Конспект по теме "Что такое дата-пайплайны и зачем они нужны"

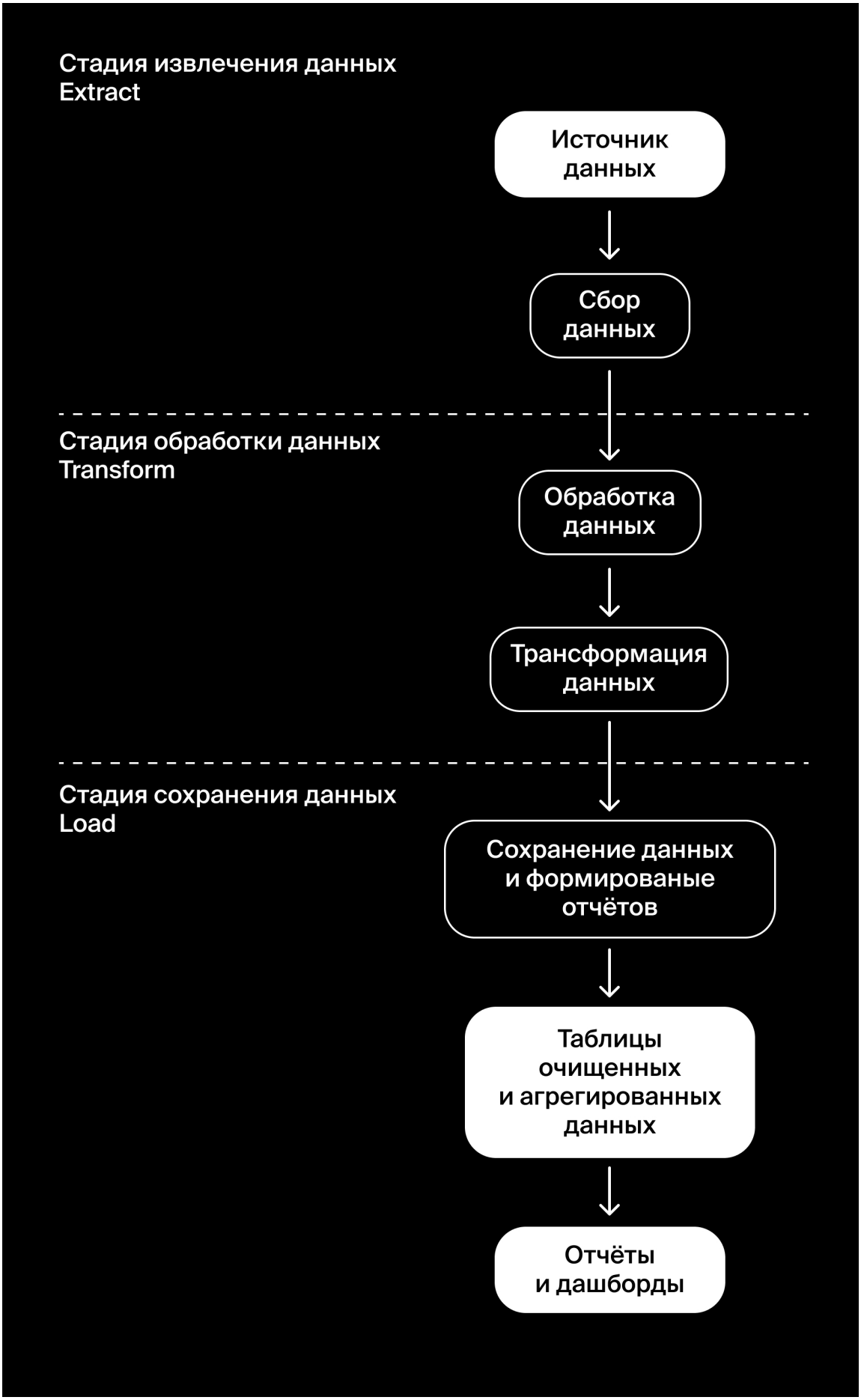
Дата-пайплайны для автоматизации дашбордов

