Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 25.М41-мм

Разработка нового представления зависимостей в проекте Desbordante

ШЛЁНСКИХ Алексей Анатольевич

Отчёт по учебной практике
в форме «Решение»

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

Санкт-Петербург
2026

Оглавление

Введение

Профилирование данных есть процесс сбора метаданных. Часто такими данными являются некоторые простые статистики, такие как количество значений, среднее значение в столбце, но также ими могут быть и различные типы более сложных закономерностей [1]. Один из основных таких типов — функциональная зависимость.

Иногда такие зависимости нужно искать на данных. Однако задачи их поиска NP-трудны. Тем не менее, на практике эти задачи решаются благодаря особенностям реальных данных и алгоритмам, которые их используют. Часто эти алгоритмы реализуются в языках, быстрых для разработки, но медленных в скорости исполнения (Java, Python).

Однако для расширения применимости важно и качество кода, исполняемого компьютером. Проект Desbordante [2] начинался как эксперимент с целью проверки, насколько можно будет улучшить производительность таких алгоритмов, использовав для их написания C++ [3].

Такая структура предполагала очень ограниченное применение вида “запуск исполняемого файла–вывод результата”. Со временем позиционирование проекта поменялось, и к нему были добавлена привязка к Python.

Из-за этого возникли различные проблемы, такие как:

- Сложности с временем жизни объектов;
- Большая сложность корректной реализации сериализации и сравнения объектов зависимостей;
- Сложность корректного написания алгоритмов.

Кроме того, в проекте на данный момент отсутствует документация, поэтому подходы, вызывающие данные проблемы, копируются и в новый код.

В данной работе делается шаг к исправлению ситуации, путём разработки нового способа представления закономерностей в коде, учитывающего требования к проекту на данный момент. Этот способ реализуется для функциональных зависимостей.

1. Постановка задачи

Целью работы является обзор текущих проблем в Desbordante [2], связанных с устаревшим способом представления закономерностей, и разработка способов, учитывающих требования к проекту в настоящем. Для её выполнения были поставлены следующие задачи:

1. Выполнить анализ проблем проекта, возникающих из-за способа хранения закономерностей;
2. Предложить вариант дизайна, который их избегает;
3. Произвести экспериментальную реализацию этого дизайна.

2. Обзор

В проекте Desbordante реализованы алгоритмы, связанные с различными закономерностями в данных. Далее будет представлен один из типов таких закономерностей.

2.1. Функциональная зависимость

На практике функциональная зависимость означает, что имея некоторые значения из записи в таблице, можно точно определить значения некоторых других значений в этой записи.

Более формально для определения вводится некоторое отношение. Набор его атрибутов далее будем обозначать как R. Через r обозначим его экземпляр (неформально это соответствует некоторой таблице). Экземпляр является множеством записей. Запись p — это функция $p : R \to D$, D — некоторое множество, через $p[A]$ обозначим значение записи в атрибуте $A \in R$, формально это применение функции $p(A)$.

Для функциональных зависимостей используется несколько похожих определений. Представим здесь одно из них.

Определение 1 ***Функциональная зависимость*** *(далее FD)* $X \to Y$ *является парой наборов атрибутов* (X, Y) *некоторого отношения* R*, то есть* $X, Y \subseteq R$*. Мы говорим, что FD* $X \to Y$ ***соблюдается*** *на* r *(*$FdHolds[(X, Y), r]$*), если:*

$$\forall (p, q) \in (r \times r)(\forall B \in X \ p[B] = q[B] \implies \forall A \in Y \ p[A] = q[A]).$$

Будем называть X — **левой частью** (LHS), Y — **правой частью** (RHS).

Задача поиска зависимостей (dependency discovery, также иногда упоминается такое название, как data mining) на некотором наборе данных является задачей поиска некоторого множества соблюдающихся на этих данных зависимостей, которые "представляют" все соблюдающиеся зависимости на них. Например, для FD очевидно, что если соблюдается $X \to Y$, то будет соблюдаться и $(X \cup A) \to Y$ $(A \in R)$, вне

зависимости от данных, поэтому зависимости, которые так выводятся, можно не указывать в ответе.

В случае задачи поиска допустимо также использовать упрощённое представление FD:

Факт 1 $FdHolds[(X, Y), r] \iff \forall A \in Y \ FdHolds[(X, \{A\}), r]$.

Факт означает, что при возвращении ответа в задаче поиска допустимо использовать представление FD, где в правой части находится только один атрибут.

Помимо задачи поиска существует также задача проверки соблюдения (validation). В этой задаче вводными выступает набор данных и некоторая зависимость, и нужно дать ответ, соблюдается ли она. Кроме этого может быть полезна и некоторая информация о том, как именно она не соблюдается.

