Kolmogorov.ai | Morphism | Руководство пользователя

Feature Store

ООО Дата Сапиенс

version 0.20.4 | 12.04.2024

Table of contents

1. About

Morphism [m f z( )m] - Инструмент класса Feature Store.

Позволяет организовать библиотеку датасетов, их связей, атрибутов и трансформов над атрибутами. Упрощает задачу типового дата-инжиниринга для экспериментов в подходе low-code self-service ETL.

Ключевые задачи инструмента:

- Управление данными
- Выполнение трансформаций над данными
- Мониторинг качества загруженных данных
- Экспорт/импорт данных и алгоритмов без потерь

Архитектура приложения позволяет не хранить данные в широких витринах, что позволяет упростить процесс работы с данными. Интерфейсы приложения дают возможность составлять алгоритмы преобразований данных, избавляя пользователя от необходимости писать `SQL`-код. Благодаря этому логика трансформаций унифицирована и может быть перенесена между несвязанными контурами данных.

2. Overview

2.1 Как часто бывает

Типичный процесс формирования датасетов для моделей машинного обучения обычно состоит из следующих этапов:

- Построение широкой (зачастую очень широкой) витрины, которая включает в себя избыточный набор всех возможных атрибутов
- Построение отдельного датасета для каждой модели на основе атрибутов широкой витрины с использованием дополнительных атрибутов из пользовательских песочниц
- Обогащение датасетов результатами работы других моделей по мере появления данных

В таком случае организация работы с данными при необходимости построения нового датасета выглядит следующим образом:

1. Data Scientist (**DS**) формулирует требования к инженеру данных на новый датасет
2. Data Engineer (**DE**) работает с требованиями, консультирует DS по источникам и алгоритмам, дорабатывает широкую витрину - разбирается в чужих скриптах, добавляет необходимые поля, перепроверяет, что существующие атрибуты не поломались
3. **DE** выполняет сборку датасета, передаёт результаты DS
4. **DS** проверяет датасет и понимает, что какие-то атрибуты рассчитаны неверно, а какие-то не были учтены изначально
5. Всё начинается сначала

Работы **DE** может также проводить **DS**, но в этом случае до половины рабочего времени **DS** будет тратиться на подготовку данных, а не на их исследование и анализ.

Такая организация работы с данными выглядит неэффективно.

2.2 Feature Store

Что меняется:

- Не используется широкая витрина. **DE** не тратит время на разбор тысяч строк кода построения широкой витрины, а выстраивает новые потоки данных
- **DE** дополняет приложение только действительно новыми данными, а не занимается историческим пересчётом сотен атрибутов при появлении очередного типового агрегата, который требуется рассчитать за новый период
- **DS** получает понятный и удобный каталог переменных с поддержкой документирования и версионирования алгоритмов - больше не надо ковыряться в сотнях разрозненных таблиц, тоннах писем и других документов
- **DS** самостоятельно создаёт датасет по параметрам, конфигурируя типовые преобразования с использованием конструктора агрегатов и пользовательских формул, что позволяет проводить больше экспериментов за меньшее время и сконцентрироваться на исследовании данных
- **DE** не занимается ручным построением датасетов, процесс сборки датасета по заданным параметрам становится автоматическим

2.2.1 Управление данными

Каждый объект, будь это таблица или столбец, регистрируется в приложении, что позволяет в будущем работать с данными через интерфейсы приложения без прямых обращений в источник данных.

Регистрация данных является подробным описанием источника данных. Все это делается для:

- Корректной работы движка приложения с данными при выполнении преобразований
- Накопление исторических данных
- Удобство поиска необходимых данных в интерфейсе приложения

Описание источника данных в SDK:

```
from FSClient.catalogue import (
    entity as ent,  # <-- Сокращение для удобства
    feature as feat,
    loader as load
)

new_loader = load.create(
    source='table'  # <-- Тип источника данных
)

new_loader.config = {
        'loader_name': 'CUSTOMER_INFO_HIST',  # <-- Название источника данных в каталоге
        'description': 'Информация по клиенту из карточки клиента АБС',
        'src_schema_name': 'stage',  # <-- Схема таблицы
        'src_table_name': 'customer_hist',  # <-- Название таблицы
        'entity': ent.CUSTOMER  # <-- Сущность таблицы
}
```

2.2.2 Выполнение трансформаций над данными

Широкий функционал приложения дает возможность строить датасеты на основе зарегистрированных данных, используя типовые преобразования с использованием конструктора и пользовательских формул. Все это помогает сконцентрироваться на исследовании данных, а не сборе.

Создание датасета в SDK:

```
my_dataset = ds.create(
    entity=ent.CUSTOMER,  # <-- Сущность датасета
    name='AGG_TRANS_CUSTOMER_DATA',
    description='Данные по транзакциям клиентов в разбивке по количеству транзакций'
)

my_dataset.add_feature(
    features=[
        feat.TRN_AMT.CT_W_V1.alias('TRN_AMT')  # <-- список версий фичей на агрегирование
    ],
    agg=[  # список агрегирующих функций
        func.sum(),
        func.max(),
        func.min()
    ],
    domain=[  # условия агрегации
        (feat.TRN_CNT.CT_W_V1 <= 10).set(alias='L10'),  # <-- количество транзакций меньше 10
        (feat.TRN_CNT.CT_W_V1 > 10).set(alias='M10')  # <-- количество транзакций больше 10
    ]
)
```

Процесс сборки датасета по заданным параметрам становится автоматическим и не требует пользовательского вмешательства.

Благодаря структурированной системе хранения данных обеспечивается высокая скорость расчета датасета, позволяя получать датасеты при разработке решений машинного обучения или для анализа "на лету".

2.2.3 Мониторинг качества загруженных данных

Приложение предоставляет набор инструментов расчета статистик для загруженных данных. Таким образом обеспечивается сквозной мониторинг качества данных на всех этапах трансформаций.

2.2.4 Экспорт/импорт данных и алгоритмов без потерь

Каталогизация в приложении осуществляется не только на уровне объектов в виде источников данных, но и на уровне логических сущностей, таких как **алгоритмы**. Алгоритмы трансформаций данных в приложении пишутся, оперируя объектами каталогов, то есть по сути ссылками на реальные объекты.

Все это позволяет отделять бизнес-логику от конкретных объектов в БД и, как следствие, переносить ее на другие объекты не только внутри одной той же БД, но и на любую другую.

3. Concepts

3.1 Тип данных

Типы данных (`fs_type`) объединяют в себе типы данных из разных СУБД (`db_type`).
Терминология:

- `fs_type` - тип данных. Логическая сущность.
- `db_type` - тип данных в СУБД. Соответствие между `fs_type` и `db_type` называется имплементацией.

При регистрации/описании объектов используются именно `fs_type`. Для корректной работы с данными необходимо настроить имплементации типов данных для диалектов СУБД.

Пример:

fs_type	db_type	db_name
AMOUNT	decimal(26,2)	postgresql
AMOUNT	decimal	spark
FLOAT	float8	postgresql
FLOAT	double	spark
INT	integer	postgresql
INT	int	spark

Таким образом одно значение `fs_type` может объединять в себе несколько имплементаций на разные диалекты, что позволяет работать одновременно с разными СУБД используя один и тот же справочник типов данных.

Базовое наполнение справочника `fs_type`, а так же имплементаций, предоставляется по умолчанию. Доступна пользовательская настройка справочников и имплементаций в API в зависимости от потребностей пользователя.

Каталог данных в SDK:

```
from FSClient.catalogue import data_type  # Каталог типов данных

data_type
```

Подробная информация о типе данных в SDK:

```
from FSClient.catalogue import data_type  # Каталог типов данных

data_type.My_INT_Data_Type
```

3.1.1 Типы данных по-умолчанию

Если в процессе работы с данными пользователь не задает `fs_type` явно, то используется справочник типов данных "по-умолчанию" (`default_fs_type`).

Справочник представляет собой соответствие между типами данных СУБД (`db_type`) и типами данных (`fs_type`).

В справочнике должны быть определены все типы данных СУБД, которые используются пользователем.

Справочник типов данных `default_fs_type` настраивается для каждого диалекта отдельно.
Пример:

db_type	default_fs_type	db_name
bigint	BIG_INT	postgresql
bigserial	BIG_INT	postgresql
integer	BIG_INT	postgresql
json	BIG_STRING	postgresql
text	BIG_STRING	postgresql
character	BIG_STRING	postgresql

3.2 Сущность

Логическая **сущность**, представляющая из себя набор из не менее чем одного ключа.
Физически каждый ключ является полем в таблице. Таким образом, остальные поля таблицы содержат в себе *данные о сущности*.

Сущность необходима для регистрации данных в приложении. Каждая сущность объединяет в себе данные из разных источников (таблиц).

Пример: CUSTOMER (CUSTOMER_ID), OFFER (CUSTOMER_ID + CAMPAIGN_ID) и т.д.

Каталог сущностей в SDK

```
from FSClient.catalogue import entity  # Каталог сущностей

entity
```

Подробная информация о сущности в SDK:

```
from FSClient.catalogue import entity  # Каталог сущностей

entity.CUSTOMER
```

3.2.1 Реестр сущности

Реестр сущности = якорная таблица, которая содержит целевой набор значений сущности, на основе которого строятся датасеты.

Пользователь может определить загрузчик реестра сущности в опциях сущности.

Требования к загрузчику реестра сущности:

- Таблица загрузчика содержит эталонный (полный и актуальный) набор значений сущности
- Загрузчик зарегистрирован под той же сущностью, по которой определяется реестр
- Тип загрузчика = "готовые переменные" (final)

Определение реестра сущности в SDK:

```
from FSClient.catalogue import (
    entity as ent,  # Каталог сущностей
    loader as load  # Каталог загрузчиков
)

edit_entity = ent.CUSTOMER.edit()  # редактирование сущности

# загрузчик готовых фичей CUSTOMER_HIST_FINAL как реестр
edit_entity.registry = load.CUSTOMER_HIST_FINAL

edit_entity.save()  # сохранение изменений сущности
```

С целью оптимизации расчета датасетов настоятельно рекомендуется определить реестр сущности для всех зарегистрированных сущностей.

Если реестр сущности не определен, но в логике задания расчета датасета предполагается его использование, он будет рассчитан автоматически:

- для операции "Слияние датасетов" - объединяются значения сущности из всех таблиц загрузчиков готовых переменных
- для операции "Узел перехода" - объединяются значения сущности из всех используемых таблиц без связи (при этом сохраняется требование обязательного наличия в конфигурации хотя бы одной переменной без связи)

Такой подход не является оптимальным с точки зрения скорости расчета, но позволяет соблюдать детерминированность рассчитанных датасетов.

3.3 Фича

Является мета сущностью и имеет только **имя** и **описание**. Фича по своей сути является бизнес-понятием, к которому привязываются поля в источнике данных (таблице). Таким образом одна фича может иметь несколько источников данных, где каждое отображение фичи в источнике называется *версией фичи*.

Каталог фичей в SDK:

```
from FSClient.catalogue import feature  # Каталог фичей

feature
```

Подробная информация о фиче в SDK:

```
from FSClient.catalogue import feature  # Каталог фичей

feature.AGE
```

3.3.1 Версия фичи (переменная)

Фича, привязанная к источнику данных, то есть к таблице. Можно создать только через загрузчик.

Пример: допустим, поступают данные о транзакциях из трех разных источников. В таком случае создается *одна* фича TRANSACTION_AMOUNT и регистрируется по версии на каждый из источников с помощью загрузчиков.

Подробная информация о версии фичи в SDK:

```
from FSClient.catalogue import feature  # Каталог фичей

feature.AGE.DATA_FROM_STORE_V1
```

3.4 Источник данных

Источниками в приложении выступают БД. Для регистрации нового источника данных необходимо убедиться, что приложение поддерживает используемый диалект.

Каждый источник данных используется как отдельный контур и данные не переносятся между источниками данных. Например, при конфигурации датасета данные, зарегистрированные из одного источника, не могут быть использованы вместе с данными, зарегистрированными из другого источника.

Каталог источников в SDK:

```
from FSClient.catalogue import datasource  # Каталог источников

datasource
```

Подробная информация об источнике в SDK:

```
from FSClient.catalogue import datasource  # Каталог источников

datasource.Postresql_Datasource
```

3.5 Загрузчик

Загрузчик является инструментом регистрации данных.

Данные, зарегистрированные в приложении, используются для создания датасетов.

3.5.1 Процесс регистрации

При регистрации загрузчика указывается:

- Источник данных
- Ссылка на даг Airflow, которым рассчитываются данные в источнике (справочно)
- Тип приемника
- Режим загрузки
- Маппинг
- Расписание

Имя загрузчика должно быть уникальным во всём каталоге.

Источник данных

Допустимые источники данных:

- Таблица в БД (table / external)
- SQL-скрипт
- CSV/XLSX-файл (file)

В отличие от CSV/XLSX-файла и SQL-скрипта, таблица в БД может быть зарегистрирована в виде ссылки без последующей физической загрузки данных из неё во внутренние структуры приложения. Ниже представлено сравнение использования загрузчика с загрузкой данных и без таковой.

ЗАГРУЗЧИК С ЗАГРУЗКОЙ ДАННЫХ (TABLE, FILE, SQL)

- Плюсы
- Поддерживается ведение истории с обеспечением принципов неизменности исторических данных (данные за загруженные даты не изменяются)
- Автоматический расчет периодов актуальности данных
- Оптимальная структура хранения загруженных данных
- Минусы
- Данные занимают дополнительное место в БД
- Требуется время на загрузку данных
- Не всегда возможно исправить загруженные исторические срезы, учесть "долеты"

ССЫЛКА НА ТАБЛИЦУ В БД (EXTERNAL)

- Плюсы
- Не требуется дополнительное место в БД
- Данные готовы для использования сразу, не надо ждать загрузку
- Возможность исправить исторические данные

- Минусы
- Данные в таблице не валидируются
- Данные в таблице находятся в зоне ответственности пользователя (система не контролирует неизменность истории, не рассчитывает индексы и т.д.)
- Дополнительные требования к структуре таблицы (поля с датами и ключами должны называться определенным образом)

ЗАГРУЗКА ДАННЫХ ЧЕРЕЗ SQL-СКРИПТ

К скрипту выдвигаются следующие требования:

- Скрипт может содержать только одну инструкцию `select` без использования CTE;
- Скрипт должен быть исполняем на том источнике данных, который указывается в загрузчике.

Тип приемника

Допустимые типы приемников:

- Переменные (hot) - создаются и загружаются новые версии в каталоге переменных
- Готовые переменные (final) - данные в разрезе сущности, готовые для использования в датасете без дополнительных манипуляций; в срезе данных по дате актуальности отсутствуют дубли по сущности; *примеры - пол клиента, номер договора и т.д.*
- Транзакционный предагрегат (trans_preagg) - предварительно агрегированные данные за определенный период (гранулярность); *примеры - количество транзакций за неделю, сумма зп-начислений за месяц и т.д.*
- Предагрегат общего назначения (common_preagg) - детализированная информация по сущности под расширенным ключом; *примеры - договоры в разрезе клиента, ответы из БКИ по клиенту и т.д*
- Сегмент (keys) - сохраняется список значений сущности, остальные поля сохраняются как переменные сегмента. Сегмент регистрируется в каталоге датасетов, переменные сегмента не сохраняются в каталоге переменных. Для датасетов, полученных таким образом, после завершения загрузки производится расчет статистик и графика плотности распределения (отключаемо). Загрузка сегмента недоступна через sql-скрипт.
- Связь сущностей (entity_link) - создается и загружается новая связь существующих сущностей

Режим загрузки

Доступны следующие режимы загрузки:

- full = загрузка полного среза
- replace = замена данных приемника данными источника; режим доступен только для источников файл, таблица и sql-скрипт

Реализовано сохранение истории изменения данных, если это применимо для выбранного типа приемника. Стратегия работы с историей зависит от режима загрузки и типа приемника:

Тип приемника	Режим загрузки	Стратегия загрузки
Готовые переменные	full	snapshot
Готовые переменные	replace	replace
Транзакционный предагрегат	full	insert
Транзакционный предагрегат	replace	replace
Предагрегат общего назначения	full	snapshot
Предагрегат общего назначения	replace	replace
Связь сущностей	full	snapshot
Связь сущностей	replace	replace
Сегмент	-	replace

Для стратегий загрузки `snapshot` и `insert` происходит последовательная загрузка срезов данных по отобранным датам актуальности. Даты актуальности отбираются из источника по условию: `дата актуальности > максимальная дата актуальности в таблице-приемнике`.

Для режима загрузки `replace` даты актуальности не анализируются, происходит простая замена данных приемника данными источника без дополнительных проверок и условий.

SNAPSHOT

У записей у которых в таблице-приемнике to_dttm=infinity, проставляется `to_dttm=<пришедшая дата> - 1 microsecond`. Новые записи вставляются с `from_dttm=<пришедшая среза>, to_dttm=infinity`

INSERT

Производится INSERT новых записей без проверок

REPLACE

Без истории. Производится REPLACE таблицы

Маппинг

При регистрации загрузчика необходимо указать соответствие загружаемых полей источника на:

- ключи выбранной сущности (для связи сущностей - ключи родительской и дочерней сущности)
- дату актуальности - дата, на которую рассчитаны данные в строке
- дату окончания актуальности - дата до которой данные в строке остаются актуальными
- переменные (только для загрузчиков переменных) - при этом указываются параметры создаваемой версии (имя, описание, тип данных). Имя версии переменной должно быть уникальным в разрезе переменной во всём каталоге.

