Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б11-мм

# Улучшение поддержки UCC в проекте Desbordante

***Демченко Артем Евгеньевич***

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
ассистент кафедры ИАС, Г. А. Чернышев

Санкт-Петербург
2023

# Оглавление

**Введение** **3**

**1. Постановка задачи** **5**

**2. Обзор** **6**

2.1. Определения . . . . . 6

2.2. Анализ изначального кода верификации UCC . . . . . 7

2.3. Реализация Pyro в Desbordante . . . . . 8

**3. Реализация** **10**

3.1. Изменение архитектуры верификации UCC . . . . . 10

3.2. Добавление тестов . . . . . 10

3.3. Исправление отдельных ошибок . . . . . 12

3.4. Реализация поиска AUCC . . . . . 12

**4. Эксперимент** **14**

4.1. Методология . . . . . 14

4.2. Сравнение PyroUCC и HyUCC . . . . . 14

4.3. Производительность Pyro . . . . . 15

**Заключение** **16**

**Список литературы** **17**

# Введение

Профилирование данных — это извлечение метаданных из заданного набора данных [1]. Этот процесс является неотъемлемой частью как и научного мира, так и IT-индустрии. Например, анализируя информацию о своих клиентах, компания может сделать выводы о выборе целевой аудитории при разработке продуктов. Существуют разные модели представления данных, дальше в работе будут рассматриваться только табличные данные.

Для профилирования табличных данных вводится понятие примитива — набора математических правил, описывающих отношения между некоторыми наборами данных таблицы. Примитив уникальная комбинация колонок (Unique Column Combination, UCC) — множество атрибутов таблицы, проекция на которые не содержит повторяющихся кортежей [4]. Алгоритмы, работающие с UCC, можно разделить на два типа: верификация и поиск. При верификации алгоритм получает на вход таблицу и множество атрибутов, для которого проверяется, является ли оно рассматриваемым примитивом в рамках данной таблицы. При поиске алгоритм получает на вход только таблицу и выделяет множество минимальных UCC. UCC называется минимальным, если при удалении из него любого непустого подмножества атрибутов результирующее множество перестает быть UCC [4]. Для любого UCC верно, что при добавлении в него нового атрибута оно остается UCC. Из этого следует, что любое множество атрибутов, являющееся UCC, можно получить из множества минимальных UCC. Алгоритмы верификации и поиска UCC используются при дедупликации и очистки данных.

В проекте Desbordante[1] — инструменте для профилирования данных с открытым исходным кодом — на момент написания работы был реализован алгоритм верификации UCC, который не был интегрирован в основную ветку репозитория, так как код не соответствовал стандартам разработки, принятым в проекте. Среди них основными являются

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

следование Google C++ Style Guide[2] и поддержка линейной и понятной для разработчика истории коммитов в системе контроля версий git. Отсутствие алгоритма верификации UCC в основной ветке проекта послужило мотивацией для рефакторинга существующего кода и адаптирования его под текущую версию проекта. Также в Desbordante реализован и интегрирован алгоритм Pyro для поиска минимальных приближенных функциональных зависимостей [6]. При этом в статье, где описывается этот алгоритм, указано, что его можно использовать для поиска минимальных UCC [3]. Это стало стимулом для реализации поиска данного примитива в проекте, максимально переиспользуя существующий код. Таким образом, была поставлена задача улучшить поддержку UCC в Desbordante, доработав существующий алгоритм верификации и адаптировав Pyro для поиска минимальных UCC.

[2] https://google.github.io/styleguide/cppguide.html

# 1 Постановка задачи

Цель работы — доработать реализацию алгоритма верификации UCC, чтобы его можно было интегрировать в основную ветку проекта, и адаптировать алгоритм Pyro для поиска минимальных UCC, тем самым улучшив поддержку примитива UCC в проекте Desbordante. Для достижения цели были поставлены следующие задачи:

1. произвести рефакторинг кода алгоритма верификации UCC по ревью, оставленным на его реализацию, по необходимости добавить новые тесты;
2. определить, какие компоненты для поиска минимальных UCC уже реализованы в алгоритме Pyro и разработать архитектуру для отдельного алгоритма, в котором будет переиспользоваться существующий код;
3. адаптировать алгоритм Pyro для поиска минимальных UCC;
4. сравнить производительность Pyro с другими алгоритмами поиска минимальных UCC, реализованными в Desbordante.

