Санкт-Петербургский государственный университет

Группа 21.Б08-мм

Реализация алгоритма HyMD на C++

Шлёнских Алексей Анатольевич

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
ассистент кафедры ИАС Г. А. Чернышев

Санкт-Петербург
2024

Оглавление

Введение

Профилирование данных — это процесс анализа данных, используемый для извлечения метаданных. Получив эту информацию можно, в свою очередь, преобразовать исходные данные некоторым полезным способом или сделать некоторые полезные выводы.

Один из видов профилирования данных — поиск закономерностей. Возможные закономерности зависят от типа рассматриваемых данных, будь то табличные, графовые, транзакционные или другие. В данной работе будут рассматриваться только табличные данные.

Пример закономерности на табличных данных — функциональные зависимости (Functional Dependencies, далее FD). Их можно использовать для задачи приведения данных в нормальную форму. При этом для задачи, например, дедупликации данных они уже не подходят, так как условие равенства атрибутов может не соблюдаться, если в данных есть ошибки. Для этой задачи могут использоваться ослабленные функциональные зависимости (Relaxed Functional Dependencies, далее RFD) [1].

Одной из разновидностей RFD являются сопоставляющие зависимости (Matching Dependencies, далее MD). Если для соблюдения FD нужно чтобы при равенстве значений атрибутов в левой части выполнялось равенство значений атрибута в правой части, то для того, чтобы соблюдалась MD, нужно чтобы при достаточной схожести значений атрибутов левой части значения атрибута в правой части тоже были достаточно схожи. Тем самым MD ослабляют критерий равенства.

Одним из примеров использования MD для дедупликации является система MDedup [5]. Для использования MD их надо найти, для чего в этой системе используется алгоритм HyMD, реализованный на платформе Metanome.

Однако платформа Metanome и алгоритм HyMD реализованы на языке Java, что значительно ограничивает производительность. Поэтому в этой работе алгоритм HyMD реализуется на C++ в рамках платформы Desbordante [2].

Платформа Desbordante — профилировщик данных, ориентированный на решение прикладных задач. Для решения задач, связанных с наукой о данных, используются средства, написанные на языке Python, поэтому ожидается, что алгоритмы, реализованные на платформе, можно будет использовать и из языка Python. Поэтому в этой работе алгоритм HyMD также связывается с Python.

1. Постановка задачи

Целью работы является реализация эффективного алгоритма поиска MD. Для этого были поставлены следующие задачи:

- Реализовать алгоритм HyMD из [4] на платформе Desbordante;
- Сравнить производительность с реализацией в Metanome;
- Связать алгоритм с Python.

2. Сопоставляющие зависимости

2.1. Основные понятия

Сопоставляющие зависимости (Matching Dependency, MD) определяются для двух отношений, далее R и S, и могут соблюдаться на паре экземпляров этих отношений, далее r и s соответственно. Здесь приведены сокращённые определения, более полные определения смотреть в [4] и [8].

Определение 1. Метрика схожести (similarity metric) — двуместная функция, результат которой — вещественное число из отрезка $[0.0, 1.0]$.

Определение 2. Значения пары считаются схожими, если результат применения метрики схожести к ним больше или равен некоторому пределу, который называется границей решения. Иначе они несхожи.

Рассматривать границы решения не из отрезка $[0.0, 1.0]$ не имеет смысла, поэтому дальше считаем, что границы решения всегда находятся в этих пределах.

Определение 3. Сопоставление столбцов (column match) — тройка, в которой первый элемент — атрибут R, второй — атрибут S, третий — метрика схожести.

Определение 4. Классификатор столбцов (column classifier) — пара из сопоставления столбцов и границы решения.

Определение 5. Записи в паре из $r \times s$ считаются сопоставленными набором классификаторов столбцов, если для каждого классификатора столбцов их значения атрибутов схожи.

Определение 6. MD есть набор классификаторов столбцов (левая часть, LHS) и ещё один классификатор столбцов (правая часть, RHS).

Определение 7. Пара записей противоречит MD, если записи пары сопоставляется её LHS, но значения атрибутов из RHS у записей пары несхожи.

Определение 8. На r и s соблюдается MD, если ни одна из пар записей из $r \times s$ не противоречит ей.

2.2. Поиск MD

2.2.1. Пространство поиска

Чтобы использовать MD на наборах данных, их нужно сначала найти. Поиск MD определим для некоторого заранее заданного набора сопоставлений столбцов и пары таблиц как подбор таких границ решения, что MD с левой частью, имеющая в левой

части эти сопоставления столбцов с соответствующими им границами решения, а в правой — одно из этих сопоставлений с собственной границей решения, соблюдается на паре таблиц.

