Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 22.Б08-мм

Реализация алгоритма верификации Denial Constraints в “Desbordante”

Аносов Павел Игоревич

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
асс. кафедры ИАС Г. А. Чернышев

Санкт-Петербург
2024

Оглавление

Введение

Desbordante [2] — это высокоэффективный наукоемкий профилировщик данных. Наукоёмкое профилирование нацелено на поиск сложных паттернов и зависимостей, которые невозможно обнаружить с использованием обычного профилирования.

Одним из важных видов таких паттернов являются запрещающие ограничения (Denial Constraints, DC). Они представляют из себя обобщение функциональных и условных функциональных зависимостей, зависимостей порядка и других, которыми не всегда удаётся обойтись при описании тех или иных зависимостей в данных. Запрещающие ограничения (ЗО) задают ограничения на комбинацию значений атрибутов которые не должны выполняться. Таким образом они позволяют находить неточности и ошибки в данных, такие как пропущенные значения, дубликаты и другие несоответствия.

На данный момент существуют различные алгоритмы для верификации ЗО — проверки выполнения заданной ЗО на множестве данных, и поиска ЗО — процесса нахождения конкретных ЗО, которые выполняются на заданных данных. Обе задачи нетривиальные и требуют комплексного подхода. В данной работе мы рассмотрим только задачу верификации ЗО и опишем добавление данной функциональности в профилировщик Desbordante.

1 Постановка задачи

Целью работы является реализация алгоритма для эффективной верификации особого вида зависимостей в данных — запрещающих ограничений, а также последующее добавление возможности верификации из Python при помощи библиотеки pybind11 [4]. Для достижения этой цели были поставлены следующие задачи:

1. Произвести обзор области верификации ЗО и выбрать наиболее подходящий алгоритм для данной задачи;
2. Реализовать класс алгоритма верификации ЗО в ядре системы (на C++);
3. Создать модульные тесты для проверки корректности работы алгоритма, а также соответствующий набор данных для тестов;
4. Сделать возможным использование верификации в Python;
5. Провести анализ нагрузочного тестирования алгоритма.

2 Обзор

Несмотря на то, что ЗО являются довольно молодой научной областью, тем не менее, в ней накопилось немало работ, описывающих алгоритмы верификации и поиска ЗО. Алгоритмы поиска в данной работе затронуты не будут, более подробную информацию и сравнительную характеристику о них можно найти в работе Ивана Морозко[1]. В данной разделе мы рассмотрим лишь алгоритмы верификации, они представляют большую ценность, так как позволяют проверять различные гипотезы о данных.

Перед началом работы необходимо было провести анализ предметной области для выявления алгоритма верификации ЗО который был бы наиболее эффективным и не требующим большого объема инфраструктуры, которую необходимо реализовать для его поддержки в программном коде.

В ходе изучения предметной области были найдены четыре работы, описывающие алгоритмы верификации ЗО, но у каждого из них были свои преимущества и недостатки.

1. Rapidash Verification [7] — поддержка только переменных гетерогенных и гомогенных ЗО (variable heterogeneous and homogeneous DC).

2. One-Pass Inconsistency Detection Algorithms [5] — поддержка переменных и константных ЗО (variable and constant DC), но только гомогенных (homogeneous).

3. FAst Constraint-based Error DeTector (FACET) [6] — поддержка только переменных гетерогенных и гомогенных ЗО (variable heterogeneous and homogeneous DC).

4. VioFinder [3] — поддержка только переменных гетерогенных и гомогенных ЗО (variable heterogeneous and homogeneous DC).

[1] https://github.com/Desbordante/desbordante-core/blob/main/docs/papers/FastADC%20-%20Ivan%20Morozko%20-%202024%20autumn.pdf

Определения константных и переменных ЗО будут приведены в следующем разделе.

Также были изучены работы, описывающие алгоритмы поиска, так как зачастую такие алгоритмы тоже применяют верификацию. К сожалению, все алгоритмы либо не применяли верификацию в явном виде, либо предполагали построение объемной инфраструктуры, что нам не подходит.

После изучения доступных вариантов алгоритмов верификации основой работы стала статья [7], так как, во-первых, гетерогенные ЗО представляют наибольшую ценность (такими ЗО, например, являются функциональные зависимости и зависимости порядка). А во-вторых, по заявлению авторов, алгоритм быстрее алгоритма FACET, представленного в статье [6], который в свою очередь быстрее алгоритма VioFinder, представленного в статье [3].

3 Предварительные сведения

Перед тем, как начать реализацию алгоритма, необходимо ввести основные определения и понятия. Подавляющее большинство определений взяты из статьи [7], исключения будут указаны явно.

Таблица 1: Основные понятия.

Понятие	**Значение**
R	таблица с исходными данными
$vars(R)$	конечное множество атрибутов (колонок) таблицы
$\|R\|$	количество строк в R
$t,\ s$	кортежи в R
$t.A$	значение атрибута A в кортеже t
предикат p	выражение вида: $s.A\ op\ t.B$, где $s,t \in R,\ A,\ B \in vars(R)$ и $op \in \{=\ ,\ \neq\ ,\ \geq\ ,\ >\ ,\ \leq\ ,\ <\}$
ЗО φ	конъюнкция предикатов: $\forall t,s \in R,\ t \neq s : \neg(p_1 \wedge \cdots \wedge p_m)$
$vars_{op}(\varphi)$	множество атрибутов гомогенного ЗО, для которых есть предикат с оператором op в ЗО