2.2. Проект Desbordante, контекст

В разделе выше был упомянут специальный вид ответа для задачи поиска. Он нужен, поскольку если представлять ответ как множество всех соблюдающихся зависимостей, он во многих случаях будет слишком большим.

Например, если таблица состоит из одной строки и 64 столбцов, то на ней соблюдается 2^{128} FD (все возможные подмножества в LHS и RHS). Если представлять ответ как множество всех соблюдающихся FD, то для такого ответа не будет достаточно памяти. Но в упомянутом специальном виде в ответ будет достаточно включить только одну зависимость: $\emptyset \to R$ — остальные выводятся без обращения к данным. Если используются только FD с одним атрибутом в правой части, то на указанных данных ответ будет выглядеть как $\{d | \exists A \in R \quad d = \emptyset \to A\}$, размер этого множества — $|R| = 64$.

Тем не менее, максимальный размер даже такого уменьшенного представления ответа растёт экспоненциально от количества атрибутов [5], поэтому в реализации стоит обращать внимание на вопросы, связанные с памятью.

Кроме того, сама задача является NP-трудной. Однако это не значит, что она совсем не решаема — реальные данные имеют некоторые особенности, как в виде ответа, так и в распределении значений, которые позволяют проектировать алгоритмы так, что решение часто можно получить относительно быстро.

Помимо этого, более качественный код, выполняемый компьютером, также может позволить улучшить производительность алгоритмов. Однако многие эталонные реализации алгоритмов из статей, их предлагающих, написаны на языках, условно предпочитающих производительности скорость разработки (Java, Python) [2].

Платформа Desbordante изначально была разработана для того, чтобы определить, насколько можно улучшить производительность, реализовав алгоритмы на C++ [3]. Изначально проект предполагал простой вариант работы — "запустить исполняемый файл–получить ответ в текстовом виде".

Однако со временем позиционирование проекта изменилось, его ключевой особенностью стала коллекция разных алгоритмов для поиска и проверки зависимостей, собранных в одном месте.

Проект теперь имеет три части. Сами алгоритмы реализованы в основной части библиотеки, которая в проекте именуется "ядром" (core), расположенной по пути `src/core`. Алгоритмы ядра тестируются в части тестов, располагающейся в `src/tests`.

Ядро проекта было связано с Python с помощью `pybind11` [4]. Эта часть кода называется "связкой с Python" (Python bindings) и находится по пути `src/python_bindings`. Для снижения входного порога было добавлено множество примеров использования[1].

Форма Python библиотеки предполагает множество вариантов использования, под которые предыдущая архитектура была не приспособлена.

В дополнение к сказанному, у проекта отсутствует документация, множество структур данных пишется ad-hoc, а значительная часть проекта написана студентами младших курсов без опыта работы в подоб-

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

ных проектах — имеет место копирование подходов, даже когда они не соответствуют текущим требованиям к проекту.

Исправление архитектуры и процессов в целом займёт очень много времени и ресурсов, это задача на будущее. В данной работе рассматривается только исправление проблем в одной конкретной области — представление закономерностей в коде. Для правильного проектирования представлений далее будут идентифицированы требования к библиотеке, важные для этой задачи.

2.3. Требования к библиотеке

2.3.1. Время жизни объекта

Одна из многих черт Python, отличающая его от C++ — автоматическое управление памятью. Это позволяет не беспокоиться о том, сколько существует объект, что, в свою очередь, порождает другие ожидания от библиотек языка.

К примеру, контракт использования итератора `std::vector` в C++ предполагает, что при его разыменовании коллекция не была удалена, иначе возникает неопределённое поведение. В Python в аналогичной ситуации итератор бы просто продлевал время жизни коллекции.

В отношении объектов, представляющих закономерности, это означает, что они должны существовать самостоятельно, их время жизни и возможность их использовать не должны зависеть от того, что происходит в других объектах. Иначе придётся прибегать к различного рода ухищрениям в связывающем коде, которые могут сильно ограничивать пользователя и которые сложно реализовать корректно.

2.3.2. Сериализация

В случае простого рабочего процесса “запуск–вывод результата” нет необходимости иметь возможность сохранять зависимости на диск. Однако для некоторых вариантов использования, например для профилировщика данных на основе этой библиотеки Reflejo [6], это упрощает

работу.

Реализация данного требования заметно проще, если объекты состоят из простых частей (строки, числа) и не имеют, например, циклов указателей.

2.3.3. Совместимость со стандартными инструментами

Python имеет такие встроенные коллекции, как `list`, `set`, `dict`. Использование последних двух предполагает наличие у объектов специальных методов, а именно проверки на равенство (`__eq__`) и получения хэша (`__hash__`). Для `list` некоторые методы, как например `list.index`, предполагают возможность сравнивать объекты (нужно реализовать метод `__eq__`). Функция `sorted` предполагает наличие метода “меньше” (`__le__`).