МАППИНГ НА ДАТЫ (ИСТОЧНИК -> ПРИЕМНИК)

- `table`, `file`, `sql` -> `hot`, `entity_link`
- **Дата актуальности**: Опционально (поле/константа)
- **Дата окончания**: Не обрабатывается
- `external` -> `hot`, `entity_link`
- **Дата актуальности**: Опционально (только поле)
- **Дата окончания**: Если указана Дата акутальности, то Обязательно, иначе не обрабатывается

- `*` -> `keys`
- **Дата актуальности**: Не обрабатывается
- **Дата окончания**: Не обрабатывается

Если дата актуальности не указана для источников `table` / `file` / `sql` загрузчиков, загружаемых в режиме полного среза, то в качестве даты актуальности проставляется фактическая дата загрузки данных (т.о. загружаемые данные действуют с момента загрузки).

Во всех остальных случаях, если дата актуальности не указана, то данные в таблице считаются действующими бессрочно.

НАСТРОЙКА РАСПИСАНИЯ

Источник	Настройка расписания
CSV/XLSX-файл	
Таблица в БД (ссылка)	
Таблица в БД (загрузка)	
SQL	

Настройка расписания осуществляется в формате пятизначной cron-строки:

```
* * * * * cron-строка расписания
- - - - -
| | | | |
| | | | ----- день недели (0–7) (воскресенье = 0 или 7)
| | | ------- месяц (1–12)
| | --------- день месяца (1–31)
| ----------- час (0–23)
------------- минута (0–59)
```

Настройка загрузчика в SDK:

```
from catalogue import loader

new_loader = loader.create(
    source: str,  # источник загрузчика: file, table, external, sql
    target: str  # тип загрузки: features, segment, entity_link
)

# Конфиг для загрузчика фичей с источником файл (`load.create(source='file', target='feature'`):
new_loader.config = {
        'loader_name': str, # имя загрузчика
        'description': Optional[str],  # описание загрузчика
        'load_mode': str, # тип загрузки, доступные: ("inc", "full")
        'delimiter': str, # разделитель в загруженном файле (для .csv)
        'entity': Entity,  # сущность для загрузчика
        'preagg_flg': bool,  # флаг таблицы предагрегатов
        'granularity_type': str = None,  # тип гранулярности; необязательный
        'granularity': int = None,  # гранулярность; необязательный

}

# Конфиг для загрузчика фичей с источником таблица/внешняя таблица/SQL-скрипт

new_loader.config = {
        'loader_name': str, # имя загрузчика
        'description': Optional[str],  # описание загрузчика
        'load_mode': str,  # тип загрузки, доступные: ("inc", "full")
        'src_schema_name': str,  # имя схемы из источника
        'src_table_name': str,  # имя таблицы из источника
        'entity': Entity,  # сущность для загрузчика
        'preagg_flg': bool,  # флаг таблицы предагрегатов
        'granularity_type': str = None,  # тип гранулярности; необязательный
        'granularity': int = None,  # гранулярность; необязательный
}

# Конфиг для загрузчика связи сущностей с источником файл (`load.create(source='file', target='feature'`):
new_loader.config = {
        'loader_name': str, # имя загрузчика
        'description': Optional[str],  # описание загрузчика
        'load_mode': str, # тип загрузки, доступные: ("inc", "full")
        'delimiter': str, # разделитель в загруженном файле (для .csv)
        'kind': str,  #  тип связи ("1:1", "1:M", "M:N")
        'parent_entity': Entity,  # родительская сущность (в типе связи - слева)
```

```
        'child_entity': Entity  # дочерняя сущность (в типе связи - справа)
}

# Конфиг для загрузчика связи сущностей с источником таблица/внешняя таблица/SQL-скрипт

load.create(source='file', target='feature')

new_loader.config = {
        'loader_name': str, # имя загрузчика
        'description': Optional[str],  # описание загрузчика
        'load_mode': str, # тип загрузки, доступные: ("inc", "full")
        'src_schema_name': str,  # имя схемы из источника
        'src_table_name': str,  # имя таблицы из источника
        'kind': str,  #  тип связи ("1:1", "1:M", "M:N")
        'parent_entity': Entity,  # родительская сущность (в типе связи - слева)
        'child_entity': Entity  # дочерняя сущность (в типе связи - справа)
}

# Конфиг для загрузчика списка ключей/сегмента с источником файл (`load.create(source='file', target='segment'`):
new_loader.config = {
        'loader_name': str, # имя загрузчика
        'description': Optional[str],  # описание загрузчика
        'load_mode': str, # тип загрузки, доступные: ("inc", "full")
        'delimiter': str, # разделитель в загруженном файле (для .csv)
        'entity': Entity,  # сущность для загрузчика
}

# Конфиг для загрузчика списка ключей/сегмента с источником таблица/внешняя таблица/SQL-скрипт

load.create(source='file', target='segment')

new_loader.config = {
        'loader_name': str, # имя загрузчика
        'description': Optional[str],  # описание загрузчика
        'load_mode': str, # тип загрузки, доступные: ("inc", "full")
        'src_schema_name': str,  # имя схемы из источника
        'src_table_name': str,  # имя таблицы из источника
        'entity': Entity,  # сущность для загрузчика
}

# Настройка расписания
new_loader.schedule = {
    'type': str,  # возможные значения:
                  # 'Hours', 'Everyday', 'Weekly', 'Monthly_last_day', 'Monthly_day_of_week', 'Custom'
    'step': int,  # количество часов в шаге; обязательный только для Hours
    'weekdays_flg': bool,  # признак "только по будням"; может заполняться только
                           # для Everyday (не обязательный), в остальных случаях не заполняется
    'day_of_week': List[int],  # перечисление дней недели (1-7); обязательный только
                               # для Weekly И Monthly_day_of_week, в остальных случаях не заполняется
    'num_of_week': int,  # номер недели; обязательный только для Monthly_day_of_week,
                         # в остальных случаях не заполняется
    'day_of_month': List[int],  # перечисление дней месяца (1-31), на которые нужно
                                # провести расчёт; обязательный только для Custom,
                                # в остальных случаях не заполняется
    'run_time': str  # время запуска в формате строки "YYYY-MM-DD hh:mm:ss"
}

# Маппинг загрузчика фичей
new_loader.mapping = {
        'from_dttm': Union[str, datetime.date],  # поле даты актуальности
        'to_dttm': Union[str, datetime.date] = None,  # поле окончания периода актуальности
                                                      # (только для External загрузчика)
        'entity': [
            { # словарь маппинга выбранного сущности
                'entity_key': Entity,  # объект Entity с детализацией до ключа
                'stg_column_name': str  # название поля для ключа из выбранного файла
            },
            ..
        ],
        'features': List[
            {
                'stg_column_name': str,  # название поля для ключа из выбранного файла
                'feature': Feature,  # объект Feature
                'feature_version_config':
                    {  # заполняется только если в 'feature' передана БФ, а не версия
                        'name': str,  # название версии фичи
                        'kind': str,  # тип фичи, доступные значения: ["feature"]
                        'description': str,  # описание версии фичи
                        'data_type': DataType  # объект DataType
                    } = None
            },
            ..
       ]  # список словарей маппинга фичей
}

# Маппинг загрузчика связи сущностей
new_loader.mapping = {
        'from_dttm': Union[str, datetime.date],  # поле даты актуальности
        'to_dttm': Union[str, datetime.date] = None,
        'parent': [
            { # словарь маппинга родительнской сущности
                'entity_key': EntityKey,  # объект EntityKey с детализацией до ключа
                'stg_column_name': str  # название поля для ключа из выбранного файла
            },
```

```
            ..
        ],
        'child': [
            { # словарь маппинга дочерней сущности
                'entity_key': EntityKey,  # объект EntityKey с детализацией до ключа
                'stg_column_name': str  # название поля для ключа из выбранного файла
            },
            ..
        ]
}

# Маппинг загрузчика сегмента/списка ключей
new_loader.mapping = {
        'entity': Union[
            [
                { # словарь маппинга выбранной сущности
                    'entity_key': Entity,  # объект Entity с детализацией до ключа
                    'stg_column_name': str  # название поля для ключа из выбранного файла
                },
                ..
            ],
            ent.ROWNUM  # в качестве сущности можно указать ROWNUM
        ]
}

new_loader.save_x_execute()
```

3.6 Связь сущностей

Связь сущностей является "мостом" между двумя разными сущностями. Применяется для использования в конфигурации датасета объектов (версия фичи, ключ, другой датасет), сущность которых отличается от сущности датасета.

Связи сущностей могут быть объединены в цепочку для перехода от одной сущности к другой.

Ключевым параметром связи сущностей является ее мощность:

- 1:1
- 1:M
- M:N

Мощность определяет отношение сущностей, что важно при переходе от одной сущности к другой. Например, если сущность `CUSTOMER` связана с сущностью `AGREEMENT` (`CUSTOMER`) `1:M` (`AGREEMENT`), то при переходе от сущности `CUSTOMER` к сущности `AGREEMENT` записи размножатся.

3.7 Датасет

Датасет - материализованный набор данных в виде таблицы в БД, рассчитанный в разрезе выбранной сущности на данных, зарегистрированных в приложении.

Назначение - получение выборок данных для:

- Задач машинного обучения
- Анализ данных
- Промежуточное звено трансформации данных

Способы получения датасета:

- Создание согласно конфигурации (конструктор)
- Загрузка из источника с помощью загрузчика (сегмент)
- Нарезка другого датасета на части (стратификация)
- Замена пустых значений (импутация) другого датасета

3.7.1 Конфигурация датасета через конструктор

Классический вариант создания датасета.
Представляет собой следующую последовательность действий:

- Выбор сущности датасета

Создание нового датасета в SDK:

```
from catalogue import dataset as ds

new_ds = ds.create(
    entity_ds: ent.CUSTOMER,  # указывается объект Entity, сущность датасета
    name: "my_giga_dataset",  # название датасета
    description: "customer transactions through the agreements",  # описание
)
```

- Выбор сегмента (опционально) - датасет-источник значений сущности.
- Сегментом может выступать любой датасет из каталога.
- Если сегмент не выбран, то в конфигурируемый датасет отбираются значения сущности всех используемых версий переменных, загруженных под сущностью датасета.

Конфигурация сегмента в SDK:

```
new_ds.segment: ds.Ivans_anaylytics_dataset  # датасет не должен быть пустым
```

- Выбор даты расчёта по состоянию на которую рассчитываются переменные:
- Актуальный срез
- Даты из календаря
- Даты из сегмента (только если выбран сегмент и в сегменте есть дата)\

Настройка даты актуальности датасета в SDK:

```
new_ds.calendar_config = ['2000-01-01 00:00:00']  # формат дат: YYYY-MM-DD hh:mm:ss
```

- Конфигурация переменных датасета - можно использовать:
- Готовые переменные из каталога
- Предагрегаты из каталога - настраиваются агрегирующие функции и условия фильтрации:
- Агрегирующие функции - max, min, avg, count, sum, count_rows (количество строк), count_distinct (количество уникальных, аналог count(1))
- В условиях фильтрации могут использоваться:
- Переменные из того же предагрегата, что и агрегируемая переменная
- Дата актуальности. Настройка периода агрегации по дате актуальности обязательна для транзакционных предагрегатов
- Переменные из сегмента
- Ключи связанных сущностей
- Расчетные переменные на основе сконфигурированных переменных из датасета (пользовательские формулы второго уровня). Доступные функции для пользовательских формул описаны ниже.

Добавление фичей в датасет в SDK:

```
# Готовая фича
new_ds = ds.create(ent.CUSTOMER, name='test_ds')
new_ds.add_feature(
    features=[
        Features.<Feature_Name>.<Feature_Version_Name>.alias('my_new_alias')
    ],
    link: List[EntityLink]
)

# Агрегированная фича
new_ds = ds.create(ent.CUSTOMER, name='test_ds')
new_ds.add_feature(
    features=[
        Features.<Feature_Name>.<Feature_Version_Name>.alias('my_new_alias')
    ],
    link: List[EntityLink],
    agg: List[Function],  # func.sum(), func.avg() etc.
    domain: List[FeatureExpression]
)

# Ключ связанной сущности
new_ds = ds.create(ent.CUSTOMER, name='test_ds')
new_ds.add_feature(
    ent.AGREEMENT.AGREEMENT_RK.alias('agreement_rk_linked').link(el.link_CUSTOMER_x_AGREEMENT)
)
```

- Выбор итоговых переменных для включения в поля датасета - например, некоторые переменные могут использоваться только для фильтрации записей, но не должны фигурировать в итоговой таблице датасета.
- Настройка условий фильтрации с использованием любых сконфигурированных переменных датасета. Синтаксис условия совпадает с пользовательскими формулами второго уровня.

Настройка фильтра в SDK:

```
new_ds.filter = local_feat.MIN_TRN_AMT_ALL + 100 > 500
```

- Выбор рассчитываемых статистических показателей для полей датасета. (опционально)

Имена переменных датасета должны быть уникальны в таблице датасета и не пересекаться с физическими названиями других полей датасета (ключи зерна, дата расчета, технические поля - *created_dttm*, *run_rk*)

Цепочки связей зерен

При конфигурировании сегмента и переменных датасета возможно использование цепочек связей сущностей.

Цепочка связей сущностей представляет собой последовательность связей по которой осуществляется переход от сущности датасета к сущности используемого объекта. Ограничение - в цепочке связей не может использоваться более

одной автоматический связи подряд.
Если сущность используемого объекта не совпадает с сущностью датасета, то указание цепочки связей является обязательным.
Если в цепочке связей для переменной присутствует связь, приводящая к размножению записей, то такая переменная всегда конфигурируется как предагрегат.

В датасете должна быть выбрана хотя бы одна версия переменной, загруженная под сущностью датасета или сегмент под сущностью датасета (без использования связей).

Функции пользовательских формул

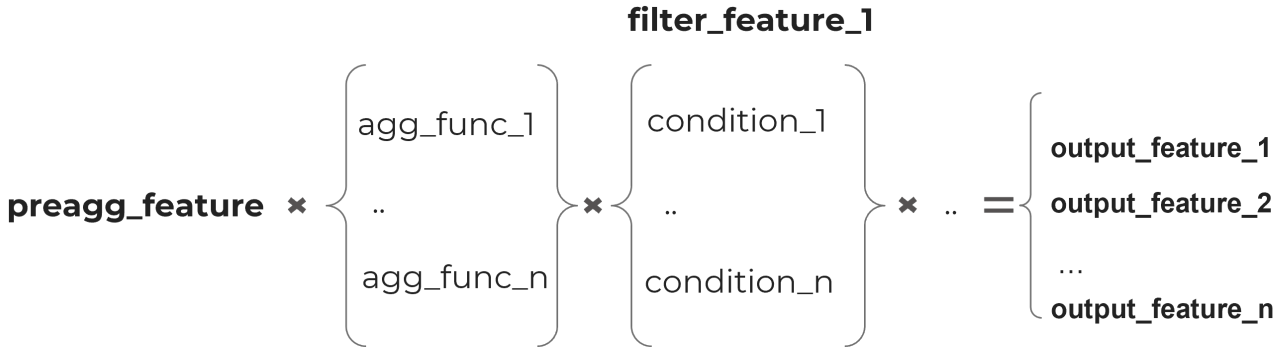
При конфигурации переменных датасета допускается использование формул с использованием математических операторов (сложение, умножение, сравнение etc.), а так же SQL функций, таких как `between`, `like`, `date_add` etc.

