Лекция 1.
Введение в анализ больших данных

Что такое большие данные?

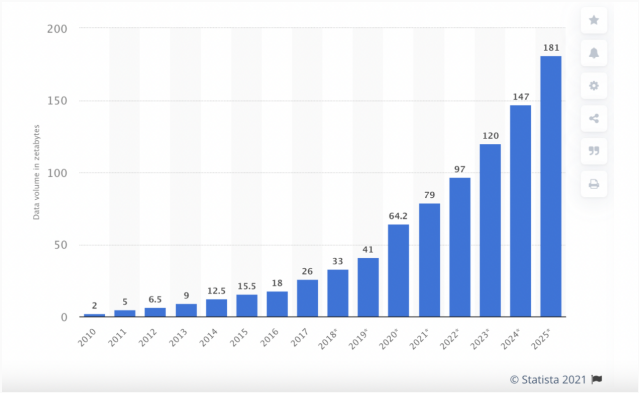
Что такое Big Data?

Big Data?

- 1Мбайт?
- 1Гбайт?
- 1Тбайт?
- 1Пбайт?
- 1Гбайт/мин?
- 1Тбайт/день?
- 1Пбайт/месяц?

Что такое Big Data?

Прогноз объема данных, созданных человечеством



https://www.statista.com/statistics/871513/worldwide-data-created/

Что такое Big Data?

Определяющие факторы больших данных:

- Volume (Объем)
- Velocity (Скорость)
- Variety (Разнообразие)
- Veracity (Правдивость)
- Value (Ценность)

Что такое Big Data?

- Архитектура
  - Data Warehouse
  - Data Lake
  - Lambda
  - ...
- Технологии
  - Хранение
  - Обработка данных
  - ...
- Инструменты
  - Администрирование
  - Business Intelligence
  - ...
- Наука
  - Математика
  - Машинное обучение
  - Глубокое обучение
  - ...

Зачем DS знать Big Data?

Что такое Big Data?

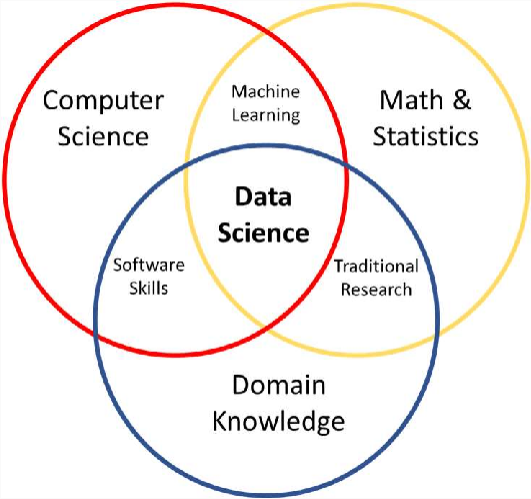
- Сбор и хранение данных
- Обработка данных
- Представление данных
- Анализ данных
- Предсказания
- Оптимизация процессов
- Монетизация

Что такое Big Data?

- Сбор и хранение данных
- Обработка данных
- Представление данных
- Анализ данных
- Предсказания
- Оптимизация процессов
- Монетизация

Data Engineer
+
Data Scientist
+
Data Analyst
+
ML Engineer
+
Data Quality Engineer

Командная работа

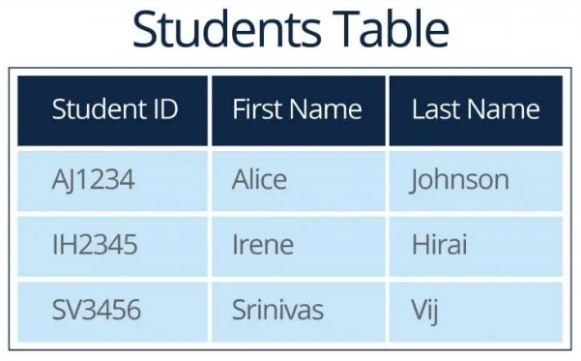


Проблемы в работе с данными для DS

1. Где взять данные?
2. Где обрабатывать столько данных?

Хранение данных. Реляционные БД

Реляционные базы данных (RDBMS)

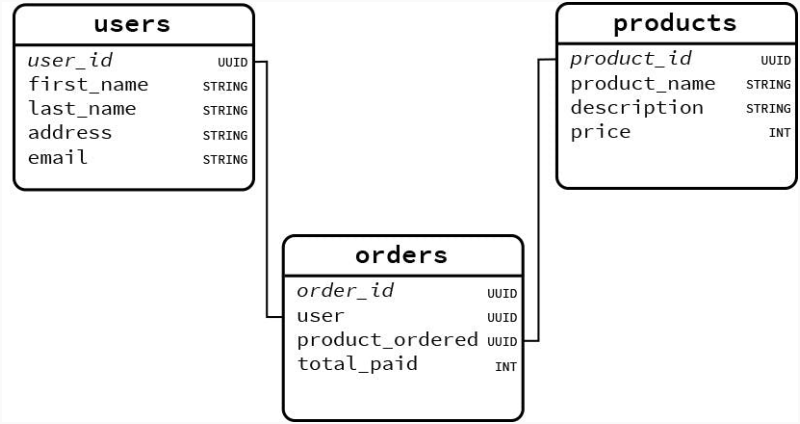


Students Table

Student ID	First Name	Last Name
AJ1234	Alice	Johnson
IH2345	Irene	Hirai
SV3456	Srinivas	Vij

https://www.smartsheet.com/database-relationships

Реляционные базы данных (RDBMS)



https://www.cockroachlabs.com/blog/database-schema-beginners-guide/

Реляционные базы данных (RDBMS)

- PostgreSQL
- MySQL
- Microsoft SQL Server

SQL — Structured Query Language

```sql
SELECT Name,
        ProductNumber,
        ListPrice AS Price
FROM Product
WHERE ProductLine = 'R'
        AND DaysToManufacture < 4
ORDER BY Name ASC;
```

Хранение данных. Не реляционные БД

Не реляционные базы данных (NoSQL)

- Key Value
- Document-oriented
- Column-oriented
- Graph

NoSQL — Key Value

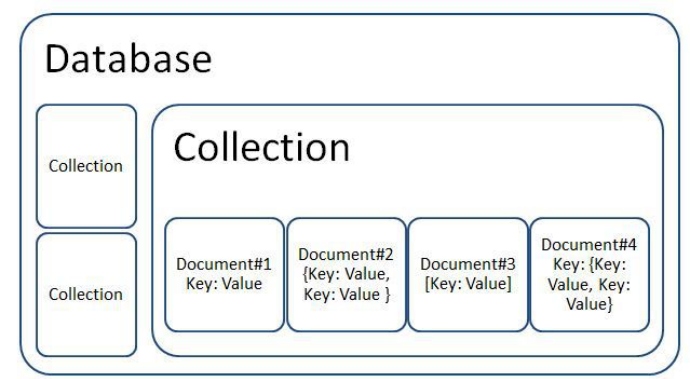
Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

https://en.wikipedia.org/wiki/Key–value_database

NoSQL — Key Value

- Memcached
- Redis
- Ignite

NoSQL — Document-oriented

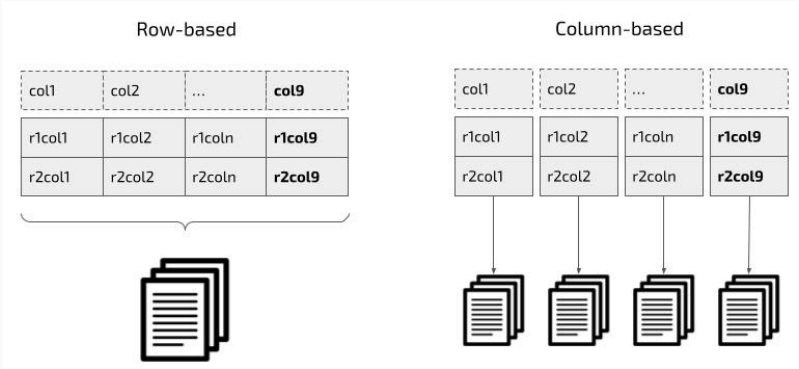


https://www.webcodegeeks.com/nosql/primer-open-source-nosql-databases/

NoSQL — Document-oriented

- MongoDB
- CouchDB
- ArangoDB
- Elasticsearch

NoSQL — Column-oriented

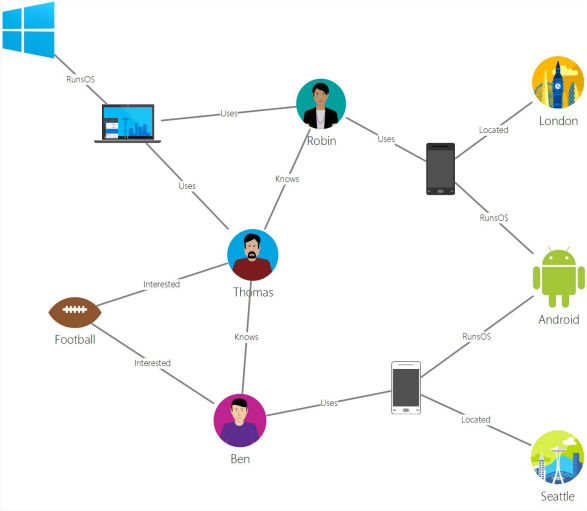


https://mariadb.com/resources/blog/new-in-mariadb-xpand-6-0-columnar-indexes-for-distributed-sql/

NoSQL — Column-oriented

- Cassandra
- HBase
- ClickHouse

NoSQL — Graph



https://analyticsindiamag.com/ai-origins-evolution/graph-database-how-graph-is-being-utilised-for-data-analytics/

NoSQL — Graph

- Neo4j
- OrientDB
- ArangoDB

SQL vs NoSQL

SQL vs NoSQL

- RDBMS => SQL
- NoSQL => ?

SQL vs NoSQL

https://www.astera.com/ru/knowledge-center/sql-vs-nosql/

Обработка данных

Обработка данных

- Извлечение
- Очищение
- Преобразование
- Выгрузка
- Анализ

Обработка данных

- Как?
  - Batch
  - Stream
- Чем?
  - Hadoop
  - Spark
  - ...

Что можно делать с данными?

