Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 23М04-мм

Реализация алгоритмов для поиска и верификации приближённых зависимостей включения в рамках платформы Desbordante

Чижов Антон Игоревич

Отчёт по учебной практике
в форме «Решение»

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

Санкт-Петербург
2024

Оглавление

Введение

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

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

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

Зависимости включения могут [2] использоваться для обнаружения несогласованности данных и восстановления целостности. На практике поиск IND может применяться для ограниченного набора задач. Аналогично функциональным зависимостям, для которых определено множество зависимостей, ключевой идеей которых является задание метрики для подсчёта ошибки, для зависимостей включения определяются приближённые IND. Приближённые зависимости можно [5, 10] применять для большего набора задач, например для обнаружения отсутствующих ограничений, проблем с целостностью данных.

Desbordante [6] представляет собой научно-ориентированный профи-

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

В данный момент Desbordante активно развивается и требует расширения набора поддерживаемых примитивов, в том числе включая поддержку приближённых IND. В рамках предыдущей работы был выполнен краткий обзор предметной области и реализован алгоритм для поиска приближенных унарных IND, а также выбран алгоритм для n-арных зависимостей. Данный алгоритм является единственным известным на данный момент для поиска приближенных зависимостей.

Также, в рамках данной работы реализован алгоритм для верификации приближенных IND. Алгоритмы верификации позволяют пользователю исследовать конкретную зависимость. В случае, если зависимость является приближенной, тогда пользователь может получить подробную статистику о данной зависимости (наиболее важная информация — список кортежей, на которых зависимость не выполняется). Алгоритмы верификации можно использовать вместе с алгоритмами поиска зависимостей для того, чтобы автоматически находить приближенные зависимости, а затем получать более подробную статистику о них с помощью алгоритмов верификации. Данный пайплайн может быть удобен для анализа данных, исправления ошибок и т.д.

[1] https://github.com/Desbordante/desbordante-core

Постановка задачи

Целью данной работы является разработка новой функциональности для приближённых зависимостей включения с целью улучшения анализа данных. Для достижения этой цели были поставлены следующие задачи.

- Выполнить обзор алгоритма Mind для поиска приближенных IND и алгоритма верификации IND.
- Реализовать алгоритм Mind и алгоритм верификации IND в рамках платформы Desbordante.
- Предложить алгоритм для возможного совместного использования алгоритмов для создания пайплайна.
- Провести тестирование алгоритмов.

1 Обзор алгоритмов

В данной секции представлен обзор алгоритма для поиска приближенных n-арных зависимостей, а также вводится алгоритм для верификации зависимостей включения (далее, INDVerifier). В заключение рассматривается алгоритм для построения пайплайна на основе совместного использования данных алгоритмов.

1.1 Алгоритм Mind

Алгоритм Mind [11] является n-арным алгоритмом для поиска приближенных IND. Оригинальный алгоритм делится на:

- поиск унарных зависимостей;
- поиск n-арных зависимостей.

Фактически, вторая часть алгоритма не зависит от первой, поскольку для второй части алгоритма необходим исключительно список унарных зависимостей. Поэтому для поиска унарных зависимостей может использоваться любой алгоритм. В рамках данной работы используется модифицированный алгоритм поиска унарных зависимостей Spider [7], который подробно рассматривался в предыдущей работе.

```
1  # Input: d a database, and J_1 the set of unary INDs.
2  # Output: all INDs satisfied by d.
3  C_2 := gen_next(J_1);
4  i := 2;
5  while C_i != do
6    forall I ∈ C_i do
7      if d ⊨_ε I then
8        J_i := J_i ∪ {I};
9    C_{i+1} := gen_next(J_i);
10   i := i + 1;
11 end while
```

```
12 return ⋃_{j<i} 𝒥_j
```

Листинг 1: Алгоритм Mind.

Процесс проверки зависимостей осуществляется по уровням. Сначала формируется набор кандидатов, основываясь на зависимостях, найденных на предыдущем уровне. Затем происходит последовательная проверка кандидатов.