Определение 1. *Пара кортежей (s,t) будет считаться нарушением (violation), если все предикаты в φ верны.*

Определение 2. *φ удерживается (holds) на R, если нет нарушений.*

Определение 3. *Предикат (predicate) называется гомогенным (homogeneous), если он представим в виде $s.A$ op $t.A$ или $s.A$ op $a.B$*

Определение 4. *Предикат называется гетерогенным (heterogeneous), если его можно представить в виде: $s.A$ op $t.B$.*

Определение 5. *Гомогенный предикат вида $s.A$ op $t.A$ называется строчным гомогенным (row-level homogeneous) предикатом.*

Определение 6. *Согласно [1] ЗО называется константной (constant DC), если она содержит предикат вида: $s.A$ op c, где c — константа.*

Определение 7. *Согласно [1] ЗО называется переменной (variable DC), если $\forall p_i$ представим в виде $s.A$ op $t.B$.*

Определение 8. *Гомогенный предикат вида $s.A$ op $s.B$ называется колоночным гомогенным (column-level homogeneous) предикатом.*

Так как большинство ЗО (зависимости порядка, функциональные зависимости и пр.) содержат только строчные гомогенные предикаты, то для удобства будем называть гомогенным ЗО такое ЗО, которая содержит только строчные гомогенные предикаты.

Определение 9. *Если ЗО Содержит только строчные гомогенные предикаты, то такое ЗО — гомогенное (homogeneous DC).*

Определение 10. *Если ЗО содержит и строчные, и колоночные гомогенные предикаты, то такое ЗО — смешанная гомогенная (mixed homogeneous DC).*

Определение 11. *Гетерогенное ЗО (heterogeneous DC) — ЗО, содержащая любые типы предикатов.*

Не умаляя общности, будем считать, что каждая колонка в R находится в максимум одном предикате гомогенного ЗО.

Пример. В таблице 2 приведен набор данных, в котором содержится информация о налоговой ставке для жителей разных штатов США. Вот некоторые правила, которые выполнятся на данном наборе данных: (1) SSN — потенциальный ключ, (2) $Zip \rightarrow State$ — функциональная зависимость и (3) для всех жителей одного штата у человека с большей заработной платой больше налоговая ставка.

Таблица 2: Налоговые ставки для жителей разных штатов США.

	SSN	Zip	Salary	FedTaxRate	State
t1	100	10108	3000	20%	NewYork
t2	101	53703	5000	15%	Wisconsin
t3	102	53703	6000	20%	Wisconsin
t4	103	53703	4000	10%	Wisconsin

Все вышеперечисленные правила можно записать в терминах запрещающих ограничений следующим образом:

1. $\varphi_1 : \forall t, s \in R,\ t \neq s : \neg(s.SSN = t.SSN)$

2. $\varphi_2 : \forall t, s \in R,\ t \neq s : \neg(s.Zip = t.Zip \wedge s.State \neq t.State)$

3. $\varphi_3 : \forall t, s \in R,\ t \neq s : \neg(s.State = t.State \wedge s.Salary < t.Salary \wedge s.FedTaxRate > t.FedTaxRate)$

Все вышеперечисленные ЗО являются гомогенными, т.е. содержат строчные гомогенные предикаты. Также приведем пример гетерогенного ЗО: $\varphi_4 : \forall t, s \in R,\ t \neq s : \neg(s.Salary < t.FedTaxRate)$. Отметим, что $vars_{=}(\varphi_3) = \{State\}$, $vars_{<}(\varphi_3) = \{Salary\}$ и $vars_{>}(\varphi_3) = \{FedTaxRate\}$.

4 Алгоритм

В данном разделе будут описаны все необходимые для реализации алгоритма понятия и структуры данных, введенные авторами статьи [7], а также принцип работы самого алгоритма и его оптимизации.

4.1 Ортогональный поиск

Сперва авторы вводят понятие ортогонального поиска, необходимого для поиска нарушений при обработке кортежей в R.

Пусть $\mathbb{N}$ — множество с заданным на нем линейным порядком, $A \subseteq \mathbb{N}^k$, $k \geq 1$ размера N. Пусть $\mathbf{L} = (\ell_1, \ldots, \ell_k)$ и $\mathbf{U} = (u_1, \ldots, u_k)$ такие, что $\mathbf{L}, \mathbf{U} \in \mathbb{N}^k$ и $\ell_i \leq u_i \ \forall \ i = 1 \ldots k$.

Определение 12. *Ортогональный диапазон — это пара* $(\mathbf{L}, \mathbf{U})$*, образующая выровненный по осям гиперкуб в* k *измерениях (в двумерном случае это, соответственно, нижняя левая и правая верхняя точки).*

Поиск по ортогональному диапазону по множеству A обозначим как $Q(A)$, тогда он будет состоять из следующего множества:

$$Q(A) = \left\{ a \in A \mid \bigwedge_{i \in k} \ell_i \ \mathrm{op}_1 \ a_i \ \mathrm{op}_2 \ u_i \right\},$$

где операторы op_1 и $op_2 \in \{<, \leq\}$.

Другими словами, $Q(A)$ содержит все точки в A, которые лежат на гиперкубе или внутри него. Задачи поиска по ортогональному диапазону состоит в определении того, является ли $Q(A)$ пустым или нет.