- Извлекать
- Загружать
- Преобразовывать
- Визуализировать
- Анализировать
- Добавить

Извлекать/Загружать

- Базы данных
  - SQL
  - NoSQL
- Файловые системы
  - HDFS
- Брокеры сообщений
  - Apache Kafka

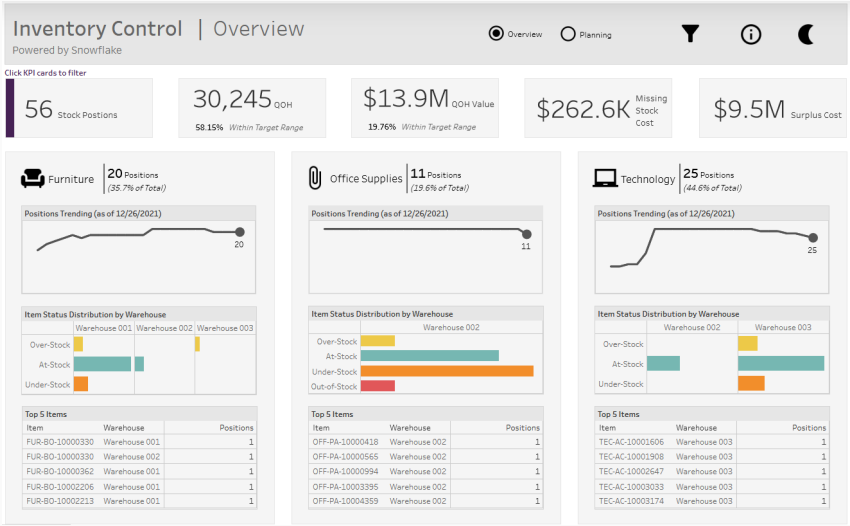
Преобразовывать

Промежуточное хранение

- Память
- База данных
- Файловая система

Визуализировать

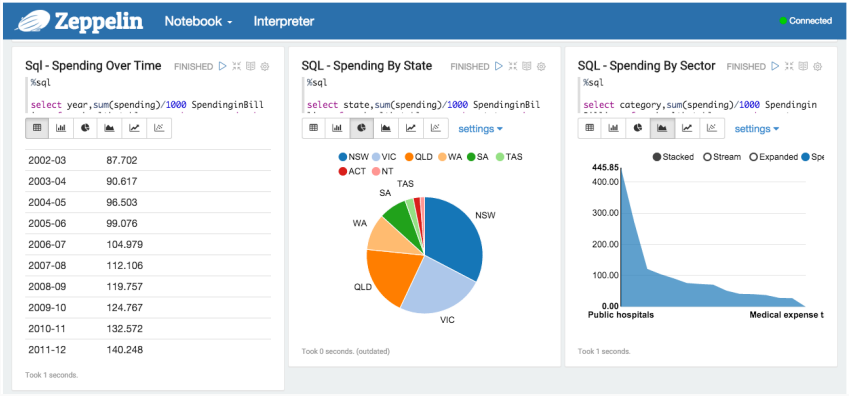
- Tableu
- Qlik
- Sisense



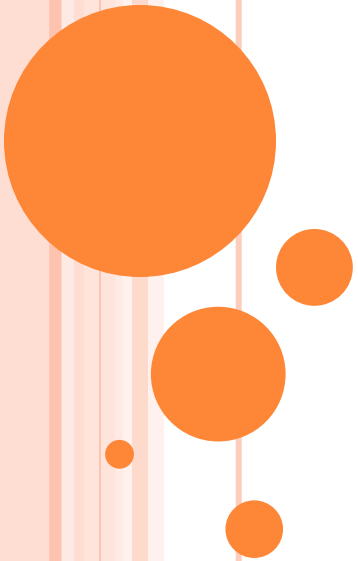
https://www.phdata.io/blog/tableau-inventory-dashboard-example/

Анализировать и добывать

- Jupyter
- CoLab
- Zeppelin



https://community.cloudera.com/t5/Community-Articles/Apache-Zeppelin-Walk-Through/ta-p/244643



Получение данных с помощью DQL

О языке DQL

- Включает только одну команду – `SELECT`
- `SELECT` используется для получения данных без их изменения
- Результат запроса – всегда таблица
- Пользователь указывает лишь вид таблицы, а не способ ее получения

Базовый запрос SELECT

Обязательные части запроса

- `SELECT [DISTINCT]` – перечисляет поля, которые должны быть возвращены в списке результатов. Для выбора всех полей можно указать *. `DISTINCT` указывается для исключения дубликатов

- `FROM` – перечисляет имена таблиц или представлений, из которых выбираются данные

Базовый запрос SELECT

Примеры:

```
SELECT *
FROM MOVIE

SELECT MOVIE_TITLE
FROM MOVIE

SELECT MOVIE_TITLE, MPAA_RATING_CODE
FROM MOVIE
```

Псевдонимы полей

- Присвоение псевдонима – переименование столбца, распространяющееся только на данный список результатов

- Записывается псевдоним через слово AS

```
SELECT
  MOVIE_GENRE_CODE AS GENRE,
  MPAA_RATING_CODE AS RATING,
  MOVIE_TITLE
FROM MOVIE
```

Сортировка результатов

- Осуществляется с помощью блока `ORDER BY` в запросе `SELECT`

- Может осуществляться по одному или нескольким полям таблицы, по возрастанию или убыванию
- `ASC` – по возрастанию (по умолчанию)
- `DESC` – по убыванию

- Можно сортировать по полям, которые не входят в результирующую таблицу

Ключевое слово WHERE

- Используется для отбора необходимых строк
- Каждая строка проверяется на соответствие указанному правилу
- Только удовлетворяющие правилу строки добавляются в результат запроса

Операторы сравнения

=	равно	
<	меньше	
<=	меньше или равно	!>
>	больше	
>=	больше или равно	!<
<>	не равно	!=

Пример:

```
SELECT MOVIE_TITLE
FROM MOVIE
WHERE MPAA_RATING_CODE = 'PG-13'
```

Оператор IS NULL

- Пустые значения ничему не равны, даже пустым значениям

- Проверки = NULL и = NOT NULL некорректны

Пример:

```
SELECT MOVIE_TITLE
FROM MOVIE
WHERE MPAA_RATING_CODE IS NOT NULL
```

Объединительные операторы

- AND (И) – истинно, если оба условия истинны
- OR (ИЛИ) – истинно, если хотя бы одно условие истинно

```
SELECT MOVIE_TITLE
FROM MOVIE
WHERE MPAA_RATING_CODE = 'PG-13' OR
      (RETAIL_PRICE_DVD >= 14.99 AND
       RETAIL_PRICE_DVD <= 19.99)
```

Оператор BETWEEN

- Используется для того, чтобы определить, попадает ли значение в заданный интервал

```
SELECT MOVIE_TITLE    FROM MOVIE
WHERE MPAA_RATING_CODE = 'PG-13' OR
  (RETAIL_PRICE_DVD >= 14.99 AND
   RETAIL_PRICE_DVD <= 19.99)

SELECT MOVIE_TITLE    FROM MOVIE
WHERE
MPAA_RATING_CODE = 'PG-13' OR
  RETAIL_PRICE_DVD BETWEEN 14.99 AND 19.99
```

Оператор LIKE

- Используется для сравнения строки с маской
- Специальные символы:
  - Подчеркивание (_) – любой символ
  - Процент (%) – любое количество символов

- Примеры:
  - _ама – соответствуют слова: мама, рама, …
  - _о_ – соответствуют слова: дол, пол, кол, …
  - %ук – соответствуют слова: лук, барсук, …
  - о%о – соответствуют слова: озеро, облако, одеяло, …

Примеры

```
SELECT *
FROM MOVIE
WHERE MOVIE_TITLE LIKE 'В %и'
```

```
SELECT *
FROM MOVIE
WHERE MOVIE_TITLE LIKE '% и %'
```

Оператор IN

- Используется для проверки того, входит ли значение в заданный список

```
SELECT *
FROM MOVIE
WHERE MPAA_RATING_CODE IN ('R','PG-13')
```

Оператор EXISTS

- Используется для проверки наличия строк во вложенном выборе

```
SELECT MOVIE_ID, MOVIE_TITLE
FROM MOVIE m
WHERE MOVIE_TITLE = 'Терминатор'
   AND EXISTS
   (SELECT MOVIE_ID
    FROM MOVIE_COPY c
    WHERE m.MOVIE_ID = c.MOVIE_ID);
```

Арифметические операторы

- +, –, *, /

```
SELECT PRICE_VHS + PRICE_DVD AS COST
FROM MOVIE
WHERE MOVIE_TITLE = 'Титаник';
```

```
   COST
-------
  35.91
```

Основные функции SQL

Символьные функции. Конкатенация строк

- Oracle – «||», MS SQL Server – «+»

```
SELECT 'Дорогой клиент '+ FNAME + ' ' +
LNAME AS CUSTOMER_SALUTATION
FROM PERSON;
```

Символьные функции. UPPER

- Переводит все символы строки в верхний регистр

```
...
WHERE UPPER(MOVIE_GENRE_CODE) = 'DRAMA'
...
```

Символьные функции. LOWER

- Переводит все символы строки в нижний регистр

```
...
WHERE UPPER(MOVIE_GENRE_CODE) = 'drama'
...
```

Символьные функции. SUBSTR

- Возвращает подстроку

- Oracle, DB2 – SUBSTR
- MS SQL Server, MySQL – SUBSTRING

- SUBSTRING(*ИМЯ_ПОЛЯ, нач_позиция, длина*)

```
...
WHERE SUBSTRING(FNAME, 1, 1) = 'Б'
...
```

Символьные функции. LENGTH

- Возвращает длину строки

- Oracle, DB2, MySQL – LENGTH
- MS SQL Server – LEN

```
SELECT MOVIE_TITLE, LEN(MOVIE_TITLE) AS
  LENGTH
WHERE LEN(MOVIE_TITLE) < 10;
```

Математические функции. ROUND

- Округляет число до заданного количества десятичных знаков.

- ROUND(*числ_выражение, кол_дес_знаков*)

```
SELECT ROUND((PRICE_VHS+PRICE_DVD)/2,2)
  AS AVG_COST
FROM MOVIE
WHERE MOVIE_TITLE = 'Титаник';
```

Другие математические функции

Function	Description
ABS	Absolute value of the given number
COS	Trigonometric cosine of the given angle (in radians)
EXP	Exponential value of the given number
POWER	Raise number to a power (both number and power are parameters)
SIN	Trigonometric sine of the given angle (in radians)
TAN	Trigonometric tangent of the given angle (in radians)

Функции преобразования. CAST

- Преобразует данные одного типа в другой
- CAST(*выражение* AS *тип_данных*)

```
SELECT '$' + CAST(PRICE_DVD AS
  VARCHAR(6)) AS PRICE
FROM MOVIE
WHERE MOVIE_TITLE = 'Титаник';

PRICE
------
$19.96
```

Агрегирующие функции и группировка строк

- ***Агрегирующие функции*** – это функции, объединяющие несколько строк в одну

- `AVG` – среднее значение столбца или выражения
- `COUNT` – количество значений в поле. Можно подсчитать количество уникальных значений с помощью `DISTINCT`
- `MAX` – максимальное значение в поле
- `MIN` – минимальное значение в поле
- `SUM` – сумма значений в поле

Агрегирующие функции и группировка строк

Примеры:

```
SELECT COUNT(*) AS NUM_MOVIES
FROM MOVIE

SELECT
  COUNT(DISTINCT(MOVIE_GENRE_CODE))
  AS NUM_GENRES
FROM MOVIE
```

Выражение GROUP BY

- Объединяет строки, отобранные запросом, в группы на основании значения одной или нескольких колонок

Пример:

```
SELECT MOVIE_GENRE_CODE,
  COUNT(*) AS COUNT
FROM MOVIE
GROUP BY MOVIE_GENRE_CODE
```

Операторы составных запросов

- UNION
- INTERSECT
- EXCEPT

Немного практики

- Найдите все фильмы в таблице `MOVIE` с рейтингом MPAA (поле `MPAA_RATING_CODE`), отличным от `R`
- Выведите список названий и цен на все фильмы с ценой DVD (`RETAIL_PRICE_DVD`) не меньше 19,99 и не больше 29,99, отсортированный по возрастанию цен
- Сколько человек имеют фамилию (таблица `PERSON`, поле `PERSON_FAMILY_NAME`), включающую прописную или строчную «а»?