Такие методы тоже проще реализовать, если объект состоит из простых частей и в нём отсутствуют сложные конфигурации ссылок.

2.3.4. Вывод для человека

Найденные закономерности может быть полезно выводить, будь то для поиска ошибок в программе или для исследования найденного.

Типовой вариант вывода для первого случая:

```
print(dependencies)
```

Для второго случая:

```
for pattern in patterns:
    print(pattern)
```

В первом случае у каждой зависимости вызывается метод `__repr__`, во втором — `__str__`. Оба являются методами приведения к строке.

2.3.5. Приемлемое потребление ресурсов

Специфика задачи поиска зависимостей предполагает очень большие ответы, в худшем случае экспоненциально растущие в размере в

зависимости от параметров исходных данных. Эти ответы нужно хранить как можно более компактно, при этом позволяя пользователю работать с ними относительно удобно.

Также сами задачи поиска являются NP-трудными, поэтому библиотека должна оптимизировать как потребление памяти, так и скорость работы алгоритмов, в частности избегая лишнего копирования.

2.3.6. Ожидания пользователя

Создание объекта и `__repr__`. В Python существует также конвенция насчёт метода `__repr__` — он должен возвращать строку, с помощью которой можно будет создать объект в коде. Объекты зависимостей обычно простые по структуре — их можно представить как наборы строк и чисел, это не, например, объекты для управления ограниченными системными ресурсами наподобие принтера, которые логично получать только с помощью специальных функций, — поэтому пользователь будет ожидать, что сможет их создать.

Соотношения между типами. Как поиск, так и проверка соблюдения работают с закономерностями. Со стороны пользователя разумно предполагать, что в алгоритмах, решающих эти задачи, представления закономерностей будут одинаковыми, или хотя бы схожими и конвертируемыми. Например, чтобы можно было легко модифицировать зависимость, и передать её алгоритму проверки. Это может пригодиться в сценарии исследования данных — проверить, почему зависимость, которая кажется правильной, не соблюдается фактически.

Использование названий столбцов. При работе с таблицами ожидается, что столбцы можно будет указывать по названию, а не только по индексу. Такое использование возможно, например, для методов DataFrame[2].

[2]https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.html

2.4. Текущее состояние

Представление закономерностей на момент выполнения работы частично соответствует указанным требованиям: объекты результатов имеют самостоятельное время жизни, их можно сериализовывать, по крайней мере некоторые из них имеют реализации специальных методов для Python, они приводятся к строке, и используются приёмы для экономии памяти.

Однако даже это соответствие требованиям достигалось путём изменений поверх существующих представлений, что привело к неоптимальным решениям. Далее рассмотрим объекты, используемые при работе с представлениями FD, чтобы проиллюстрировать проблемы.

2.4.1. `RelationalSchema`

`RelationalSchema` — объект схемы таблицы. Хранит в себе список столбцов и название таблицы[3].

```
class RelationalSchema {
private:
    std::vector<std::unique_ptr<Column>> columns_;
    std::string name_;
```

Название таблицы хранится в `std::string`. Столбцы хранятся в `std::unique_ptr<Column>`. `std::unique_ptr` использовался с 2021 года, до этого использовался `std::shared_ptr`[4].

2.4.2. `Column`

`Column` — объект столбца в некоторой таблице[5].

```
class Column {
private:
    std::string name_;
```

[3] см. src/core/model/table/relational_schema.h

[4] см. коммит 7d285f716c0a81e0c43be8331e5199f08624141c

[5] см. src/core/model/table/column.h

```
IndexType index_;
RelationalSchema const* schema_;
```

Название столбца также хранится разумно — в `std::string`. А вот `schema_` — это рекурсивная ссылка на объект схемы, в котором данный столбец хранится, `index_` — это индекс столбца в `schema_->columns_`. Из-за этого схему приходится конструировать вызовами методов, а потом не изменять на протяжении всего использования алгоритма[6]:

```
schema_ = std::make_unique<RelationalSchema>(
    input_data.GetRelationName());
for (ColumnIndex col_idx = 0; col_idx < num_columns;
     ++col_idx) {
    auto column = Column(schema_.get() /*!!*/,
        input_data.GetColumnName(col_idx), col_idx);
    schema_->AppendColumn(std::move(column)); /*!!*/
}
```

Кроме того, такое хранение не позволяет сериализовать столбец сам по себе, ведь тогда нужно каким-то образом корректно обработать указатель на схему, которым он не владеет, а значит и удалять не может. Некорректная обработка может привести к утечкам памяти. Из-за этого приходится организовывать сериализацию более сложным образом, который будет описан далее.