# 2 Обзор

## 2.1 Определения

Для всех определений из этого раздела будем использовать следующие обозначения [2]: $R = \{C_1, C_2, \ldots, C_m\}$ — схема отношения с $m$ атрибутами, где $C_i$ — сами атрибуты; $r \subseteq C_1 \times C_2 \times \cdots \times C_m$ — тело отношения. Следующие два определения взяты из статьи [2].

**Определение 1.** Множество атрибутов $K \subseteq R$ называется уникальной комбинацией столбцов (UCC) тогда и только тогда, когда

$$\forall t_1, t_2 \in r : (t_1 \neq t_2 \Rightarrow t_1[K] \neq t_2[K])$$

Если множество атрибутов X не является UCC, то мы получаем, что

$$\exists t_1, t_2 \in r : (t_1 \neq t_2 \wedge t_1[K] = t_2[K])$$

В таком случае будем говорить, что такие $t_1, t_2$ нарушают UCC.

**Определение 2.** UCC называется минимальным тогда и только тогда, когда

$$\forall K' \subset K : (\exists t_1, t_2 \in r : (t_1[K'] = t_2[K'] \wedge (t_1 \neq t_2))$$

**Пример 1.** *В таблице 1 множество атрибутов {Год рождения} не является UCC, третий и пятый кортежи нарушают UCC. А множество {Имя, Год рождения} является UCC.*

**Пример 2.** *В таблице 1 {Имя, Год рождения} не является минимальным UCC, так как {Имя} ⊂ {Имя, Год рождения} и {Имя} является UCC, в то время как {Имя} будет являться минимальным UCC.*

Для определенных выше примитивов мы не допускаем ни одного нарушения заданного отношения. Реальные данные, полученные в результате различных измерений, могут содержать неточности, опечатки, ошибки. В таком случае, если мы предполагаем, что семантически множество атрибутов $X$ должно являться UCC, но таблица содержит

Таблица 1: Данные для примеров

| Имя | Год рождения | Почтовый индекс | Город |
|---|---|---|---|
| Артем | 1990 | 125167 | Москва |
| Кирилл | 1981 | 117452 | Москва |
| Юлия | 1985 | 191014 | Санкт-Петербург |
| Ксения | 1999 | 198335 | Санкт-Петербург |
| Александр | 1985 | 420000 | Казань |

несколько ошибочных кортежей, то алгоритмы поиска точных UCC не выявят его как UCC. Чтобы учитывать такие ситуации, были введены приближенные уникальные комбинации столбцов (Approximate UCC, AUCC). Следующие два определения взяты из статьи [3].

**Определение 3.** Ошибкой кандидата $X$ называется величина

$$e(X, r) = \frac{|\{(t_1, t_2) \in r^2 : t_1 \neq t_2 \wedge t_1[X] = t_2[X]\}|}{|r|^2 - |r|}$$

Будем говорить, что множество атрибутов $X$ является AUCC на $r$ с ошибкой $e_{max}$, если $e(X, r) \leq e_{max}$.

**Пример 3.** *Пользуемся таблицей 1. Пусть $X$ ={Год рождения}, тогда* $e(X, r) = \frac{2}{5^2-5} = 0.1$.

Для AUCC также верно, что $e(X, r) \geq e(X \cup A, r)$ для множества атрибутов $X$ и атрибута $A$ [3]. Пользуясь этим, введем определение минимального AUCC.

**Определение 4.** AUCC с ошибкой $e_{max}$ называется минимальным, если при удалении из него любого непустого подмножества атрибутов ошибка результирующего множества будет строго больше чем $e_{max}$.

## 2.2 Анализ изначального кода верификации UCC

В изначальном коде было реализовано два класса — `UCCVerifier` и `StatsCalculator`. Объект класса `StatsCalculator` был приватным

полем класса `UCCVerifier` и хранил данные, характеризующие результаты работы алгоритма (например, номера кортежей, которые нарушают UCC). `StatsCalculator` имел методы, возвращающие результаты работы алгоритма, и метод `PrintStatistics(...)`, осуществляющий вывод этих результатов в консоль. `UCCVerifier` отвечал непосредственно за работу алгоритма. Основные публичные методы этого класса — `Execute(...)`, `LoadData(...)` и несколько методов, возвращающие результат работы алгоритма, которые при вызове вызывали соответствующий метод класса `StatsCalculator`.