На Листинге 2 представлен алгоритм генерации кандидатов на языке SQL.

```
1  # Input: 𝒥_i, INDs of size i.
2  # Output: C_{i+1}, candidates of size i+1.
3  insert into C_{i+1}
4  select p.lhs.rel[p.lhs[1], p.lhs[2], ..., p.lhs[i], q.lhs[i]]
5         ⊆ p.rhs.rel[p.rhs[1], p.rhs[2], ..., p.rhs[i], q.rhs[i]]
6  from 𝒥_i p, 𝒥_i q
7  where p.lhs.rel = q.lhs.rel and p.rhs.rel = q.rhs.rel
8      and p.lhs[1] = q.lhs[1] and p.rhs[1] = q.rhs[1]
9      and ...
10     and p.lhs[i−1] = q.lhs[i−1] and p.rhs[i−1] = q.rhs[i−1]
11     and p.lhs[i] < q.lhs[i] and p.rhs[i] <> q.rhs[i]
12 for all I ∈ C_{i+1} do
13   for all J ≺ I and J of size i do
14     if J ∉ 𝒥_i then
15       C_{i+1} := C_{i+1} \ {I}
16     end if
17   end for
18 end for
```

Листинг 2: Алгоритм gen_next.

Для генерации кандидатов $i+1$ уровня перебираются всевозможные комбинации зависимостей i уровня. Зависимости должны отличаться только последним атрибутом. На завершающем этапе для каждого кандидата проверяется, что все его подзависимости размера i выполняют-

ся. На Листинге 3 представлен упрощенный (более высокоуровневый) алгоритм генерации кандидатов.

```
1  # Input: J_i, INDs of size i.
2  # Output: C_{i+1}, candidates of size i+1.
3  forall (p, q) ∈ J_i × J_i:
4      if not p.startswith(q):
5          continue
6
7      if not (p.lhs.last_column() < q.lhs.last_column() and
8              p.rhs.last_column() ≠ q.rhs.last_column()):
9              continue
10
11     candidate := create_candidate(p, q)
12     if can_prune_candidate(candidate, prev_raw_inds):
13         continue
14
15     C_{i+1} := C_{i+1} ∪ {candidate}
16 end for
```

Листинг 3: Упрощенный алгоритм gen_next.

Последняя часть алгоритма заключается в проверке валидности кандидата. В оригинальной статье [11] приводится следующий SQL запрос 4:

```
1 select X from r
2 where not exists (
3     select * from s
4     where r.A_1 = s.B_1 and ... and r.A_n = s.B_n
5 );
```

Листинг 4: Проверка точного кандидата.

Фактически, для проверки точной зависимости необходимо выполнить проекцию обоих отношений (в одном случае выполняется проекция на атрибуты, которые используются в левой части зависимости, в

другом — в правой части). Затем необходимо проверить, что все кортежи из левого отношения также есть и в правом отношении.

Для поиска приближенных зависимостей предлагается использовать следующие запросы 5:

```
1 # count error rows
2 select count(distinct r.A_1, ..., r.A_n) as disqualifying_rows
3 from r
4 where not exists (
5     select * from s
6     where r.A_1 = s.B_1 and ... and r.A_n = s.B_n
7 );
8 # calculate lhs cardinality
9 select count(distinct r.A_1, ..., r.A_n) from r;
```

Листинг 5: Проверка приближенного кандидата.

После чего для проверки валидности кандидата, достаточно поделить количество уникальных кортежей ($disqualifying\_rows$), которые не содержатся в правом отношении на мощность левого множества ($lhs\_cardinality$). Неформально, значение ошибки для зависимости включения можно определить как количество строк, которые нужно удалить из левого отношения для того (независимо от их количества появлений), чтобы получить датасет, на котором зависимость выполняется.

1.2 Алгоритм INDVerifier

Алгоритмы верификации зависимости включения предназначены для проверки конкретных зависимостей, задаваемых пользователем. В отличие от алгоритмов майнинга, которые автоматически выявляют зависимости в данных, здесь пользователь самостоятельно задает зависимость для проверки. При этом пользователь может получить подробную информацию об ошибке в случае, если зависимость не выполняется. Это полезно для анализа данных, поскольку предоставляет возможность понять, какие именно значения нарушают зависимость.