При копировании `Column` будет скопировано и название столбца. Этот класс используется в реализации алгоритмов. Обратим внимание, что для алгоритмов поиска FD не важно, как называются столбцы, они должны проверять только равенство значений, ведь для определения FD не важно, какими именно являются сами атрибуты. То есть использование этого класса вносит риск излишнего копирования, которое может порождать такие неожиданности, как зависимость времени исполнения алгоритма от длины названия столбца таблицы.

[6] см. пример в src/core/algorithms/ind/faida/preprocessing/abstract_column_store.cpp

2.4.3. Vertical

`Vertical` — выборка столбцов из некоторой схемы.

```
class Vertical {
private:
    boost::dynamic_bitset<> column_indices_;
    RelationalSchema const* schema_;
```

Здесь хранится указатель на схему и битовая маска её столбцов. Не владеющий указатель также осложняет сериализацию.

Не владеющий указатель на схему используется для получения строкового представления этой выборки[7]:

```
std::string Vertical::ToString() const {
    std::string result = "[";
    /*...*/
    for (size_t index = /* итерация по column_indices_ */) {
        result += schema_->GetColumn(index)->GetName() /*!!*/;
        if (column_indices_.find_next(index) !=
            boost::dynamic_bitset<>::npos) {
            result += ' ';
        }
    }

    result += ']';

    return result;
}
```

Ещё раз заметим, что алгоритмы поиска функциональных зависимостей не используют и не должны использовать названия столбцов. При этом многие из них используют именно `Vertical` в своей реализации. Фактически им нужна только битовая маска. Эта идея будет использована далее.

[7] см. src/core/model/table/vertical.cpp

2.4.4. FD

FD[8] — объект, представляющий функциональную зависимость в Desbordante.

```
class FD {
private:
    Vertical lhs_;
    Column rhs_;
    std::shared_ptr<RelationalSchema const> schema_;
```

Сама функциональная зависимость представлена в варианте, где RHS представлена одним столбцом. LHS представлена выборкой из схемы. Объект закономерности владеет схемой в `schema_`.

Контекст, в котором создаётся этот объект, — представление результатов поиска FD. В такой ситуации этих объектов создаётся очень много, но таблица, для которой они актуальны, всего одна. Поэтому одним объектом схемы владеют все созданные объекты `FD`, что позволяет использовать меньше памяти. Таким образом, каждый объект `FD` является самостоятельной единицей с собственным временем жизни, но схема не копируется во все из них.

Для того, чтобы FD можно было поместить в `set` в `Python`, для связывающего кода был реализован метод проверки на равенство.

Из-за рекурсивности устройства `Column` проверку на равенство не получилось реализовать с помощью операторов `operator==`, ведь тогда надо было бы либо сравнивать содержащиеся в них `schema_`, дополнительно избегая бесконечной рекурсии, либо учитывать внешние условия — `Column` сравнивается только как часть объектов результатов. Последнее было бы сложно для понимания и хрупко. Поэтому `FD` сравниваются путём приведения их к кортежу из имён столбцов, их составляющих[9], и сравнения этих кортежей[10]:

```
py::class_<FD>(fd_module, "FD")
```

[8] см. src/core/algorithms/fd/fd.h
[9] см. src/core/algorithms/fd/fd.cpp
[10] см. src/python_bindings/fd/bind_fd.cpp

```
    /*...*/
    // TODO: implement proper equality check for FD
    .def("__eq__", [](FD const& fd1, FD const& fd2) {
        return fd1.ToNameTuple() == fd2.ToNameTuple();
    })

tuple<vector<string>, string> FD::ToNameTuple() const {
    std::tuple<std::vector<std::string>, std::string> name_tuple;
    auto& [lhs_names, rhs_name] = name_tuple;
    std::vector<Column const*> const& lhs_columns =
        lhs_.GetColumns();
    lhs_names.reserve(lhs_columns.size());
    for (Column const* column : lhs_columns) {
        lhs_names.push_back(column->GetName());
    }
    rhs_name = rhs_.GetName();
    return name_tuple;
}
```

Это не до конца корректно, поскольку в таблицах могут присутствовать столбцы с одинаковыми названиями, но на практике эта ситуация редка.

`FD::schema_` владеет указателем в составляющих частях `FD`, а именно в `lhs_.schema_` и `rhs_.schema_`. Устройство такое для того, чтобы можно было корректно сериализовывать объекты результата.