Стоит обратить внимание, что, согласно определению выше, почти на всех наборах данных континуально много соблюдающихся MD. Ограничим множество рассматриваемых зависимостей.

Определение 9. Назовём границу решения естественной для заданного сопоставления столбцов и таблиц r и s, если она является результатом применения метрики схожести к значениям атрибутов сопоставления некоторой пары записей из $r \times s$.

Определение 10. Назовём MD естественной для r и s, если каждая граница решения во всех составляющих её классификаторах столбцов является естественной для сопоставления столбцов из этого классификатора.

Искать соблюдающиеся MD будем среди множества естественных MD, остальные можно вывести из них при необходимости. Но даже так рассматриваемых зависимостей слишком много — $(|r| * |s|)^{n+1}$, где n — количество классификаторов столбцов, в худшем случае.

Определение 11. Назовём MD тривиальной (trivial), если граница решения в её правой части не больше границы решения для того же сопоставления столбцов в левой части.

Тривиальные MD соблюдаются на любой паре таблиц, а значит искать их не имеет смысла.

Заметим, что как при уменьшении границы решения в правой части у соблюдающейся MD, так и при увеличении одной из границ решения в её левой части, она не перестаёт быть соблюдающейся. Воспользовавшись этим, зададим частичный порядок на множестве поиска MD.

Определение 12. Пусть φ, φ' — некоторые MD на рассматриваемом множестве. Пишем $\varphi' \preceq \varphi$, если:

1. Все границы решения левой части φ' не больше границ решения левой части соответствующих сопоставлений столбцов в φ;
2. В правых частях φ и φ' одно и то же сопоставления столбцов;
3. Граница решения правой части φ' не меньше границы решения правой части φ.

При этом говорим, что φ' — обобщение (generalization) φ, φ — специализация (specialization) φ'.

Аналогичные термины также будем использовать для левых частей некоторых MD, но в этом случае определение частичного порядка для них не требует последних двух пунктов.

Определение 13. Называем MD минимальной соблюдающейся, если она является минимальной относительно отношения частичного порядка $\preceq$ на множестве соблюдающихся на r и s естественных сопоставляющих зависимостей.

Стоит заметить, что для полного описания множества соблюдающихся на паре таблиц MD достаточно только минимальных соблюдающихся MD, их и будем искать. Таким образом, было построено пространство поиска.

2.2.2. Отсечение ненужных MD

Даже такое пространство поиска очень велико. При этом не все минимальные соблюдающиеся MD имеют практический интерес, поэтому в [4] предлагаются несколько критериев интересности, которые позволяют значительно уменьшить пространство поиска.

Определение 14. Мощность (cardinality) MD — это количество ненулевых границ решения в её левой части.

Определение 15. Поддержка (support) MD — это количество пар из $r \times s$, записи в которых сопоставляет её левая часть.

В [4] предлагается отсекать MD со слишком большой мощностью или слишком маленькой поддержкой. Также предлагаются ещё три критерия интересности:

- Предлагается отсекать те MD, в которых граница решения для сопоставления столбцов из правой части в левой части ненулевая;
- Предлагается не рассматривать зависимости, у которых какая-то из ненулевых границ решения в левой части слишком маленькая;
- Предлагается ограничить количество рассматриваемых границ решения для сопоставлений столбцов из левой части.

В реализации для Desbordante присутствуют все критерии интересности, кроме достаточной малости мощности.

2.3. Стратегии поиска MD

После задания частичного порядка можно использовать стратегии поиска зависимостей на данных из ранних работ, адаптировав их под MD.

Обход решётки. Эта стратегия была описана в [7] для алгоритма поиска FD TANE. С помощью частичного порядка на зависимостях алгоритм представляет пространство поиска как решётку, затем исследует её, отбрасывая те её части, исследование которых не приведёт к обнаружению новых зависимостей. Для FD это значило экспоненциальный рост размера решётки при увеличении количества атрибутов, для MD рост размера решётки с ростом количества сопоставлений столбцов более быстрый, чем для FD, ведь границ решения может быть больше двух.

Выведение из пар записей. Эта стратегия была предложена для алгоритма поиска FD FDEP. Путём сравнения пар записей выводятся несоблюдающиеся зависимости, из которых выводятся соблюдающиеся зависимости. Для FD время поиска в худшем случае масштабируется как $|r|^2$ (FD определены только для одной таблицы), для MD — $|r| * |s|$.

Гибридный подход. Один из алгоритмов поиска FD, использующий такую стратегию — HyFD [6]. Она была предложена для разрешения недостатков масштабируемости двух предыдущих стратегий. В ней стратегии используются поочерёдно, возможно, обмениваясь полезной друг для друга информацией. Используемая стратегия меняется, когда становится "неэффективной".

Последняя стратегия используется в HyMD.