Пользовательские функции и формулы в SDK:

```
func.and_(
    ds_feat.str_feature.not_like('%A'),
    ds_feat.num_feature > 1000
).alias('TEST_LVL2_FEATURE')

func.coalesce(
    ds_feat.num_feature,
    ds_feat.num_feature_2,
    else_=-1
).alias('TEST_LVL2_FEATURE')


ds_feat.SOME_FEATURE.lead(
    offset=1,
    partition_by=[
        ds_feat.CITY
    ],
    order_by=[
        ds_feat.CUSTOMER_AGE
    ]
).alias('LEAD2_SOME_FEATURE')

func.ltrim(
    ds_feat.str_feature_1
).alias('TEST_LVL2_FEATURE')

func.date_part(
    ds_feat.date_feature,
    'minute'
).alias('TEST_LVL2_FEATURE')
```

Пример конфигурирования предагрегата

В процессе конфигурирования предагрегатов производится настройка агрегирующих функций и условий фильтрации агрегируемых данных. Итоговый набор переменных датасета генерируется по формуле:

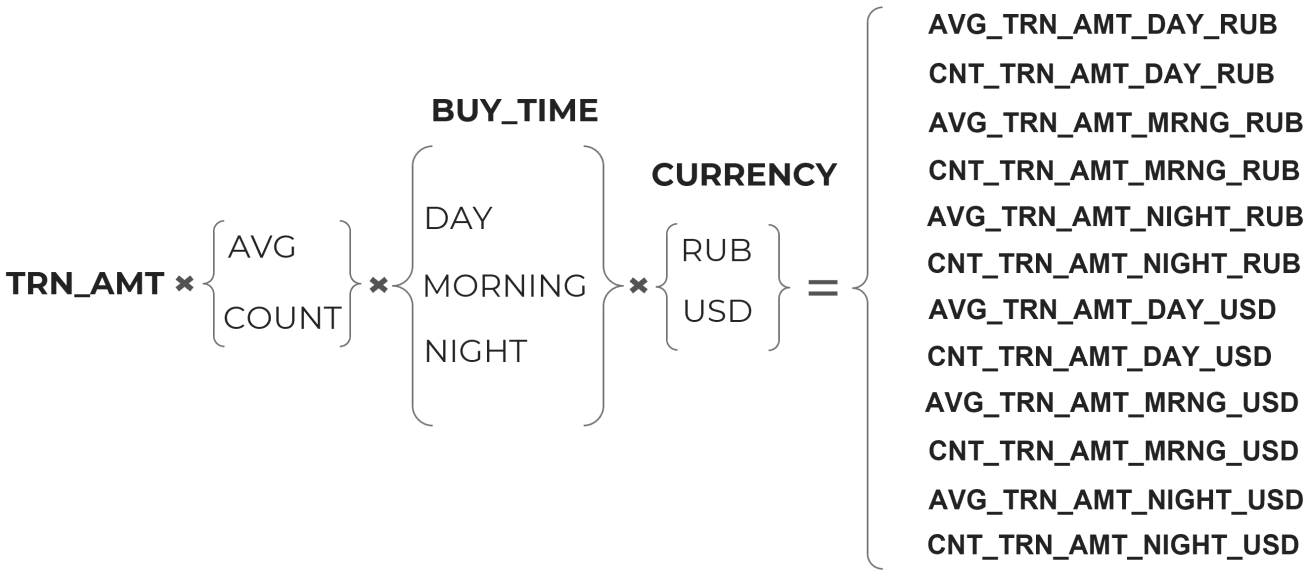


Пример - конфигурация предагрегата TRN_AMT:

- Агрегирующие функции - AVG, COUNT
- Условия фильтрации по версии фичи BUY_TIME:
- = MORNING
- = DAY
- = NIGHT
- Условия фильтрации по версии фичи CURRENCY:
- = RUB
- = USD

Итоговый набор переменных датасета:

1. AVG_TRN_AMT_DAY_RUB (`AVG` & `BUY_TIME = DAY` & `CURRENCY = RUB`)
2. CNT_TRN_AMT_DAY_RUB (`COUNT` & `BUY_TIME = DAY` & `CURRENCY = RUB`)
3. AVG_TRN_AMT_MRNG_RUB (`AVG` & `BUY_TIME = MORNING` & `CURRENCY = RUB`)
4. CNT_TRN_AMT_MRNG_RUB (`COUNT` & `BUY_TIME = MORNING` & `CURRENCY = RUB`)
5. AVG_TRN_AMT_NIGHT_RUB (`AVG` & `BUY_TIME = NIGHT` & `CURRENCY = RUB`)
6. CNT_TRN_AMT_NIGHT_RUB (`COUNT` & `BUY_TIME = NIGHT` & `CURRENCY = RUB`)
7. AVG_TRN_AMT_DAY_USD (`AVG` & `BUY_TIME = DAY` & `CURRENCY = USD`)
8. CNT_TRN_AMT_DAY_USD (`COUNT` & `BUY_TIME = DAY` & `CURRENCY = USD`)
9. AVG_TRN_AMT_MRNG_USD (`AVG` & `BUY_TIME = MORNING` & `CURRENCY = USD`)
10. CNT_TRN_AMT_MRNG_USD (`COUNT` & `BUY_TIME = MORNING` & `CURRENCY = USD`)
11. AVG_TRN_AMT_NIGHT_USD (`AVG` & `BUY_TIME = NIGHT` & `CURRENCY = USD`)
12. CNT_TRN_AMT_NIGHT_USD (`COUNT` & `BUY_TIME = NIGHT` & `CURRENCY = USD`)



3.7.2 Загрузка датасета из источника

Добавление нового датасета в каталог датасетов возможно через загрузчик (назначение загрузчика - segment).

Датасет, полученный таким образом, может быть использован при конфигурации нового датасета в конструкторе.

3.7.3 Стратификация

Получение новых датасетов из частей другого датасета из каталога в результате его нарезки датасета по заданным правилам.
Стратификация осуществляется в три этапа:

1. Группировка по указанным полям датасета
2. Сортировка данных перед нарезкой
3. Нарезка датасета, полученного в результате манипуляций в п.1 и п.2, на указанные в конфигурации части

Для стратификации доступны датасеты, полученные через конструктор или зарегистрированные загрузчиком.

Стратификация в SDK:

```
from FSClient.catalogue import dataset  # Каталог датасетов

ds_feautures = ds.My_Dataset.get_feautures()  # получение каталога фичей стратифицируемого датасета

ds.My_Dataset.split(
    parts=[  # выбор размера и названий выборок
        (<int размер выборки1>, <str, название выборки1>),
        ..
        (<int размер выборкиN>, <str, название выборкиN>)
    ],
    group=[  # optional - группировка по фичам датасета
        ds_feautures.my_feature_1,
        ..
        ds_feautures.another_feature_n
    ] = None,
    sort=[  # optional - сортировка исходного датасета перед стратификацией
            # по значениям фичей или ключей с заданными параметрами
        (ds.My_Dataset.entity.my_key_1, 'asc nulls_last'),  # сортировка по ключу
                                                            # доступная конфигурация asc/desc nulls_last/nulls_first
                                                            # Default value 'asc nulls_last'
        (ds_feautures.my_feature_1, 'desc'),  # сортировка по фиче
        ..
    ] = None,
    name=str  # базовое имя датасета, при значении не None заменяет название исходного датасета
)
```

3.7.4 Импутация

Создаёт новый датасет на базе существующего с заменой пустых значений в столбцах на результат импутитирующей функции.

Требования:

- Исходный датасет должен быть непустым
- Структура нового датасета полностью совпадает со структурой исходного датасета
- Поля из исходного датасета, не определенные в правилах импутации, копируются в новый датасет as-is
- Если переданы поля группировки, то данные для импутации агрегируются в разрезе переданных полей, иначе рассчитываются общие агрегаты для всего датасета
- После завершения расчета нового датасеты выполняется расчет базовых статистических показателей

Доступные функции для импутации:

- **mean** – среднее арифметическое
- **min** – минимальное значение
- **max** – максимальное значение
- **median** – медианное значение
- **mode** - мода, самое частое значение
- **const** – константа из "VALUE"

Импутация в SDK:

```
from FSClient.catalogue import (
    dataset,
    function
)

new_ds = dataset.My_Dataset.impute(  # Returns an imputation dataset obj.
    name = 'new_dataset',
    description = 'new_dataset description'
)
output = new_ds.get_features()  # local features catalogue from current dataset
new_ds.add_rule(
    features=[output.any_feature, 'my_another_feature'],  # feature from output catalogue
                                                          # or feature name as str
    func=function.median(),  # avg, median, mode, max, min, const from function catalogue

    # feature from output catalogue
    # or feature name as str
    # or datasets entity key
    # or report_dttm
    group=[output.some_feature, new_ds.entity.customer_entity_key, 'report_dttm'],
    const=50  # constant value of imputation
)
new_ds.save_x_execute()
```

4. How-to

4.1 Подготовка данных

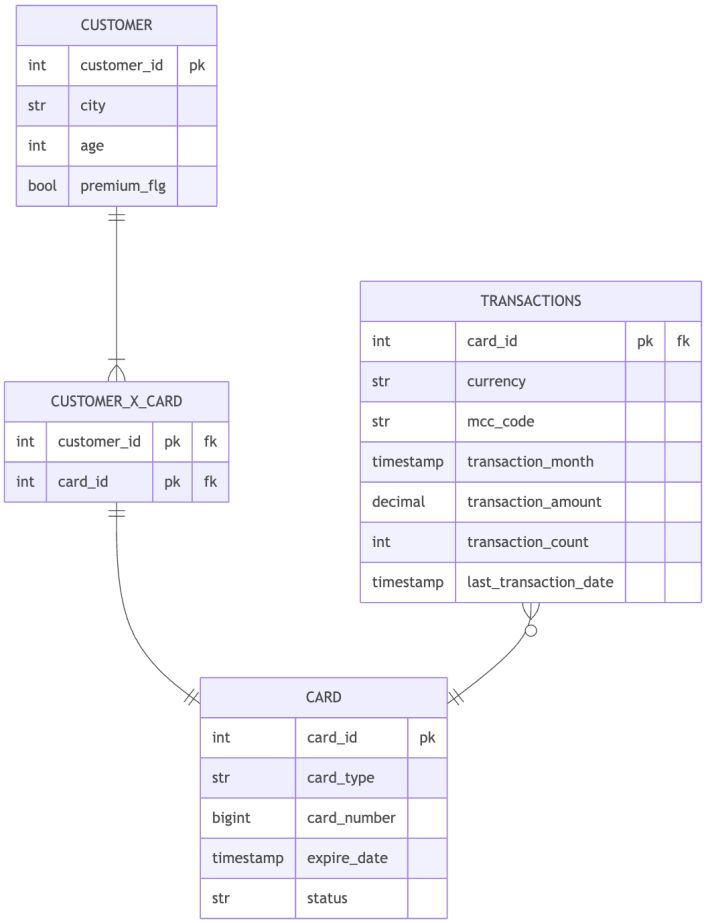
Рассмотрим банковское хранилище, в котором имеются данные о клиенте, его картах и транзакциях по картам.

4.1.1 Концептуальная схема

Перед тем, как регистрировать данные в приложении, необходимо спроектировать концептуальную схему хранения данных:

1. Выделить набор сущностей
2. Определиться с атрибутами сущностей
3. Определить связи между сущностями

Пример подготовленной концептуальной схемы для нашего примера:



Опишем получившиеся сущности:

- Клиент - характеристики клиента. Набор ключей определяется Primary keys исходной таблицы, значит ключ будет один - `customer_id`
- Карта - характеристики карты. Ключ - `card_id`

Атрибуты сущностей по таблицам:

- Таблицы CUSTOMER и CARD содержат в себе справочные данные о своих сущностях. Такие данные уже представлены в готовом виде.
- Таблица TRANSACTIONS содержит данные о транзакциях по картам, которые предварительно агрегированы до карты в разрезе валюты (`currency`), торговой категории (`mcc_code`) и месяца транзакции (`transaction_month`). Для использования данных в аналитике их потребуется агрегировать в разрезе сущности, т.к. получившаяся структура данных является предагрегатом.

Связи сущностей:

- Таблица CUSTOMER_X_CARD является "мостом" между сущностями Клиент и Карта. Она используется для получения информации о картах клиента и наоборот. Тип связи 1:M, где на одного клиента может быть зарегистрировано несколько карт.

4.1.2 Требования к таблицам данных

Следующим этапом будет соотнесение физических данных (таблиц) с их представлениями в приложении.

CUSTOMER

Содержит в себе справочные данные о клиентах банка.

Сущность - Клиент.
Тип загрузчика - загрузчик готовых фичей.

Требования к таблице:

- В атрибуте `customer_id` нет пропусков
- Значения `customer_id` уникальны
- Типы данных атрибутов таблицы соответствуют своей природе, например `age` (возраст) - целое число

Для регистрации загрузчика в WEB-интерфейсе нужно выбрать "Готовые переменные".

CARD

Содержит в себе справочные данные о картах клиента.

Сущность - Карта.
Тип загрузчика - загрузчик готовых фичей.

Требования к таблице:

- В атрибуте `card_id` нет пропусков
- Значения `card_id` уникальны
- Типы данных атрибутов таблицы соответствуют своей природе, например `expire_date` (дата окончания срока действия карты) - дата, `card_id` - целое число

Для регистрации загрузчика в WEB-интерфейсе нужно выбрать "Готовые переменные".

TRANSACTIONS

Содержит в себе данные о транзакциях по картам.

Сущность - Карта.
Тип загрузчика - загрузчик предагрегатов.

Требования к таблице:

- В атрибутах `card_id` нет пропусков
- Типы данных атрибутов таблицы соответствуют своей природе, например `transaction_amount` (размер транзакции) - decimal. Важно, чтобы `card_id` имел одинаковую размерность с `card_id` в таблице CARD: если `CARD.card_id` - integer, то и `TRANSACTIONS.card_id` - integer (не varchar и не decimal)

Для регистрации загрузчика в WEB-интерфейсе нужно выбрать "Предагрегат".

CUSTOMER_X_CARD

"Мост" между сущностями Клиент и Карта.

Сущность - поскольку таблица связывает сущности, в ней имеются две полноценные сущности - CUSTOMER, CARD.
Тип загрузчика - загрузчик связи сущностей.

Требования к таблице:

- В атрибутах `customer_id` и `card_id` нет пропусков
- Сочетание атрибутов `customer_id` и `card_id` - уникально
- Типы данных `customer_id` и `card_id` соответствуют типам данных аналогичных полей в таблицах CUSTOMER и CARD

Для регистрации в WEB-интерфейсе нужно выбрать "Связь сущностей"

4.1.3 Общие требования и рекомендации к таблицам

Выше был рассмотрен конкретный пример работы с данными и ключевые требования к разным типам загрузчиков. Для корректной работы приложения выделим основные **требования** ко всем регистрируемым таблицам:

- В атрибутах ключей не должно быть null-ов
- Типы данных полей ключей в таблицах источника должны соответствовать типам данных, которые определены для этих ключей
- Согласовывать типы данных одинаковых атрибутов между таблицами, например:
- Размерность полей с плавающей точкой должны быть одинаковой
- Соблюдать порядок целочисленных значений (int, bigint или smallint) для измеряемых значений
- Во всех таблицах сущности должны присутствовать атрибуты всех ключей сущности
- Нет дублей по ключам для загрузчиков готовых переменных
- Если для подготовки данных требуются сложные операции, то их нужно выполнять перед загрузкой в витрины и не вносить внутрь представления:
- Предварительная агрегация
- Сложная фильтрация
- Долгие JOIN-ы

Рекомендации:

- Использовать в атрибутах ключей целочисленные типы данных
- Не создавать сущности с тремя и более ключами
- Не плодить много сущностей - это снизит скорость работы с данными и ухудшит опыт работы с продуктом. Лучше заранее уделить время денормализации витрин и выделении ключевых сущностей. Как правило, порядка 20 сущностей достаточно для сложных банковских хранилищ.
- При проектировании сущностей стоит учесть, что приложение будет выполнять все JOIN-ы именно по ключам этих сущностей.
- Использовать версионность данных (период актуальности) только там, где необходимо хранить историю. В примере выше это может быть таблица TRANSACTIONS, когда данные поступают регулярно и предварительно агрегируются за конкретный период (например атрибут transaction_amount содержит в себе сумму транзакций за последнюю неделю в разрезе конкретной карты)
- Стараться не использовать представления и хранить данные для приложения в таблицах

4.2 Регистрация сущности

Пример регистрации сущности с созданием нового ключа и использованием существующего. Дополнительно указывается ключ дистрибуции, который можно выбрать только из ключей, входящих в состав сущности. Ключи дистрибуции работают только при условии поддержки дистрибуции БД.

Создание сущности в SDK:

```
from fsclient.catalogue import (
    entity as ent,
    data_type as dttp
)

customer_agreement = ent.create()

customer_agreement.config = {
    'name': 'CUSTOMER_AGREEMENT',     # Название сущности
    'description': 'Клиент-договор',  # Описание сущности
    'entity_keys': [
        {
            'name': 'AGREEMENT_RK',  # Название нового ключа
            'description': 'Суррогатный ключ договора',  # Описание ключа
            'data_type': dttp.INT    # Тип данных ключа
        },
        ent._keys.CUSTOMER_RK  # <-- Существующий ключ
    ]
}

customer_agreement.storage_key = [  # ключи дистрибуции (опционально)
    ent._keys.CUSTOMER_RK
]

customer_agreement.save()  # Регистрация новой сущности
```

4.2.1 Автоматические связи сущностей

При регистрации сущности с использованием существующих ключей, будут созданы связи между регистрируемой сущностью и всеми остальными сущностями, имеющими используемые ключи.

Тип связи в этом случае определяется по следующим правилам:

- `1:1` - полное пересечении ключей между сущностями
- `1:M` - ключи одной из сущностей полностью входят в набор ключей другой сущности, у которой есть минимум один ключ, не входящий в набор ключей первой сущности
- `M:N` - ключи сущностей пересекаются частично, то есть у каждой сущности из связи есть минимум один ключ, отсутствующий у другой

Созданные автосвязи появляются в каталоге связей сущностей.

4.3 Регистрация фичи

Пример регистрации фичи в SDK:

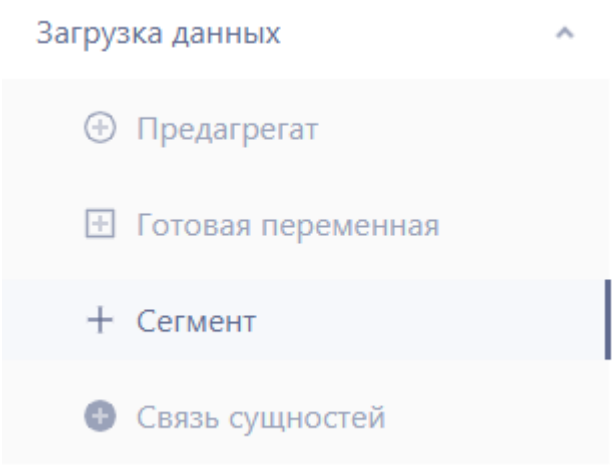
```
from fsclient.catalogue import feature

my_new_feature = feat.create()

my_new_feature.config = {
    'name': 'TRANSACTION_AMT',                # Название
    'description': 'Размер транзакции, руб.',  # Описание
}

my_new_feature.save()  # Регистрация новой фичи
```

4.4 Загрузка данных

4.4.1 Загрузка данных

В разделе описаны примеры регистрации сущностей приложения, которые описывают данные и связи между ними.