Сериализация происходит на стороне привязки к Python[11] с помощью механизмов pickle[12]:

```
py::class_<FD>(fd_module, "FD")
    /*...*/
    .def(py::pickle(
            // __getstate__
```

[11] см. src/python_bindings/fd/bind_fd.cpp

[12] https://docs.python.org/3/library/pickle.html

```
            [](FD const& fd) {
                /*...*/
            },
            // __setstate__
            [](py::tuple t) {
                /*...*/
            }));
```

При сериализации `Column` и `Vertical` схема не запоминается:

```
// __getstate__
[](FD const& fd) {
    py::tuple schema_state =
        table_serialization::SerializeRelationalSchema(
            fd.GetSchema().get());
    py::tuple lhs_state =
        table_serialization::SerializeVertical(fd.GetLhs());
    py::tuple rhs_state =
        table_serialization::SerializeColumn(fd.GetRhs());
    return py::make_tuple(std::move(schema_state),
                          std::move(lhs_state),
                          std::move(rhs_state));
},

py::tuple SerializeVertical(Vertical const& v) {
    std::vector<unsigned int> idx_vec =
        v.GetColumnIndicesAsVector();
    return py::make_tuple(std::move(idx_vec) /*schema_??*/);
}

py::tuple SerializeColumn(Column const& c) {
    return py::make_tuple(c.GetName(), c.GetIndex(),
                          /*schema_??*/);
}
```

Указатель на схему для составляющих частей FD получается только при десериализации целой FD:

```
// __setstate__
[](py::tuple t) {
    /*...*/
    std::shared_ptr<RelationalSchema const> schema =
            table_serialization::DeserializeRelationalSchema(
                    t[0].cast<py::tuple>());
    Vertical lhs = table_serialization::DeserializeVertical(
            t[1].cast<py::tuple>(), schema.get() /*!!*/);
    Column rhs = table_serialization::DeserializeColumn(
        t[2].cast<py::tuple>(), schema.get() /*!!*/);
    return FD(lhs, rhs, std::move(schema));
}
```

Такой метод усложняет понимание кода. Кроме того, в объектах закономерностей, использующих любой из этих классов, придётся прибегать к подобной схеме для корректности сериализации. В условиях, когда в проекте нет документации, такой код будет копироваться, а неопытность программистов будет дополнительно увеличивать вероятность ошибок. Данное устройство является неудовлетворительным с точки зрения поддерживаемости кода, особенно в условиях, в которых развивается данный проект, и требует замены.

Подобные проблемы актуальны и для других объектов представления результатов, однако в этой работе будет разработан подход только для исправления FD, поскольку объём работ очень велик — Desbordante поддерживает более 15 типов закономерностей. Рассуждения и результаты этой работы можно будет использовать и для исправления устройства остальных объектов закономерностей.

2.4.5. Проверка соблюдения

Помимо поиска FD, Desbordante также предоставляет функциональность по проверке их соблюдения.

Пример проверки соблюдения FD:

```
# прочитать данные
data = pd.read_csv(table, header=[0])
# создать алгоритм
algo = desbordante.afd_verification.algorithms.Default()
# загрузить данные в алгоритм
algo.load_data(table=data)
# выполнить проверку соблюдения FD из столбца с индексом 0
# в столбец с индексом 2
algo.execute(lhs_indices=[0], rhs_indices=[2])
```

Указать название столбца здесь нельзя и никакой связи с объектом FD здесь нет. Когда столбцов много, чтобы правильно задать индекс, нужно, к примеру, вызвать метод, что неудобно и противоречит ожиданиям пользователей (см. раздел 2.3.6).

2.4.6. FdAlgorithm

FDAlgorithm — класс, от которого наследуются все алгоритмы поиска FD. Этот класс реализует методы для хранения FD.

```
class FDAlgorithm : public Algorithm {
protected:
    config::MaxLhsType max_lhs_;

    /* Collection of all discovered FDs
     * Every FD mining algorithm should place discovered
     * dependecies here. Don't add new FDs by accessing this
     * field directly, use RegisterFd methods instead
     */
    util::PrimitiveCollection<FD> fd_collection_;

    /* Registers new FD.
     * Should be overrided if custom behavior is needed
```

```
     */
    virtual void RegisterFd(
        Vertical lhs, Column rhs,
        std::shared_ptr<RelationalSchema const> const& schema) {
        if (lhs.GetArity() <= max_lhs_)
            fd_collection_.Register(
                std::move(lhs), std::move(rhs), schema);
    }

    virtual void RegisterFd(FD fd_to_register) {
        if (fd_to_register.GetLhs().GetArity() <= max_lhs_)
            fd_collection_.Register(std::move(fd_to_register));
    }
```

Все `FD` хранятся в `util::PrimitiveCollection`, который является списком с мьютексом. Поле `max_lhs_` присутствует для ограничения длины LHS добавляемых зависимостей, что позволяет уменьшить размер результата, если FD с большой LHS не нужны. Также оно может учитываться алгоритмом для того, чтобы не искать такие FD вовсе и ускорить работу.

В интерфейс класса `FDAlgorithm` добавляет методы получения результата в различных формах (`FdList`, `SortedFdList`). Наследникам предоставляются методы для добавления результата (`RegisterFd`).