Изначальный код не проходил тесты CI, а именно проверку статическим анализатором clang-format[3], большинство коммитов не соответствовали правилам, принятым в проекте. У некоторых функций были неправильные описания в комментариях. Также имена переменных не всегда отражали их назначения. Присутствовали ошибки, специфичные для языка C++, например, подключение заголовочных файлов там, где этого не требовалось. Также стоит отметить, что тестирование алгоритма происходило только на маленькой таблице (восемь строк и пять столбцов). Все эти проблемы были отражены в ревью и нуждались в исправлении.

## 2.3 Реализация Pyro в Desbordante

Алгоритм Pyro в Desbordante был реализован с целью поиска минимальных приближенных функциональных зависимостей (AFD) [6]. Это примитив для профилирования табличных данных, не рассматриваемый в рамках данной работы, поэтому мы ограничимся только самыми важными определениями. Будем считать, что функциональная зависимость (FD) $X \to Y$ удерживается, если из равенства любых двух кортежей на множестве атрибутов $X$ следует их равенство на множестве атрибутов $Y$. $X$ называют левой частью FD, а $Y$ — правой. Дальше будем рассматривать FD с одним атрибутом в правой части, обозначим его $A$. Для таких FD, как и для UCC, вводится некоторая функция

[3] https://clang.llvm.org/docs/ClangFormat.html

ошибки $e(X \to A, r)$ и считается, что AFD с ошибкой $e_{max}$ удерживается, если $e(X \to A, r) \leq e_{max}$. AFD $X \to A$ с ошибкой $e_{max}$ называется минимальной, если при удалении из множества $X$ любого непустого подмножества атрибутов ошибка результирующей зависимости $e$ будет строго больше чем $e_{max}$.

Заметим, что любое множество атрибутов X является кандидатом как на AUCC, так и на AFD, разве что во втором случае нам нужно добавить один атрибут в качестве правой части функциональной зависимости. Благодаря этому факту Pyro можно использовать как для поиска AFD, так и для поиска AUCC, а также для поиска обоих примитивов за один запуск алгоритма [3]. Таким образом, Pyro принимает таблицу и заранее заданную величину наибольшей возможной ошибки $e_{max}$ и осуществляет поиск приближенных зависимостей (AFD и/или AUCC), ошибка которых не превосходит $e_{max}$. То, какие зависимости будут искаться, зависит от количества созданных пространств поиска (класс `SearchSpace`) для обхода решетки возможных кандидатов. Решетка возможных кандидатов — всевозможные подмножества множества атрибутов, обход которых осуществляется алгоритмом в определенном порядке [3]. В рамках нашей задачи устройство решетки и порядок обхода не имеет значения. Чтобы найти минимальные AUCC, Pyro достаточно создать одно пространство поиска. Чтобы найти минимальные AFD, Pyro создает $m$ пространств поиска, где $m$ — количество атрибутов таблицы (для AFD каждое пространство поиска характеризуется атрибутом, которое будет в правой части функциональной зависимости). В изначальной реализации в Desbordante создавалось $m + 1$ пространств поиска, то есть осуществлялся поиск как AFD, так и AUCC. Однако в классе `Pyro` не было публичных методов, которые возвращали бы найденные AUCC.

# 3 Реализация

## 3.1 Изменение архитектуры верификации UCC

Было решено удалить класс `StatsCalculator`, так как его метод `PrintStatistics(...)` никак не использовался в проекте, а нужная алгоритму логика содержалась только в методе `CalculateStatistics(...)`. Поэтому поля для хранения результатов, а именно `num_error_rows_` и `err_clusters_` были перенесены в класс `UCCVerifier` и переименованы. Также в класс `UCCVerifier` был перенесен метод `CalculateStatistics(...)`. Методы получения результатов работы алгоритма теперь напрямую реализованы в `UCCVerifier`.

## 3.2 Добавление тестов

Изначально класс тестировался только на таблице с восемью строками и пятью столбцами, нужно было добавить более реальный тестовый сценарий. Код основной логики теста можно увидеть в листинге 1. Для тестирования используется реализованный в Desbordante алгоритм HyUCC [5] и уже существующие в проекте датасеты. Фреймворк для написания тестов — googletest[4]. С помощью алгоритма HyUCC находятся все минимальные UCC (`mined_uccs`). Затем для каждого `current_ucc` из `mined_uccs` запускается алгоритм верификации UCC (то есть методы класса `UCCVerifier`, это происходит в функции `CreateAndExecuteUCCVerifier(...)`) и проверяется, что `UCCVerifier` определил множество колонок как UCC. Следующим шагом для каждого `current_ucc` из `mined_uccs` проверяется, что при удалении одного столбца из `current_ucc` `UCCVerifier` определяет получившееся множество столбцов как не-UCC. Весь этот сценарий повторяется для трех разных датасетов.