Меню загрузчиков данных по содержанию:

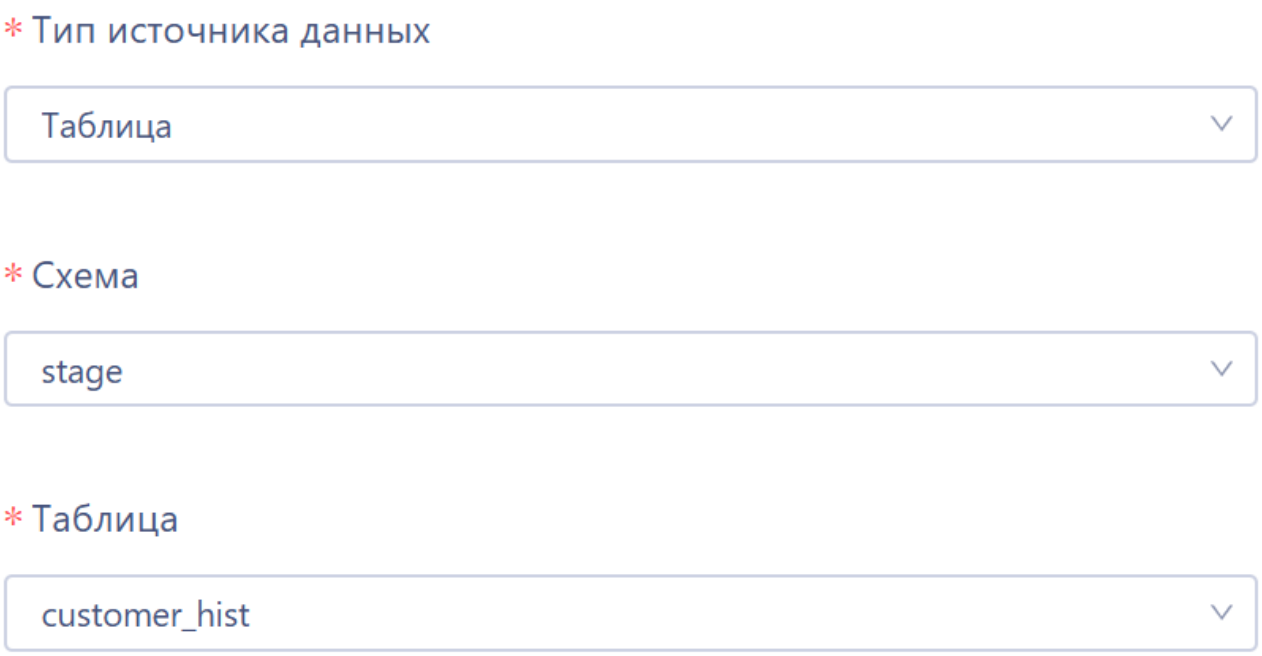


Источник

Выбор источника является первым этапом регистрации данных. В зависимости от типа источника требуются разные данные об источнике.

ТАБЛИЦА/ССЫЛКА НА ТАБЛИЦУ

- В поле «Тип источника данных» выбрать «Таблица»
- Выбрать схему из доступных
- Указать таблицу



Пример конфигурации загрузчика из таблицы в SDK:

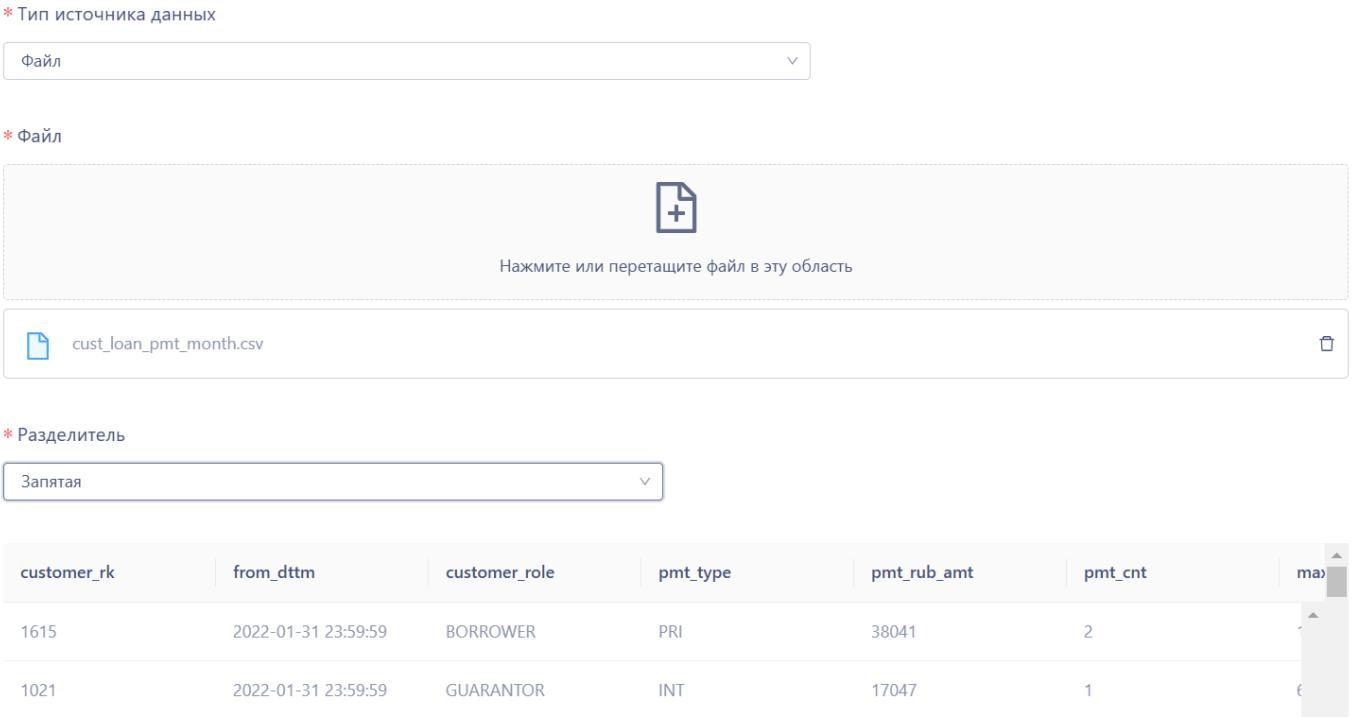
```
new_loader = load.create(
    source='table',    # <-- Тип источника данных
    target='segment'  # <-- Тип загрузчика
)

new_loader.config = {
    # Название загрузчика/сегмента
    'loader_name': 'CUSTOMER_FREE_DATA',
    # Пользовательское описание загружаемого объекта
    'description': 'Датасет из пользовательской песочницы',
    # Схема таблицы
    'src_schema_name': 'dataset',
    # Название таблицы
    'src_table_name': 'CUSTOMER_DATASET_1010',
    # Сущность таблицы
    'entity': ent.CUSTOMER
}
```

Пример конфигурации загрузчика ссылкой на таблицу в SDK:

```
new_loader = load.create(
    source='external'  # <-- Тип источника данных "Ссылка на таблицу"
)

new_loader.config = {
        # см. конфигурацию загрузчика из таблицы
}
```

ФАЙЛ

- В поле «Тип источника данных» выбрать «Файл»
- Загрузить файл с локального устройства в формате `.csv` / `.xlsx`
- Указать разделитель (актуально для файлов типа `.csv`)



Пример конфигурации загрузчика из файла в SDK:

```
new_loader = load.create(
    source='file',    # <-- Тип источника данных
    target='segment'  # <-- Тип загрузчика
)

new_loader.config = {
    # Название загрузчика/сегмента
    'loader_name': 'CUSTOMER_FREE_DATA',
    # Пользовательское описание загружаемого объекта
    'description': 'Датасет из пользовательской песочницы',
    # Разделитель файла
    'delimiter': ';',
    # Сущность таблицы
    'entity': ent.CUSTOMER
}

new_loader.select_file('data/customer_dataset.csv')
```

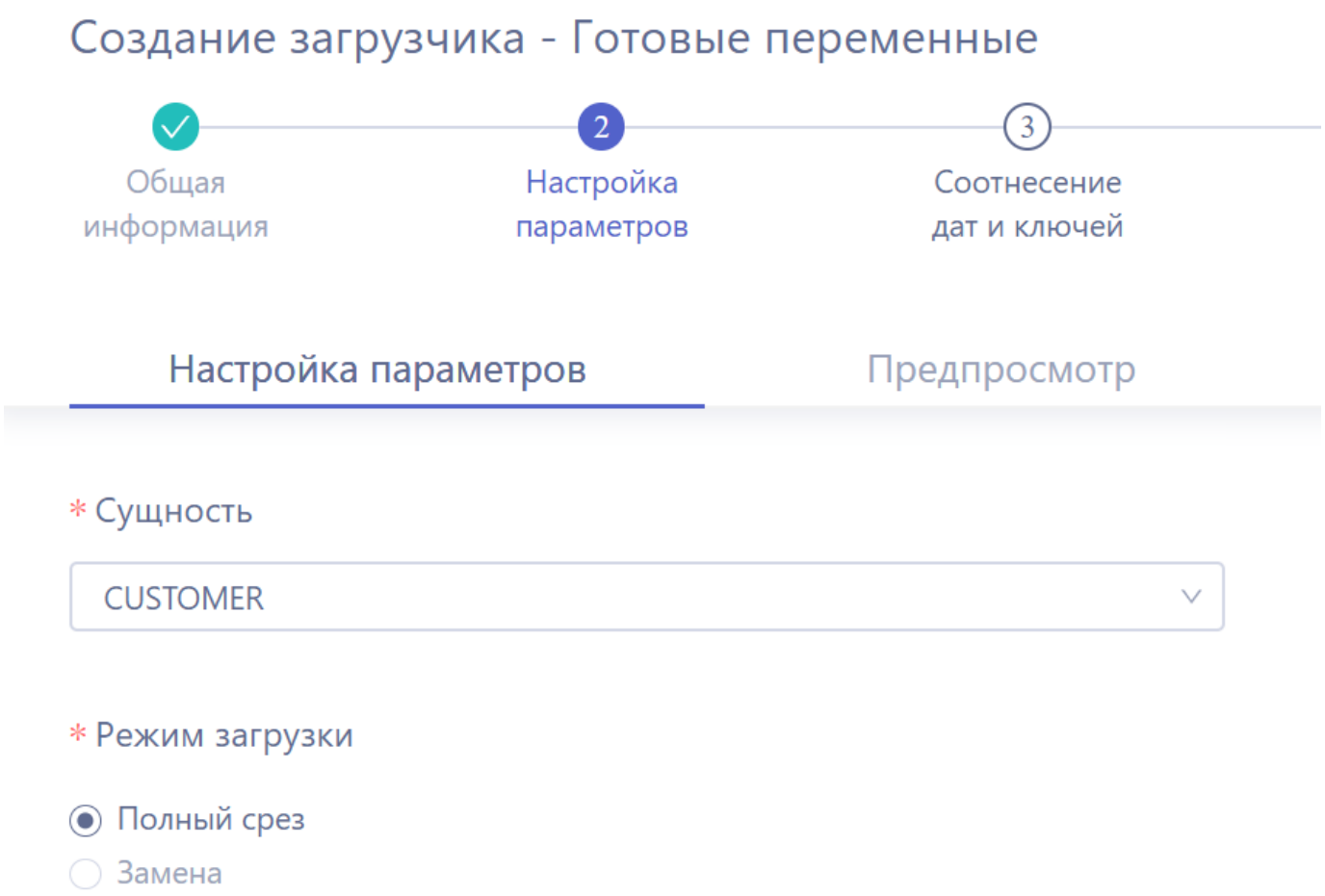
4.4.2 Загрузка фичей

В статье рассматривается регистрация версий фичей, которые используются для создания датасетов.

Готовые фичи

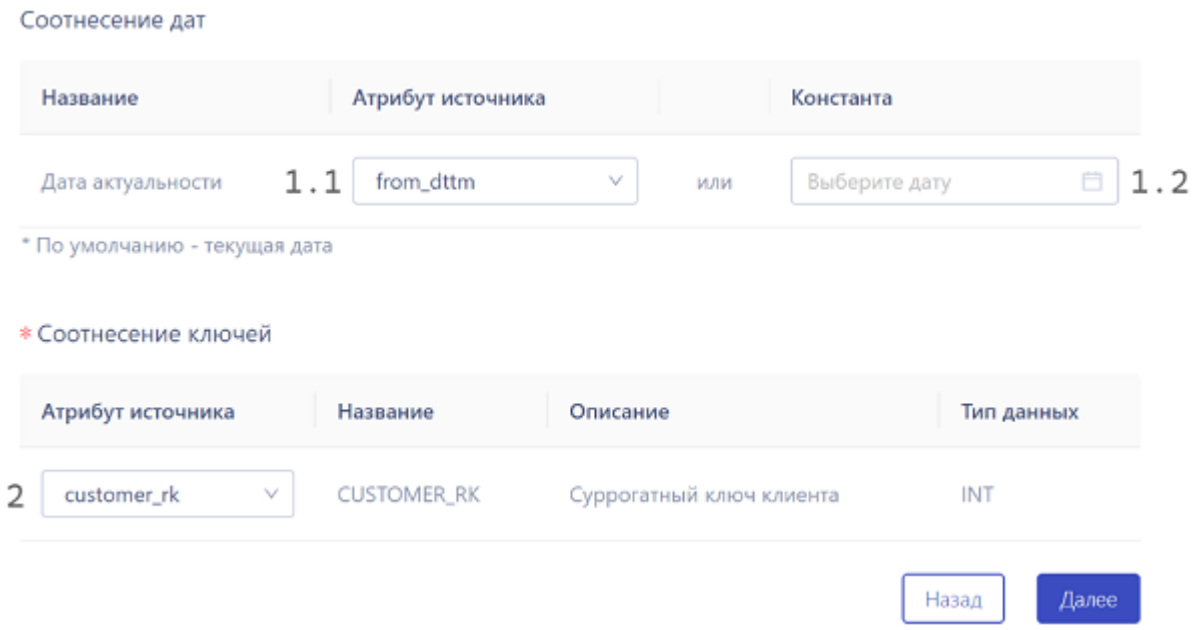
Настройка параметров:

- Выбрать сущность загрузчика
- Режим загрузки - полный срез («full») или замена («replace»)



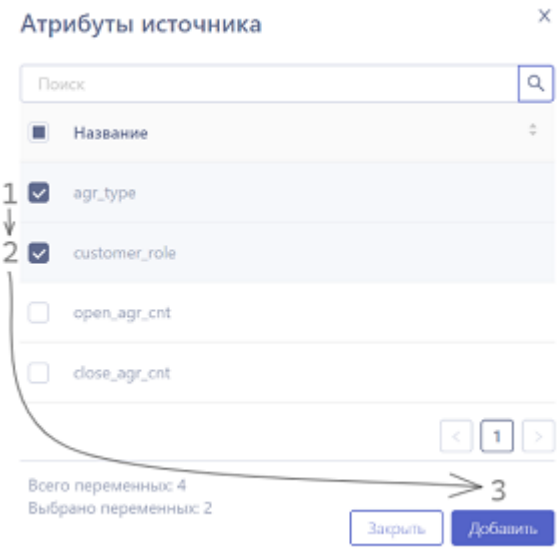
Соотнесение дат и ключей:

- Выбрать поле из источника, которое соответствует дате актуальности **(1.1)** или ввести дату вручную **(1.2)**
- Выбрать поля для ключей сущности из источника **(2)**

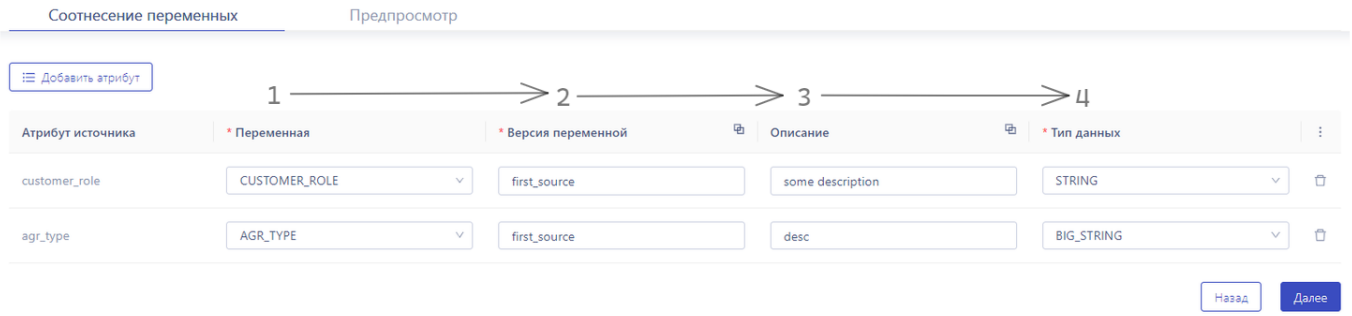


Соотнесение фичей и атрибутов источника:

Для добавления столбцов из источника «Добавить атрибут», где выбрать поля **(1, 2)** и добавить их в конфигурацию **(3)**:

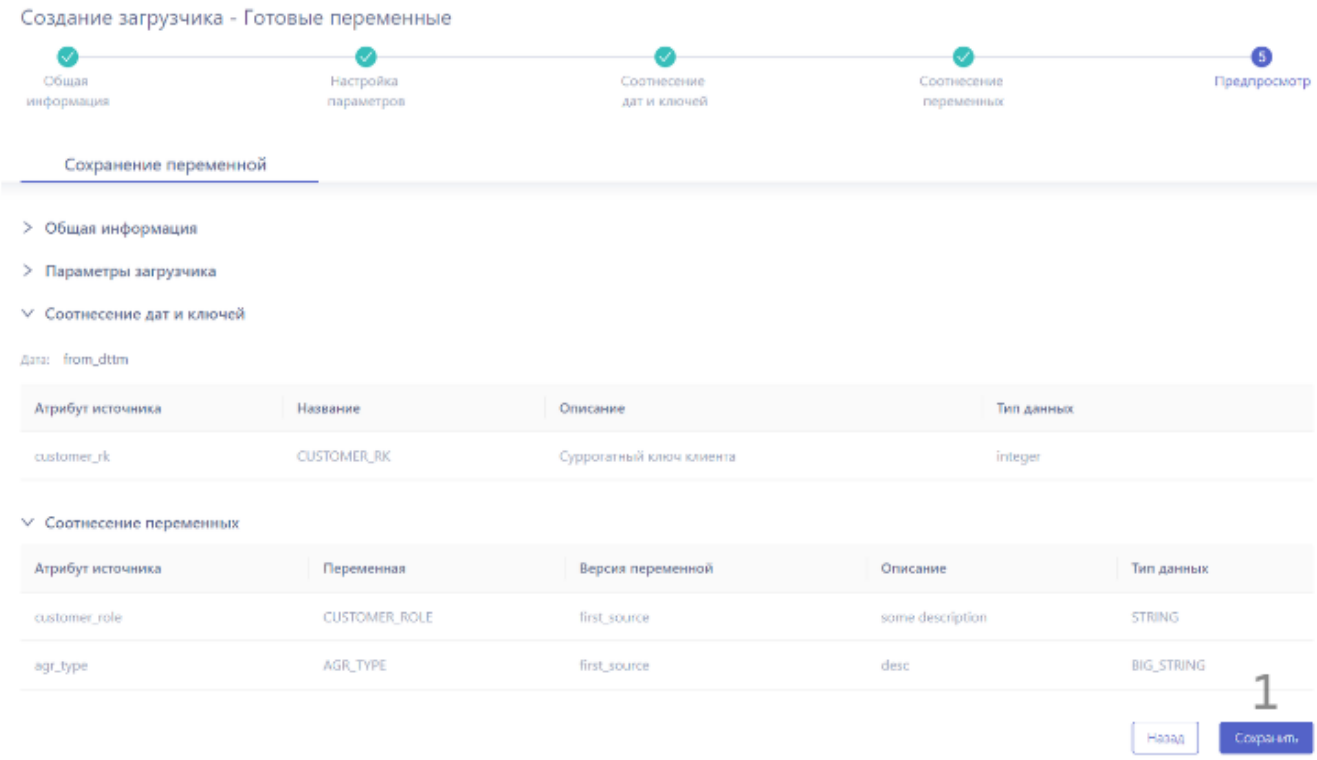


- Соотнести фичи **(1)**, зарегистрированные в приложении, к выбранным полям источника
- Ввести название версии фичи (переменной) **(2)**
- Описание переменной **(3)** (необязательное)
- Указать тип данных **(4)**



Предпросмотр:

Последним шагом создания загрузчика является предпросмотр, где можно увидеть всю конфигурацию загрузчика. Для регистрации нажать кнопку «Сохранить» **(1)**.



Пример конфигурации загрузчика готовых фичей в SDK:

```
from FSClient.catalogue import (  # <--  импорт необходимых каталогов
    entity as ent,
    feature as feat,
    loader as load
)

new_loader = load.create(
    source='table'  # <-- Тип источника данных
)

# Настройка конфигурации загрузчика
new_loader.config = {
    # Название загрузчика в каталоге
    'loader_name': 'CUSTOMER_INFO_HIST',
    # Пользовательское описание загрузчика
    'description': 'Информация по клиенту из карточки клиента АБС',
    # Схема таблицы
    'src_schema_name': 'stage',
    # Название таблицы
    'src_table_name': 'customer_hist',
    # Сущность таблицы
    'entity': ent.CUSTOMER
```

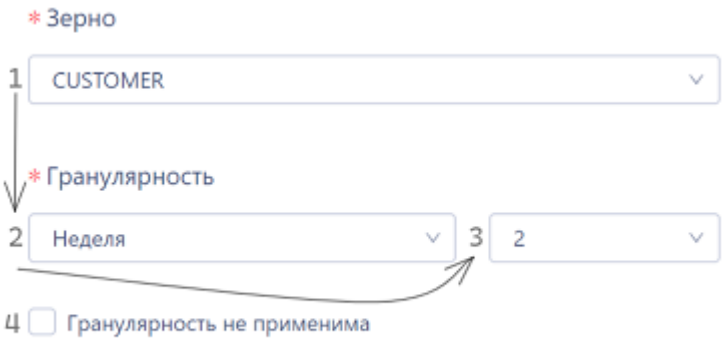
```
}

# Маппинг полей источника на сущности каталога
new_loader.mapping = {
    # в данном примере дату актуальности (from_dttm) не указываем
    # в таком случае период актуальности данных будет от текущей даты до бесконечности
    'entity': [
        {
            # Выбор ключа сущности из каталога
            'entity_key': ent.CUSTOMER.CUSTOMER_RK,
            # Название поля из таблицы, соответствующее выбранному ключу
            'stg_column_name': 'customer_rk'
        }
    ],
    'features': [
        {
            # Фича из каталога
            'feature': feat.CUSTOMER_AGE,
            # Название поля из таблицы
            'stg_column_name': 'customer_age',
            'feature_version_config': {
                # Название версии переменной в каталоге
                'name': 'V1_MAIN',
                # Пользовательское описание версии переменной
                'description': 'Карточка клиента из АБС: Возраст клиента',
                # Тип данных поля из таблицы
                'data_type': dttp.INT
            }
        },
        {
            'stg_column_name': 'gender',
            'feature': feat.GENDER,
            'feature_version_config': {
                'name': 'V1_MAIN',
                'description': 'Карточка клиента из АБС: Пол',
                'data_type': dttp.STRING
            }
        },
        {
            'stg_column_name': 'city',
            'feature': feat.CITY,
            'feature_version_config': {
                'name': 'V1_MAIN',
                'description': 'Карточка клиента из АБС: Наименование города',
                'data_type': dttp.STRING
            }
        }
   ]
}

# Расписание
new_loader.schedule = "0 9 1-7 * 1"  # первый понедельник каждого месяца в 9 утра

# Регистрация загрузчика
new_loader.save_x_execute()  # <-- регистрация (данных) загрузчика
```

Предагрегаты

Настройка параметров:

- Выбрать сущность загрузчика **(1)**
- Гранулярность: тип периода (неделя, месяц etc.) **(2)**
- Гранулярность: число, отвечающее за количество недель, месяцев etc. **(3)**
- Если у данных нет гранулярности, выставить чекбокс **(4)**

На примере ниже указана гранулярность данных в две недели.



Соотнесение дат и ключей:

- Выбрать поле из источника, которое соответствует дате актуальности или ввести дату вручную
- Выбрать поля для ключей сущности из источника

Соотнесение фичей и атрибутов источника:

Для добавления столбцов из источника «Добавить атрибут», где выбрать поля и добавить их в конфигурацию.

- Соотнести фичи, зарегистрированные в приложении, к выбранным полям источника
- Ввести название версии фичи (переменной)
- Описание переменной (необязательное)
- Указать тип данных

Предпросмотр:

Последним шагом создания загрузчика будет предпросмотр, где можно все увидеть всю конфигурацию загрузчика. Для регистрации нажать кнопку «Сохранить» **(1)**.

Пример конфигурации загрузчика предагрегатов в SDK:

```
from FSClient.catalogue import (  # <--  импорт необходимых каталогов
    entity as ent,
    feature as feat,
    loader as load
)

new_loader = load.create(
    source='table'  # <-- Тип источника данных
)

new_loader.config = {
        # Название загрузчика в каталоге
        'loader_name': 'CUST_SALARY_MONTH',
        # Пользовательское описание загрузчика
        'description': 'ЗП-начисления за месяц из КХД',
        # Схема таблицы
        'src_schema_name': 'public',
        # Название таблицы
        'src_table_name': 'cust_salary_month',
        # Сущность таблицы
        'entity': ent.CUSTOMER,
        # Флаг предагрегата
        'preagg_flg': True,
        # Тип гранулярности - календарный месяц
        'granularity_type': 'calendar_month',
        # Количество календарных месяцев гранулярности
        'granularity': 1
}

# Маппинг
new_loader.mapping = {
    'from_dttm': 'date_start',
    'to_dttm': 'actual_period_end_dttm',
    'entity': [
        {
            # Выбор ключа сущности из каталога
            'entity_key': ent.CUSTOMER.CUSTOMER_RK,
```

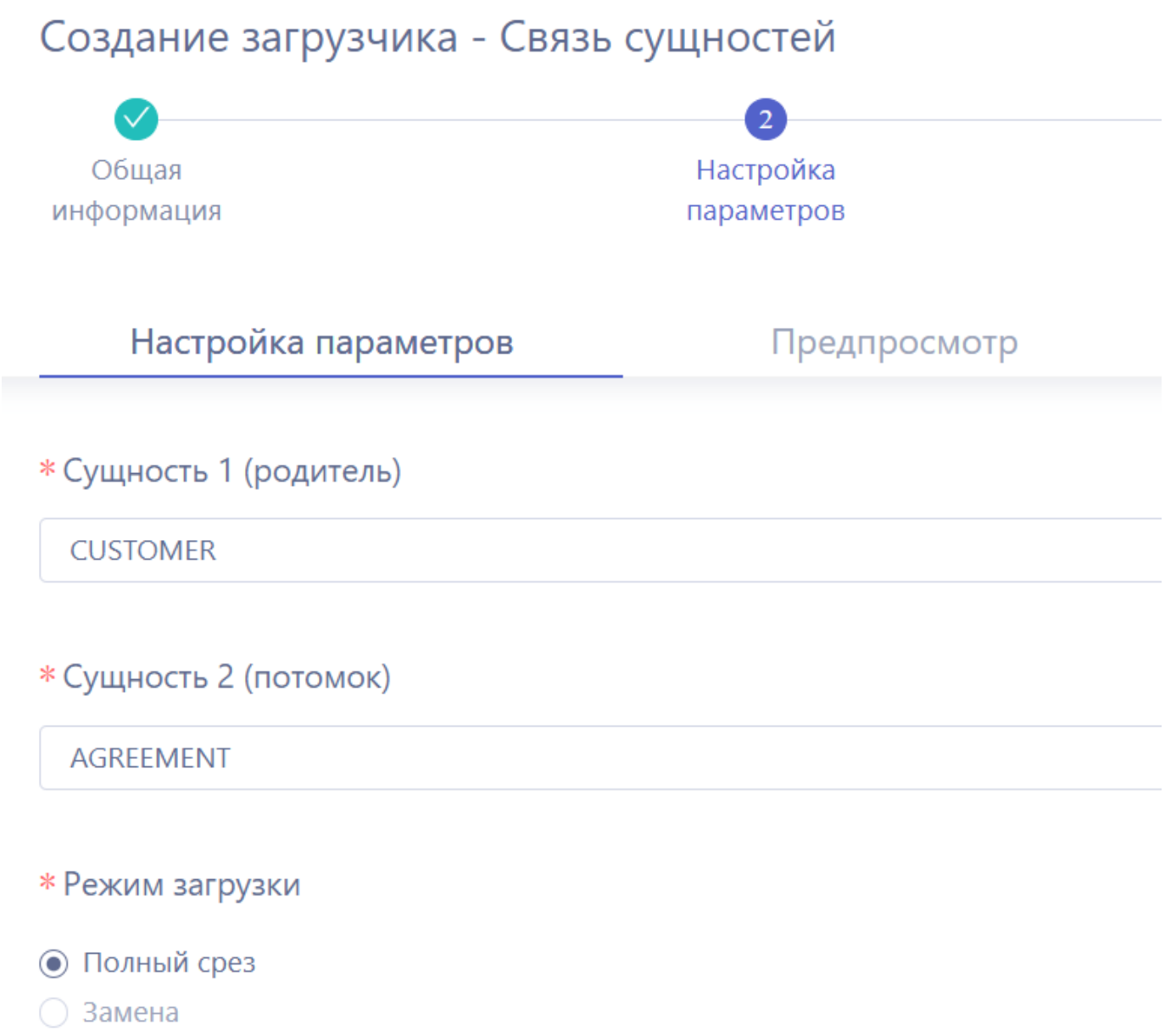
```
            # Название поля из таблицы, соответствующее выбранному ключу
            'stg_column_name': 'customer_rk'
        }
    ],
    'features': [
        {
            # Фича из каталога
            'feature': feat.SAL_SRC_NM,
            # Название поля из таблицы
            'stg_column_name': 'sal_src_nm',
            'feature_version_config': {
                # Название версии переменной в каталоге
                'name': 'SP_M_V1',
                # Пользовательское описание версии переменной
                'description': 'ЗП-начисления за месяц из КХД: Источник зп-начисления',
                # Тип данных поля из таблицы
                'data_type': dttp.STRING
            }
        },
        {
            'stg_column_name': 'sal_paym_amt',
            'feature': feat.SAL_PAYM_AMT,
            'feature_version_config': {
                'name': 'SP_M_V1',
                'description': 'ЗП-начисления за месяц из КХД: Сумма зп-начислений',
                'data_type': dttp.AMOUNT
            }
        },
        {
            'stg_column_name': 'sal_paym_cnt',
            'feature': feat.SAL_PAYM_CNT,
            'feature_version_config': {
                'name': 'SP_M_V1',
                'description': 'ЗП-начисления за месяц из КХД: Количество зп-начислений',
                'data_type': dttp.INT
            }
        }
    ]
}

new_loader.save_x_execute()  # <-- регистрация (данных) загрузчика
```

4.4.3 Связь сущностей

Описание процесса регистрации связи сущностей. С ее помощью можно использовать в датасете переменные, зарегистрированные под сущностью, отличной от сущности датасета.

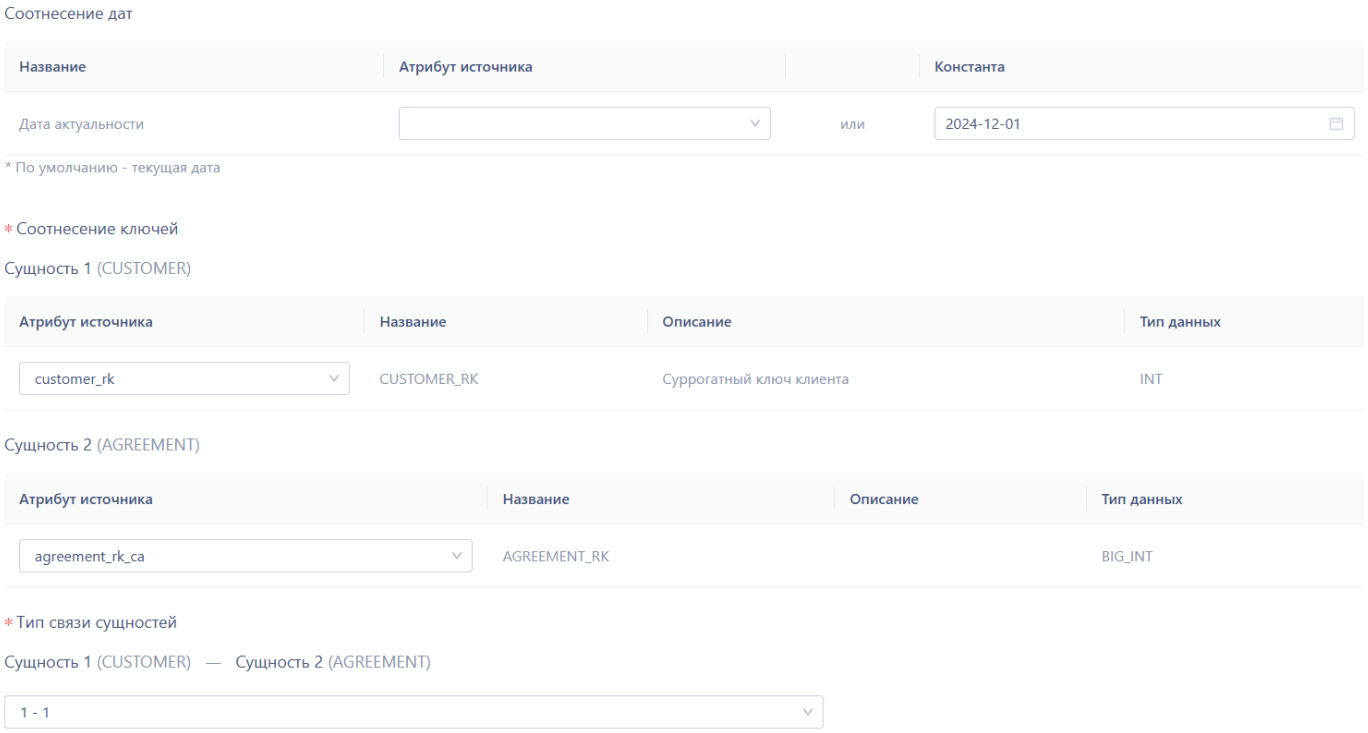
Настройка параметров:

- Выбрать сущность-родитель и сущность-потомок
- Режим загрузки - полный срез («full») или replace («replace»)



Соотнесение дат и ключей:

- Выбрать поле из источника, которое соответствует дате актуальности или ввести дату вручную
- Соотнесение ключей — выбрать поля для ключей сущности родителя и потомка из источника
- Указать тип связи между сущностями (1:1, 1:M, M:N)



Предпросмотр:

Последним шагом создания загрузчика будет предпросмотр, где можно все увидеть всю конфигурацию загрузчика. Для регистрации нажать кнопку «Сохранить».