Пример. Рассмотрим ортогональный поиск на примере таблицы 2. Пусть A — множество двумерных точек, полученных путем проекции исходной таблицы на атрибуты $Salary$ и $FedTaxRate$. Пусть $\mathbf{L} = (3500, 5)$ и $\mathbf{U} = (4500, 22)$. Тогда ортогональный запрос $(\mathbf{L}, \mathbf{U})$ вернет все точки такие, что $Salary$ находится в промежутке от 3500 до 4500 и $FedTaxRate$ в промежутке от 5 до 22. В исходной таблице только t_4 удовлетворяет заданным условиям. Таким образом результат ортогонального поиска $(\mathbf{L}, \mathbf{U})$ по таблице — множество из одной точки: $\{(4000, 10)\}$.

Будем предполагать, что структура для ортогонального поиска поддерживает следующие операции:

1. $booleanRangeSearch(\mathbf{L}, \mathbf{U})$: возвращает булево значение, если существует точка, находящаяся внутри параллельного осям ограничивающего гиперкуб, заданного $\mathbf{L}$ и $\mathbf{U}$.

2. insert(t): выполняет вставку k-мерного кортежа в структуру данных.

Существует два основных вида структур данных для ортогонального поиска: деревья диапазонов (range trees) и k-d деревья. В данной работе было отдано предпочтение k-d дереву, так как по количеству занимаемой памяти оно выигрывает в большее количество раз, чем дерево диапазонов выигрывает по времени. Кроме того, так как подходящих уже реализованных решений для ортогонального поиска на языке C++ не нашлось, выбор пал на k-d деревья из-за простоты в реализации.

4.2 Алгоритм верификации

В данной части мы приступим к непосредственному разбору алгоритма, представленного в статье [7]. Не умаляя общности, будем считать, что все предикаты содержат только операторы равенства ($=$) и неравенства ($<, \leq, >, \geq$), но не операторы не равно ($\neq$) и что ЗО — гомогенное. Позже эти ограничения будут сняты.

Алгоритм 1 описывает детали верификации гомогенного ЗО φ с заданной таблицей R. На строке 1 мы вычисляем k — количество колонок, которые находятся в предикатах с неравенством. Если φ содержит только предикаты с равенством, то $k = 0$.

Затем проходимся по всем кортежам из R. Для каждого кортежа t в R мы сначала проецируем его на все колонки, которые есть в предикатах с равенством, чтобы вычислить v — проекцию кортежа на колонки с равенством. Если v еще не встречалось ранее, то мы помещаем ее в хэш-таблицу H и инициализируем.

Algorithm 1: Верификация ЗО

Input: Таблица R, гомогенное ЗО ϕ
Output: True/False

k $\leftarrow|$ vars$(\phi)\setminus$ vars $_{=}(\phi)\mid$
$H \leftarrow$ пустая хэш-таблица
foreach $t \in R$ **do**
  $v \leftarrow \pi_{\text{vars }=(\phi)}(t)$
  if $v \notin H$ **then**
    if $k \neq 0$ **then**
      $H[v] \leftarrow$ new OrthogonalRangeSearch()
    else
      $H[v] \leftarrow 0$
  if $k \neq 0$ **then**
    L, U $\leftarrow$ SearchRange (t)
    $\mathbf{L}', \mathbf{U}' \leftarrow$ InvertRange $(\mathbf{L}, \mathbf{U})$ `/*` op_1 `и` op_2 `для booleanRangeSearch выбираются на основе` *op* `в предикатах с неравенством */`
    if $H[v]$.*booleanRangeSearch* $(\mathbf{L}, \mathbf{U}) \vee$ $H[v]$.*booleanRangeSearch* $(\mathbf{L}', \mathbf{U}')$ **then**
      return *false*
    $H[v]$.insert $\left(\pi_{\text{vars }(\phi)\setminus \text{vars}=(\phi)}(t)\right)$
  else
    $H[v] \leftarrow H[v] + 1$
    if $H[v] > 1$ **then**
      return *false*
return *true*
Procedure `SearchRange(`t`)`
  $\mathbf{L} \leftarrow (-\infty, \ldots, -\infty), \mathbf{U} \leftarrow (\infty, \ldots, \infty)$ `/*` **L** `и` **U** `оба размера` k, `индексированы предикатами неравенства` p_i. `*/`
  foreach *предикат* $p_i \in$ *предикаты неравенства* $\text{in}\,\phi$ **do**
    if $p_i.op$ *is* $< or \leq$ **then**
      $\mathbf{U}_i \leftarrow \min\{\mathbf{U}_i, \pi_{p_i.\text{col}}(t)\}$
    if $p_i.op$ *is* $> or \geq$ **then**
      $\mathbf{L}_i \leftarrow \max\{\mathbf{L}_i, \pi_{p_i.\text{col}}(t)\}$
    return $\mathbf{L}, \mathbf{U}$
Procedure `InvertRange(L, U)`
  $\text{U}' \leftarrow \text{L}, \text{L}' \leftarrow \text{U}$
  изменить $-\infty$ на ∞ и ∞ на $-\infty$ в $\mathbf{U}'$ и $\mathbf{L}'$ соответственно
  return L', U'

- Если ЗО содержит предикаты с неравенством (строки 10–15), то мы строим структуру для ортогонального поиска размерности k. Затем выполняем вставку k-мерной точки в дерево, т.е. кортежа, полученного проекцией t на колонки с неравенствами (строка 15). Но перед вставкой также необходимо проверить, что новая точка не будет удовлетворять всем предикатам с неравенством, т.е. образовывать нарушение с какой-либо точкой, уже находящейся в дереве по заданному ключу.

- Если ЗО содержит только предикаты с равенством (строки 16–19), то достаточно проверить, что существует два кортежа, чьи проекции равны, что гарантировало бы нарушение. Это выполнено с помощью счетчика в хэш-таблице, который увеличивается на единицу (строки 17–19).