Кроме того, предоставляются методы `GetJsonFDs`, `FDsToJson` — первый использует второй.

```
    /* возвращает набор ФЗ в виде JSON-а. По сути, это просто
     * представление фиксированного формата для сравнения
     * результатов разных алгоритмов. JSON - на всякий случай,
     * если потом, например, понадобится загрузить список в
     * питон и как-нибудь его поанализировать
     * */
    std::string GetJsonFDs() const;
```

В Python объекты FD передаются напрямую и используются как обычные объекты.

Метод `Fletcher16` вычисляет контрольную сумму Флетчера строки с представлением FD в виде JSON, используется только в тестах.

```
// считает контрольную сумму Флетчера - нужно для
// тестирования по хешу
unsigned int Fletcher16();
```

Последние три метода могли быть реализованы и вне этого класса, но для удобства реализации были добавлены в него самого. Хэширование результата применяется для упрощения кода тестирования, чтобы не хранить большие наборы зависимостей в коде.

В проекте и для других типов закономерностей часто применяется схема, в которой все алгоритмы поиска для какого-то типа закономерности наследуются от некоторого общего типа с названием по схеме `ТипЗакономерностиAlgorithm`, что может запутать новых участников проекта. Увидев такую схему наследования легко посчитать, что в проекте вообще все алгоритмы, работающие с некоторым типом закономерности, должны иметь общий класс. Это замедляет разработку и тратит время рецензентов кода[13].

Помимо этого, от этого класса наследуются классы, которые выполняют поиск формально другого типов закономерностей (`Pyro`, `Tane`), из-за чего при возвращении результатов пользователю теряется информация.

2.4.7. Другие классы

`Algorithm`. Все классы алгоритмов в Desbordante наследуются от класса `Algorithm`. Этот класс предоставляет некоторую полезную функциональность для облегчения задания параметров алгоритмов. Интерфейс исполнения алгоритмов предполагает последовательный вызов трёх методов: конструктора, метода загрузки данных (`LoadData`) и метода исполнения (`Execute`).

[13]пример в https://github.com/Desbordante/desbordante-core/pull/537#discussion_r2095872921

`PliBasedFDAlgorithm`. Большинство алгоритмов поиска FD имеют одинаковые стратегии считывания данных, поэтому для них метод `LoadData` вынесен в этот класс, все такие алгоритмы наследуются от него.

3. Описание решения

Исправление указанных проблем будет происходит путём замены инфраструктуры, связанной с представлением FD в коде. Далее будут определены и отделены явно её части — типы представления, типы хранения, внутренние типы алгоритмов и типы ввода.

Данная работа предлагает распространить такой подход и на остальные типы закономерностей.

3.1. Представление результата поиска

3.1.1. Разделение на типы представления и внутренние типы

Предлагаемое решение будет использовать идею из раздела 2.4.3, упомянутую при описании `Vertical`. Фактически для алгоритмов класс `Vertical` является битовой маской, которой они манипулируют, член `schema_` им не нужен. Объект схемы в классе `Vertical` требуется только для получения его строкового представления.

В этой работе предлагается распространить это соображение на обращение с объектами зависимостей в целом, жёстко разделив типы на типы представления и внутренние типы. Целью первых будет обеспечение удобства пользователя и ничего больше, вторых — обеспечение работы алгоритма и ничего больше. Допустима ситуация, когда внутреннего эквивалента для типа представления вообще может не быть.

Рассмотрим тип представления для FD. FD по определению является парой наборов атрибутов какого-то отношения. В коде атрибут можно представить как название столбца, но на практике бывают случаи, когда два столбца в таблице имеют одинаковые названия. В таком случае их нужно различать, что можно сделать с помощью числового идентификатора. Очень простой вариант реализации такого идентификатора — индекс столбца в таблице. Итого, тип представления для FD может быть определён как показано на Рис. 1. `table_name` — название таблицы, `lhs` и `rhs` — наборы атрибутов левой и правой частей FD соответственно.

```
struct Attribute {
    std::string name;
    Index id;
};
struct FunctionalDependency {
    std::string table_name;
    std::vector<Attribute> lhs;
    std::vector<Attribute> rhs;
    /* методы */
};
```

Рис. 1: Тип представления FD

Внутренний тип FD не будет иметь никакой связи с типом представления, внутри алгоритмов будет использоваться только то, что нужно алгоритму. Конкретно в случае алгоритмов поиска FD это будут заменяющие `Column` и `Vertical` индекс столбца (`IndexType`) и битовая маска (`boost::dynamic_bitset<>`) соответственно.