[4] https://github.com/google/googletest

**Листинг 1: Основная логика тестирования публичных методов класса UCCVerifier на неразмеченных датасетах**

```
UCCVerifierWithHyUCCParams const& p(GetParam());

algos::StdParamsMap hyucc_params_map = p.GetHyUCCParamsMap();
std::list<model::UCC> const& mined_uccs = hyucc->UCCList();

// run ucc_verifier on each UCC from mined_ucc (UCC must hold)
for (auto const& current_ucc : mined_uccs) {
    auto verifier = CreateAndExecuteUCCVerifier(
                        p.GetUCCVerifierParamsMap(),
                        current_ucc.GetColumnIndicesAsVector());
    EXPECT_TRUE(verifier->UCCHolds());
}

// Case to prevent false positives
// run ucc_verifier on each UCC from mined_ucc with one column
   index removed
// (UCC must not hold because HyUCC returns minimal UCCs)
for (auto const& current_ucc : mined_uccs) {
    std::vector<unsigned int> current_ucc_vec =
            current_ucc.GetColumnIndicesAsVector();
    if (current_ucc_vec.size() < 2) {
        continue;
    }
    current_ucc_vec.pop_back();
    auto verifier = CreateAndExecuteUCCVerifier(
                        p.GetUCCVerifierParamsMap(),
                        std::move(current_ucc_vec));
    EXPECT_FALSE(verifier->UCCHolds());
}
```

## 3.3 Исправление отдельных ошибок

Также были внесены следующие изменения:

1. форматирование кода;
2. исправление названий публичных методов, например, `GetNumErrorRows(...)` заменен на `GetNumRowsViolatingUCC(...)`;
3. удаление включений ненужных заголовочных файлов;
4. изменение истории коммитов, чтобы каждый коммит соответствовал общему стилю наименования и имел название, отражающее его содержание.

## 3.4 Реализация поиска AUCC

В проекте каждый класс, реализующий алгоритм, наследуется от базового класса `Algorithm`. При этом каждый класс осуществляет поиск или верификацию только одного примитива. По этой причине было рассмотрено два решения. Первое — добавить методы возврата найденных AUCC в уже существующий класс `Pyro` и использовать его в качестве приватного поля в двух новых классах, которые будут отвечать за поиск AFD и AUCC, но это решение не было использовано в окончательной версии из-за архитектуры существующих классов. Второе решение — оставить класс `Pyro` без влияющих на его структуру изменений и реализовать класс `PyroUCC`, который бы наследовался от `UCCAlgorithm`. Это решение было применено на практике. На рисунке 1 частично представлена иерархия классов. Желтым цветом выделен метод, который был изменен, красным цветом выделен добавленный класс. Из метода `ExecuteInternal(...)` класса `Pyro` было удалено пространство поиска, которое отвечало за поиск AUCC. Общие компоненты, которые используют классы `Pyro` и `PyroUCC`, были вынесены в отдельную директорию `pyrocommon`.