Примеры

```
SELECT COUNT(*)
FROM PERSON
WHERE PERSON_FAMILY_NAME LIKE 'B%a'
```

```
SELECT COUNT(*)
FROM CUSTOMER_ACCOUNT
WHERE DATE_TERMINATED IS NOT NULL
```

Примеры

```
SELECT
  MOVIE_GENRE_CODE AS GENRE,
  MPAA_RATING_CODE AS RATING,
  MOVIE_TITLE
FROM MOVIE
WHERE
  MOVIE_GENRE_CODE IN ('ActAd', 'Drama')
  AND MPAA_RATING_CODE = 'PG-13'
ORDER BY
  MOVIE_GENRE_CODE, MPAA_RATING_CODE
```

Комбинация данных из нескольких таблиц

Соединения

- ***Соединение*** – операция реляционной базы данных, комбинирующая поля из одной или более таблиц в едином результате запроса
- В поле `FROM` перечисляется более одной таблицы или представления

```
SELECT MOVIE_ID AS ID,
       MOVIE_GENRE_DESC AS GENRE,
       MOVIE_TITLE
FROM MOVIE, MOVIE_GENRE
ORDER BY MOVIE_ID
```

Результат выполнения запроса

ID	GENRE	MOVIE_TITLE
1	Мультфильмы	Таинственная река
1	Боевики	Таинственная река
1	Детские фильмы	Таинственная река
1	Документальные фильмы	Таинственная река
1	Драма	Таинственная река
2	Мультфильмы	Гнев
2	Боевики	Гнев
2	Детские фильмы	Гнев
2	Документальные фильмы	Гнев
2	Драма	Гнев
3	Мультфильмы	Холодная гора
…	…	…

Результат выполнения запроса

- Полученный результат – ***декартово произведение*** – каждая строка одной таблицы соединена с каждой строкой другой таблицы

- В данном запросе СУБД не знает, как сопоставить строки

Эквисоединения

- ***Эквисоединение*** (или ***внутреннее соединение***) – это соединение, в котором одно или более полей одной таблицы (обычно внешний ключ) совмещаются с аналогичными полями другой таблицы (обычно первичный ключ) при условии равенства

- Способы:
  - условие в блоке `WHERE`
  - `JOIN`

Использование WHERE

- Похоже на исключение нежелательных строк в простом запросе SELECT

```
SELECT MOVIE_ID AS ID,
       MOVIE_GENRE_DESC AS GENRE,
       MOVIE_TITLE
FROM   MOVIE, MOVIE_GENRE
WHERE  MOVIE.MOVIE_GENRE_CODE =
       MOVIE_GENRE.MOVIE_GENRE_CODE
ORDER BY MOVIE_ID
```

Результат выполнения запроса

ID	GENRE	MOVIE_TITLE
1	Драма	Таинственная река
2	Боевики	Гнев
3	Драма	Холодная гора
…	…	…

Псевдонимы таблиц

- Назначаются в блоке `FROM` после имени таблицы

```
SELECT A.MOVIE_ID AS ID,
       B.MOVIE_GENRE_DESC AS GENRE,
       A.MOVIE_TITLE
FROM   MOVIE AS A, MOVIE_GENRE AS B
WHERE  A.MOVIE_GENRE_CODE =
       B.MOVIE_GENRE_CODE
ORDER BY A.MOVIE_ID
```

Использование JOIN

- JOIN оформляется в виде ссылки на таблицу после FROM
- JOIN объединяет список таблиц после FROM и условия соединения в единую конструкцию

- Синтаксис:

```
имя_таблицы [INNER] JOIN имя_таблицы
{ON условие |
  USING (имя_поля [, имя_поля])}
```

Использование JOIN

- ON – позволяет указать условие, подобно указанному после WHERE

- USING – перечисляет имена полей, которые должны использоваться для совпадающих строк
- USING работает только тогда, когда столбцы в обеих таблицах имеют одинаковые имена

- **USING не работает в MS SQL Server, работает в Oracle и MySQL**

Пример

```
SELECT MOVIE_ID AS ID,
       MOVIE_GENRE_DESC AS GENRE,
       MOVIE_TITLE
FROM MOVIE JOIN MOVIE_GENRE
ON MOVIE.MOVIE_GENRE_CODE =
  MOVIE_GENRE.MOVIE_GENRE_CODE
ORDER BY MOVIE_ID
```

Пример

```
SELECT MOVIE_ID AS ID,
       MOVIE_GENRE_DESC AS GENRE,
       MOVIE_TITLE
FROM MOVIE JOIN MOVIE_GENRE
USING (MOVIE_GENRE_CODE)
ORDER BY MOVIE_ID
```

Естественное соединение

- ***Естественное соединение*** основывается на всех полях двух таблиц, имена которых совпадают.

```
SELECT MOVIE_ID, MOVIE_GENRE_DESC,
       MOVIE_TITLE
FROM MOVIE NATURAL JOIN MOVIE_GENRE
ORDER BY MOVIE_ID
```

- **Не работает в MS SQL Server, работает в Oracle и MySQL**

Внешние соединения

- ***Внешнее соединение*** включает несовпавшие строки как минимум одной из таблиц в результаты запроса.

- Три основных типа:
  - Левое внешнее соединение
  - Правое внешнее соединение
  - Полное внешнее соединение

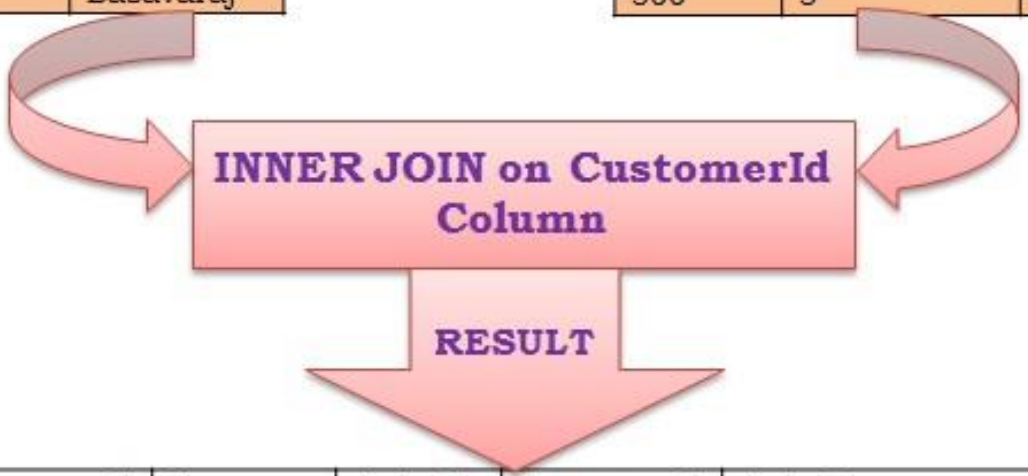
INNER JOIN

Customers

CustomerId	Name
1	Shree
2	Kalpana
3	Basavaraj

Orders

OrderId	CustomerId	OrderDate
100	1	2014-01-29 23:56:57.700
200	4	2014-01-30 23:56:57.700
300	3	2014-01-31 23:56:57.700



CustomerId	Name	OrderId	CustomerId	OrderDate
1	Shree	100	1	2014-01-30 23:48:32.850
3	Basavaraj	300	3	2014-02-01 23:48:32.853

LEFT OUTER JOIN

Customers

CustomerId	Name
1	Shree
2	Kalpana
3	Basavaraj

Orders

OrderId	CustomerId	OrderDate
100	1	2014-01-29 23:56:57.700
200	4	2014-01-30 23:56:57.700
300	3	2014-01-31 23:56:57.700

RESULT

CustomerId	Name	OrderId	CustomerId	OrderDate
1	Shree	100	1	2014-01-30 23:48:32.850
2	Kalpana	NULL	NULL	NULL
3	Basavaraj	300	3	2014-02-01 23:48:32.853

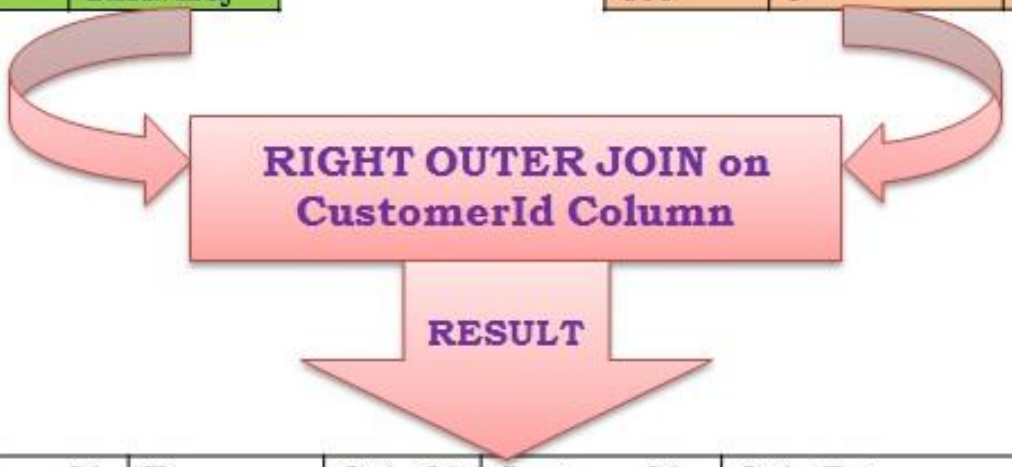
RIGHT OUTER JOIN

Customers

CustomerId	Name
1	Shree
2	Kalpana
3	Basavaraj

Orders

OrderId	CustomerId	OrderDate
100	1	2014-01-29 23:56:57.700
200	4	2014-01-30 23:56:57.700
300	3	2014-01-31 23:56:57.700



CustomerId	Name	OrderId	CustomerId	OrderDate
1	Shree	100	1	2014-01-30 23:48:32.850
NULL	NULL	200	4	2014-01-31 23:48:32.853
3	Basavaraj	300	3	2014-02-01 23:48:32.853

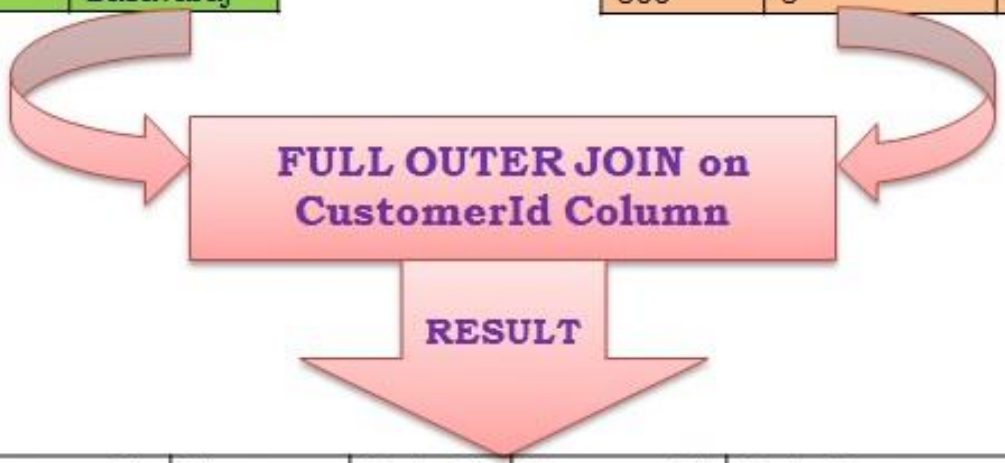
FULL OUTER JOIN

Customers

CustomerId	Name
1	Shree
2	Kalpana
3	Basavaraj

Orders

OrderId	CustomerId	OrderDate
100	1	2014-01-29 23:56:57.700
200	4	2014-01-30 23:56:57.700
300	3	2014-01-31 23:56:57.700