3.1.2. Типы хранения

Количество объектов представления может быть довольно велико — поэтому схема не хранилась в каждом объекте `FD` ранее. Хранить сейчас все FD в виде, определённом выше, было бы расточительно. Для того, чтобы учесть требование минимизации расхода памяти, предлагается также третий тип классов — типы для хранения ответов. Их внутреннее устройство будет зависеть от устройства алгоритма, а их задачей будет минимизация дополнительных затрат на запись ответа в форме, которую возможно преобразовать в тип представления. В этом смысле может быть тяжело чётко разграничить типы для хранения и внутренние типы — один тип может использоваться для обеих целей.

Хотя в общем случае для различных алгоритмов типы хранения могут отличаться в зависимости от их внутреннего устройства, для всех алгоритмов поиска FD можно определить типы хранения, как показано на Рис. 2. Здесь `TableHeader` — замена `RelationalSchema` без рекурсивности. `StrippedFd` — замена класса `FD`, учитывающая, что информация

```
struct TableHeader {
    std::string table_name;
    std::vector<std::string> column_names;
};
struct StrippedFd {
    boost::dynamic_bitset<> lhs;
    boost::dynamic_bitset<> rhs;
};

class FdStorage {
    TableHeader table_header_;
    Container<StrippedFd> fds_;
    /* методы */
};
```

Рис. 2: Типы хранения для алгоритмов поиска FD

о названиях столбцов — которая важна только для представления — хранится отдельно. `Container` — это некоторый контейнер, в котором хранятся представления каждой FD. Это может быть как `std::vector`, так и `std::list`, так и какой-либо другой подходящий контейнер.

Стоит отметить, что в оригинальном коде в классе, выступающем в роли хранилища — `PrimitiveCollection`[14] — присутствует (помимо коллекции результатов) мьютекс. Некоторые алгоритмы пишут в хранилище параллельно, это тоже нужно учесть, в предлагаемой реализации мьютекс будет стоять в самом алгоритме.

Методы, нужные от хранилища — чтение со стороны пользователя и добавление со стороны алгоритма. Есть смысл использовать класс-строитель.

Самый удобный способ прочитать хранилище — перевести все хранящиеся в нём закономерности в тип представления. Это будет первый метод чтения.

Но если закономерностей очень много, такой метод использует много памяти. Поэтому разумно также предоставить пользователю методы для чтения результатов по одному. Для этого была добавлена итерация

[14] см. src/core/util/primitive_collection.h

по хранилищу, которая была реализована[15] с помощью встроенного в C++20 `std::ranges::views::transform`.

На стороне связки с Python итерация была реализована[16] с помщью `pybind11::make_iterator`[17].

В предлагаемом варианте FD могут добавляться в произвольном порядке в зависимости от реализации алгоритма. Для постоянства вывода набора зависимостей может пригодиться операция "приведения в нормальный вид", где логически одинаковые ответы будут иметь одинаковые элементы.

Пример: пусть имеется два хранилища со следующими FD:

1. - $(\text{SSN}, \text{LastName}) \to (\text{Address}, \text{FirstName})$
   - $(\text{ID}) \to (\text{Income})$
2. - $(\text{ID}) \to (\text{Income})$
   - $(\text{SSN}, \text{LastName}) \to (\text{FirstName})$
   - $(\text{SSN}, \text{LastName}) \to (\text{Address})$

Логически это одинаковые ответы, они указывают на одни и те же равенства атрибутов, но элементы в самих хранилищах отличаются. Предлагаемый метод приведёт их к одинаковому виду:

- $(\text{ID}) \to (\text{Income})$
- $(\text{SSN}, \text{LastName}) \to (\text{Address}, \text{FirstName})$

3.2. Проверка соблюдения зависимости

Проблемы с интерфейсом проверки соблюдения FD были в том, что нельзя ссылаться на столбец с помощью названия и класс `FD` никак не связан с проверкой соблюдения `FD`. То есть ожидается возможность вызвать код, подобный приведённому ниже.

[15] см. src/core/algorithms/fd/fd_storage.h
[16] см. src/python_bindings/fd/bind_fd.cpp
[17] см. https://pybind11.readthedocs.io/en/stable/reference.html

```
fd_verifier.execute(fd=FunctionalDependency(
    ("SSN"), ("LastName", "FirstName")))
```

Но для создания класса `FunctionalDependency` нужно указывать атрибуты целиком, поэтому зададим похожий класс `FdInput`.

```
struct FdInput {
    std::vector<std::string> column_names_lhs;
    std::vector<std::string> column_names_rhs;
};
```

Вариант только со строками не будет работать правильно во всех ситуациях, поскольку в таблице может быть несколько столбцов с одинаковым названием — тогда произвести проверку соблюдения просто не получится. Поэтому ещё нужно дать вариант действий в таких ситуациях. Самый простой вариант — дать возможность указывать индекс столбца вместо названия.

```
struct FdInput {
    std::vector<std::variant<std::string, Index>> lhs;
    std::vector<std::variant<std::string, Index>> rhs;
};
```

Условимся называть такие классы типами ввода. Для цели, указанной в разделе 2.3.6, предлагается добавить функциональность перевода типа представления в тип ввода. Для класса `FunctionalDependency` это выглядит как специальный метод `ToInput`[18]. В текущей реализации он заполняет поля `FdInput` индексами столбцов, но можно реализовать и более сложную логику, где используются названия, если нет повторений, чтобы улучшить читаемость при выводе полученного объекта `FdInput`.