Пример конфигурации загрузчика связи сущностей в SDK

```
from FSClient.catalogue import (  # <--  импорт необходимых каталогов
    entity as ent,
    feature as feat,
    loader as load
)

new_loader_link = load.create(
    source='file',        # <-- Тип источника данных
    target='entity_link'  # <-- Загрузка связи сущностей
)

# Настройка конфигурации загрузчика
new_loader_link.config = {
        # Название загрузчика (Связь в каталоге будет называться так же)
        'loader_name': 'CUSTOMER_x_AGREEMENT_1_1',
        # Пользовательское описание загрузчика
        'description': 'Связь между клиентом и договором',
        # Разделитель
        'delimiter': ',',
        # Сущность родитель
        'parent_entity': ent.CUSTOMER,
        # Сущность потомок
        'child_entity': ent.AGREEMENT,
        # Тип связи (допустимые: (1:1, 1:M, M:N))
        'kind': '1:1'
}

# Выбор локального файла
new_loader.select_file('data/customer_agreement.csv')

# Маппинг полей источника на сущности каталога
new_loader_link.mapping = {
    'from_dttm': 'from_dttm',
    'parent': [  # маппинг родительской сущности
        {
            'entity_key': ent.CUSTOMER.CUSTOMER_RK,
            'stg_column_name': 'customer_rk'
        }
    ],
    'child': [  # маппинг сущности потомка
        {
            'entity_key': ent.AGREEMENT.AGREEMENT_RK,
            'stg_column_name': 'agreement_rk'
        }
```

```
    ]
}

# Расписание
new_loader_link.schedule = "0 0 * * *"  # каждый день в полночь

# Регистрация
new_loader_link.save_x_execute()  # <-- регистрация (данных) загрузчика
```

4.4.4 Сегмент

Сегмент — список ключей определенной сущности с переменными. Маппинг фичей при регистрации сегмента не требуется. Все поля, не встречающиеся в маппинге ключей, размечаются как переменные получившегося датасета. После успешной регистрации загрузчика в каталоге появится одноименный датасет.

Настройка параметров:

- Выбрать сущность загружаемых данных

Соотнесение ключей:

- Выбрать поля для ключей сущности. Для сегментов с источником "Ссылка на таблицу" поля ключей из источника должны называться так же, как и сами ключи называются в каталоге.

Предпросмотр:

Последним шагом создания загрузчика будет предпросмотр, где можно все увидеть всю конфигурацию загрузчика. Для регистрации нажать кнопку «Сохранить».

Пример конфигурации загрузчика сегмента в SDK

```
from FSClient.catalogue import (  # <--  импорт необходимых каталогов
    entity as ent,
    loader as load
)

new_loader = load.create(
    source='external',    # <-- Тип источника данных
    target='segment'  # <-- Тип загрузчика
)

# Настройка конфигурации загрузчика
new_loader.config = {
        # Название загрузчика/сегмента
        'loader_name': 'CUSTOMER_FREE_DATA',
        # Пользовательское описание загружаемого объекта
        'description': 'Датасет из пользовательской песочницы',
        # Схема таблицы
        'src_schema_name': 'dataset',
        # Название таблицы
        'src_table_name': 'CUSTOMER_DATASET_1010',
        # Сущность таблицы
        'entity': ent.CUSTOMER
}

# Маппинг полей источника на сущности каталога
new_loader.mapping = {
    'entity': [
        {
            # Выбор ключа сущности из каталога
            'entity_key': ent.CUSTOMER.CUSTOMER_RK,
            # Название поля из таблицы, соответствующее выбранному ключу
            'stg_column_name': 'customer_rk'
        }
    ]
}

# Регистрация загрузчика
new_loader.save_x_execute()
```

4.5 Создание датасета

Пример регистрации датасета с использованием разных видов трансформаций данных, после которого будет разобран каждый шаг подробнее.

```
new_dataset = ds.create(
    # Сущность нового датасета
    entity=ent.CUSTOMER,
    # Название датасета в каталоге
    name='PURCHASE_NORMAL',
    # Пользовательское описание датасета
    description='str'
)

# Готовые фичи
new_dataset.add_feature(
    features=[
        feat.CUSTOMER_NAME.ADM_V1.alias('NAME'),
        feat.AGE.ADM_V1.alias('AGE')
    ]
)

# Агрегация с условием фичей предагрегатов
new_dataset.add_feature(
    features=[
        feat.PRICE.STORE_1.alias('PRICE')
    ],
    agg=[  # список агрегирующих функций
        func.sum(),
        func.max(),
        func.min()
    ],
    domain=[
        (feat.BONUS_PAYMENT.STORE_1 == True).set(alias='BP'),
        (feat.CATEGORY.STORE_1 == 'FOOD').set(alias='FOOD'),
        (feat.CATEGORY.STORE_1 == 'CLOTHES').set(alias='CLOTHES')
    ],
    link=[
        el.CUSTOMER_X_PURCHASE_STORE_1
    ]
)

# Составные фичи (фичи 2-го уровня)
ds_feat = new_dataset.get_features() # <-- каталог фичей датасета

new_dataset.add_feature(
    (
        (ds_feat.SUM_PRICE_BP_FOOD - ds_feat.MIN_PRICE_BP_FOOD) / (ds_feat.MAX_PRICE_BP_FOOD - ds_feat.MIN_PRICE_BP_FOOD)
    ).alias('PRICE_BP_FOOD_FOOD_NORMA')
)

# Фильтрация датасета
ds_feat = new_dataset.get_features()

new_dataset.filter = (ds_feat.AGE > 18) & (ds_feat.AGE <= 65)

# Регистрация
new_dataset.save_x_execute()
```

4.5.1 Готовые фичи

```
new_dataset.add_feature(
    features=[
        feat.CUSTOMER_NAME.ADM_V1.alias('NAME'),
        feat.AGE.ADM_V1.alias('AGE')
    ]
)
```

Готовые фичи добавляются в датасет as-is с возможностью указать алиас для них в итоговом датасете, как указано в примере.

Если фичи зарегистрированы под другой сущностью, в аргумент `link` передается связь сущностей или цепочка связей сущностей. Цепочка связей всегда должна приводить от сущности датасета к сущности фичи или фичей, указанных в `features`.

Использование связи сущностей может потребовать агрегации готовой фичи в следующих случаях:

- Использована связь M:N
- Использована связь 1:M, где дочерняя сущность (M) - сущность фичи

В таких ситуациях работать с фичами как с предагрегатами (см. "Предагрегаты").

Альтернативный вариант добавления готовых фичей в датасет:

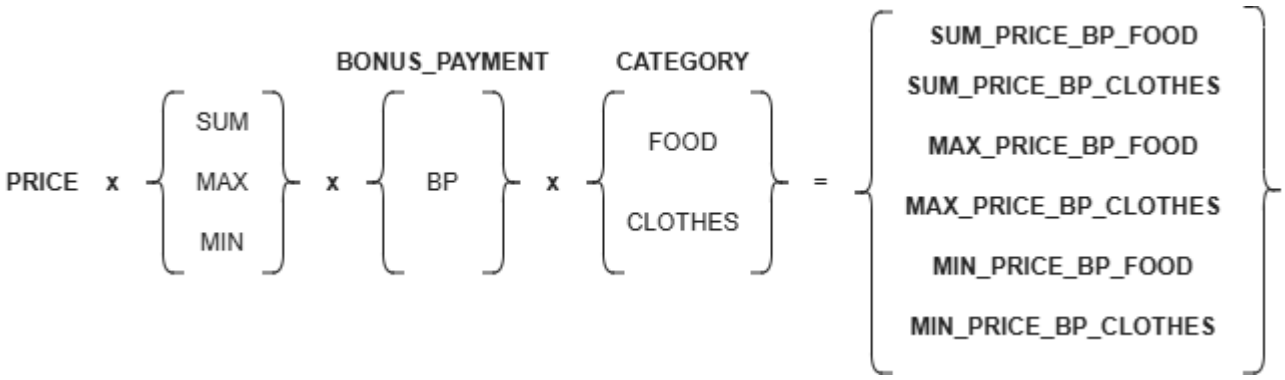
```
new_dataset.add_feature(
    feat.CUSTOMER_NAME.ADM_V1.alias('NAME'),
    feat.AGE.ADM_V1.alias('AGE')
)
```

4.5.2 Предагрегаты

```
new_dataset.add_feature(
    features=[
        feat.PRICE.STORE_1.alias('PRICE')
    ],
    agg=[  # список агрегирующих функций
        func.sum(),
        func.max(),
        func.min()
    ],
    domain=[
        (feat.BONUS_PAYMENT.STORE_1 == True).set(alias='BP'),
        (feat.CATEGORY.STORE_1 == 'FOOD').set(alias='FOOD'),
        (feat.CATEGORY.STORE_1 == 'CLOTHES').set(alias='CLOTHES')
    ],
    link=[
        el.CUSTOMER_X_PURCHASE_STORE_1
    ]
)
```

Для добавления предагрегатов в датасет требуется указать функции агрегации в аргумент `agg`. Набор доступных функций агрегации — см. глава "Функции агрегации".

Условия, передаваемые в аргумент `domain`, будут использованы в операторе `CASE` внутри указанных функций агрегации. Для составления условий могут быть использованы функции модуля `functions` (см. глава "Функции трансформации"). В примере выше мы получим 6 фичей в датасете:



Фичи, которые могут быть использованы в условиях:

- Готовые фичи
- Любые фичи, которые загружены одним загрузчиком с фичами в списке `features`

В аргументе `link` передается связь сущностей или цепочка связей сущностей если используемый предагрегат зарегистрирован под другой сущностью. Цепочка связей всегда должна приводить от сущности датасета к сущности фичи или фичей, указанных в `features`.

В аргумент `features` при агрегации можно передать несколько фич-предагрегатов. В таком случае к ним будут применены одинаковые функции, условия и связи сущностей. При этом фичи должны быть зарегистрированы под одинаковой сущностью.

4.5.3 Составные фичи (фичи 2 уровня)

```
ds_feat = new_dataset.get_features() # <-- каталог фичей датасета

new_dataset.add_feature(
    (
        (ds_feat.SUM_PRICE_BP_FOOD - ds_feat.MIN_PRICE_BP_FOOD) / (ds_feat.MAX_PRICE_BP_FOOD - ds_feat.MIN_PRICE_BP_FOOD)
    ).alias('PRICE_BP_FOOD_FOOD_NORMA')
)
```

Составные фичи получаются из фичей датасета. Для создания такой фичи требуется получить каталог фичей датасета `ds_feat`. Доступный набор функций см. глава "Функции трансформаций".

Для составных фичей обязательно указания алиаса, как в примере.

4.5.4 Фильтрация датасета

```
ds_feat = new_dataset.get_features()

new_dataset.filter = (ds_feat.AGE > 18) & (ds_feat.AGE <= 65)
```

Фильтрация датасета осуществляется фичами датасета. Возможности трансформаций полностью соответствуют функционалу получения составных фичей.

Переданное выражение попадает в условие `WHERE` при расчете датасета, поэтому его результат должен быть булевым.

4.5.5 Функции агрегирования

Все функции представлены в модуле `function`, пример использования:

```
new_dataset.add_feature(
    features=[
        feat.PRICE.STORE_1.alias('PRICE')
    ],
    agg=[
        func.sum(),
        func.max(),
        func.min()
    ]
)
```

- `sum()`
- `avg()`
- `max()`
- `min()`
- `count()`
- `count_distinct()`

4.5.6 Функции трансформаций

Все функции представлены в модуле `function`, пример использования функции `between`:

```
ds_feat = my_ds.get_features()

my_ds.add_feature(
    (
        func.between(ds_feat.AGE, 18, 35)
```

```
    ).alias('test_feature_between')
)
```

- `and_(<feature expression (bool) 1>, .., <feature expression (bool) N>)`
- `or_(<feature expression (bool) 1>, .., <feature expression (bool) N>)`
- `all(<feature 1>, .. <feature N>)`
- `between(<feature>, <left value>, <right value>)`
- `case((<WHEN feature expression 1>, <THEN value 1>), .. , else_=<ELSE value, default NULL>)`
- `ceil(<feature>)`
- `coalesce(<feature 1>, .. , <feature N>, _else=<ELSE value>)`
- `concat(<feature 1>, .., <feature N>)`
- `concat_ws(<feature 1>, .., <feature N>, sep=<separate value>)`
- `date_add(`
  `<feature>, year=<years count>, month=<months count>, week=<weeks count>, day=<days count>,`
  `hour=<hours count>, minute=<minutes count>, second=<seconds count>`
  `)`
- `date_sub(`
  `<feature>, year=<years count>, month=<months count>, week=<weeks count>, day=<days count>,`
  `hour=<hours count>, minute=<minutes count>, second=<seconds count>`
  `)`
- `date_part(<feature>, unit=<unit value ex. 'century', 'day', 'month'>)`
- `date_trunc(<feature>, field=<trunc value ex. 'minute', 'year'>)`
- `floor(<feature>)`
- `length(<feature>)`
- `like(<feature>, <like value>)`
- `like_any(<feature>, <like value>)`
- `like_not(<feature>, <like value>)`
- `ln(<feature>)`
- `log(<feature>)`
- `ltrim(<feature>)`
- `replace(<feature>, to_replace=<replace value>, replacement_string=<replase TO value>)`
- `reverse(<feature>)`
- `rtrim(<feature>)`
- `sqrt(<feature>)`
- `strpos(<feature>, substring=<sub string value>)`
- `substr(<feature>, start_pos=<START value, defatult=1>, length=<LENGTH value, default=None>)`
- `lower(<feature>)`
- `translate(<feature>, str_to_replace=<replace value>, replacement_str=<replace TO value>)`
- `trim(<feature>, trim_str=<trim value>, trim_type=<one of ['leading', 'trailing', 'both']>)`
- `trunc(<feature>, digit=<DIGIT value, default 0>)`
- `upper(<feature>)`

4.6 Подготовка данных

Для работы с Morphism и создания датасетов требуется наполнить его данными. Ниже описан подход к проектированию архитектуры загружаемых объектов.

4.6.1 Pipeline подготовки данных для формирования датасета

Порядок действий для загрузки данных в Morphism представлен на следующей схеме. Далее более детально описан каждый из этапов.

Схема описывает процесс для ситуации, когда в приложении отсутствуют не только необходимые данные, но и структуры (сущности, фичи).

Уже созданные сущности и фичи можно переиспользовать при загрузке новых данных. Если в приложении уже настроена загрузка всех нужных данных и они покрывают все требования к новому датасету, процесс можно свести к выполнению только 1, 6, 7 шага.

4.6.2 Что можно грузить в Axiom

По "форме"

Доступна настройка следующих источников:

- Таблица в БД
- Ссылка на таблицу в БД
- CSV / XLSX файл
- SQL-скрипт

Подробнее см. в разделе "Загрузчики"

По "содержанию"

Наполнение таблиц можно разделить на 3 типа. У всех типов различные возможности по использованию при генерации датасетов.

Тип входных данных	Описание	Агрегация	Работа с историей	Пример
Готовые переменные	Данные, рассчитанные в целевом виде, используются в датасетах "как есть".	Одно значение сущности - одна запись на дату.	Нет, используются данные только на дату построения датасета	Пол, дата рождения, место работы
Предагрегат - транзакционный	Транзакционные данные, предварительно агрегированные (если применимо) за определенный фиксированный период - гранулярность.	Возможно несколько записей на дату для одного значения сущности.	Да, возможно использование данных за разные периоды и агрегирование нужного количества срезов.	Количество транзакций за неделю; Сумма ЗП-начислений за месяц
Предагрегат - общего назначения	Детализированная информация по сущности под расширенным ключом.	Возможно несколько записей на дату для одного значения сущности.	Да, данные агрегируются только на дату, не за период. Можно выбрать дату из истории в формате "N периодов от даты построения датасета".	Договоры клиента; Ответы БКИ по клиенту (сущность - Клиент)

Обязательные атрибуты

Сущность - один или несколько атрибутов, являющихся PRIMARY KEY таблицы источника, в разрезе которых будет рассчитываться будущая модель. (см. подробнее)

Дата актуальности (опционально) - дата, на которую актуальны данные.