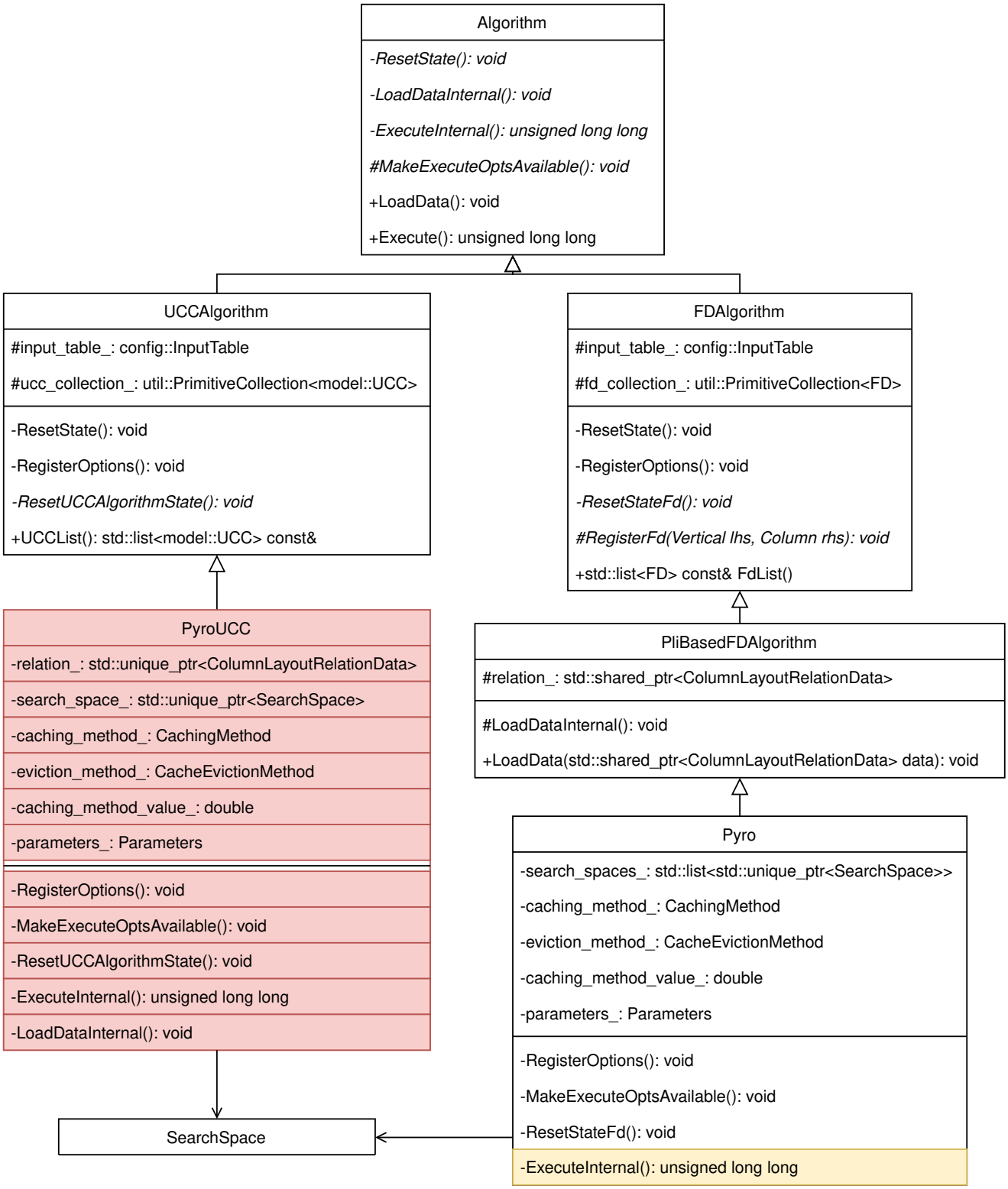


Рис. 1: Иерархия классов

# 4 Эксперимент

## 4.1 Методология

В этом разделе описаны условия, при которых проходили измерения в следующих двух разделах. Процессор — 11th Gen Intel(R) Core(TM) i5-11500 @ 2.70GHz (12 ядер), оперативная память 16ГБ. Операционная система — GNU/Linux, дистрибутив — Ubuntu 22.04.3 LTS, версия gcc 11.4.0. Также для всех сценариев была выставлена максимальная частота процессора, а именно 4.60GHz, с помощью команды `cpufreq-set`. Для тестирования были взяты датасеты из проекта Desbordante: 1 — breast_cancer, 2 — CIPublicHighway, 3 — EpicMeds, 4 — EpicVitals, 5 — LegacyPayors, 6 — neighbors100k, 7 — iowa1kk. Для тестирования каждого алгоритма он запускался на этих датасетах 15 раз, после чего строился доверительный интервал с уровнем доверия 0.95 для каждого датасета. Под временем работы алгоритма в этом разделе понимается время работы метода `Execute(...)`(т.е. число, которое возвращает этот метод). Таким образом, мы не учитываем затраты ресурсов на загрузку данных.

## 4.2 Сравнение PyroUCC и HyUCC

При $e_{max} = 0$ поиск минимальных AUCC сводится к поиску минимальных UCC. Поэтому мы можем проанализировать, какой из алгоритмов поиска минимальных UCC, реализованных в Desbordante, эффективнее. Было проведено сравнение времени работы алгоритмов HyUCC и PyroUCC (PyroUCC запускался с опцией $e_{max} = 0$).