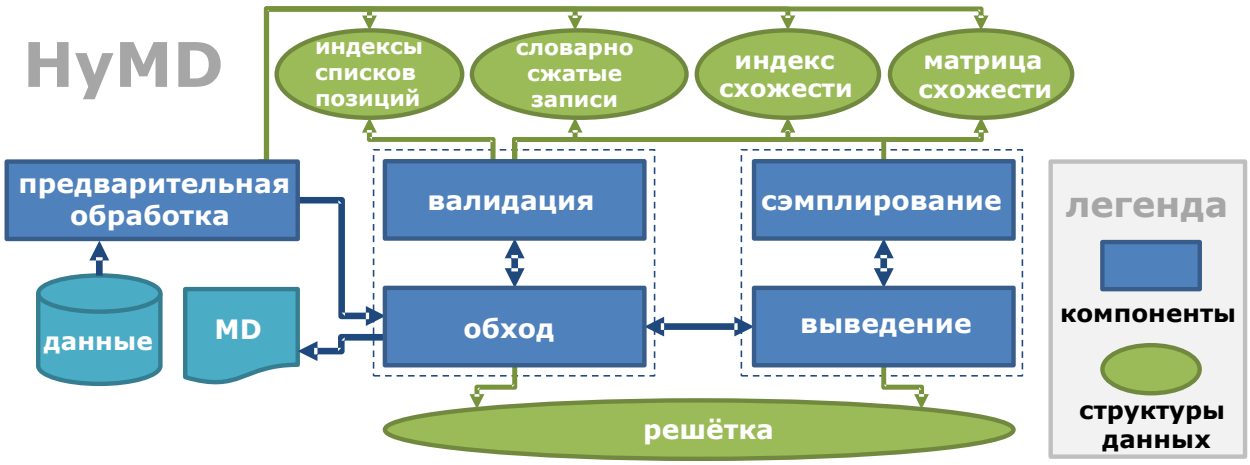

Рис. 1: Источник: [4]

3. HyMD

HyMD использует гибридную стратегию поиска зависимостей, адаптируя её под поиск MD. Алгоритм состоит из нескольких компонентов, показанных на Рис. 1.

3.1. Индексы

Перед началом поиска зависимостей данные преобразуются на этапе предварительной обработки. Сначала одновременно создаются индексы списков позиций (position list index, PLI) и словарно сжатые записи (dictionary compressed records). Для этого алгоритм читает записи из входных таблиц по одной и записывает значение в каждом столбце в PLI (один для каждой колонки), записывая набор идентификаторов значений, каждый из которых получен из соответствующего PLI, во второй индекс.

Индекс списков позиций для каждой колонки содержит списки идентификаторов записей, в которых встречается значение (кластеры). В каждом таком индексе идентификатор значения сопоставлен записям, в которых это значение встречается. Реализован как массив массивов целых чисел.

Словарно сжатые записи является массивом сжатых записей — массивов идентификаторов значений. В этом индексе идентификатору записи сопоставляется набор значений в этой записи. Индекс реализован как массив массивов.

После этого составляются индексы и матрицы схожести для каждого сопоставления столбцов. Алгоритм подсчитывает схожести между значениями, встречающимися в столбцах из сопоставления столбцов. Далее в матрицу схожести записываются схожести значения из r со значением из s, если схожесть не слишком мала, а в индекс схожести для каждого значения из r и схожести записываются множества идентификаторов записей из s.

Схожести могут считаться как между всеми парами значений, так и между лишь

некоторых из них для уменьшения расчётов за счёт потери точности. Сейчас в варианте реализации из Desbordante подсчитываются все схожести. Кроме того, подсчёт схожестей независим от других таких же подсчётов, поэтому его можно распараллелить.

Матрица схожести сопоставляет идентификаторам значений из таблиц r и s их схожесть. Реализуется как массив ассоциативных массивов (в ассоциативном массиве ключ — идентификатор значения из s, а значение — схожесть). Отсутствие ключа означает, что схожесть нулевая.

Индекс схожести сопоставляет значению из r и схожести все записи из s, у которых схожесть в значении атрибута такая же или бóльшая. Может быть реализован как массив ассоциативных массивов с порядком (в ассоциативном массиве ключ — схожесть, а значение — множество идентификаторов записей).

После завершения предварительной обработки алгоритм начинает работу с выведения из пар записей, переключаясь между ним и обходом решётки, заканчивая поиск MD обходом решётки.