Дата окончания актуальности (опционально) - дата, до которой актуальны данные. Может быть указана только при наличии даты актуальности.

При загрузке данных в Axiom можно указать в качестве источника дат периода актуальности:

- Атрибут из загружаемой таблицы/файла;
- Фиксированную дату;
- Текущую дату загрузки (используется по умолчанию, если пользователь ничего не указал).

Для двух последних пунктов в источнике не требуется наличие атрибута с датой актуальности.

4.6.3 Проектирование объектов с исходными данными

Исходные требования к датасету

Подход к проектированию входной структуры данных будет описываться на примере в банковской бизнес-области. Этот набор фичей содержит информацию по клиенту, его договорам, транзакциям.

Этот пример охватывает разные виды данных, при необходимости его можно экстраполировать на другие предметные области.

Клиентская информация

Название	Описание	Период расчета
Клиентская информация		-
GENDER	Пол	-
AGE	Возраст	-
PROPERTY_CNT	Количество видов собственности	-
Договоры		-
FIRST_OPEN_DT_MONTHS_CNT	Количество месяцев с даты открытия первого кредита	-
LOAN_CNT_PERIOD	Общее количество кредитных договоров за период	6M, 12M
LOAN_CNT_RATIO	Отношение количества кредитных договоров на отчетную дату к количеству за период	6M, 12M, 24M
Транзакции		-
TRN_INPUT_CASH_AMT	Сумма транзакций внесения наличных по карте за период	6M, 12M, 24M
TRN_INPUT_NONCASH_AMT	Сумма транзакций внесения безнал по карте за период	6M, 12M, 24M
TRN_ABROAD_AMT	Сумма расходных транзакций за границей за период	3M, 6M, 12M
TRN_VARIANCE	Дисперсия расходных транзакций за период	3M, 6M, 12M

Рекомендуется "схлопывать" требования до основных сущностей, например "сумма транзакции", "количество договоров" и т.д. Необходимые фильтры перечислять в соседних столбцах. Так будет удобнее формировать датасеты и проектировать источники переменных для Axiom.

Определение сущности

Сущность - один или несколько атрибутов по которым необходимо рассчитывать модель.

Примеры сущностей:

- ID клиента - если интересен скор по каждому клиенту в общем;
- ID клиента +ID точки продаж - если рассчитываем модели для каждой точки продаж отдельно.
- ID точки продаж - если интересны показатели точек, без детализации по клиентам.

Определение набора базовых фичей

Имея требования к датасету, нужно определить набор фичей.

На примере требований выше описывается процесс формирования списка фичей, указывается к какому типу входных данных оптимальнее отнести каждую переменную.

Требование	Фича	Тип данных	Комментарий
Пол	Пол	Готовая переменная	Не требует преобразований
Возраст	Дата рождения	Готовая переменная	Периоды (возраст, срок, стаж и т.п.) рекомендуется хранить в Axiom в виде даты начала действия периода и рассчитывать при формировании датасета
Количество видов собственности	Вид собственности	Предагрегат общего назначения	Расчет количества настраивается агрегирующей функцией в разрезе зерна
Количество месяцев с даты открытия первого кредита	Дата открытия первого кредитного договора	Готовая переменная	Вариант 1
Количество месяцев с даты открытия первого кредита	Дата открытия первого договора	Предагрегат общего назначения	Вариант 2, если потенциально могут быть полезны не только кредитные договора
Общее количество кредитных договоров за период	Номер договора	Предагрегат общего назначения	-
Отношение количества кредитных договоров на отчетную дату к количеству за период	Номер договора	Предагрегат общего назначения	-
Сумма транзакций внесения наличных по карте за перио	Сумма транзакции	Транзакционный предагрегат	Итоговые переменные для датасета рассчитываются при создании датасета. Пользователь указывает: агрегирующую функцию (обязательно), условия фильтрации по доменам,периоды расчета
Сумма транзакций внесения безнал по карте за период	Сумма транзакции	Транзакционный предагрегат	Итоговые переменные для датасета рассчитываются при создании датасета. Пользователь указывает: агрегирующую функцию (обязательно), условия фильтрации по доменам,периоды расчета
Сумма расходных транзакций за границей за период	Сумма транзакции	Транзакционный предагрегат	Итоговые переменные для датасета рассчитываются при создании датасета. Пользователь указывает: агрегирующую функцию (обязательно), условия фильтрации по доменам,периоды расчета
Дисперсия расходных транзакций за период	Дисперсия расходных транзакций за период 1, ..., Дисперсия расходных транзакций за период N	Готовая переменная	Переменные, для расчета которых требуются детальные данные, рекомендуется рассчитывать перед зарузкой в Axiom и хранить в качестве готовой переменной

Гранулярность в транзакционных предагрегатах

Гранулярность - глубина, за которую агрегируются данные в предагрегате перед загрузкой в Axiom. Гранулярность указывается целым числом периодов; доступные периоды: час, день неделя, месяц, год.

Выбор гранулярности остается за пользователем, проектирующим входящие объекты для Axiom. Руководствоваться стоит минимально необходимым периодом, за который будут формироваться переменные для датасетов. Если модели обычно используют транзакционные фичи за периоды в 1, 3, 6 и 12 месяцев, рекомендуется рассчитывать данные в предагрегате с гранулярностью 1 месяц.

Избыточно мелкая гранулярность увеличивает объем хранимых данных, но дает больше вариативности для агрегирования при создании итоговых фичей в датасет.

Результирующие объекты

Перед загрузкой данных в Axiom необходимо продумать разделение по таблицам.

Для примера датасета выше рекомендуемый набор входных витрин для Axiom приведен в таблице ниже. В каждой витрине должны присутствовать обязательные поля: ключ(-и) сущности и (опционально) даты актуальности.

Таблица	Тип таблицы	Сущность	Переменная	Описание переменной
CUSTOMER_INFO	Готовые переменные	Клиент	SEX	Пол
CUSTOMER_INFO	Готовые переменные	Клиент	BIRTH_DT	Дата рождения
POPERTY	Предагрегат общего назначения	Клиент	PROPERTY_NM	Вид собственности
LOAN	Готовые переменные	Клиент	FIRST_OPEN_DT	Дата открытия первого кредитного договора
AGREEMENT	Предагрегат общего назначения	Клиент	AGREEMENT_NUM	Номер договора
AGREEMENT	Предагрегат общего назначения	Клиент	AGREEMENT_TYPE	Тип договора
TRANSACTION	Транзакционный предагрегат	Клиент	TRN_AMT	Сумма транзакции
TRANSACTION	Транзакционный предагрегат	Клиент	TRN_DIRECTION	Направление транзакции (in/out)
TRANSACTION	Транзакционный предагрегат	Клиент	TRN_COUNTRY	Страна проведения транзакции
TRANSACTION_CALC	Готовые переменные	Клиент	VARIANCE_1	Дисперсия расходных транзакций за период 1
TRANSACTION_CALC	Готовые переменные	Клиент	...	...
TRANSACTION_CALC	Готовые переменные	Клиент	VARIANCE_N	Дисперсия расходных транзакций за период N

5. Before start

Установка:

```
# из репозитория
pip install FSClient --index-url https://__token__:<token secret>@git.angara.cloud/api/v4/projects/535/packages/pypi/simple

# .whl/tar.gz (должен находиться в той же папке, что и ноутбук .ipynb)
pip install FSClient.whl
```

Подключение к приложению из SDK необходимо выполнять в начале каждой сессии перед импортом каталогов:

```
from FSClient import _connect

_connect(
    url='https://<%morphism_url>/api',
    username='some_user',
    password='some_password'
)
```

5.1 Подготовка к регистрации данных

Проверить, что выполнена инициализирующая загрузка справочников типов данных.

Для работы с сущностями, фичами и другими объектами в приложении используется система каталогов. Начало работы с каталогом сущностей:

```
from FSClient.catalogue import entity
```

Для просмотра каталога сущностей необходимо вызвать каталог, который вернет каталог в формате таблицы:

```
entity  # <-- просмотр каталога
```

Создание новых сущностей осуществляется через вызов функции `create` из соответствующего каталога.

Начнем с данных о клиенте. В первую очередь необходимо подготовить мета-сущность клиента в приложении. Название сущности и ключа **не обязательно** должно быть таким же, как в таблице с данными, т.к. при регистрации данных происходит сопоставление ключей и полей в явном виде.
Выполняем создание новой сущности `CUSTOMER`, у которой будет один ключ - `CUSTOMER_RK`:

```
customer = entity.create()  # <-- получение объекта новой сущности

customer.config = {
    'name': 'CUSTOMER',  # <-- имя сущности
    'description': 'Клиент',  # <-- пользовательское описание сущности
    'entity_keys': [  # <-- список ключей сущности
        {
            'name': 'CUSTOMER_RK',  # <-- название ключа
            'description': 'Суррогатный ключ клиента',  # <-- пользовательское описание ключа
            'data_type': dttp.INT  # <-- тип данных ключа
        }
    ]
}

customer.save()  # <-- регистрация сущности
```

Создание фичи осуществляется через каталог фичей, выполняем импорт каталога фичей:

```
from FSClient.catalogue import feature
```

Для определения признаков из таблицы необходимо зарегистрировать фичи, с которыми мы их свяжем в приложении:

```
new_feat = feature.create()
new_feat.config = {
        'name': 'GENDER',  # <-- Название фичи
        'description': 'Пол'  # <-- Пользовательское описание фичи
    }
new_feat.save()  # <-- регистрация фичи
```

Название фичи так же **не обязательно** должно быть таким же, как поле в таблице с данными.

```
new_feat = feature.create()
new_feat.config = {
        'name': 'AGE',
        'description': 'Возраст клиента'
    }
new_feat.save()
new_feat = feature.create()
new_feat.config = {
        'name': 'CITY',
        'description': 'Наименование города'
    }
new_feat.save()
```

Теперь можно приступить к регистрации таблицы CUSTOMER_INFO. Процесс регистрации данных является описанием таблицы. Это позволяет выполнять операции с данными без участия пользователя в дальнейшем.

Первый шаг в регистрации загрузчика - выбор источника данных (БД) и адреса таблицы (схема + название). Название и описание загрузчика являются мета-полями и предназначены для идентификации источника данных в каталоге приложения.

В нашем случае источником данных является таблица в БД. Подробнее о других источниках будет далее.

```
from FSClient.catalogue import (
    entity as ent,  # <-- Для удобства использования каталог в коде используем сокращения
    feature as feat,
    loader as load
)

new_loader = load.create(
    source='table'  # <-- Тип источника данных
)

new_loader.config = {
        # Название источника данных в каталоге
        'loader_name': 'CUSTOMER_INFO_HIST',
        # Пользовательское описание источника данных
        'description': 'Информация по клиенту из карточки клиента АБС',
        # Схема таблицы
        'src_schema_name': 'stage',
        # Название таблицы
        'src_table_name': 'customer_hist',
        # Сущность таблицы
        'entity': ent.CUSTOMER
}
```

После выбора источника данных необходимо определить соответствие между полями таблицы и логическими сущностями приложения:

- В аргументе `entity` указываем, какие поля таблицы источника соответствуют ключам выбранной сущности
- В аргументе `features` связываем фичи из каталога приложения с полями таблицы

```
new_loader.mapping = {
    'entity': [
        {
            # Выбор ключа сущности из каталога
            'entity_key': ent.CUSTOMER.CUSTOMER_RK,
            # Название поля из таблицы, соответствующее выбранному ключу
            'stg_column_name': 'customer_rk'
        }
    ],
    'features': [
        {
            # Фича из каталога
            'feature': feat.CUSTOMER_AGE,
            # Название поля из таблицы
            'stg_column_name': 'customer_age',
            'feature_version_config': {
                # Название версии фичи в каталоге
                'name': 'V1_MAIN',
                # Пользовательское описание версии фичи
                'description': 'Карточка клиента из АБС: Возраст клиента',
                # Тип данных поля из таблицы
                'data_type': dttp.INT
            }
        },
        {
            'stg_column_name': 'gender',
            'feature': feat.GENDER,
            'feature_version_config': {
                'name': 'V1_MAIN',
                'description': 'Карточка клиента из АБС: Пол',
```

```
                'data_type': dttp.STRING
            }
        },
        {
            'stg_column_name': 'city',
            'feature': feat.CITY,
            'feature_version_config': {
                'name': 'V1_MAIN',
                'description': 'Карточка клиента из АБС: Наименование города',
                'data_type': dttp.STRING
            }
        }
    ]
}
```

Процесс связки мета-сущностей из приложения с полями в таблице с данными, проделанный на предыдущем шаге, называется **маппинг**.

Когда выбран источник и произведен маппинг, следует проверить введенную информацию и после зарегистрировать:

```
new_loader.save_x_execute()  # <-- регистрация (данных) загрузчика
```

5.2 Первый датасет

Датасет - таблица, с использованием данных, зарегистрированных в приложении.

После успешной регистрации загрузчика мы можем создать датасет, используя данные из источника данных, указанного в нашем загрузчике.

Для создания датасета нужен каталог датасетов:

```
from FSClient.catalogue import dataset as ds
```

Инициализируем новый датасет:

```
my_dataset = ds.create(
    # Сущность будущего датасета
    entity=ent.CUSTOMER,
    # Название датасета в каталоге
    name='my_first_dataset',
    # Пользовательское описание датасета
    description='Срез клиентов старше 40 лет'
)
```

Добавим туда данные о клиенте:

```
my_dataset.add_feature(
    feat.GENDER.V1_MAIN,
    feat.CITY.V1_MAIN,
    # через alias можно явно определить название поля в датасете
    feat.AGE.V1_MAIN.alias('CUSTOMER_AGE')
)
```

При добавлении данных в датасет мы выбираем именно **версию фичи** (в нашем случае версии фич мы назвали `V1_MAIN`), т.к. именно она является физическим источником данных бизнес-логики фичи.

Чтобы не копировать источник данных, только в виде датасета, отфильтруем клиентов по возрасту.

Для начала нам следует получить локальный каталог полей датасета:

```
ds_feat = my_dataset.get_features()  # <-- получение локального каталога полей датасета
```

Зададим условие фильтрации данных в нашем датасете "Клиенты, возраст которых старше 40 лет":

```
my_dataset.filter = ds_feat.CUSTOMER_AGE > 40
```

Для регистрации датасета и запуска расчета выполняем соответствующую функцию:

```
my_dataset.save_x_execute()
```

Обновим каталог, чтобы посмотреть информацию о датасете и узнать статус расчета:

```
from FSClient.catalogue import update

update()  # <-- Обновление каталога
ds.my_first_dataset  # <-- просмотр информации о датасете
```

5.3 Регистрация предагрегатов

При регистрации источника данных транзакционных предагрегатов в загрузчике появляются новые характеристики данных, такие как: тип гранулярности, значение гранулярности.

Гранулярность данных - минимальный интервал, в разрезе которого собраны данные в строке таблицы.

Таблица `CUST_CARD_TRANS_WEEK` содержит данные о транзакциях клиента, разбитые по неделям, значит гранулярность данных "1 неделя".

```
new_loader = load.create(source = 'table')

new_loader.config = {
    'loader_name': 'CUST_CARD_TRANS_WEEK',
    'description': 'Транзакции по картам (за неделю) по данным процессинга',
    'src_schema_name': 'stage',
    'src_table_name': 'cust_card_trans_week',
    'preagg_flg': True,  # <-- флаг данных-предагрегатов
    'granularity_type': 'week',  # <-- тип гранулярности
    'granularity': 1,  # значение гранулярности
    'entity': ent.CUSTOMER
}

new_loader.mapping = {
    'from_dttm': 'from_dttm',
    'entity': [
        {
            'entity_key': ent.CUSTOMER.CUSTOMER_RK,
            'stg_column_name': 'customer_rk'
        }
    ],
    'features': [
        {
            'stg_column_name': 'mcc_categ',
            'feature': feat.MCC_CATEG,
            'feature_version_config': {
                'name': 'CT_W_V1',
                'kind': 'feature',
                'description': 'Транзакции по картам (за неделю): Категория MCC',
                'data_type': dttp.STRING
            }
        },
        {
            'stg_column_name': 'channel',
            'feature': feat.CHANNEL,
            'feature_version_config': {
                'name': 'CT_W_V1',
                'kind': 'feature',
                'description': 'Транзакции по картам (за неделю): Канал совершения транзакции',
                'data_type': dttp.STRING
            }
        },
        {
            'stg_column_name': 'trn_amt',
            'feature': feat.TRN_AMT,
            'feature_version_config': {
                'name': 'CT_W_V1',
                'kind': 'feature',
                'description': 'Транзакции по картам (за неделю): Сумма транзакций',
                'data_type': dttp.AMOUNT
            }
        },
        {
            'stg_column_name': 'trn_cnt',
            'feature': feat.TRN_CNT,
            'feature_version_config': {
                'name': 'CT_W_V1',
                'kind': 'feature',
                'description': 'Транзакции по картам (за неделю): Количество транзакций',
                'data_type': dttp.INT
            }
        },
        {
            'stg_column_name': 'last_trn_dt',
            'feature': feat.LAST_TRN_DT,
            'feature_version_config': {
                'name': 'CT_W_V1',
                'kind': 'feature',
```

```
                'description': 'Транзакции по картам (за неделю): Дата совершения последней транзакции',
                'data_type': dttp.DATETIME
            }
        }
    ]
}
```

Нам известно, что данные в таблице источнике обновляются раз в месяц, поэтому выставим для загрузчика расписание с обновлением в конце каждого месяца. Таким образом приложение будет хранить актуальные данные и накапливать историю с момента регистрации данных.

```
new_loader.schedule = {  # <-- задаем расписание обновления данных
    'type': 'monthly_last_day',  # <-- тип расписания "последний день каждого месяца"
    'run_time': '23:59:59'  # <-- время запуска обновления данных
}
```

Регистрация загрузчика:

```
new_loader.save_x_execute()
```

5.4 Датасет с агрегатами

Первый источник данных, который мы регистрировали, представлял из себя факты о клиенте, то есть данные в "готовом виде".