Основа автоматизации — **дата-пайплайны** (конвейеры данных). Дата-пайплайн — специальная программа, которая вызывается по расписанию. Она собирает, объединяет, трансформирует и сохраняет данные автоматически. Пайплайны позволяют:

- парсить данные в Интернете и сохранять результат в базу данных;
- собирать информацию о визитах и покупках пользователей из корпоративных систем и формировать отчёты для когортного анализа;
- отслеживать аномалии в поведении пользователей;
- анализировать A/B-тесты;
- заспамить коллег письмами о росте продаж и посещений.

И всё это — без малейшего вмешательства человека!

Элементы пайплайна описывает аббревиатура **ETL** (*extract, transform, load* — извлеки, трансформируй, сохрани):


Стадия извлечения данных
Extract
Источник данных
Сбор данных
Стадия обработки данных
Transform
Обработка данных
Трансформация данных
Стадия сохранения данных
Load
Сохранение данных и формированые отчётов
Таблицы очищенных и агрегированных данных
Отчёты и дашборды

На стадии извлечения данных (*Extract*) пайплайн собирает информацию из разных источников.

На этапе обработки данных (*Transform*) информация приводится к единому стандарту.На этом же этапе данные категоризируют. Затем происходит трансформация данных: они приводятся к формату, удобному для построения отчётов или хранения агрегированных сведений.

На последнем шаге (*Load*) пайплайн сохраняет агрегированные данные в таблицы БД и, если нужно, формирует отчёты и рассылки.

Дата-пайплайн проектируют так, чтобы все его этапы можно было перезапускать снова, всякий раз получая стабильный, повторяющийся результат.

Агрегация данных и создание таблиц в БД

На практике исходные данные для анализа хранят в базах. Чаще всего, это **необработанные логи** — журналы событий.

Аналитику сырые данные нужны редко: для отчётов и выводов хватает агрегированных. **Агрегирование** или **агрегация** — процесс группировки и уменьшения размера данных.

Чтение большого числа записей из БД может занимать часы! Лучше завести новую таблицу, в которую сохранять сгруппированные данные.

Чтобы сохранить агрегированные данные, создают таблицу командой **CREATE TABLE.** Вот её синтаксис:

```
CREATE TABLE имя_таблицы (первичный_ключ тип_данных,
                          имя_столбца1 тип_данных,
                          имя_столбца2 тип_данных,
                          …);
```

Некоторые из типов данных в PostgreSQL :

- **INT** — целое число;
- **REAL** — число с плавающей точкой;
- **VARCHAR(n)** — строка, где n — её максимальная длина;
- **TIMESTAMP** — дата и время;

- **SERIAL** — специальный тип данных для значений первичных ключей. Напомним, первичным ключом называют уникальный номер записи в таблице. Первичный ключ обозначают выражением:

```
CREATE TABLE имя_таблицы (название поля с первичным ключом serial PRIMARY KEY);
```

При добавлении в таблицу новой записи, СУБД сама проставляет в поле типа SERIAL номер строки на единицу больше, чем номер предыдущей вставленной строки.

Вертикальные и горизонтальные таблицы

При проектировании таблицы агрегированных данных важно помнить, что она должна быть как можно более вертикальной. Добиться этого можно, если следовать правилу: каждому признаку отводить отдельный столбец. Такой метод размещения данных намного лучше автоматизируется и масштабируется.

С «горизонтальной» таблицей может возникнуть проблема: при добавлении каждой новой колонки в пайплайне, а также в зависящих от него дашбордах и отчётах, придётся заново определять тип столбца и правила обработки пропусков. Эту задачу, конечно, можно решить в коде. Однако гораздо проще строить вертикальный вариант таблицы — в нём добавление новых данных не приведёт к переопределению их структуры.