В отличие от большинства других алгоритмов верификации, в случае с IND может удерживаться как в рамках одного отношения, так и в рамках двух отношений. Наибольший интерес представляет именно проверка выполнения зависимости между двумя отношениями. Например, такая ситуация возникает, когда предполагается, что правая часть зависимости является ключом, а левая часть использует использует его значения. В этом контексте алгоритмы верификации обеспечивают возможность проверки, действительно ли заданное отношение выполняет необходимые условия, что имеет важное значение для обеспечения целостности данных.

```
1  # Input: d a database, and I - IND to check
2  # Output: stats - statistics about IND.
3
4  lhs_stream := get_projected_stream(d, I.lhs.table, I.lhs.indices)
5  rhs_stream := get_projected_stream(d, I.rhs.table, I.rhs.indices)
6  stats := init_stats()
7
8  forall lhs_row ∈ lhs_stream do
9      if lhs_row holds I in rhs_stream then
10       continue
11
12     stats := update_stats(stats, lhs_row)
13 end forall
14
15 return stats
```

Листинг 6: Алгоритм INDVerifier.

На Листинге 6 представлен алгоритм для верификации IND. Проверка валидности может выполняться аналогично тому, как она выполняется в алгоритме Mind, однако алгоритм также должен собирать статистку о строках, на которых зависимость не выполняется.

1.3 Совместное использование алгоритмов

Алгоритмы майнинга приближенных зависимостей включения и верификации приближенных зависимостей включения могут использоваться для автоматического анализа данных с помощью создания пайплайна:

- На первом этапе применяется алгоритм майнинга для поиска приближенных зависимостей с заданным порогом ошибки. Обнаруженные зависимости, значение ошибки которых достаточно низко, может говорить о возможных ошибках.
- На втором этапе применяется алгоритм верификации для верификации каждой приближенной зависимости, с целью получения кластеров (другими словами, строк, на которых зависимость нарушается.

```
1  Input: d a database, and ϵ − error threshold.
2  Output: Verified INDs with statistics.
3
4  inds := Mind(d, ϵ)
5
6  forall I ∈ inds do
7      if I.error = 0 then
8          continue
9
10     stats := INDVerifier(d, I)
11     # show exact error and statistics
12     show_stats(stats)
13 end for
```

Листинг 7: Пайплайн из алгоритмов Mind и INDVerifier

2 Реализация алгоритмов

2.1 Алгоритм Mind

Важно отметить, что алгоритм в оригинальной статье определяется в терминах реляционных баз данных. То есть, для генерации и проверки кандидатов используются SQL запросы. При этом данное решение имеет несколько недостатков:

- Некорректная работа в случае с нулевыми и пустыми значениями;
- Отсутствие возможности повторно использовать промежуточные вычисления для проверки кандидатов, из-за чего время проверки кандидатов может значительно меняться.

При реализации методов для генерации кандидатов и проверки их валидности использовались аналогичные алгоритмы без использования SQL.