CustomerId	Name	OrderId	CustomerId	OrderDate
1	Shree	100	1	2014-01-30 23:48:32.850
2	Kalpana	NULL	NULL	NULL
3	Basavaraj	300	3	2014-02-01 23:48:32.853
NULL	NULL	200	4	2014-01-31 23:48:32.853

Внешние соединения

- Общий синтаксис внешнего соединения:

```
имя_таблицы {RIGHT | LEFT |FULL} [OUTER] JOIN
имя_таблицы
{ ON условие | USING (имя_поля [,имя_поля]) }
```

Внешние соединения. Пример

```
SELECT MOVIE_GENRE_DESCRIPTION AS GENRE,
       MOVIE_TITLE
FROM MOVIE_GENRE LEFT OUTER JOIN MOVIE
ON MOVIE_GENRE.MOVIE_GENRE_CODE =
  MOVIE.MOVIE_GENRE_CODE
```

Рефлексивные соединения

- ***Рефлексивное соединение*** – соединение таблицы с самой собой

```
SELECT PERSON_ID, EMPLOYEE_HOURLY_RATE
  AS HOURLY_RATE, SUPERVISOR_ID
FROM EMPLOYEE;
```

```
PERSON_ID    HOURLY_RATE    SUPEVISOR_ID
-----------------------------------------
         1             15
         2           9.75               1
        10           9.75               1
```

Рефлексивные соединения. Пример

```
SELECT A.PERSON_ID, A.EMPLOYEE_HOURLY_RATE
  AS HOURLY_RATE, B.EMPLOYEE_HOURLY_RATE AS
  SUPV_HOURLY_RATE
FROM EMPLOYEE A JOIN EMPLOYEE B
 ON A.SUPERVISOR_ID = B.PERSON_ID
```

- ***Использование псевдонимов необходимо***

Перекрестные соединения

- ***Перекрестное соединение*** – стандартный синтаксис для декартова произведения.

```
SELECT MOVIE_ID AS ID,
       MOVIE_GENRE_DESC AS GENRE,
       MOVIE_TITLE
FROM MOVIE CROSS JOIN MOVIE_GENRE
ORDER BY MOVIE_ID
```

Вложенные запросы

Вложенные запросы

- Вложенные запросы представляют собой подчиненные запросы `SELECT`
- Обычно помещаются после `WHERE` в качестве способа ограничения строк, возвращаемых в списке результатов внешнего запроса

- Вложенные запросы должны быть заключены в скобки
- Все, что можно сделать с помощью вложенных запросов, можно сделать с помощью соединений

Вложенные запросы

- ***Некоррелированный вложенный запрос*** – это вложенный запрос, в котором внутренний запрос не обращается к содержащему его внешнему запросу

- ***Коррелированный вложенный запрос*** – это вложенный запрос, в котором внутренний запрос обращается к значениям, полученным с помощью внешнего запроса

Некоррелированные вложенные запросы

Пример. Вывести список всех языков, на которых в видеопрокате нет фильмов

```
SELECT LANGUAGE_CODE, LANGUAGE_NAME
FROM LANGUAGE
WHERE LANGUAGE_CODE NOT IN
  (SELECT DISTINCT LANGUAGE_CODE
   FROM MOVIE_LANGUAGE)
ORDER BY LANGUAGE_CODE
```

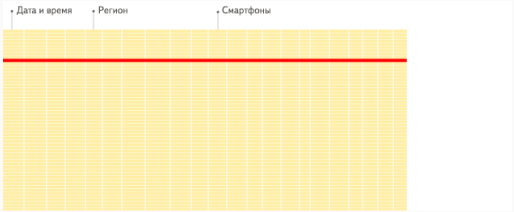
Коррелированные вложенные запросы

Пример. Вывести список всех клиентов, заплатившим более 15 долларов за прокат

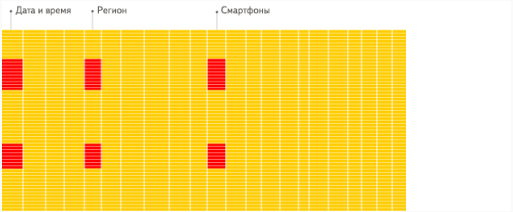
```
SELECT DISTINCT CUSTOMER_ACCOUNT_ID
FROM CUSTOMER_TRANSACTION AS A
WHERE 15 <
  (SELECT SUM(RENTAL_FEE)
   FROM MOVIE_RENTAL AS B
   WHERE A.TRANSACTION_ID =
         B.TRANSACTION_ID )
```

Лекция 2.
ClickHouse

Реляционные БД vs Колоночные БД



Реляционная СУБД



Аналитическая СУБД: ClickHouse

https://clickhouse.tech/docs/ru/

https://habr.com/ru/company/yandex/blog/303282/

https://habr.com/ru/company/vk/blog/430168/

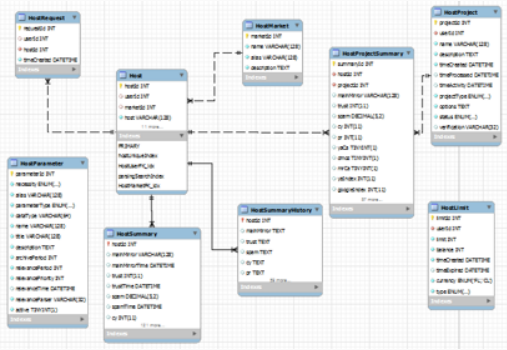
Преимущества и особенности (основные)

- Использование
  - Подходит для OLAP (online analytical processing) сценария работы
    Читая данные по колонкам, намного быстрее выбираем нужные данные, быстрее их сжимаем и больше данных помещаются в кэш => суперскорость (обработать 5 колонок из 100 в 20 раз быстрее, чем в реляционных базах).
- Стоимость
  - ClickHouse — СУБД с открытым исходным кодом (open source)
    Можно напрямую влиять на продукт через PR на GitHub
    Проще писать собственные драйверы для связки ClickHouse с различными интерфейсами
- Поддержка SQL
- Больше преимуществ: `https://clickhouse.tech/docs/ru/introduction/distinctive-features/`

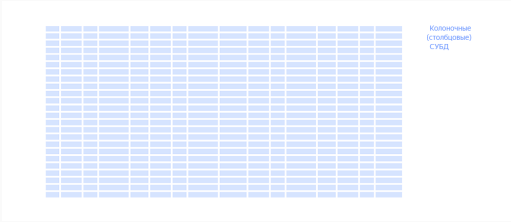
Преимущества и особенности (основные)

- Проблемы
  - Невозможно удалить записанные данные построчно, но возможно по партиции **ALTER TABLE HITS DROP PARTITION '2020-02-19'**
  - Не подходит для хранения транзакционных данных
  - Отсутствие оконных функций (постепенно исправляется в новых версиях)
  - Тяжелые операции с JOIN
  - Каждая колонка — отдельная нагрузка на оперативную память сервера, SELECT * является очень тяжелым запросом даже с лимитом для широких таблиц

Реляционные БД vs Колоночные БД



Типичная структура таблиц в реляционной БД



В ClickHouse такую структуру может заменять одна широкая таблица

Создание таблицы

```
CREATE TABLE default.ads_data
(
    `date`                  Date,
    `time`                  DateTime,
    `event`                 LowCardinality(String),
    `platform`              LowCardinality(String),
    `ad_id`                 Int32,
    `client_union_id`       Int32,
    `campaign_union_id`     Int32,
    `ad_cost_type`          LowCardinality(String),
    `ad_cost`               Float32,
    `has_video`             Int8,
    `target_audience_count` Int64
) ENGINE = MergeTree PARTITION BY date ORDER BY (time, ad_id) SAMPLE BY ad_id SETTINGS index_granularity = 8192
```

Движки

Семейства:

MergeTree	Log	Движки для интеграции	Специальные движки
MergeTree	TinyLog	Kafka	Distributed
ReplacingMergeTree	StripeLog	MySQL	MaterializedView
SummingMergeTree	Log	ODBC	Dictionary
AggregatingMergeTree		JDBC	Merge
CollapsingMergeTree			File
VersionedCollapsingMergeTree			Null
GraphiteMergeTree			Set
			Join
			URL
			View
			Memory
			Buffer

https://clickhouse.tech/docs/ru/engines/table-engines/

Use cases

MergeTree — самый функциональный движок для таблиц, к которым совершаются постоянные аналитические запросы

- Данные пишутся в фоновом режиме
- Хранит данные, отсортированные по первичному ключу
- Позволяет оперировать партициями, если задан ключ партиционирования
- Поддерживает репликацию данных (ReplicatedMergeTree)
- Поддерживает сэмплирование данных

Use cases

Buffer — используется при большом количестве INSERT-ов, которые не успевают обрабатываться, как дополнительная возможность добиться консистентности и снизить риск потери данных

- Буферизует записываемые данные в оперативке, периодически сбрасывая их в другую таблицу. При чтении, производится чтение данных одновременно из буфера и из другой таблицы

JDBC — движок для чтения внешних БД (например, прочитать MySQL таблицу), также JDBC драйвер позволяет подключаться к CH из pyspark джобы или интерфейсов вроде DataGrip

- Позволяет ClickHouse подключаться к внешним базам данных с помощью JDBC

Use cases

Distributed — используется для хранения огромных таблиц, которые не помещаются на один сервер. Ускоряет скорость обработки запроса за счет распределения частей запроса по шардам (серверам)

MaterializedView. Материализованное представление — это по сути такая же таблица, которая хранит данные, взятые их другой таблицы

Dictionary — данные в формате справочников, которые хранятся в памяти, что обеспечивает моментальный доступ к ним. Данные из словарей используются в SELECT запросах к оригинальной таблице через функцию dictGet()

Возможности MergeTree

```
CREATE TABLE default.ads_data
(
    `date`                Date,
    `time`                DateTime,
    `event`               LowCardinality(String),
    `platform`            LowCardinality(String),
    `ad_id`               Int32,
    `client_union_id`     Int32,
    `campaign_union_id`   Int32,
    `ad_cost_type`        LowCardinality(String),
    `ad_cost`             Float32,
    `has_video`           Int8,
    `target_audience_count` Int64
) ENGINE = MergeTree PARTITION BY date ORDER BY (time, ad_id) SAMPLE BY ad_id SETTINGS index_granularity = 8192
```

Проще всего делать партиционирование по дням или месяцам (по месяцам дешевле по памяти)

Первичный ключ не обязательно должен быть уникальным. Первичный ключ обычно совпадает с ключом сортировки

Полезно прописывать колонку для сэмплирования, но если нужно будет сделать выборку по другой колонке, то можно использовать функции хэширования и фильтрацию по остатку от деления
Например, берем каждого второго пользователя cityHash64(user_id, randConstant()) % 2 = 0

Типы данных

Утяжеляет таблицу

- String
- Float32/64
- Nullable()

Альтернатива

- LowCardinality(String)
- Int32/64
- 0

ClickHouse применяет словарное кодирование в столбцы типа LowCardinality. Если словарь содержит менее 10 000 различных значений, ClickHouse в основном показывает более высокую эффективность чтения и хранения данных. Если же словарь содержит более 100 000 различных значений, ClickHouse может работать хуже, чем при использовании обычных типов данных

Также поддерживаются массивы, кортежи, вложенные структуры (Nested)

Синтаксис

Выражение WITH, доступное в SQL, отличается своими возможностями:

1. Рекурсивные запросы не поддерживаются
2. Если в качестве выражения используется подзапрос, то результат должен содержать ровно одну строку
3. Результаты выражений нельзя переиспользовать во вложенных запросах

В дальнейшем, результаты выражений можно использовать в секции SELECT

Синтаксис

```
WITH
  (
    SELECT sum(bytes)
    FROM system.parts
    WHERE active
  ) AS total_disk_usage
SELECT
  (sum(bytes) / total_disk_usage) * 100 AS table_disk_usage,
  table
FROM system.parts
GROUP BY table
ORDER BY table_disk_usage DESC
LIMIT 10;
```

Синтаксис

HAVING — очень полезное выражение, работает также как в SQL и позволяет фильтровать результат группировки сразу после GROUP BY без дополнительного подзапроса

```
SELECT session_id
FROM message
WHERE bot_id = 'botId-1'
GROUP BY session_id
HAVING count() > 3;
```

Синтаксис

1. Функции для работы с массивами

   https://clickhouse.com/docs/ru/sql-reference/functions/array-functions

2. Функции агрегации с примерным результатом

   Например, uniq() выдает примерный результат, но быстрее
   uniqExact() выдает точный результат, но медленнее

3. Функции для работы с текстом URL

   https://clickhouse.com/docs/ru/sql-reference/functions/url-functions

4. ...