Пример. Рассмотрим таблицу 2 и ЗО $\varphi_3 : (s.State = t.State \wedge s.Salary < t.Salary \wedge s.FedTaxRate > t.FedTaxRate)$, которая содержит один предикат с оператором равенства и два — с оператором неравенства.

Алгоритм 1 начнет с предикатов с равенством и поместит t_1 в хэш-таблицу ключ $t_1.State = NewYork$. Так как дерево поиска для данного ключа пусто, ортогональный поиск вернет значение $false$ и затем поместит $(t_1.Salary, t_1.FedTaxRate)$ в дерево. Затем обрабатывается t_2 и помещается в хэш-таблицу с другим ключом, так как $t_2.State = Wisconsin$.

Затем алгоритм выполняет ортогональный поиск, который снова возвращает $false$, так как дерево по данному ключу пусто. Потом выполняется вставка $(5000, 15)$ в дерево и обрабатывается новый кортеж. Когда обрабатывается t_3, он помещается в хэш-таблицу по тому же ключу, что и t_2, так как у них одинаковое значение в колонке $State$. Поэтому необходимо рассмотреть оставшиеся предикаты в ЗО, чтобы выяснить, будет ли t_3 образовывать нарушение с каким-либо кортежем по тому же ключу в таблице, обработанным раннее.

Нарушение может быть в двух случаях: 1) Если кортеж, уже нахо-

дящийся в дереве (t_2) имеет $Salary$ меньше чем 6000 и $FedTaxRate$ больше чем 20% или же 2) если у t_2 $Salary$ больше чем 6000, а $FedTaxRate$ меньше 20%.

Чтобы это выяснить, выполняется ортогональный поиск по прямому диапазону: $\mathbf{L} = (-\infty, 20)$ и $\mathbf{U} = (6000, \infty)$ (случай 1) и инвертированному — $\mathbf{L}' = (6000, -\infty)$ и $\mathbf{U}' = (\infty, 20)$ (случай 2). Так как t_2 не находится ни в одном из диапазонов, оба ортогональных поиска возвращают $false$, и затем выполняется вставка $(6000, 20)$ в дерево.

Наконец, обрабатывается t_4, который нужно поместить по тому же ключу, что и t_2 и t_3. Таким образом, выполняется ортогональный поиск: $\mathbf{L} = (-\infty, 10)$ и $\mathbf{U} = (4000, \infty)$ (и поиск по инвертированному диапазону $\mathbf{L}' = (4000, -\infty), \mathbf{U}' = (\infty, 10)$), но ни одна точка не лежит в указанных диапазонах. Оба ортогональных поиска возвращают $false$, и алгоритм завершает работу со значением $true$ (строка 20).

Чтобы продемонстрировать пример нарушения, рассмотрим таблицу 2 с измененным значением t_4 с $FedTaxRate = 22\%$. При обработке кортежа t_4 будет выполняться ортогональный поиск с $\mathbf{L} = (-\infty, 22), \mathbf{U} = (4000, \infty)$ и $\mathbf{L}' = (4000, -\infty), \mathbf{U}' = (\infty, 22)$. Тогда t_2 и t_3 будут образовывать нарушение с t_4 так как в обоих кортежах $Salary$ больше чем 4000, но $FedTaxRate$ меньше чем 22%, поэтому алгоритм вернет значение $false$ на строчке 14.

4.3 Обобщения и оптимизации

В данном разделе авторы постепенно убирают ограничения на ЗО, введенные раннее. Примеры и доказательства приведены в данной работе не будут, их можно найти в оригинальной статье [7].

4.3.1 Гетерогенные предикаты с неравенством

Сперва авторы статьи модифицируют алгоритм так, чтобы он мог обрабатывать гетерогенные предикаты, то есть предикаты вида $s.A\ op\ t.B$, где op либо $<, \leq, >$ либо $\geq$. Пусть φ — это ЗО, содержащее строчные гомогенные предикаты и некоторое количество гетерогенных предика-

Algorithm 2: Генерация диапазонов для ортогонального поиска при обработке очередного кортежа r с гетерогенным предикатом с неравенством

Input: Кортеж r из таблицы R, φ
Output: Прямой и инвертированный диапазоны

```
1  L ← (−∞, ..., −∞), U ← (∞, ..., ∞)   /* L, U, L', U' индексированы атрибутами R которые участвуют в предикатах с неравенствами */
2  L' ← (−∞, ..., −∞), U' ← (∞, ..., ∞)
3  Procedure CreateBothSearchRanges(r, phi)
4      foreach предикат с неравенством s.C op t.Dφ do
5          if op if ≤ or ≤ then
6              U.C ← min{U.C, r.D}
7              L'.D ← max{L'.D, r.C}
8          if op is > or ≥ then
9              L.C ← max{L.C, r.D}
10             U'.D ← min{U'.D, r.C}
11         return L, U, L', U'
```

тов с неравенством. Главное отличие от предыдущего случая состоит в том, что мы обобщаем вычисление диапазонов: для прямого — $(\mathbf{L}, \mathbf{U})$ и для инвертированного — $(\mathbf{L}', \mathbf{U}')$. В алгоритме 2 показано обобщенное вычисление диапазона. Главная идея состоит в том, что если в φ содержится предикат $s.C < t.D$, то когда мы обрабатываем новый кортеж r, верхней границей для атрибута C будет значение $r.D$ для прямого поиска, а нижней границей для атрибута D — $r.C$ для инвертированного поиска. Когда $C = D$ мы получаем исходный алгоритм. Заметим, что алгоритм теперь может обрабатывать случаи, когда атрибут содержится в нескольких предикатах. Таким образом, гетерогенный предикат с равенством $s.C = t.D$ может быть записан в виде: $s.C \leq t.D \wedge s.C \geq t.D$.