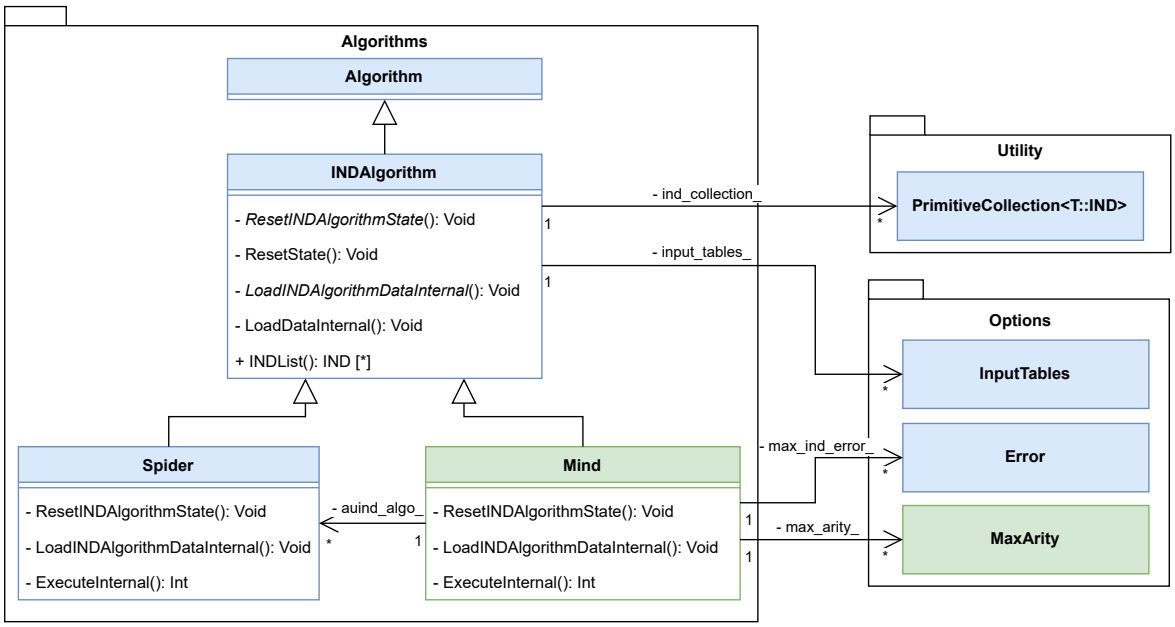

Рис. 1: Иерархия классов для алгоритма Mind.

На Рис. 1 представлена иерархия классов для алгоритма Mind. Алгоритм Mind использует алгоритм Spider для майнинга унарных зависимостей включения, а для поиска зависимостей большей размерности

применяется непосредственно алгоритм Mind. На каждом уровне выполняется генерация кандидатов, после чего происходит их проверка. Алгоритм генерации кандидатов является хорошо известным и не будет рассматриваться далее.

Наибольший интерес представляет реализация проверки валидности кандидатов. Алгоритм отличается в случае проверки точной зависимости включения и приближенной. На Листинге 8 представлен алгоритм для проверки точных зависимостей:

```
1  Input: d a database, I — IND candidate.
2  Output: true, if I holds, false otherwise.
3
4  lhs_stream := get_projected_stream(d, I.lhs.table, I.lhs.indices)
5  rhs_stream := get_projected_stream(d, I.rhs.table, I.rhs.indices)
6  rhs_hash_set := create_hash_set(rhs_stream)
7
8  forall lhs_row ∈ lhs_stream do
9      if lhs_row ∉ rhs_hash_set then
10         return false
11     end if
12 end forall
13
14 return true
```

Листинг 8: Алгоритм проверки кандидата IND

Сначала выполняется создание хэш таблицы для проекции правого отношения на $I.rhs.indices$. После чего выполняется проход по всем кортежам проекции левого отношения на $I.lhs.indices$.

На Листинге 9 представлен алгоритм для проверки приближенных зависимостей:

```
1  Input: d a database, I — IND candidate, ε — error threshold.
2  Output: (holds, error), where holds is true iff I holds.
3
4  lhs_stream := get_projected_stream(d, I.lhs.table, I.lhs.indices)
```

```
5  rhs_stream := get_projected_stream(d, I.rhs.table, I.rhs.indices)
6  rhs_hash_set := create_hash_set(rhs_stream)
7  lhs_hash_set := create_hash_set(lhs_stream)
8
9  lhs_cardinality := cardinality(lhs_hash_set)
10 disqualify_row_limit := ⌊lhs_cardinality * ϵ⌋ + 1
11 disqualify_row_count := 0
12
13 forall lhs_row ∈ lhs_hash_set do
14     if lhs_row ∉ rhs_hash_set then
15         disqualify_row_count := disqualify_row_count + 1
16         if disqualify_row_count == disqualify_row_limit then
17             return (false, None)
18         end if
19     end if
20 end forall
21
22 error := disqualify_row_count/lhs_cardinality
23 if error ≤ ϵ then
24     return (true, error)
25 else
26     return (false, None)
27 end if
```

Листинг 9: Алгоритм проверки кандидата AIND

В отличие от оригинального алгоритма, здесь используется раннее терминирование в случае, когда количество ошибок достигает лимита. Также, при проверке приближенной зависимости необходимо подсчитывать мощность левого спроецированного множества. Из-за чего хэш таблица создается не только для правого отношения, а также и для левого.