Ещё несколько трюков ClickHouse

#1 Семплирование

```
SELECT
  count()*10 as visits,
  sum(PageViews)*10 as hits,
  uniq(UserID)*10 as users,
  URL as url
FROM visits_table SAMPLE 1/10
```

#1 Семплирование

```
SELECT
  count() as visits,
  sum(PageViews) as hits,
  uniq(UserID) as users,
  URL as url
FROM visits_table SAMPLE 1/10 OFFSET {i}/10
-- где i = 0, 1, ..., 9
GROUP BY url
```

#2 Интервалы для дат

```
SELECT
  today(),
  today() - 7 AS prev_week_date,
  now(),
  now() - 7 * 24 * 60 * 60 AS prev_week_date_time
```

```
┌─today()─┬─prev_week_date─┬──────────────now()─┬─prev_week_date_time─┐
│ 2017-12-08 │     2017-12-01 │ 2017-12-08 16:00:27 │ 2017-12-01 16:00:27 │
└──────────┴────────────────┴─────────────────────┴─────────────────────┘
```

#2 Прошлый месяц

```
SELECT
  today(),
  toStartOfMonth(today() - toDayOfMonth(today())) AS prev_month_start,
  prev_month_start + toDayOfMonth(today() - 1) AS prev_month
```

```
┌─today()─┬─prev_month_start─┬─prev_month─┐
│ 2017-12-08 │       2017-11-01 │ 2017-11-08 │
└────────────┴──────────────────┴────────────┘
```

#2 Функция INTERVAL

```
SELECT
  today(),
  today() - INTERVAL 1 MONTH AS prev_month
```

```
┌─today()─┬─prev_month─┐
│ 2017-12-08 │ 2017-11-08 │
└────────────┴────────────┘
```

#3 Фильтрация по интервалу дат

```
SELECT
  count() as visits,
  sum(PageViews) as hits,
  uniq(UserID) as users
FROM visits_table
WHERE (Date >= toDate('2017-06-01'))
  AND (Date <= toDate('2017-08-31'))
```

#3 BETWEEN

```
SELECT
  count() as visits,
  sum(PageViews) as hits,
  uniq(UserID) as users
FROM visits_table
WHERE Date BETWEEN '2017-06-01' AND '2017-08-31'
```

#4 Условия и преобразования

```
SELECT
  if(UserAgent LIKE '%Yabrowser%', 'yabrowser',
    if(UserAgent LIKE '%Firefox%', 'firefox',
      if(UserAgent LIKE '%Opera%', 'opera',
        if(UserAgent LIKE '%Chrome%', 'google_chrome', 'other')
      )
    )
  ) as browser,
  count() as visits
FROM visits_table
GROUP BY browser
```

#4 Функция multiIf

```
SELECT
  multiIf(
    UserAgent LIKE '%Yabrowser%', 'yabrowser',
    UserAgent LIKE '%Firefox%', 'firefox',
    UserAgent LIKE '%Opera%', 'opera',
    UserAgent LIKE '%Chrome%', 'google_chrome',
    'other'
  ) as browser,
  count() as visits
FROM visits_table
GROUP BY browser
```

#5 Дерево принятия решений на CH

```
SELECT *,
  multiIf(
    Sex = 'female', 1,
    Age > 9.5, 0,
    SibSp > 2.5, 0, 1
  ) as survived
FROM titanic_data
```

#5 Градиентный бустинг на CH

- CatBoost – open-source framework от Яндекса, основанный на градиентном бустинге
- применяем модель как `modelEvaluate('model_name', feature1, ..., featureN)`
- `https://clickhouse.com/docs/en/sql-reference/functions/other-functions#catboostevaluate`

#6 Модификаторы агрегатных функций

- If — позволяет откинуть часть строк при расчете агрегатной функции
- Array — работает с элементами массивов как с значениями
- ForEach — работает на уровне элементов массива

#6 Модификатор — If

```
SELECT
  uniqIf(UserID, Browser = 'YandexBrowser') as browser_users,
  uniq(UserID) as total_users,
  countIf(Browser = 'YandexBrowser') as browser_visits,
  count() as total_visits,
  round(100*browser_users/total_users, 2) as users_share,
  round(100*browser_visits/total_visits, 2) as visits_share
FROM visits_table
```

#6 Модификатор — Array

```
SELECT
  sumArray(a) AS sum1,
  sumArrayArray(b) AS sum2
FROM
(
  SELECT
    [1, 2, 3, 4, 5, 6] AS a,
    [[1, 2], [3], [4, 5, 6]] AS b
)
```

```
┌─sum1─┬─sum2─┐
│   21 │   21 │
└──────┴──────┘
```

#6 Модификатор — ForEach

```
SELECT
  avgForEach(test_results),
  quantileForEach(0.5)(test_results)
FROM
(
  SELECT arrayJoin([[1, 1, 1, 1, 1], [1, 0, 1, 1, 1], [1, 0, 1, 0, 0], [1,
  1, 1, 1, 1], [1, 1, 1, 0, 0], [0, 0, 0, 0, 0], [1, 1, 0, 0, 1], [0, 1,
  1, 1, 0], [1, 1, 1, 0, 1], [1, 0, 0, 0, 0]]) AS test_results
)
```

avg_results	median_results
[0.8,0.6,0.7,0.4,0.5]	[1,1,1,0,0.5]

ClickHouse и Python

```
!pip install clickhouse_connect

import clickhouse_connect
client = clickhouse_connect.get_client(host='HOSTNAME.clickhouse.cloud',
                                       port=8443,
                                       username='default',
                                       password='your password')
result = client.query('SELECT max(key), avg(metric) FROM new_table')
result.result_rows

Out[1]: [(2000, -50.9035)]
```

Лекция 3.
Apache Hadoop

Apache Hadoop

- Экосистема для распределенного хранения и распределенной обработки
- Обработка больших объемов данных
- Разработан на Java
- Open-source
- Принцип Data locality

Реляционные БД vs Колоночные БД

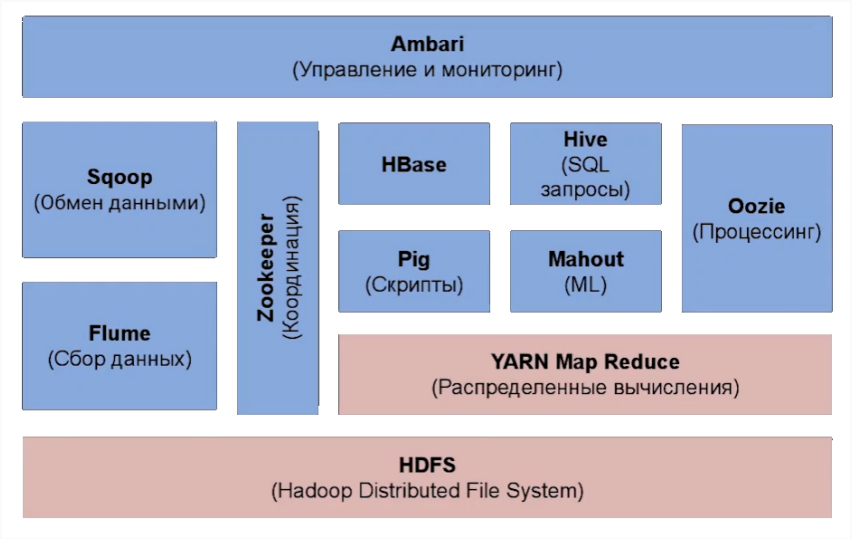
Плюсы:

- Масштабируемость
- Гибкость
- Отказоустойчивость
- Способен работать с большими объемами данных

Минусы:

- Проблемы безопасности
- Не разумен для большого числа данных маленького объема
- Высокая цена содержания инфраструктуры

Компоненты


Ambari
(Управление и мониторинг)
Sqoop
(Обмен данными)
Flume
(Сбор данных)
Zookeeper
(Координация)
HBase
Hive
(SQL
запросы)
Pig
(Скрипты)
Mahout
(ML)
Oozie
(Процессинг)
YARN Map Reduce
(Распределенные вычисления)
HDFS
(Hadoop Distributed File System)

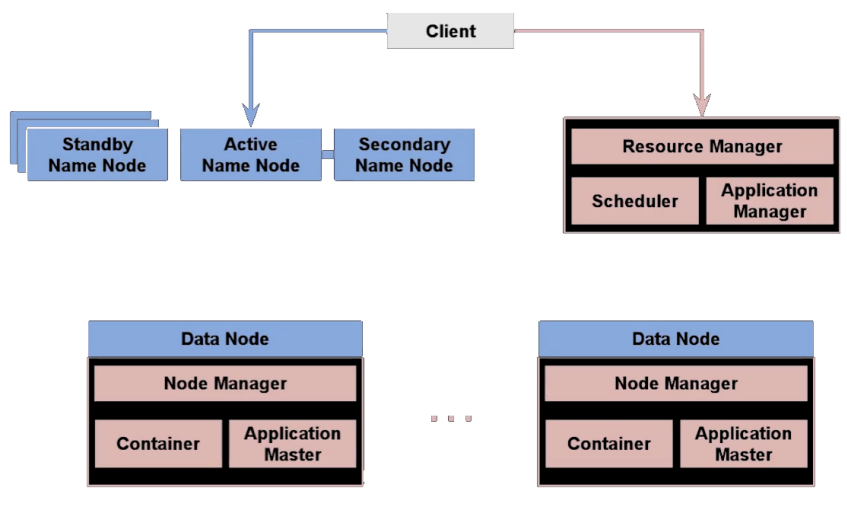
Базовые компоненты

- **HDFS** — специальная распределённая файловая система, которая позволяет хранить огромные объёмы данных
- **YARN** (Map Reduce Framework) — осуществление распределённых вычислений внутри кластера

Дополнительные компоненты

- Ambari — менеджер: управление кластером и его мониторинг
- Sqoop — выполнение процесса обмена данными
- Flume — выполнение процесса сбора данных
- Zookeeper — координатор между сервисами кластера
- HBase — колонкоориентированное хранилище, используется внутри экосистемы Hadoop
- Pig — компонент, позволяющий разрабатывать скрипты на определённом языке и применять их к распределённым данным внутри кластера
- Hive — реляционное представление над данными, предоставляющее возможности осуществлять SQL запросы
- Mahout — содержит алгоритмы машинного обучения и способен выполнять их над распределёнными данными
- Oozie — система планирования процессов для выполнения задач Hadoop

