACM International Conference on Information and Knowledge Management. — CIKM '12. — New York, NY, USA : Association for Computing Machinery, 2012. — P. 2094–2098. — URL: https://doi.org/10.1145/2396761.2398580.

[7] Discovering functional and inclusion dependencies in relational databases / Martti Kantola, Heikki Mannila, Kari-Jouko Räihä, Harri Siirtola // International Journal of Intelligent Systems. — 1992. — Vol. 7, no. 7. — P. 591–607. — https://onlinelibrary.wiley.com/doi/pdf/10.1002/int.4550070703.

[8] Efficient Discovery of Functional Dependencies from Incremental Databases / Loredana Caruccio, Stefano Cirillo, Vincenzo Deufemia, Giuseppe Polese // The 23rd International Conference on Information Integration and Web Intelligence. — iiWAS2021. — New York, NY, USA : Association for Computing Machinery, 2022. — P. 400–409. — URL: https://doi.org/10.1145/3487664.3487719.

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

[10] Kruse Sebastian, Papenbrock Thorsten, Dullweber Christian et al. Fast Approximate Discovery of Inclusion Dependencies. — 2017. — 03.

[11] Goncharov D. Примитивы профилирования данных в Desbordante: обзор и сравнение существующих инструментов. — 2024. — https://github.com/Desbordante/desbordante-core/blob/f6afc52aae477da7f2f137c7fef2624e861f8244/docs/papers/Data profiling survey - Daniil Goncharov - 2024 spring.pdf.

[12] Inclusion Dependency Discovery: An Experimental Evaluation of Thirteen Algorithms / Falco Dürsch, Axel Stebner, Fabian Windheuser et al. // Proceedings of the 28th ACM International Conference on Information and Knowledge Management. — CIKM '19. — New York, NY, USA : Association for Computing Machinery, 2019. — P. 219–228. — URL: https://doi.org/10.1145/3357384.3357916.

[13] Marchi Fabien De, Lopes Stéphane, Petit Jean-Marc. Unary and n-ary inclusion dependency discovery in relational databases // Journal of Intelligent Information Systems. — 2009. — P. 53–73. — URL: https://doi.org/10.1007/s10844-007-0048-x.

[14] Marchi Fabien De, Petit Jean-Marc. Zigzag: a new algorithm for mining large inclusion dependencies in databases // Proceedings of the Third IEEE International Conference on Data Mining. — ICDM '03. — USA : IEEE Computer Society, 2003. — P. 27.

[15] Smirnov A. Реализация алгоритмов для поиска зависимостей включения в рамках платформы Desbordante. — 2023. — https://github.com/Desbordante/desbordante-core/blob/f6afc52aae477da7f2f137c7fef2624e861f8244/docs/papers/Faida - Alexandr Smirnov - BA thesis.pdf.

[16] Srikant Ramakrishnan, Vu Quoc, Agrawal Rakesh. Mining association rules with item constraints // Proceedings of the Third International Conference on Knowledge Discovery and Data Mining. — KDD'97. — AAAI Press, 1997. — P. 67–73.

[17] Tane: An Efficient Algorithm for Discovering Functional and Approximate Dependencies / Yká Huhtala, Juha Kärkkäinen, Pasi Porkka, Hannu Toivonen // The Computer Journal. — 1999. — Vol. 42, no. 2. — P. 100–111.