2.2 Алгоритм INDVerifier

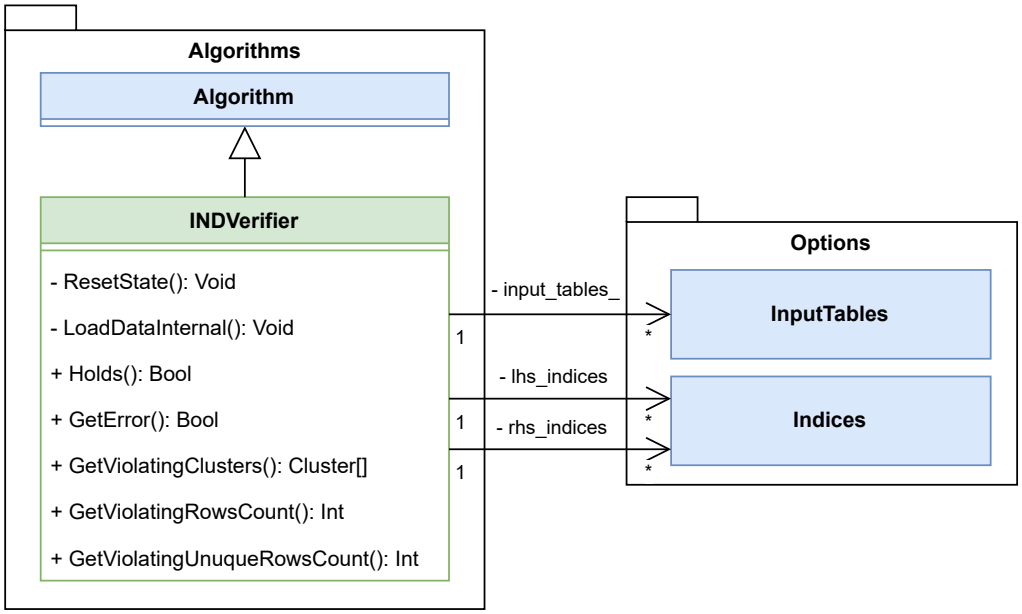

Рис. 2: Иерархия классов для алгоритма INDVerifier.

Иерархия классов представлена на Рис. 2. Алгоритм принимает одну/две таблицы, а также индексы левой части зависимости и индексы для правой части. Если указана одна таблица, то зависимость ищется в рамках одного отношения, в противном случае $lhs\_indices$ относятся к первой таблице, а $rhs\_indices$ — ко второй.

Для реализации алгоритма верификации IND используется схожий алгоритм валидации, как и в алгоритме Mind, о котором говорилось ранее. Однако, в данном случае также происходит сбор статистики о кластерах, на которых зависимость не выполняется. Верификатор возвращает список из кластеров, где кластер (т.е. набор индексов) соответствует одному уникальному значению в левой части отношения.

Для пользователя наибольший интерес представляют те зависимости, на которых зависимость практически полностью удерживается, то есть для которых верификатор находит мало кластеров.

Алгоритм INDVerifier реализован на языке C++, а также добавлен интерфейс к нему для взаимодействия через Python, подобно аналогичным алгоритмам верификации в Desbordante.

3 Пример совместного использования алгоритмов

В данной секции рассматривается пример использования пайплайна из алгоритма для поиска приближенных зависимостей и алгоритма верификации IND.

id	customer_id	product
1	101	Laptop
2	102	Phone
3	103	Tablet
4	104	Monitor
5	108	Keyboard
6	201	Mouse
7	102	Charger

Таблица 1: Orders

id	name	country
101	Alice	USA
102	Bob	UK
103	Charlie	Canada
104	David	Germany
105	Eve	France