Архитектура

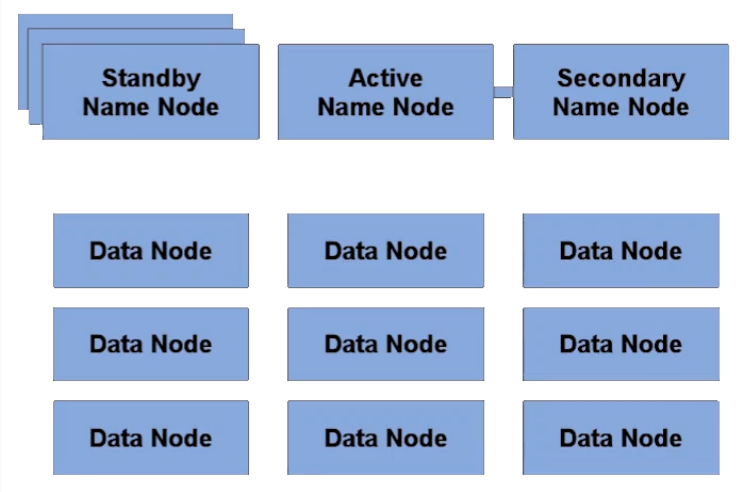


Компоненты архитектуры Hadoop-кластера

- **Name Node** — компонент, в котором хранится мета-информация о хранящихся данных; делится на три типа: Active, Standby и Secondary
- **Data Node** — хранит блоки данных
- **Resource Manager** — обладает информацией о занятых и свободных ресурсах, которые можно использовать для вычислений; эту роль выполняет YARN

HDFS

Архитектура



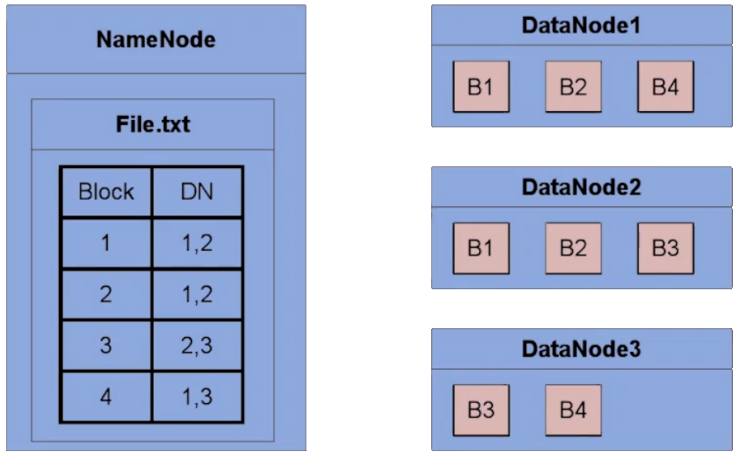
Основные компоненты архитектуры HDFS

- **Name Node** — метаинформация о данных, хранящихся в кластере
  - **Active Name Node** — главная, среди остальных, мастер-нода, с ней производятся основные взаимодействия; в кластере может быть только одна Active Name Node
  - **Standby Name Node** — резервная нода; в любой момент времени готова заменить Active Name Node
  - **Secondary Name Node** — выделенный узел, который взаимодействует с Active Name Node и выполняет операции сохранения контрольных точек о файловой системе; не может заменить Active Name Node
- **Data Node** — хранение блоков данных

Name Node

- Данные кластера недоступны, если нет ни одной Name Node
- Name Node хранит все метаданные:
  - расположение файлов в HDFS
  - информацию о правах на доступ
  - названия отдельных блоков
  - расположение блоков в кластере

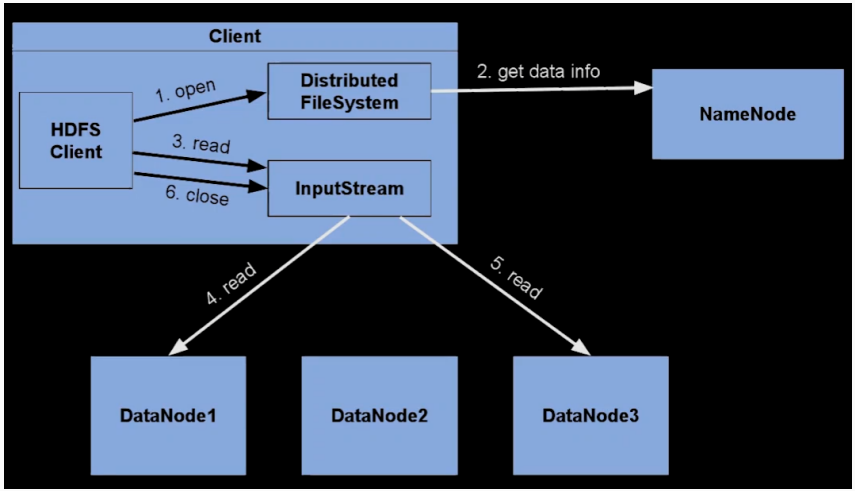
Хранение метаинформации о файлах, хранящихся в HDFS



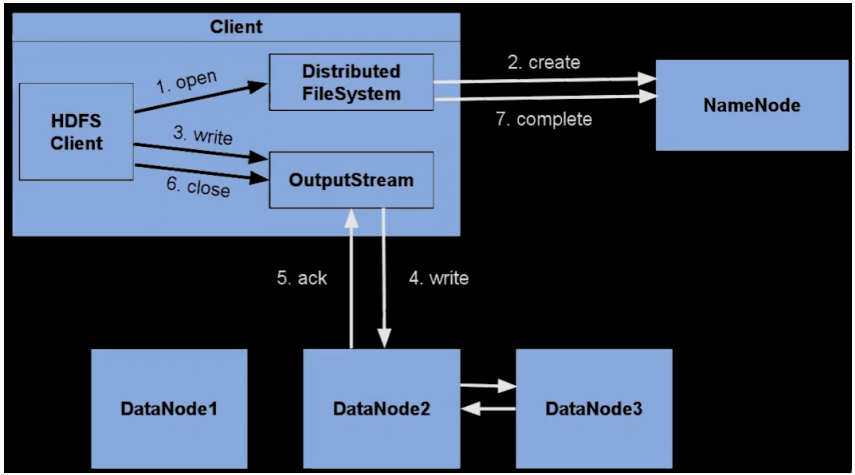
Data Node

- Блоки данных хранятся в виде стандартных файлов на узлах
- Узлы данных могут быть добавлены в кластер или удалены, а данные могут быть перебалансированы по узлам данных
- Информация о местонахождении данных может использоваться совместно с YARN для локального выполнения операций на узлах
- Каждый блок хранится на нескольких разных узлах для резервирования
- Контролирует доступ к блокам
- Взаимодействует с Name Node

HDFS — чтение


Client
HDFS Client
1. open
3. read
6. close
Distributed FileSystem
InputStream
2. get data info
NameNode
4. read
5. read
DataNode1
DataNode2
DataNode3

HDFS — запись

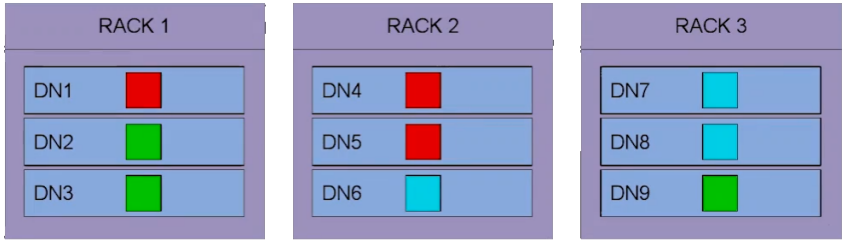

Client
HDFS Client
1. open
3. write
6. close
Distributed FileSystem
OutputStream
2. create
7. complete
NameNode
5. ack
4. write
DataNode1
DataNode2
DataNode3

HDFS — репликация

Replication (Репликация) — хранение копий блока на разных DataNode

- блоки реплицируются на Data Node (по умолчанию Replication Factor = 3, т.е. блоки хранятся одновременно на трех разных Data Node'ax)
- репликация обеспечивает надёжность и доступность данных

HDFS — осведомленность о стойке



Rack awateness (Осведомлённость о стойке) — это режим репликации, в котором репликация происходит приоритетней на узлах из другой стойки.

Реплицирование на нескольких стойках обеспечивает более отказоустойчивое хранение

HDFS — проблемы

Сохранение большого числа файлов маленького размера.

- маленький файл значительно меньше, чем размер блока HDFS (по умолчанию: 128 МБ)
- Name Node хранит метаданные в памяти (150 байт на блок)
- 10 миллионов файлам требуются метаданные 1.5ГБ

Это чревато:

- проблемами в кластере
- переполнением памяти в Name Node
- проблема обработки при выполнении параллельных процессов

HDFS — интерфейсы взаимодействия

- CLI — Command Line Intarface — Console
  - Hadoop CLI
- GUI — Graphic User Interface (Browser)
  - Ambari
  - Hue
- API
  - Java
  - C
  - REST API

HDFS — CLI — основные команды

- Загрузить файл

```
hdfs dfs -put data.txt /tmp/data.txt
```

- Скачать файл

```
hdfs dfs -get /tmp/data.txt /download
```

- Содержимое директории

```
hdfs dfs -ls /tmp
```

- Создать директорию

```
hdfs dfs -mkdir /tmp/test
```

- Удалить директорию

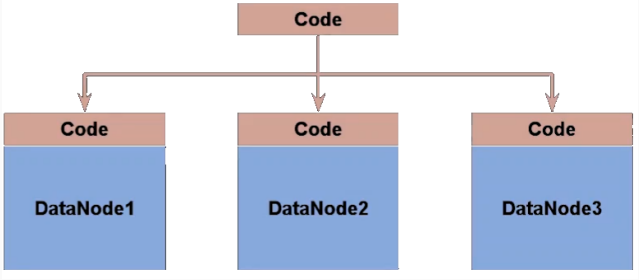
```
hdfs dfs -rm -R /tmp/test
```

MapReduce

Data locality

Принцип, применяющийся при работе с большими данными

- перемещение логики вычислений к данным
- задачи (MapReduce) копируются в узлы, данные не копируются к задачам
- минимализация нагрузки на сеть иди передачу данных