При создании таблицы что-то может пойти не так. Тогда нужно удалить таблицу командой **DROP TABLE**:

```
DROP TABLE имя_таблицы;
```

Когда неправильные таблицы удалили, а нужную оставили, раздавать права доступа к ней командой **GRANT** :

```
GRANT ALL PRIVILEGES ON TABLE table_name TO anthony;
-- здесь anthony - имя пользователя, которому нужен доступ к таблице
```

- **TO** определяет, кому выдают права;
- **ON TABLE** указывает, на какую таблицу выдают права;
- **ALL PRIVILEGES** означает, что пользователь получает все возможные права на таблицу: может читать, записывать и удалять данные.

Помимо прав на таблицу, нужно раздать права для работы с первичными ключами. Когда задаём поле с первичными ключами типа SERIAL, СУБД автоматически создаёт объект **SEQUENCE**. В нём хранятся данные о том, как генерировать идентификаторы для первичного ключа. Такие объекты получают имена автоматически и имеют вид: `имяТаблицы_имяКлюча_seq`.

```
GRANT USAGE, SELECT ON SEQUENCE table_name_id_seq TO anthony;
```

`GRANT USAGE` означает, что пользователю разрешено добавлять в `table_name_id_seq` новые значения. `GRANT SELECT` позволяет читать данные из таблицы.

Создание скрипта пайплайна

Хотя коллеги могут помочь вам с выгрузкой, старайтесь всегда получать прямой доступ к базам данных:

- Это ускорит вашу работу: не придётся ждать, когда сделают выгрузку;
- Без прямого доступа к данным автоматизация невозможна. Кто-то заболел или ушёл в отпуск, выгрузка не произошла, и автоматизация сломалась;
- В БД всегда много интересного.

Библиотека **sqlalchemy** позволяет читать данные из таблиц БД в датафреймы *Pandas* и сохранять данные из *Pandas* в БД одной командой.

Подключимся к БД через sqlalchemy:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
```

```
# импортируем библиотеки
import pandas as pd
from sqlalchemy import create_engine

# Задаем параметры подключения к БД,
# их можно узнать у администратора БД.
db_config = {'user': 'my_user',         # имя пользователя
             'pwd': 'my_user_password', # пароль
             'host': 'localhost',       # адрес сервера
             'port': 5432,              # порт подключения
             'db': 'db_name'}             # название базы данных

# Формируем строку соединения с БД.
connection_string = 'postgresql://{}:{}@{}:{}/{}'.format(db_config['user'],
                                                         db_config['pwd'],
                                                         db_config['host'],
                                                         db_config['port'],
                                                         db_config['db'])
# Подключаемся к БД.
engine = create_engine(connection_string)

# Формируем sql-запрос.
query = ''' SELECT column1, column2, column3
            FROM table_name
        '''

# Выполняем запрос и сохраняем результат
# выполнения в DataFrame.
# Sqlalchemy автоматически установит названия колонок
# такими же, как у таблицы в БД. Нам останется только
# указать индексную колонку с помощью index_col.
data_raw = pd.io.sql.read_sql(query, con = engine, index_col = 'column1')

print(data_raw.head(5))
```

Чтобы подключиться к базе данных, нужно указать:

- СУБД;
- Расположение базы: IP-адрес сервера, где разместили БД, и порт, к которому нужно подключиться;
- Имя пользователя и пароль для подключения;
- Название интересующей БД. На одном сервере могут хранить несколько баз.

При добавлении строк в таблицу базы данных, sqlalchemy автоматически анализирует совпадение первичных ключей в таблице БД с индексами датафрейма. Поведение sqlalchemy в этом случае подчинено параметру

if_exists . Совпадающие строки могут быть перезаписаны `if_exists = 'replace'` или продублированы новыми копиями `if_exists = 'append'`.

```
df.to_sql(name = 'table_name', con = engine, if_exists = 'append', index = False))
```

Иногда в ходе работы пайплайна, нужно удалить старые строки из таблицы. Для этого используют SQL-команду **DELETE**:

```
DELETE FROM имя_таблицы WHERE условия_для_поиска_записей_которые_нужно_стереть;
```