Таблица 2: Сравнение производительности PyroUCC и HyUCC в мс

| Датасет | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| PyroUCC | 17 | 84 | 3064 | 1916 | 162 | 16 | 867 |
| HyUCC | 2 | 607 | 1515 | 1003 | 804 | 52 | 2110 |

В таблице 2 все приведенные значения имеют относительную погрешность не более 2%. В ходе анализа полученных результатов было

уставлено, что алгоритмы показывают разную производительность в зависимости от конкретных данных, поэтому какой-либо общей закономерности выявлено не было.

## 4.3 Производительность Pyro

Так как в ходе работы реализация одного из методов класса `Pyro` была изменена, есть смысл проверить, не ухудшилась ли производительность новой реализации. Датасет 7 (iowa1kk) не был использован для измерений, так как на машине не хватило оперативной памяти для первого тестового запуска алгоритма на этом датасете.

Таблица 3: Сравнение новой и старой версии Pyro в мс

| Датасет | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Новая | $467 \pm 3$ | $1205 \pm 70$ | $11028 \pm 678$ | $2845 \pm 61$ | $274 \pm 8$ | $42 \pm 3$ |
| Старая | $577 \pm 1$ | $1156 \pm 46$ | $11659 \pm 1106$ | $3202 \pm 139$ | $343 \pm 8$ | $49 \pm 3$ |

Можем убедиться, что новая реализация не отразилась негативно на производительности алгоритма. Доверительные интервалы либо пересекаются, либо верхняя граница для новой версии меньше нижней границы для старой.

# Заключение

В ходе данной работы основная цель была достигнута — код верификации UCC был доработан и интегрирован в проект, а также алгоритм Pyro был адаптирован для поиска минимальных приближенных UCC. В итоге в проекте улучшилась поддержка примитива UCC. Результаты:

1. Произведен рефакторинг кода верификации UCC и добавлена логика тестирования для неразмеченных данных.
2. Предложена архитектура алгоритма для поиска минимальных приближенных UCC на основе архитектуры Pyro.
3. Реализован класс для поиска минимальных приближенных UCC с переиспользованием уже реализованных компонентов, частично изменена реализация Pyro для поиска приближенных FD.
4. Измерена производительность нового алгоритма, по сравнению с HyUCC результаты измерений не показали никаких закономерностей. Также показано, что новая реализация Pyro для приближенных FD по производительности не хуже старой.

Код доступен по ссылкам:

1. рефакторинг верификации UCC[5]
2. поиск приближенных UCC[6]

Возможная дальнейшая работа:

1. детально проанализировать алгоритмы PyroUCC и HyUCC с целью определения причины расхождений в производительности;
2. реализовать верификацию приближенных UCC.

[5]https://github.com/Mstrutov/Desbordante/pull/290
[6]https://github.com/Mstrutov/Desbordante/pull/324

# 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] Abedjan Ziawasch, Naumann Felix. Advancing the discovery of unique column combinations. — 2011. — 09. — P. 1565–1570.

[3] Kruse Sebastian, Naumann Felix. Efficient Discovery of Approximate Dependencies // Proc. VLDB Endow. — 2018. — mar. — Vol. 11, no. 7. — P. 759–772. — URL: https://doi.org/10.14778/3192965.3192968.

[4] Papenbrock Thorsten, Naumann Felix. A Hybrid Approach for Efficient Unique Column Combination Discovery // Datenbanksysteme für Business, Technologie und Web. — 2017. — URL: https://api.semanticscholar.org/CorpusID:38902150.

[5] Полынцов М. А. Реализация алгоритма поиска UCC в рамках платформы Desbordante. — URL: https://github.com/Mstrutov/Desbordante/blob/main/docs/papers/HyUCC%20-%20Michael%20Polyntsov%20-%20BA%20thesis.pdf (дата обращения: 19 декабря 2023 г.).

[6] Струтовский М. А. Реализация современных алгоритмов для поиска зависимостей в базах данных. — URL: https://github.com/Mstrutov/Desbordante/blob/main/docs/papers/Pyro%20-%20Maxim%20Strutovsky%20-%202020%20spring.pdf (дата обращения: 19 декабря 2023 г.).