4.3.2 Предикаты с оператором не равно (disequality)

Любой предикат $s.A \neq t.B$ может быть записан в виде: $(s.A < t.B) \vee (s.A > t.B)$. Заметим, что $\neg(\varphi \wedge s.A \neq t.B) \iff \neg(\varphi \wedge s.A < t.B) \wedge \neg(\varphi \wedge s.A > t.B)$. Таким образом, ЗО, содержащее ℓ предикатов, может быть записано в равносильной форме как конъюнкция 2^ℓ ЗО, не содержащих

операторов не равно.

4.3.3 Только один предикат с неравенством (inequality)

Если ЗО содержит строчные гомогенные предикаты с равенством, и только один предикат (гомогенный или гетерогенный) с неравенством, то тогда верификацию можно выполнить за линейное время. В алгоритме 3 представлено, как это можно сделать. Как и предыдущем алгоритме мы начинаем разделять кортежи на множества с одинаковой проекцией ЗО на колонки с равенством. Пусть предикат с неравенством — $s.A\ op\ t.B$. Главная идея состоит в том, чтобы отслеживать минимум и максимум значений колонок A и B для каждого кортежа с соответствующими равными ключами в хэш-таблице.

Algorithm 3: Верификация ЗО, когда оно содержит строчные гомогенные предикаты с равенством и один предикат с неравенством.

Input: Таблица $\mathbf{R}, \phi$ содержащее только предикаты вида $s.C = t.C$ и один предикат p вида $s.A\ op\ t.B$
Output: True/False

$\min_A, \max_A, \min_B, \max_B \leftarrow$ пустая хэш-таблица
foreach $t \in R$ **do**
  $v \leftarrow \pi_{\text{vars} =(\phi)}(t)$
  if $v \notin \min_{\text{A}}$ **then**
    $\min_{\text{A}}[v], \min_{\text{B}}[v] \leftarrow +\infty, +\infty$
    $\max_A[v], \min_B[v] \leftarrow -\infty, -\infty$
  if $(p.op \in \{<, \leq\} \wedge \min_A[v]\ op\ t.B) \vee (p.op \in \{>, \geq\} \wedge \max_A[v]\ op\ t.B$ **then**
    return *false*
  if $(p.op \in \{<, \leq\} \wedge t.A\ op\ \max_B[v]) \vee (p.op \in \{>, \geq\} \wedge t.A\ op\ \min_B[v])$ **then**
    return *false*
  $\min_{\text{A}}[v] \leftarrow \min\{\min_{\text{A}}[v], t[\text{ A}]\}$
  $\min_{\text{B}}[v] \leftarrow \min\{\min_{\text{B}}[v], t[\text{ B}]\}$
  $\max_{\text{A}}[v] \leftarrow \max\{\max_{\text{A}}[v], t[\text{ A}]\}$
  $\max_{\text{B}}[v] \leftarrow \max\{\max_{\text{B}}[v], t[\text{ B}]\}$
return *true*

Когда мы встречаем кортеж, которого ещё нет в хэш-таблице, инициализируется минимум и максимум $+\infty$ и $-\infty$ соответственно (строки 4–6). Затем в строках 7–10 выполняется проверку между всеми кортежами (в хэш таблице по текущему ключу), встреченными раннее и новым кортежем. Если ни одна из проверок не выполняется, то обновляется минимум и максимум в хэш-таблице по текущему ключу (строки 11–14). Алгоритм проходит по всей таблице за один раз, что даёт временную сложность $O(|\mathbf{R}|)$. Хоть оптимизация и простая, но она может быть очень полезна: большинство популярных ограничений, таких как функциональные зависимости, содержат ровно один предикат с неравенством.

4.3.4 Смешанные гомогенные предикаты

И наконец, авторы разрешают смешанные гомогенные предикаты: теперь ЗО может содержать предикаты вида $s.A$ *op* $s.B$ и $s.A$ *op* $t.A$. Пусть $\forall s, t : \neg\varphi(s, t)$ — ЗО. Сначала переписывается φ в следующем виде: $\varphi_S(s) \wedge \varphi_T(t) \wedge \varphi_{ST}(s, t)$, где φ_S содержит все предикаты, которые содержат только s (и не t), φ_T содержит все предикаты, которые содержат только t (и не s), и φ_{ST}, которое содержит и s, и t. Тогда ЗО $\forall s, t : \neg\varphi$ может быть записано в равносильной форме:

$$\begin{aligned}\forall s, t : \neg\varphi &\Leftrightarrow \forall s, t : \neg\left(\varphi_S(s) \wedge \varphi_T(t)\right) \vee \neg\varphi_{ST}(s, t) \\ &\Leftrightarrow \forall s, t : \left(\varphi_S(s) \wedge \varphi_T(t)\right) \Rightarrow \neg\varphi_{ST}(s, t) \\ &\Leftrightarrow \forall s \in \mathbf{S} : \forall t \in \mathbf{T} : \neg\varphi_{ST}(s, t),\end{aligned}$$

где $\mathbf{S}$ — множество всех кортежей в $\mathbf{R}$, таких что φ_S верно, и $\mathbf{T}$ — множество всех кортежей в $\mathbf{R}$, таких что φ_T верно. Отметим, что $\mathbf{S}$ и $\mathbf{T}$ могут пересекаться.