3.2. Решётка

Эта структура моделирует пространство поиска. С её помощью выполняются операции, используемые при поиске зависимостей[1]:

- Проверка, есть ли в решётке обобщение данной MD. Реализуется как проверка всех зависимостей в решётке, у которых левая часть обобщает данную;
- Добавление MD, если в решётке нет её обобщения (`addIfMin`). Может быть реализован как проверка с помощью метода выше с последующим добавлением, если обобщений нет, но на деле проверка происходит одновременно с добавлением;
- Поиск рассматриваемых MD, которым противоречит данная пара записей (`findViolated`). Реализуется как обход всех обобщений левой части и добавление тех, у которых граница решения правой части слишком большая;
- Получение максимальных границ решения правой части для обобщений нескольких MD с одинаковой LHS (`getLowerBoundaries`). Реализуется как обход всех обобщений левой части и поиск самых больших правых частей;
- Получение зависимостей из определённой части решётки (`getLevel`). Операция собирает все зависимости из неё.

[1]Список не совпадает с [4], приведённые здесь операции более близки к реальной реализации как в Metanome, так и в Desbordante

Часть решётки в последнем пункте называется уровнем, его номер является входом этой операции. В [4] представляется два определения уровня (мощность и глубина, cardinality и depth), но на деле определение может быть любым таким, что для него корректно работает обход решётки. В Desbordante реализовано только определение уровня как мощности.

Для метода `getLowerBoundaries` в [4] предлагается останавливать поиск, если все границы решения становятся равными 1.0, однако при полностью корректной реализации это невозможно. Этот метод вызывается только во время валидации для некоторой левой части и некоторых правых частей после их удаления из решётки (после получения результата они восстанавливаются). И если даже одна из границ решения правой части становится равной 1.0, то это означает, что одна из валидируемых зависимостей является специализацией — нарушен инвариант алгоритма, в решётке должны быть только минимальные из рассматриваемых зависимостей.

В Desbordante, как и в Metanome, решётка реализована как префиксное дерево.

3.3. Выведение из пар записей

В терминах Рис. 1 за эту стратегию отвечают компоненты *сэмплирование* и *выведение*. Стратегию начинает компонент *выведение*.

Стоит отметить, что выведение на самом деле работает не с самими парами записей, а со списками схожестей значений атрибутов каждого сопоставления столбцов относительно метрики у некоторой пары. Каждый такой ранее необработанный список подаётся как вход методу решётки `findViolated` и затем у возвращённых зависимостей понижается граница решения в правой части и в решётку вставляются их специализации.

Коллекция списков выше может быть обработана не полностью, если алгоритм посчитает, что выведение из пар записей стало неэффективным, и переключит стратегию. Эта проверка выполняется перед обработкой каждого такого списка.

Сама проверка может отличаться в зависимости от конкретной реализации алгоритма. В [4] предлагается останавливаться, когда отношение нарушенных зависимостей к количеству проверенных пар становится слишком мало, но в реализации из Metanome процесс останавливается, если количество проверенных списков схожестей больше 100 или сэмплирование было вызвано более одного раза после перехода на эту стратегию. Реализация в Desbordante использует то же самое условие для проведения объективных сравнений производительности с Metanome.

Процессу выше подвергаются три коллекции списков — рекомендации, оставшиеся списки, полученные от сэмплирования списки. Рекомендации — это полученные от фазы обхода решётки пары сжатых записей, которые считаются самыми интересными для проверки. Сэмплирование отдаёт списки, полученные после подсчёта схо-

жестей значений одной записи из таблицы r с записями из таблицы s, но выведение может прерваться, поэтому перед их обработкой обрабатываются списки, оставшиеся от предыдущего сэмплирования.

Получения всех значений схожести независимы, поэтому сэмплирование, как и вычисление списков схожестей для рекомендаций, можно распараллелить. В Desbordante это не реализовано, но есть в Metanome.

3.4. Обход решётки

В терминах Рис. 1 за эту стратегию отвечают компоненты *валидация* и *обход*. Стратегия начинается с компонента обхода.

При обходе решётки получаются все необработанные ранее зависимости текущего уровня, которые затем обрабатываются. При этом в зависимости от определения уровня из них может понадобиться выбирать те, что не были выбраны ранее и не специализируют друг друга. Для первого обработанные зависимости текущего уровня запоминаются, для второго оставшиеся зависимости минимизируются.

Определение 16. Минимизацией назовём получение из набора MD таких зависимостей, которые останутся минимальными относительно $\preceq$ в модифицированном наборе, где все границы решения правой части прошлых MD установлены в какое-то одно значение.

Все полученные зависимости, сгруппированные по их левой части, затем валидируются с помощью компонента "валидация". Под валидацией понимается определение таких границ решения RHS MD, при которой они соблюдаются на данных, и определение, достаточная ли поддержка у LHS. Каждая валидация независима от других, поэтому их можно выполнять параллельно.