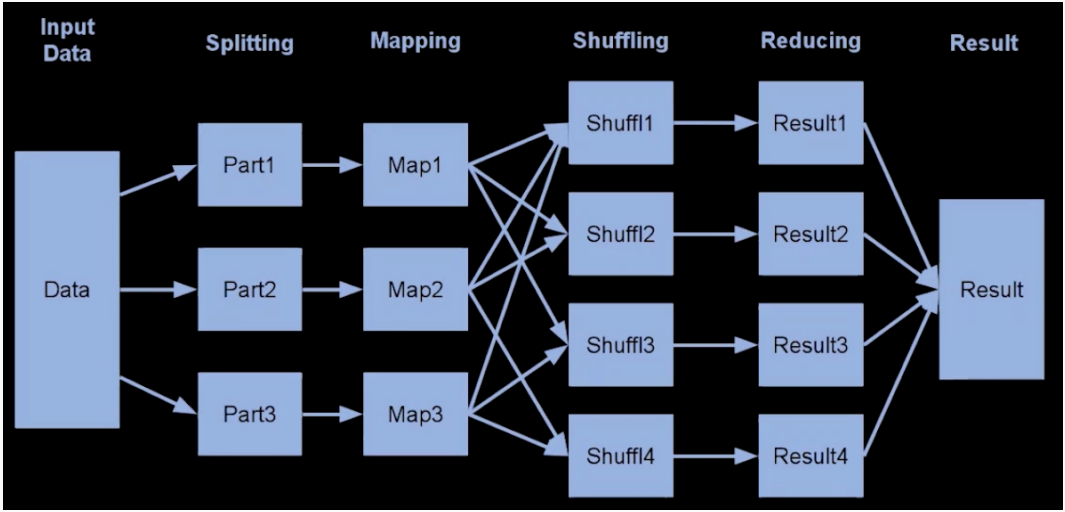


Map Reduce (MR)

MapReduce — модель распределённой обработки данных и среда исполнения, работающая на кластерах, состоящих из большого числа машин

- Автоматическое распараллеливание и распределение
- Отказоустойчивость
- Позволяет абстрагироваться при разработке алгоритмов распределённой обработки данных
- Обычно разрабатывается на Java

Map Reduce (MR)



Этапы любой MapReduce задачи

1. На первом этапе нужно определить **Input Data** — данные, которые будем обрабатывать
2. Стадия **Splitting** — разбиваем данные на некоторые части (партиции) для их последующей обработки
3. Стадия **Mapping**, на которой данные преобразуются в пары Ключ:Значение на каждой из партиций
4. Стадия **Shuffling** — создаются пары Ключ:Значение таким образом, чтобы на одной партиции были собраны все пары с одним ключом
5. Стадия **Reducing** — преобразование всех пар с одним и тем же ключом по какому-то закону в одну единственную пару Ключ:Значение
6. И наконец необходимо получить **Result** — результат, собрав все данные со всех партиций

Этапы любой MapReduce задачи

Для реализации такого подхода необходимо определить два закона:

1. Операция Map — по какому закону будем собирать данные в пары Ключ:Значение
2. Операция Reduce — по какому закону будем агрегировать пары с одинаковым ключом

Логика же других операций является достаточно общей и не зависит от тех алгоритмов, которые мы хотим реализовать

- Входные данные хранятся в HDFS, распределяются по узлам и реплицируются
- Приложение отправляет работу (Job) (map, reduce) на трекер работы (job tracker): приложение будет взаимодействовать с ресурсным менеджером (скорее всего, с YARN), и именно он будет контролировать ход задач алгоритма
- Разбивает входные данные
- Планирует и контролирует различные map и reduce задачи
- Выполняет различные map и reduce задачи

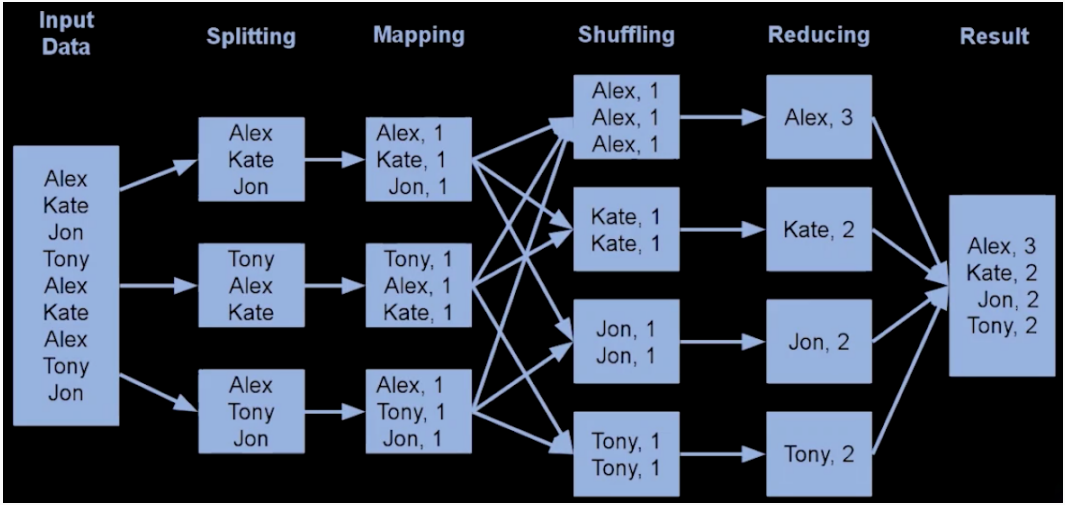
Map

- Выполняется параллельно
- Создает KV Ключ:Значение пары
- Обрабатывает блоки входных файлов

Reduce

- Получает пары Ключ:Значение, отсортированные по ключам
- Агрегирует данные
- Собирает результат
- Может быть распараллелен

Map Reduce. Пример



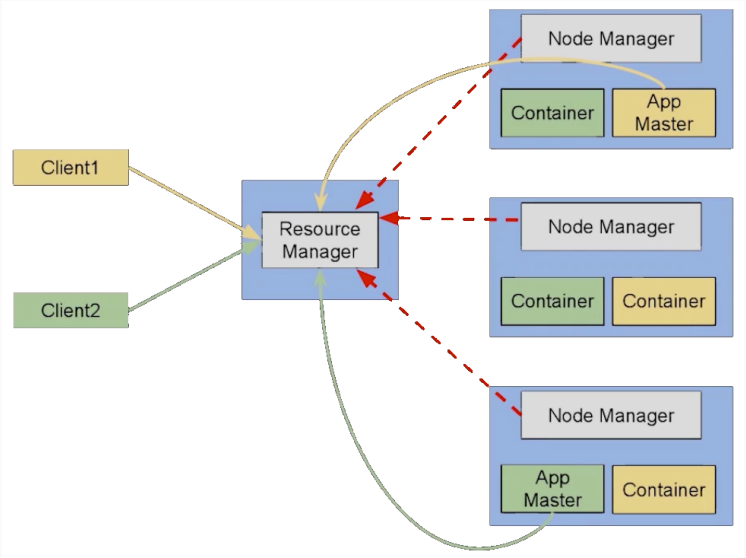
Map Reduce. Пример

1. **Input Data**: Имена людей.
2. **Splitting**: Разбитие данных на части для их последующей параллельной обработки
3. **Mapping**: Преобразование входных данных в пары Ключ:Значение. В данном случае ключами будут имена, а значением — 1 (поскольку встретили это имя пока в единственном экземпляре).
4. **Shuffling**: Сбор пар Ключ:Значение таким образом, чтобы каждое имя (каждый ключ) находилось в одной партиции
5. **Reducing**: Наборы пар Ключ:Значение группируем по ключу и высчитываем количество (count) таких пар — это и будет новое значение для пары
6. **Result**: Сбор результатов со всех партиций. Получаем список имён с количеством их вхождений в текст

YARN

- **Client**
  - Запуск задач
- **Resource Manager (RM)**
  - Управляет ресурсами на уровне кластера
- **Node Manager (NM)**
  - Управляет ресурсами на уровне одной ноды
- **Application Manager (AM)**
  - Управляет жизненным циклом приложения
  - Свой для каждого приложения
- **Container**
  - Запущенное приложение

YARN

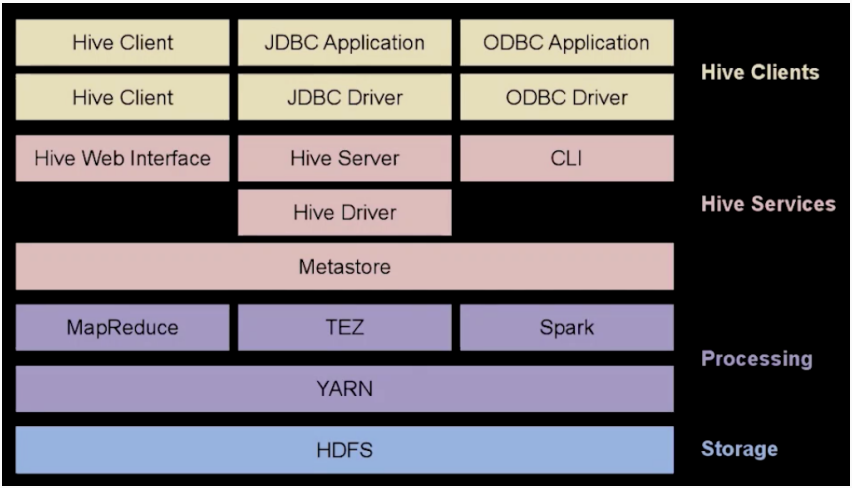

Node Manager
Container
App Master
Client1
Resource Manager
Node Manager
Client2
Container
Container
Node Manager
App Master
Container

Apache Hive

Apache Hive

1. Warehouse system — не является БД, а выполняет роль некоторой надстройки, которая работает над структурированными данными
2. Open Source
3. Базируется на Hadoop
4. Применяется для выполнения запросов и анализ больших наборов данных
5. Доступ к данным через SQL-подобные запросы (HiveQL)
6. Позволяет обобщать и агрегировать данные

Архитектура

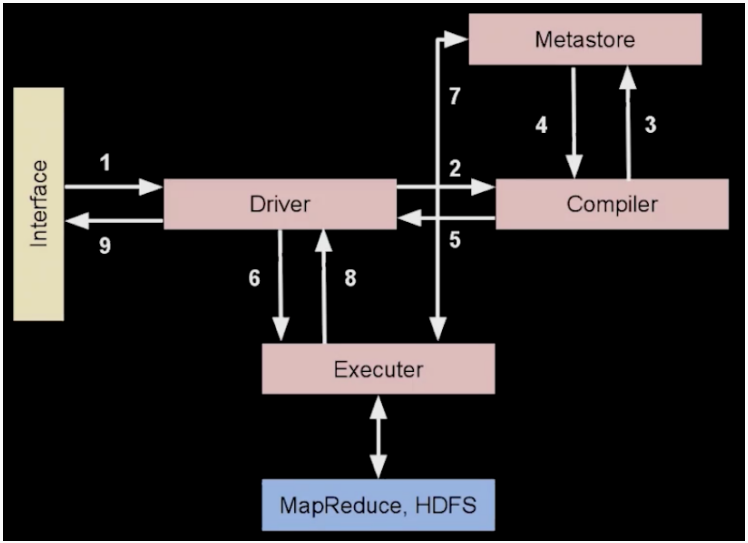


Архитектура

1. **Hive Client** — который позволяет работать с Hive
2. **Hive Services** — выполняют некоторую дополнительную логику для обеспечивания функционирования этого фреймворка
3. **Processing** — используются какие-либо движки для выполнения распределённых задач
4. **Storage** — файловая система (HDFS)

Следует отметить **Metastore**, который хранит информацию о данных, доступных для обработки, и Hive Driver, который выполняет транслитерацию SQL-языка в некоторый код, который впоследствии необходим для выполнения распределённых задач