Загрузчик CUST_CARD_TRANS_WEEK работает с гранулярными по неделям данными по транзакциям. Для анализа таких данных необходимо выполнять агрегирование, т.к. в "сыром" виде они слишком объемны и нечитаемы.

Инициализируем новый датасет:

```
my_dataset = ds.create(
    entity=ent.CUSTOMER,  # <-- Сущность датасета
    name='AGG_TRANS_CUSTOMER_DATA',
    description='Данные по транзакциям клиентов в разбивке по количеству транзакций'
)
```

! Выбор сущности датасета открывает доступ ко всем данным, зарегистрированным под этой сущностью. В будущем будет показано, как можно использовать в датасете данные, зарегистрированные под другой сущностью.

Добавим данные о клиенте:

```
my_dataset.add_feature(
    feat.GENDER.V1_MAIN.alias('CUSTOMER_GENDER'),
    feat.CITY.V1_MAIN.alias('CUSTOMER_CITY'),
    feat.AGE.V1_MAIN.alias('CUSTOMER_AGE')
)
```

Выполним конфигурацию агрегатов в новом датасете:

```
from FSClient.catalogue import function as func  # <-- каталог функций

my_dataset.add_feature(
    features=[
        feat.TRN_AMT.CT_W_V1.alias('TRN_AMT')  # <-- список версий фичей на агрегирование
    ],
    agg=[  # список агрегирующих функций
        func.sum(),
        func.max(),
        func.min()
    ],
    domain=[
        # условие на количество транзакций меньше 10
        (feat.TRN_CNT.CT_W_V1 <= 10).set(alias='L10'),
        # условие на количество транзакций больше 10
        (feat.TRN_CNT.CT_W_V1 > 10).set(alias='M10')
    ]
)
```

Результат агрегации версии фичи `TRN_AMT.CT_W_V1` по заданным выше параметрам: 3 агрегирующих функции и два противоположных условия по одному и тому же параметру. Итого 3 x 2 = 6 столбцов с данными:

- `SUM_TRN_AMT_L10` - Сумма транзакций клиента за недели, в которых меньше 10 транзакций;
- `MAX_TRN_AMT_L10` - Максимальные траты за неделю среди недель, у которых меньше 10 транзакций;
- `MIN_TRN_AMT_L10` - Минимальные траты за неделю среди недель, у которых меньше 10 транзакций;
- `SUM_TRN_AMT_M10` - Сумма транзакций клиента за недели, в которых больше 10 транзакций;
- `MAX_TRN_AMT_M10` - Максимальные траты за неделю среди недель, у которых больше 10 транзакций;
- `MIN_TRN_AMT_M10` - Минимальные траты за неделю среди недель, у которых больше 10 транзакций;

Для регистрации датасета и запуска расчета выполняем соответствующую функцию:

```
my_dataset.save_x_execute()
```

Посмотрим на получившиеся данные. Для этого необходимо вызвать функцию `get_dataframe` и передать в нее пользовательское подключение к БД:

```
ds.AGG_TRANS_CUSTOMER_DATA.get_dataframe(conn)
```

<скриншот из жупитера с датафреймом>

5.5 Регистрация данных из файла

Помимо таблиц в БД поддерживаются и другие источники, например, данные в формате .csv/.xlsx.

Зарегистрируем данные о договорах клиента, где сущностью является уже не клиент, а клиент+договор. Для этого нам потребуется создать новую сущность с **двумя** ключами, где один уже существует (`CUSTOMER_RK`):

```
customer_agreement = ent.create()

customer_agreement.config = {
    'name': 'CUSTOMER_AGREEMENT',
    'description': 'Клиент-договор',
    'entity_keys': [
        {
            'name': 'AGREEMENT_RK',
            'description': 'Суррогатный ключ договора',
            'data_type': dttp.INT
        },
        ent._keys.CUSTOMER_RK  # <-- Переиспользуем существующий ключ
    ]
}

customer_agreement.save()  # <-- Регистрация новой сущности
```

Итого у нас в каталоге две сущности: `CUSTOMER` и `CUSTOMER_AGREEMENT`, имеющих один общий ключ. Для приложения все сущности, у которых есть общий ключ, соединяются автоматической связью, т.к. через общий ключ данные могут быть связаны. Связь между CUSTOMER и CUSTOMER_AGREEMENT 1:M, т.к. на одного клиента может приходиться несколько значений клиент+договор.

Связи сущностей могут быть использованы для добавления в датасет версий фичей, сущность которых отличается от сущности датасета. Для этого при их добавлении в датасет нужно указать связь между сущностью датасета и сущностью версии.

Процесс маппинга при регистрации данных из файла не меняется. В конфигурации заменяются несколько аргументов:

```
new_loader = load.create(source='file')

new_loader.select_file('CUST_AGREEM_FINAL.csv')  # <-- загружаем файл с данными
```

```
new_loader.config = {
    'loader_name': 'CUST_AGREEM_FINAL',
    'description': 'test',
    'delimiter': ';',  # <-- вместо схемы и таблицы указываем разделитель в файле
    'entity': ent.CUSTOMER_AGREEMENT
}
```

! Разделитель указывается в соответствии с тем, какой разделитель полей используется в файле.

```
new_loader.mapping = {
    'from_dttm': 'from_dttm',  # <-- Поле с датой начала периода актуальности данных
    'entity': [
        {
            'entity_key': ent.CUSTOMER_AGREEMENT.CUSTOMER_RK,
            'stg_column_name': 'customer_rk'
        },
        {
            'entity_key': ent.CUSTOMER_AGREEMENT.AGREEMENT_RK,
            'stg_column_name': 'agreement_rk'
        }
    ],
    'features': [
        {
            'stg_column_name': 'agreement_type',
            'feature': feat.AGREEMENT_TYPE,
            'feature_version_config': {
                'name': 'V1_FINAL',
                'kind': 'feature',
                'description': 'test',
                'data_type': dttp.STRING
            }
        },
        {
            'stg_column_name': 'contract_cost',
            'feature': feat.CONTRACT_COST,
            'feature_version_config': {
                'name': 'V1_FINAL',
                'kind': 'feature',
                'description': 'test',
                'data_type': dttp.BIG_INT
            }
        },
        {
            'stg_column_name': 'vip_flg',
            'feature': feat.VIP_FLG,
            'feature_version_config': {
                'name': 'V1_FINAL',
                'kind': 'feature',
                'description': 'test',
                'data_type': dttp.BIG_INT
            }
        }
   ]
}
```

! Период актуальности данных - определяется двумя датами (from_dttm, to_dttm) и является временным окном, в которое данные в конкретной строчке актуальны и могут быть использованы. Дата начала периода актуальности определяется либо полем в источнике данных, либо задается как значение в формате YYYY-MM-DD hh:mm:ss. Дата конца периода актуальности считается бесконечно далекой датой, если не указана.

Регистрация загрузчика:

```
new_loader.save_x_execute()
```

5.6 Датасет с фичами из другой сущности

```
my_dataset = ds.create(
    entity=ent.CUSTOMER_AGREEMENT,  # <-- Сущность датасета
    name='CUSTOMER_and_AGREMENT_DATA',
    description='Данные о клиенте и его договорах'
)
```

<Объяснить почему в качестве основной сущности выбрали CUSTOMER_AGREEMENT?>

Добавим данные о договорах клиента:

```
my_dataset.add_feature(
    feat.AGREEMENT_TYPE.V1_FINAL.alias('AGR_TYPE'),
    feat.CONTRACT_COST.V1_FINAL.alias('AGR_COST')
)
```

Добавим данные о клиенте, используя автоматическую связь сущностей (сущность датасета - CUSTOMER_AGREEMENT, сущность данных о клиенте - CUSTOMER)

```
from FSClient.catalogue import entity_link as el  # <-- Импортируем каталог связей сущностей

my_dataset.add_feature(
    features=[
        feat.GENDER.V1_MAIN.alias('CUSTOMER_GENDER'),
        feat.CITY.V1_MAIN.alias('CUSTOMER_CITY'),
        feat.AGE.V1_MAIN.alias('CUSTOMER_AGE')
    ],
    link=[
        # Указываем связь, с помощью которой версии фичей будут добавлены в датасет
        el.auto_link_CUSTOMER_x_CUSTOMER_AGREEMENT_1_M
    ]
)
```

Выполняем регистрацию датасета и запуск расчета:

```
my_dataset.save_x_execute()
```

5.7 Ручные связи сущностей

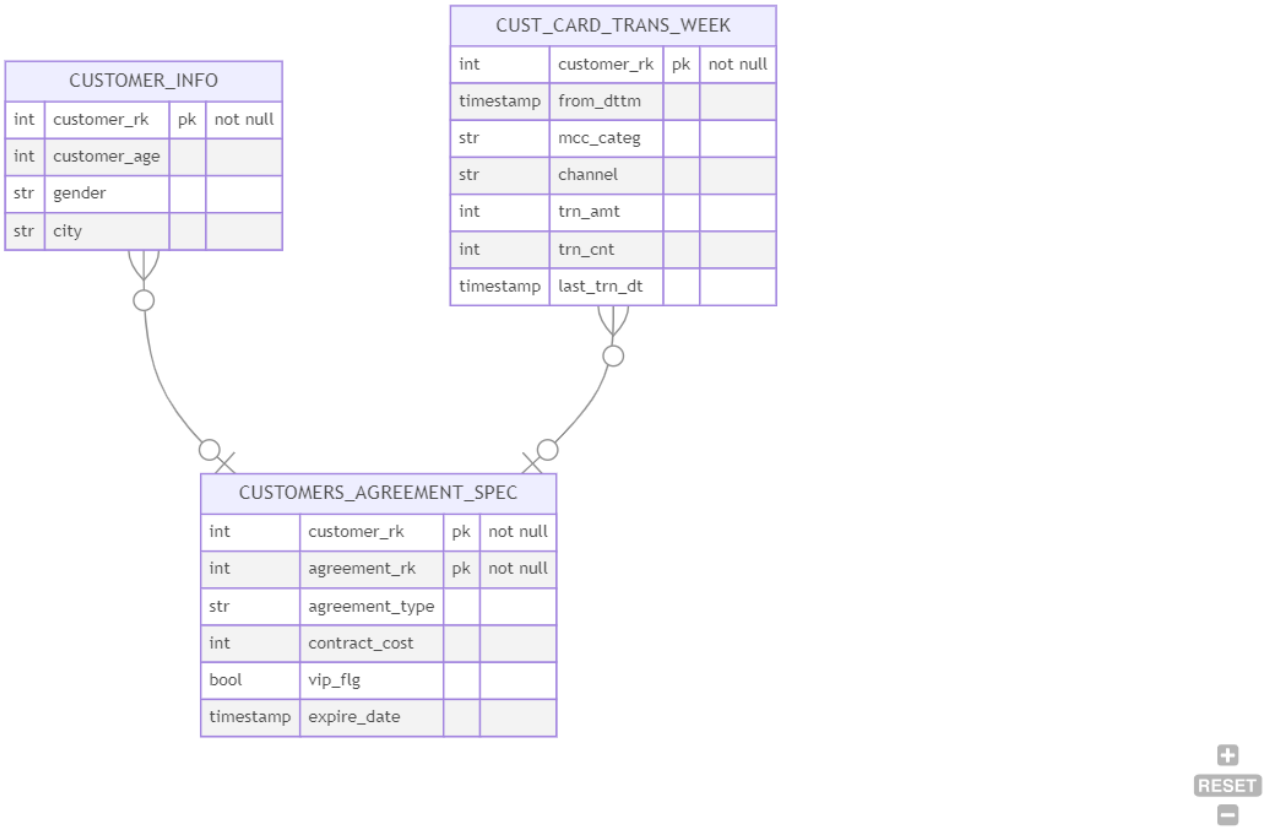
Помимо автоматических связей, которые образуются между сущностями с общими ключами, существуют ручные связи, которые представляют из себя таблицу с данными, в которой присутствуют оба набора ключей связываемых сущностей. Таким образом с помощью ручной связи можно соединить любые сущности.

Ручная связь зерен это источник данных о связи сущностей, поэтому для регистрации в приложении используем загрузчик.

Зарегистрируем связь M:N (многие ко многим) между сущностями CUSTOMER и CUSTOMER_AGREEMENT, т.к. известно, что в таблице `customer_x_customer_agreement_inhouse` содержатся договора между клиентами фирмы, что значит на один и тот же договор приходится больше 1 клиента и на каждого клиента не меньше 1 договора.

```
new_entity_link = load.create(source='table', target='entity_link')

new_entity_link.config = {
    'loader_name': 'CUSTOMER_X_CUSTOMER_AGREEMENT_M_N',
    'description': 'Связи клиентов с договорами внутри фирмы',
    'src_schema_name': 'stage',
    'src_table_name': 'customer_x_customer_agreement_inhouse',
    'kind': 'M:N',
    'parent_entity': ent.CUSTOMER,
    'child_entity': ent.CUSTOMER_AGREEMENT
}

new_entity_link.mapping = {
    'from_dttm': 'from_dttm',
    'parent': [  # <-- задаем маппинг для первой сущности CUSTOMER
        {
            'entity_key': ent.CUSTOMER.CUSTOMER_RK,
            'stg_column_name': 'customer_rk'
        }
    ],
    'child': [  # <-- задаем маппинг для второй сущности CUSTOMER_AGREEMENT
        {
            'entity_key': ent.CUSTOMER_AGREEMENT.CUSTOMER_RK,
            'stg_column_name': 'customer_rk_ca'
        },
        {
            'entity_key': ent.CUSTOMER_AGREEMENT.AGREEMENT_RK,
            'stg_column_name': 'agreement_rk_ca'
        }
    ]
}

new_entity_link.save_x_execute()
```

5.8 Исходные данные



- `CUSTOMER_INFO` - данные по конкретному клиенту - пол, возраст, город.
- `CUST_CARD_TRANS_WEEK` - данные о количестве и сумме транзакций клиента за неделю в разрезе товарных категорий и каналов распространения.
- `CUSTOMERS_AGREEMENT_SPEC` - данные о договорах между клиентом и фирмой: тип договора, стоимость, признак "Вип", дата окончания договора.

6. Definitions

Термин	Описание

- **API**: Application Programming Interface
- **Morphism**: Система класса Feature Store, осуществляющая загрузку, инвентаризацию и стандартизированное предоставление данных для использования в ML разработке
- **CSV**: Comma-separated values
- **Data Quality** : Подсистема контроля качества данных
- **DE**: Data Engineer
- **DS**: Data Scientist
- **DWH** / **КХД**: Data Warehouse (корпоративное хранилище данных)
- **ETL**: Extract-Transform-Load - процесс извлечения данных из источника, их трансформации и последующей загрузки в приёмник
- **FS**: Feature Store
- **GUI**: Graphical User Interface (графический интерфейс)
- **ML**: Machine Learning (машинное обучение)
- **Pandas Dataframe**: Формат представления табличных данных средствами библиотеки Pandas
- **PK**: Primary Key (первичный ключ)
- **S3**: Amazon S3 (файловое хранилище)
- **SDK**: Software Development Kit
- **SQL**: Structured Query Language
- **Актуальный срез**: Срез данных, актуальных на момент запроса
- **Алиас** / **Alias**: Название компонента конфигурации датасета (переменная / агрегирующая функция / условие фильтрации), используемое при генерации физического названия поля датасета
- **БД**: База данных
- **Вектор**: Датасет
- **Дата актуальности**: Дата, на которую актуальны данные
- **Датасет**: Материализованный набор данных (таблица в реляционной БД) для использования в модели машинного обучения
- **Каталог**: Реестр зарегистрированных сущностей - каталог переменных, каталог датасетов и т.д.
- **Ключ сущности**: Поле для связи данных внутри приложения - CUSTOMER_RK (ключ клиента), AGREEMENT_RK (ключ договора) и т.д.
- **Маппинг**: Соотнесение (например, полей источника на ключи сущности)
- **Переменная** / **Фича**: Логическая сущность, объединяющая различные алгоритмы её расчёта (версии переменных) - пол клиента, дата рождения, признак VIP и т.д.
- **Пользовательская формула**: Макрос, позволяющий применять дополнительную логику для расчёта полей датасета. Задаётся в виде кода в стандартизированном формате
- **Сегмент**: Список ключей, по которому производится расчёт датасета
- **Сущность**: Сущность, в разрезе которой рассчитаны данные - клиент, договор и т.д. Объединяет под собой один или несколько ключей сущности
- **Таймфрейм** / **Timeframe**: Временное окно для агрегации данных
- **Тонкий клиент** / **PyClient**: Подключаемая в Jupyter Notebook библиотека для работы с функциями приложения через API