Способ валидации зависит от мощности MD. Если мощность нулевая, то сопоставляются абсолютно все записи и сразу ясно, что границы решения правых частей должны быть самыми маленькими естественными. Поддержка левой части в этом случае — $|r| * |s|$.

Если мощность ненулевая, то процесс валидации заключается в понижении границ решения RHS до наименьших среди сопоставляемых LHS пар записей и подсчёта поддержки. Для этого последовательно рассматриваются особые пары множеств записей из r и из s. В множестве записей из r у всех записей одинаковый набор значений атрибутов, для которых в LHS есть хотя бы одно сопоставление столбцов с ненулевой границей решения. Множество записей из s состоит из записей, у которых значения в соответствующих атрибутах похожи на соответствующие значения из этого набора. Затем для каждой RHS граница решения уменьшается до наименьшей среди пар из декартова произведения этих множеств. До или после прохода пар к поддержке добавляется произведение мощностей упомянутых множеств.

Если граница решения опускается до значения, полученного с помощью метода `getLowerBoundaries`, или значения в левой части для соответствующего сопоставления столбцов или ниже, то сразу понятно, что зависимость потом придётся удалить как неминимальную. При этом если все рассматриваемые MD оказались такими и поддержка левой части достаточна, процесс валидации можно прекратить.

Если мощность равна единице, то записи, сопоставляемые левой частью, можно получить, пройдя по PLI столбца r из сопоставления столбцов, в котором граница решения ненулевая для левой части, и для каждого значения получив достаточно похожие записи из s с помощью индекса схожести.

Если мощность больше единицы, то для получения сопоставляемых записей нужно записи в кластерах PLI одного из сопоставлений столбцов с ненулевой границей решения сначала сгруппировать по значениям в таких сопоставлениях столбцов, затем получить множества похожих на них записей из s путём пересечения множеств из индекса схожести для каждого значения и границы решения.

Во время валидации каждой правой части перед началом сравнений схожестей записей из множеств записи из r группируются по значению атрибута столбца из r в рассматриваемом сопоставлении столбцов. Точно также можно групировать и записи из s, но реализации алгоритма этого не делают.

Кроме того во время валидации каждой правой части также собираются рекомендации для выведения из пар записей. В рекомендации попадают те из проверенных пар сжатых записей, у которых схожесть значений для одной из проверяемых RHS меньше той, что была в решётке перед валидацией.

После завершения валидации границы решения у зависимостей в решётке понижаются до полученных значений, а бывшие ранее в решётке зависимости специализируются, то есть в решётку добавляются зависимости, у которых LHS является специализацией старой LHS, а RHS такая же. Это происходит последовательно, ведь при специализации LHS может появиться зависимость, которая является специализацией другой LHS, с которой предыдущая была несравнимой относительно $\preceq$.

Процесс выше происходит только если валидация показала, что левая часть поддерживается. Если она не поддерживается, то эта левая часть просто отмечается как неподдерживаемая. Для этого существует ещё одна решётка, которая позволяет проверять, была ли левая часть отмечена как неподдерживаемая или, соответственно, отметить левую часть как неподдерживаемую.

Также собранные рекомендации добавляются в общую коллекцию рекомендаций, что происходит одновременно со специализацией, ведь это независимые процессы. Коллекцию рекомендаций потом получает выведение из пар записей.

4. Реализация HyMD

Составные части алгоритма представлены на Рис. 2.

Класс `Algorithm` предоставляет полезные всем алгоритмам функции такие как конфигурация. От него унаследован класс алгоритма поиска MD `MdAlgorithm`, который предоставляет пользователям класса единый интерфейс получения результатов. Пока алгоритм поиска MD только один — HyMD.

Для чтения данных используются имеющиеся средства платформы Desbordante.

Индексы списков позиций и словарно сжатые записи составляется в методе LoadDataInternal. В коде полученные данные содержатся в классе `CompressedRecords`. Этот класс содержит в себе классы `DictionaryCompressor` для каждой таблицы.

Для составления индексов алгоритм читает таблицу по одной строке и методом `DictionaryCompressor::AddRecord` добавляет значения из них во все PLI, составляя каждую запись из идентификаторов значений, возвращаемых каждым PLI. PLI представлены классом `KeyedPositionListIndex`, сжатые записи — `std::vector<CompressedRecord>`, где `CompressedRecord` — это `std::vector<ValueIdentifier>`, а `ValueIdentifier` — `std::size_t` — индекс значения, возвращаемый PLI. Индекс сжатой записи — это `RecordIdentifier`.

Дальнейшая работа выполняется в `ExecuteInternal`. Сначала создаются другие два индекса — матрицы схожести и индексы схожести. Это происходит в методе `SimilarityData::CreateFrom`. Метод принимает `CompressedRecords`, сопоставления столбцов и другие данные, нужные `SimilarityData` для работы.