3.3. `FDAlgorithm`

`FDAlgorithm` за вычетом методов для представления в JSON представляет собой хранилище. Внутреннее устройство алгоритмов может

[18] см. src/core/algorithms/fd/fd.h

отличаться, а значит и оптимальные типы хранилищ могут быть разными.

В данном случае устройство типов хранилищ во всех алгоритмах будет одинаковое, но их создание будет различаться — некоторым не нужно дополнительное ограничение на длину LHS, так как в них уже встроена функция не искать FD со слишком большими LHS, другим не нужны мьютексы, так как добавление FD в результат происходит в одном потоке.

Уберём наследование функциональности от общего класса. Вместо этого добавим во все классы тип хранилища и методы его получения. В зависимости от требования класса меняться будет тип строителя хранилища. Хранилище будем хранить как `std::shared_ptr`, чтобы не копировать при получении результата. Повторение кода будет устраняться по мере надобности.

4. Реализация и анализ решения

По ссылке https://github.com/Desbordante/desbordante-core/pull/681 представлена экспериментальная реализация решения.

Отличительной особенностью предложенного решения является то, что все объекты, связанные с FD, являются простыми. Отсутствуют сложные зависимости между классами, время жизни каждого объекта самостоятельное.

`model::FunctionalDependency` — класс представления функциональной зависимости — сериализуется, как и оригинальный `FD`, но весь код сериализации находится в одном файле. В отличие от предыдущей реализации, сложных ухищрений со связанными временами жизни применять не приходится, поскольку все объекты стандартные.

Получилось реализовать полностью корректную проверку на равенство FD на стороне ядра библиотеки с помощью `operator==`, который представляет собой простое почленное сравнение. Операция `__eq__` реализуется с помощью стандартных средств `pybind11`, вызывающих этот оператор. Операция `__hash__` реализована как хэш от кортежа, содержащего элементы класса.

Реализованы конструкторы для `FunctionalDependency` и всех её составляющих частей. Также реализованы методы приведение к строке `__str__` и `__repr__`, причём строка, возвращаемая последним, может быть использована для создания этих объектов.

Для одного из алгоритмов поиска была сильно упрощена операция сбора результатов[19], так как `StrippedFd` была ближе к внутреннему представлению этого алгоритма, чем ранее использовавшаяся `FD`.

Требования к памяти теперь даже ниже предыдущих в большинстве случаев, поскольку теперь каждая FD хранится в `FdStore` как пара битовых масок, вместо `Vertical`, `Column` и указателя на схему.

Качественная реализация нормализация хранилища требует дополнительного исследования — пока что неясно, как она должна будет работать для типов хранилищ, отличных от представленного — поэтому

[19] см. src/core/algorithms/fd/fdep/fd_tree_element.cpp

пока что эта операция реализована не была.

Методы хэширования результата в новой реализации были перенесены в часть тестов, так как они используются только там. Для того, чтобы не вычислять хэши заново, представление результата, которое хэшируется, имитирует представление в прошлой реализации. Это несколько усложнило код, но взаимодействие с этой частью проекта не является частым, поэтому это приемлемо.

Что касается проверки соблюдения, ранее она выполнялась с помощью указания индексов:

```
fd_verifier.execute(lhs_indices=[0], rhs_indices=[2])
```

Теперь эту операцию можно выполнить следующим образом:

```
fd_verifier.execute(fd=FdInput(["id"], ["name"]))
```

В новой реализации логически существует тип FD, передаваемый проверке соблюдения, и можно указывать на столбцы с помощью их названия. Эти два изменения делают код, написанный с использованием библиотеки Desbordante, более читаемым.

Заключение

В ходе работы были проработаны проблемы проекта Desbordante, связанные со способом представления закономерностей.

- Был выполнен анализ требований к проекту и проблем с текущим представлением закономерностей;
- Были разработаны принципы устройства представления закономерностей;
- В качестве эксперимента был реализован подход, использующий эти принципы.

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] 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] 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.

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

[5] Mannila Heikki, Räihä Kari-Jouko. On the complexity of inferring functional dependencies // Discrete Applied Mathematics. — 1992. — Vol. 40, no. 2. — P. 237–243. — URL: https://www.sciencedirect.com/science/article/pii/0166218X92900315.

[6] desbordante-reflejo. — URL: https://github.com/Desbordante/desbordante-reflejo (дата обращения: 2026-02-22).