Таблица 2: Customers

Для демонстрации примера используются таблицы *orders* 1 и *customers* 2. Рассмотрим скрипт для построения пайплайна на Рис. 10, написанный на языке Python с использованием библиотеки *desbordante*:

```
1  import desbordante
2  import pandas as pd
3
4  def print_clusters(algo, table, indices):
5      print("Number of clusters violating IND:",
6            f"{len(algo.get_violating_clusters())}")
7      for i, cluster in enumerate(algo.get_violating_clusters(),
8                                  start=1):
9          print(f"#{i} cluster:")
10         for el in cluster:
11             values = " ".join([f"{table[table.columns[idx]][el]}"
                   for idx in indices])
12             print(f"{el}: {values}")
13
```

```
14
15 TABLE_NAMES = ['orders.csv', 'customers.csv']
16 TABLES = [pd.read_csv(name) for name in TABLE_NAMES]
17 ERROR = 0.4
18
19 print(f"Run AIND algorithm with error threshold {ERROR}")
20 algo = desbordante.aind.algorithms.Default()
21 algo.load_data(tables=TABLES)
22 algo.execute(error=ERROR)
23
24 print("List of found AINDs:")
25 inds = algo.get_inds()
26 ainds = list(filter(lambda i: i.get_error() != 0.0, inds))
27 for aind in ainds:
28     print(aind)
29
30 print("Run AIND Verifier for each AIND:")
31 for aind in ainds:
32     (lhs_col, lhs_indices) = aind.get_lhs().to_index_tuple()
33     (rhs_col, rhs_indices) = aind.get_rhs().to_index_tuple()
34     lhs_table = TABLES[lhs_col]
35     rhs_table = TABLES[rhs_col]
36
37     verifier = desbordante.aind_verification.algorithms.Default()
38     verifier.load_data(tables=[lhs_table, rhs_table])
39     verifier.execute(lhs_indices=lhs_indices,
40                      rhs_indices=rhs_indices)
41     print("AIND:", aind)
42     print_clusters(verifier, lhs_table, lhs_indices)
43     print()
```

Листинг 10: Скрипт для совместно использования алгоритмов

В рамках пайплайна сначала происходит выполнение алгоритма для

поиска приближенных зависимостей (строки 20-22), а также для каждой приближенной зависимости запускается алгоритм верификации с выводом информации о кластерах (строки 31-43), на которых зависимость нарушается. На Листинге 11 представлен вывод после выполнения скрипта:

```
1  Run AIND algorithm with error threshold 0.4
2  List of found AINDs:
3  (orders, [customer_id]) -> (customers, [id]) with error = 0.33
4  (customers, [id]) -> (orders, [customer_id]) with error = 0.2
5
6  Run AIND Verifier for each AIND
7  AIND: (orders, [customer_id]) -> (customers, [id]) with error =
       0.33
8  Number of clusters violating IND: 2
9  #1 cluster:
10 5: 201
11 #2 cluster:
12 4: 108
13
14 AIND: (customers, [id]) -> (orders, [customer_id]) with error =
       0.2
15 Number of clusters violating IND: 1
16 #1 cluster:
17 4: 105
```

Листинг 11: Вывод скрипта при выполнении AIND алгоритма

Алгоритму майнинга приближенных зависимостей удалось найти две приближенные зависимости. Вторая зависимость не представляет интереса, поскольку *customers.id* является первичным ключом. Первая же зависимость должна удерживаться, поскольку в данном случае *orders.customer_id* является ссылкой на первичный ключ *customers.id*. Очевидно, что пятый и четвертый кортежи являются некорректными, поскольку они ссылаются на некорректный индекс.

4 Тестирование алгоритмов

Для тестирования алгоритма Mind переиспользованы уже существующие тесты для алгоритмов поиска IND. Для этого была выполнена реорганизация тестов с добавлением возможности запуска одинаковых тестов для всех алгоритмов майнинга IND. Кроме того, было добавлено 3 юнит-теста.

Для алгоритма верификации IND добавлено 14 тестов с проверкой корректности подсчета статистик.

Заключение

По итогам работы получены следующие результаты:

- Выполнен обзор алгоритма Mind для поиска приближенных IND и алгоритма верификации IND.
- Реализованы новые алгоритмы для работы с приближенными зависимостями включения, а именно:
  - алгоритм поиска приближенных IND Mind;
  - алгоритм для верификации приближенных IND INDVerifier;
  - добавлен интерфейс для алгоритма INDVeifier для взаимодействия через Python.
- Предложен алгоритм для создания пайплайна из алгоритмов с примером его работы.
- Выполнено тестирование алгоритмов, а именно:
  - адаптированы существующие тесты для IND алгоритмов;
  - добавлено 14 тестов для алгоритма верификации IND.

Код доступен на Github[2,3].

[2] https://github.com/Desbordante/desbordante-core/pull/401
[3] https://github.com/Desbordante/desbordante-core/pull/467

Список литературы

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

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

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

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

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

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

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

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

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

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

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