По конфигурации создаются классы `SimilarityMetric`, которым даются столбцы из сопоставления, преобразованные из списка строк в список объектов типа, требуемого метрикой, и правый PLI для создания индексов схожести. Метод `SimilarityMetric::MakeIndexes` создаёт структуру, содержащую всю нужную алгоритму информацию о сопоставлении столбцов, а именно:

- Границы решения левой части для этого сопоставления столбцов. Они обязательно естественные, но не это не обязательно все естественные границы;
- Самая низкая схожесть среди проверенных пар. Нужна для валидации MD с левой частью нулевой мощности;
- Матрица схожести. Тип: `std::vector<std::unordered_map<ValueIdentifier, double> >`;
- Индекс схожести. Тип: `std::vector<std::map<double, std::unordered_set<RecordIdentifier> > >`

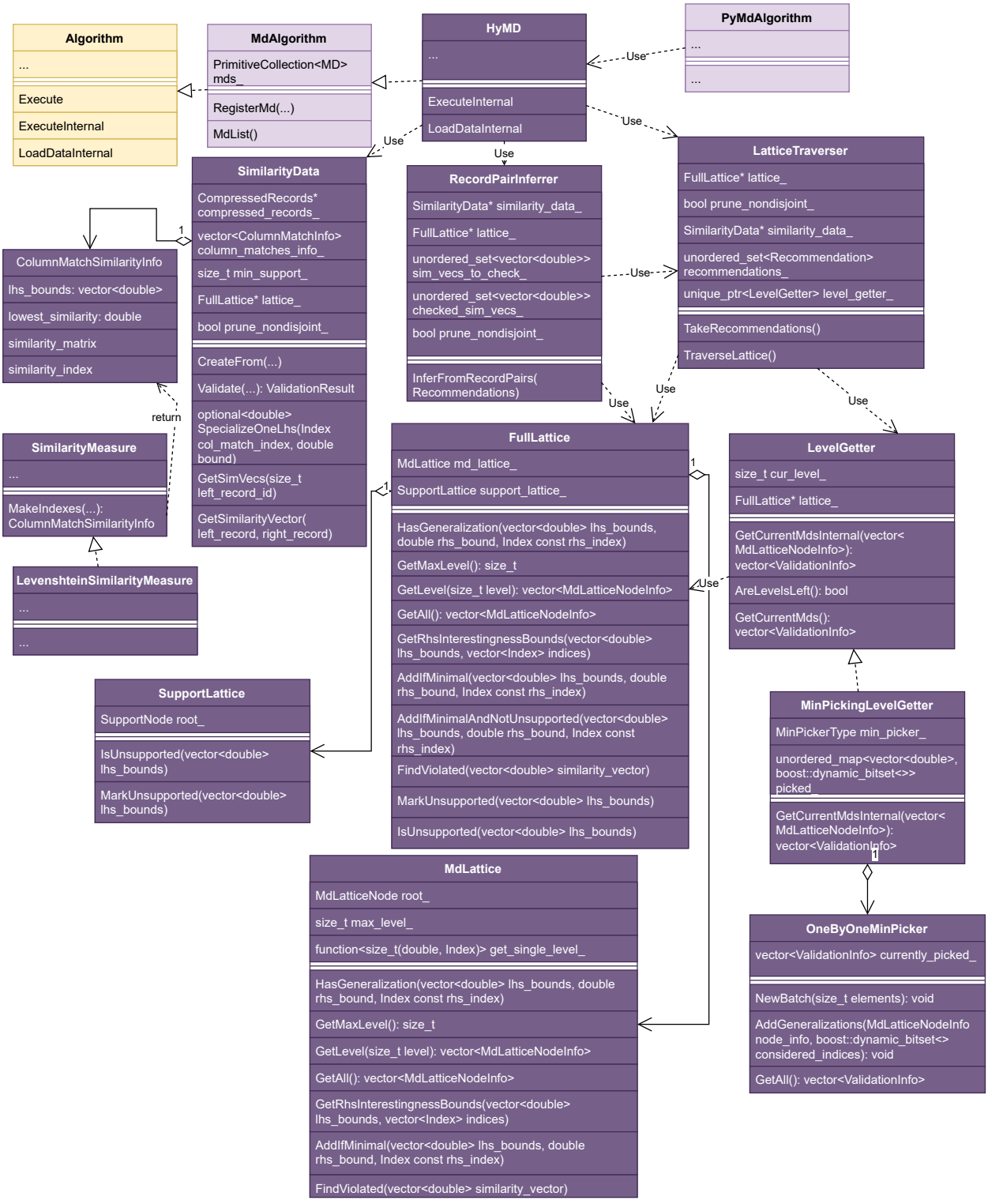

Рис. 2: Схема некоторых классов, используемых в реализации HyMD в Desbordante. Жёлтым отмечена имеющаяся инфраструктура, светло-фиолетовым — то, что сделал автор, но не относится напрямую к алгоритму, тёмно-фиолетовым — части алгоритма.