Вводятся две структуры данных для ортогонального поиска (такие же, как и структура H в алгоритме 1) $H_{\mathbf{S}}$ и $H_{\mathbf{T}}$ для точек в $\mathbf{S}$ и $\mathbf{T}$ соответственно. Для каждого кортежа (то есть точки) $q \in \mathbf{R}$, сначала проверяется, принадлежит ли она $\mathbf{S}$ и $\mathbf{T}$ (то есть просто проверяется выполнимость набора предикатов на кортеже).

1. Если $q \in \mathbf{S}$, то выполняется ортогональный поиск по $H_{\mathbf{T}}$ чтобы найти какую-нибудь точку r такую, что $\varphi_{ST}(q, r)$ верно. Если находится такая точка, то зависимость не выполняется и алгоритм возвращает $false$. Если же ортогональный поиск вернул $false$ (не нашлось ни одной точки в ортогональном диапазоне, образующей нарушение с текущей точкой), то выполняется вставка q в $H_{\mathbf{S}}$.

2. Аналогично, если $q \in \mathbf{T}$, выполняется ортогональный поиск по H_{S}, чтобы найти точку r, такую, что $\varphi_{ST}(r, q)$ верно. Если такой точки нет, выполняется вставка q в H_{T}. Иначе алгоритм вернет $false$.

5 Реализация

5.1 k-d дерево

K-d дерево — это структура данных с разбиением пространства для упорядочивания точек в k-мерном пространстве. Единственное отличие от двоичного дерева поиска состоит в том, что сравнение на каждом из уровней дерева происходит по соответствующей компоненте точки.

Подходящих уже готовых реализаций не нашлось, поэтому необходимо было реализовать собственное k-d дерево, которое бы поддерживало необходимые методы для корректной работоспособности алгоритма, а также позволяло бы работать с шаблонными типами точек различной размерности.

Ниже приведены диаграммы 1, 2 реализованного класса алгоритма верификации, а также вспомогательных структур и классов.

1. KDTree — основной класс, которые будет использоваться для поиска и вставки точек.

2. Node — вложенный класс, который необходим для хранения самих точек и указателей на левое и правое поддеревья.

3. Rect — структура, которая представляет собой ортогональный диапазон для поиска по дереву.

4. bound_type — множество значений, обозначающих включение или исключение граничной точки ортогонального диапазона при поиске.

5.2 Алгоритм

Перед началом реализации алгоритма, важной задачей было написание хорошей и гибкой инфраструктуры. Она должна была бы описывать предикаты, участвующие в ЗО, операторы, операнды в предикатах и т.п. Большая часть этой инфраструктуры уже была реализована[2]