После получения этой информации для всех сопоставлений столбцов создаётся объект `SimilarityData`, который её хранит.

Также перед выполнением создаются объект `FullLattice`, который содержит в себе решётку MD (`MdLattice`) и решётку поддержки LHS (`SupportLattice`), и объекты, которые реализуют две стратегии поиска MD: `LatticeTraverser` и `RecordPairInferrer`.

Для получения зависимостей для проверки `LatticeTraverser` использует класс `LevelGetter`. `MinPickingLevelGetter` является пока единственной имплементацией этого абстрактного класса, предназначенной для случая, когда полученные зависимости нужно дополнительно минимизировать.

Саму минимизацию выполняет объект `OneByOneMinPicker`, который получает набор с требуемым свойством, сравнивая каждую зависимость исходного друг с другом. Ещё реализован `LatticeMinPicker`, который организует зависимости в решётку, подобную `MdLattice`, тем самым уменьшая количество сравнений.

Компоненты на Рис. 1 можно сопоставить методам из реализации: за компонент валидации отвечает метод `SimilarityData::Validate`, за обход — `LatticeTraverser::TraverseLattice`, за выведение — `RecordPairInferrer::InferFromRecordPairs`, за сэмплирование — `SimilarityData::GetSimVecs`.

После выполнения алгоритма полученные из решётки зависимости добавляются в коллекцию MD класса `MdAlgorithm`, откуда их потом можно получить извне с помощью метода `MdList`.

Также присутствует связка алгоритма с Python, которую обеспечивает шаблон `PyMdAlgorithm`.

5. Использование

Алгоритм позволяет задавать 5 опций:

- `left_table` — первая таблица;
- `right_table` — вторая таблица;
- `prune_nondisjoint` — отсечение по пересечению атрибутов. По умолчанию включено;
- `min_support` — минимальная поддержка. Стандартное значение, если задана одна таблица — `размер таблицы` + 1, иначе — 1;
- `column_matches` — список сопоставлений столбцов. Сопоставления столбцов задаются как кортеж из названия столбца первой таблицы, названия столбца второй таблицы и метрики схожести. По умолчанию для одной таблицы — тройки `(столбец, столбец, нормализованное расстояние Левенштейна)` для каждого столбца таблицы, для двух — тройки для каждой пары из декартова произведения атрибутов с нормализованным расстоянием Левенштейна как метрикой схожести.

Для сопоставлений столбцов пока что доступна единственная метрика — нормализованное расстояние Левенштейна. Пример использования в Python:

```
>>> import desbordante as desb
>>> hymd = desb.HyMD()
>>> matches = [(str(i), str(i), desb.LevenshteinSimilarity(0.45))
               for i in range(2, 7)]
>>> hymd.load_data(left_table=('adult.csv', ';', False))
>>> hymd.execute(column_matches=matches, prune_nondisjoint=False)
>>> mds = hymd.get_mds()
>>> for md in mds:
...     print(md)
...
[,,1,,]->1@1
[,1,,,]->2@1
[,0.5,0.5,,]->1@0.6
[,0.714286,0.5,,]->1@0.727273
[,0.75,0.5,,]->1@1
[,0.75,0.5,,]->2@1
[0.857143,,,,0.533333]->4@1
[0.857143,,,0.5,0.533333]->0@1
```

```
[0.857143,,,0.833333,0.533333]->3@1
[0.857143,,0.5,,0.533333]->0@1
[0.857143,,1,0.5,]->0@1
[0.857143,0.5,,,0.533333]->0@1
[0.857143,0.5,,0.5,]->0@1
[0.714286,,0.5,0.5,0.533333]->4@1
[0.714286,1,,0.5,0.533333]->4@1
```

Если нужно получить MD на двух таблицах, то вторую таблицу можно указать, передав в `load_data` ещё один параметр `right_table`.

Конструктор `LevenshteinSimilarity` имеет три параметра — минимальную схожесть, равенство NULL-значений, количество рассматриваемых границ решения LHS.

Вместо названий столбцов стоят их индексы, приведённые к строке, поскольку у переданной таблицы отсутствуют названия столбцов, а стандартные названия столбцов в Desbordante — их индексы.

В примере значения второго, третьего, четвёртого, пятого и шестого столбцов сопоставляются самим себе с помощью нормализованного расстояния Левенштейна с порогом схожести в 0.45.