[2]https://github.com/ol-imorozko/Desbordante/commits/FastADC/

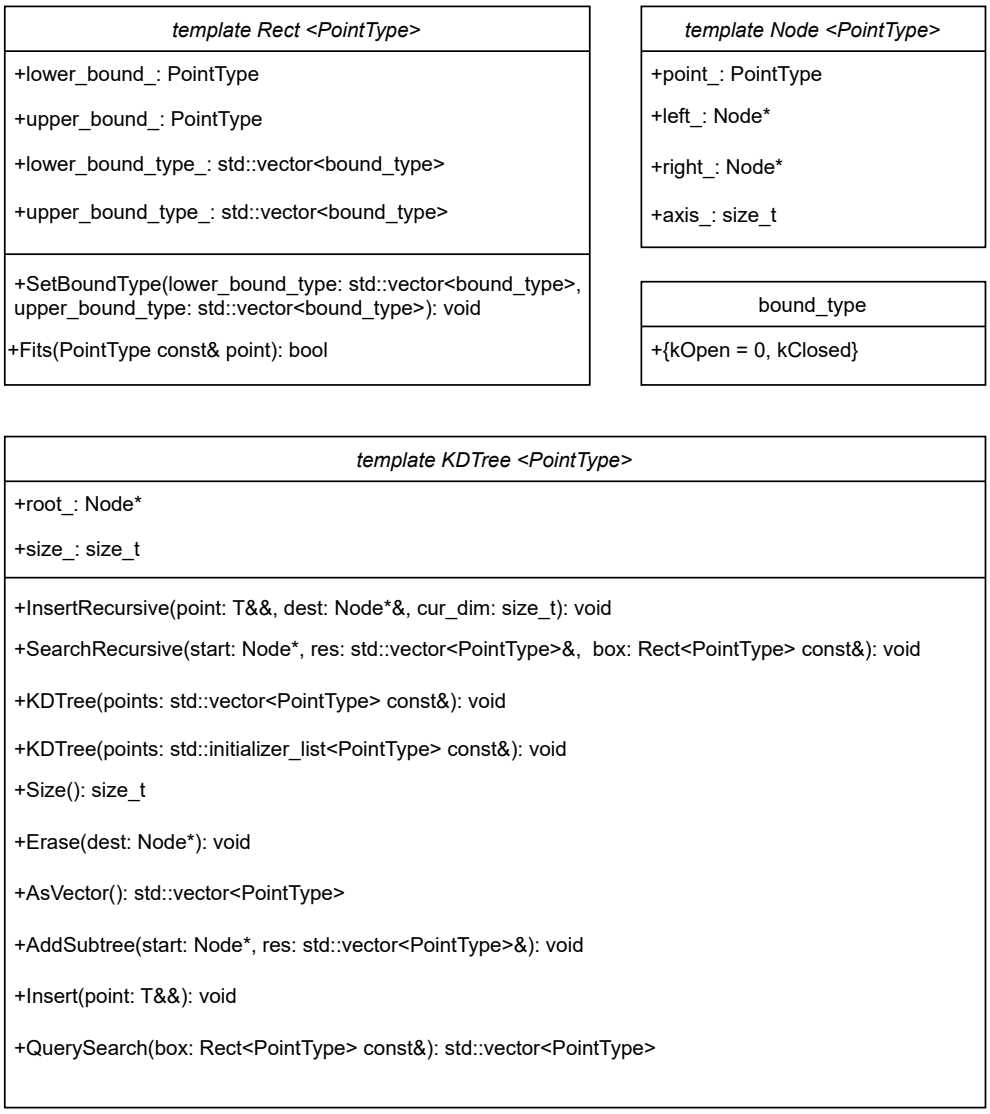

Рис. 1: Диаграмма k-d дерева.

Иваном Морозко, за что выражаю ему большую благодарность.

1. DCType — множество, классифицирующее входное ЗО.
2. DC — класс, описывающий входное ЗО.
3. DCVerifier — класс, выполняющий верификацию входного ЗО.
4. Operator — класс, описывающий оператор в предикате.
5. ColumnOperand — класс, представляющий левый и правый операнды в предикате.

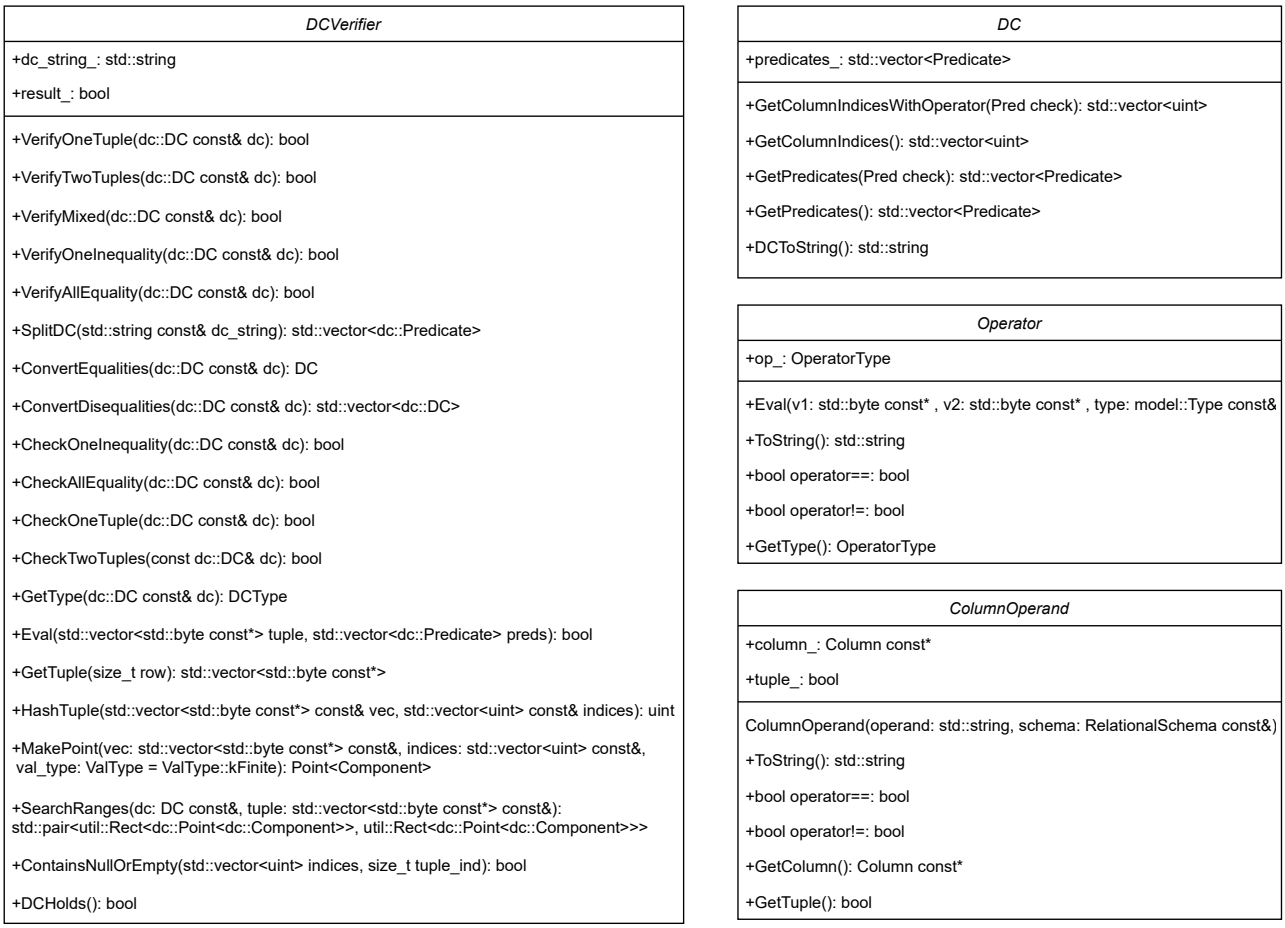

Рис. 2: Диаграмма класса алгоритма и вспомогательных классов.