Строковое представление зависимостей показывает найденные зависимости в формате *[[граница решения], ...] -> индекс@граница решения*, где в левой части находятся границы решения для сопоставлений столбцов левой части, а справа индекс сопоставления столбцов и граница решения правой части для него. Если в левой части граница решения пропущена, то она нулевая.

Таблица 1: Сравнение производительности реализаций HyMD

Реализация	flight	restaurant	adult	CORA
Java	178	0.7	26.3	225
C++	55	0.13	7.3	141

6. Эксперименты

В таблице 1 указаны результаты замеров производительности алгоритма HyMD. Система: Arch linux, glibc 2.38, gcc 13.2.1, 16 GiB RAM, Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz (800MHz–4500MHz, 6 cores, 12 threads).

Java имплементация запускалась на OpenJDK 21 с максимальным размером кучи в 8 ГБ. Использование других отличных от стандартных опций или других JDK не тестировались, поскольку были сочтены нецелесообразными. В [3] такие эксперименты уже проводились и они не привели к значительным улучшениям производительности.

Перед каждым измерением кэши сбрасывались путём записи 3 в `/proc/sys/vm/drop_caches`. С помощью `cpupower` на время экспериментов минимальная частота процессора была установлена в значение 4.5GHz. На время экспериментов также был выключен файл подкачки командой `swapoff -a`. Эксперименты для каждой реализации выполнялись после перезагрузки системы, без тяжеловесных посторонних процессов.

Измерялось время от начала выполнения алгоритма до его окончания. Каждое измерение повторялось 10 раз, в таблице приведены средние значения.

Для всех таблиц была использована стандартная конфигурация, где набор сопоставлений столбцов это тройка *(столбец, столбец, нормализованное расстояние Левенштейна)* для каждого столбца таблицы, минимальная поддержка — *размер таблицы + 1*, отсечение зависимостей с пересекающимися атрибутами включено, для всех метрик схожести минимальное значение — 0.7 и для левых частей используются все границы решения.

На всех таблицах удалось достичь ускорения. Ускорение варьируется от 1.6 (CORA) до 5.4 (restaurant) раз.

В случае набора данных CORA около 85% времени выполнения уходит на проверку наличия обобщений в решётке в реализации на C++. В Java реализации на эту операцию уходит около 90% времени работы. Возможно, такая маленькая разница в производительности объясняется тем, что хорошо работает JIT компилятор в Java.

Доля времени в работе же объясняется тем, что на этом наборе данных крайне много естественных границ решения и в конце выполнения, когда в решётке находится уже очень много MD, приходится проверять много левых частей. Сейчас метод `addIfMinimal` проверяет все обобщения левой части, переданной в него. При этом в этот метод на самом деле передаются только специализации MD, которые были ранее

в решётке, а значит не были специализациями, поэтому можно не проверять те левые части, которые являются обобщениями прошлой левой части.

Заключение

В ходе работы был реализован алгоритм поиска сопоставляющих зависимостей:

- Алгоритм HyMD исследован и интегрирован в Desbordante;
- Было получено ускорение в поиске MD;
- Алгоритм был связан с Python.

Ссылка на pull request: https://github.com/Mstrutov/Desbordante/pull/327.

Список литературы

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

[2] Strutovskiy Maxim, Bobrov Nikita, Smirnov Kirill, and Chernishev George. Desbordante: a Framework for Exploring Limits of Dependency Discovery Algorithms // 2021 29th Conference of Open Innovations Association (FRUCT). — 2021. — P. 344–354.

[3] Strutovskiy Maxim, Bobrov Nikita, Smirnov Kirill, and Chernishev George. Desbordante: a Framework for Exploring Limits of Dependency Discovery Algorithms // 2021 29th Conference of Open Innovations Association (FRUCT). — 2021. — P. 344–354.

[4] Schirmer Philipp, Papenbrock Thorsten, Koumarelas Ioannis, and Naumann Felix. Efficient Discovery of Matching Dependencies // ACM Transactions on Database Systems. — 2020. — Aug. — Vol. 45, no. 3. — P. 1–33. — Access mode: https://doi.org/10.1145/3392778.

[5] Koumarelas loannis, Papenbrock Thorsten, and Naumann Felix. MDedup: Duplicate Detection with Matching Dependencies // Proc. VLDB Endow. — 2020. — jan. — Vol. 13, no. 5. — P. 712–725. — Access mode: https://doi.org/10.14778/3377369.3377379.

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

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

[8] Шлёнских Алексей. Обзор сопоставляющих зависимостей и алгоритма их поиска HyMD. — 2023. — Access mode: https://github.com/Mstrutov/Desbordante/blob/main/docs/papers/HyMDreview-ShlyonskikhAlexey-2023spring.pdf (online; accessed: 2024-01-09).