6. Predicate — класс, описывающий предикаты в ЗО, состоящих и двух операндов и оператора.

7. Component — компонента точки.

8. Point — класс, представляющий точки, вставляемые в дерево поиска. Представляет собой массив компонент.

В ходе реализации алгоритма запрещающие ограничения были классифицированы на пять типов, для каждого из которых был написан свой эффективный метод верификации.

5.3 Pybind11

С помощью библиотеки pybind11 была добавлена возможность использования алгоритма верификации в Python. В качестве доступных опций алгоритм принимает указанное пользователем запрещающее ограничение.

Пример использования:
example.py:

```
import desbordante as db

# Denial Constraint
dc = "!(s.Salary < t.Salary and s.State = t.State and
s.FedTaxRate > t.FedTaxRate)"

# Creating a verificator and loading data in algortihm
verificator = db.dc_verification.algorithms.Default()
verificator.load_data(table=("TestDC1.csv", ',', True))
# Algorithm execution
verificator.execute(denial_constraint=dc)

# Obtaining the result
result: bool = verificator.dc_holds()

print("DC " + dc + " holds: " + str(result))
```

5.4 Тестирование

Во время работы были созданы два набора данных для тестирования и написано 18 тестов для отладки алгоритма, которые проверяют корректность работы алгоритма на различных типах запрещающих ограничений.

6 Эксперимент

В данном разделе описаны результаты проведенного нагрузочного тестирования проверки запрещающих ограничений в Python.

В качестве набора данных для тестирования были выбраны adult.csv (32 тысячи строк, 15 столбцов) и iowa1kk.csv (1 миллион строк, 23 столбца), так как в них присутствуют данные и числовых типов, и строчных.

В итоговое время проверки входит только проверка соответствующей зависимости, не включая время на загрузку данных из набора данных. В таблице 3 приведены результаты нагрузочного тестирования.

Таблица 3: Время проверки ЗО.

Набор данных	Запрещающее ограничение	Удерживается	Время (с)
adult.csv	$!(t.3 = s.3 \wedge t.4 \neq s.4 \wedge t.2 = s.2)$	Да	2.40
adult.csv	$!(t.3 = s.3 \wedge t.4 \neq s.4)$	Да	0.47
adult.csv	$!(t.11 = s.2 \wedge t.13 = s.7 \wedge s.5 = t.5)$	Да	5.42
adult.csv	$!(s.0 = s.2 \wedge s.2 >= s.4 \wedge t.1 = t.3)$	Да	0.17
iowa1kk.csv	$!(t.0 = s.0 \wedge t.2 = s.2 \wedge s.3 = t.3)$	Нет	0.56
iowa1kk.csv	$!(t.2 < t.8)$	Да	11.47

Так как алгоритм подразумевает прохождение по всем кортежам из исходной таблицы и проверку выполнения запрещающего ограничения для каждого кортежа, то чем больше размер входных данных, тем больше времени затратит алгоритм на проверку ЗО. Также нужно учитывать то, что для того чтобы узнать, удерживается ли ЗО, нужно обязательно пройтись по всем кортежам исходной таблицы, а для того, чтобы понять, что ЗО не удерживается, не всегда необходимо проверять всю таблицу, что верно для последних двух экспериментов в таблице 3.

Заключение

По итогам учебной практики стало возможным использование алгоритма верификации. По результатам выполнения работы:

1. Произведён обзор предметной области верификации и выбран наиболее подходящий алгоритм;
2. Реализован алгоритм верификации ЗО в ядре системы (на C++);
3. Созданы наборы данных и модульные тесты, проверяющие корректность работы алгоритма;
4. Были добавлены привязки для C++ методов с помощью pybind11, что позволило использовать верификацию в Python;
5. Проведено и проанализировано нагрузочное тестирование алгоритма.

Исходный код[3] доступен на GitHub, PR 444. Изменения на стадии рассмотрения.

[3]https://github.com/Desbordante/desbordante-core/pull/444

Список литературы

[1] Chu Xu, Ilyas Ihab F., Papotti Paolo. Discovering denial constraints // Proc. VLDB Endow. — 2013. — aug. — Vol. 6, no. 13. — P. 1498–1509. — URL: https://doi.org/10.14778/2536258.2536262.

[2] 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.

[3] Efficient Detection of Data Dependency Violations / Eduardo H. M. Pena, Edson R. Lucas Filho, Eduardo C. de Almeida, Felix Naumann // Proceedings of the 29th ACM International Conference on Information & Knowledge Management. — CIKM '20. — New York, NY, USA : Association for Computing Machinery, 2020. — P. 1235–1244. — URL: https://doi.org/10.1145/3340531.3412062.

[4] Jakob Wenzel, Rhinelander Jason, Moldovan Dean. pybind11 – Seamless operability between C++11 and Python. — 2017. — URL: https://github.com/pybind/pybind11.

[5] One-Pass Inconsistency Detection Algorithms for Big Data / Meifan Zhang, Hongzhi Wang, Jianzhong Li, Hong Gao // IEEE Access. — 2019. — Vol. 7. — P. 22377–22394.

[6] Pena Eduardo H. M., de Almeida Eduardo C., Naumann Felix. Fast detection of denial constraint violations // Proc. VLDB Endow. — 2021. — dec. — Vol. 15, no. 4. — P. 859–871. — URL: https://doi.org/10.14778/3503585.3503595.

[7] Rapidash: Efficient Detection of Constraint Violations / Zifan Liu, Shaleen Deep, Anna Fariha et al. // Proc. VLDB Endow. — 2024. — may. — Vol. 17, no. 8. — P. 2009–2021. — URL: https://doi.org/10.14778/3659437.3659454.