Принцип работы



HiveQL

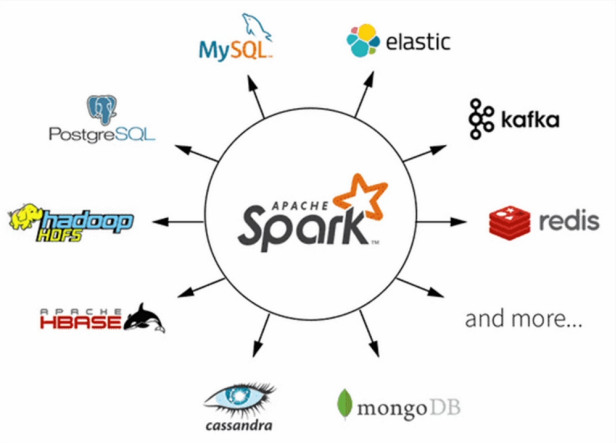
```
CREATE EXTERNAL TABLE IF NOT EXISTS
employee (
  id STRING,
  name STRING,
  surname STRING,
  age INT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
LOCATION '/employee/';
```

Лекция 4.
Введение в Spark

План занятия

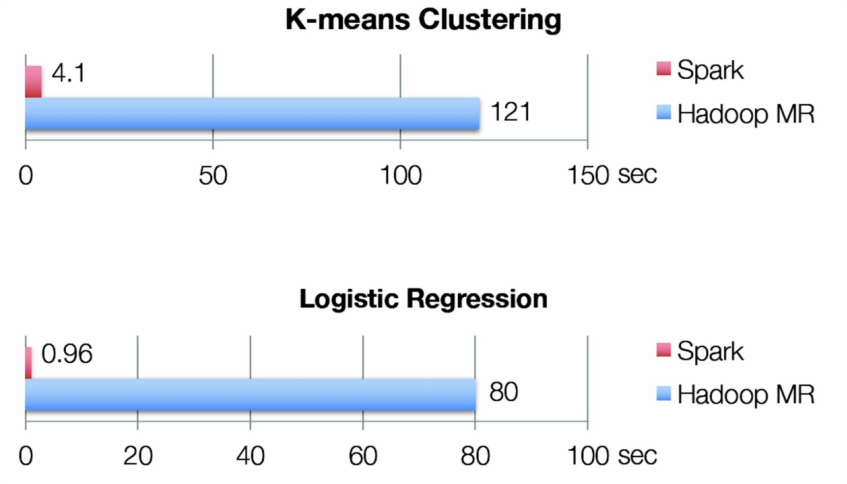
- Что из себя представляет Spark?
- В чем отличие от MapReduce?
- Какие виды Spark API бывают?

Apache Spark



- Самый популярный фреймворк распределённых вычислений
- Написан на Scala, Java
- Поддерживает API на Scala, Java, SQL, Python, R, C#, F#, ...

Spark vs Hadoop MR



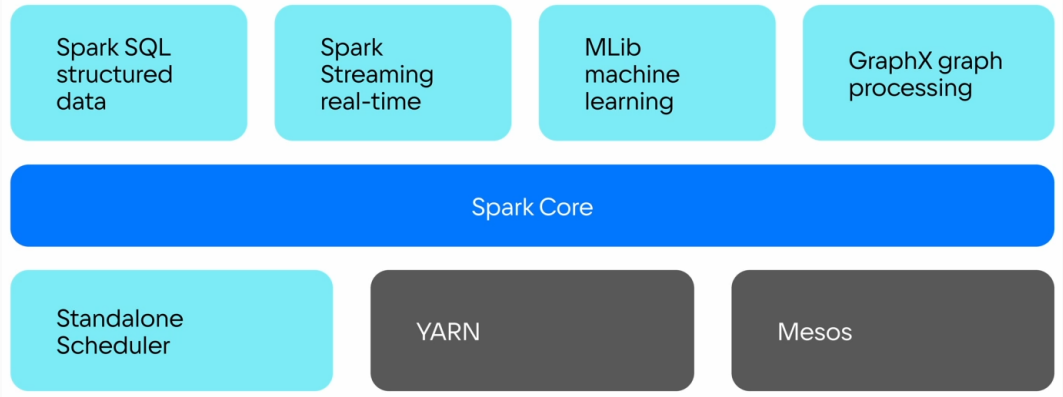
Почему Spark?

MapReduce vs **Spark**

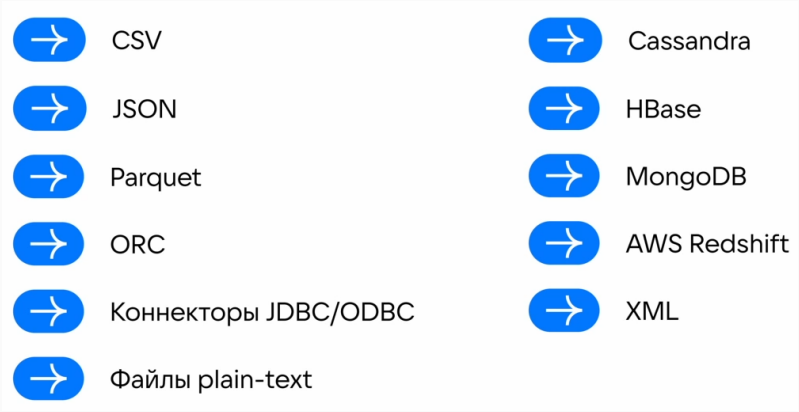
- Классический MapReduce работает медленно
- Мапперы пишут данные на диск редьюсера
- Нет автоматической оптимизации запросов

- Spark работает значительно быстрее
- Хранение промежуточных результатов в памяти
- Оптимизирует запросы
- Не привязан к HDFS
- Не привязан к менеджерам ресурсов

Компоненты Spark



Spark Data Sources



Архитектура Apache Spark

Application

Пользовательское приложение, написанное на Spark. Состоит из драйвера и исполнителя

Driver program

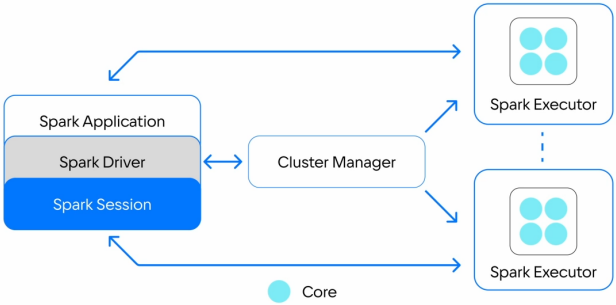
Программа с функцией main(), которая создаёт и запускает SparkContext

Deploy mode

Драйвер Cluster находится внутри кластера. Драйвер Client находится вне кластера

Executor

Процесс, запущенный для выполнения приложения на рабочей ноде



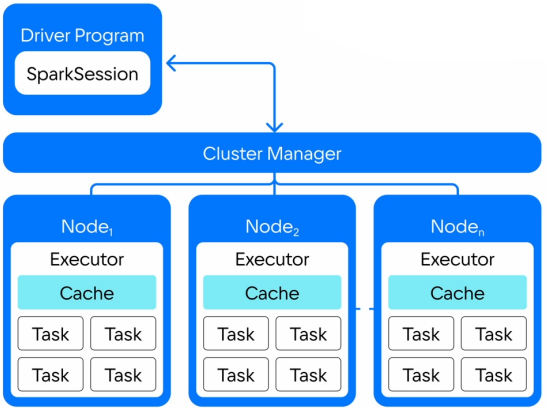
Apache Spark. Процесс работы приложения

Driver

1. Инициализирует приложение
2. Запрашивает ресурсы
3. Формирует физический план
4. Передаёт сериализованный код задач
5. Отслеживает их выполнение
6. Завершает приложение
7. Запрашивает освобождение ресурсов

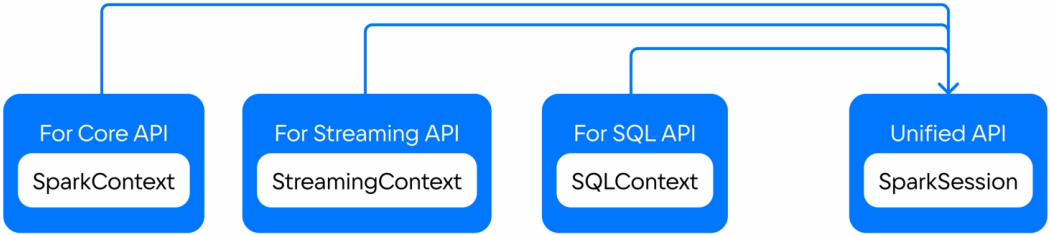
Executors

1. Получает от драйвера конкретную задачу
2. Отправляет статус выполнения на драйвер
3. Завершает выполнение по команде драйвера



Apache Spark. SparkSession

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder.appName("BlablaSession»).getOrCreate()
val people = spark.read.parquet("/some/path»)
```

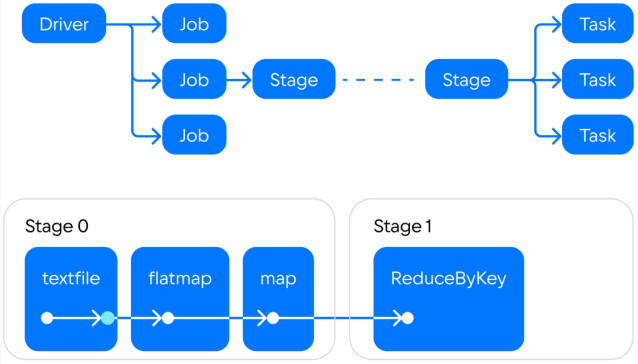


Apache Spark. Термины

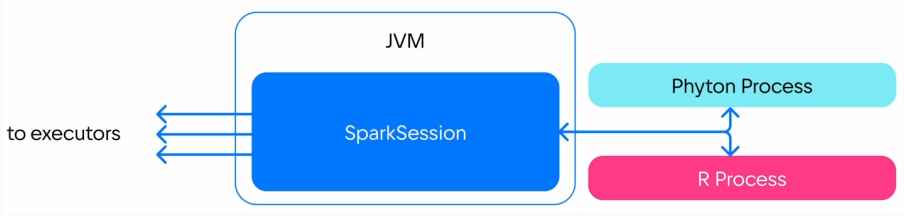
Task. Самая маленькая единица действий, которая будет отправлена на executor

Job. Параллельное вычисление, состоящее из нескольких стейджей, которое запускается в ответ на действие в Spark (например, save, collect)

Stage. Набор задач, на которое делится каждое задание (аналогично этапам Map и Reduce в MapReduce)

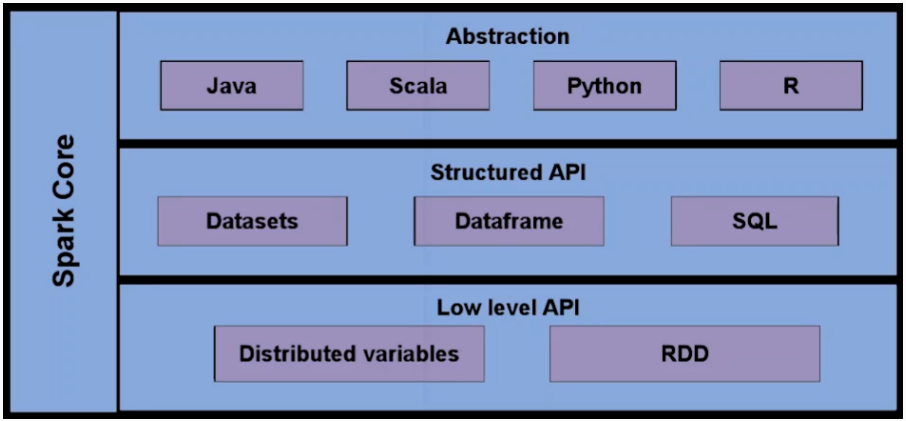


Apache Spark. External Processes

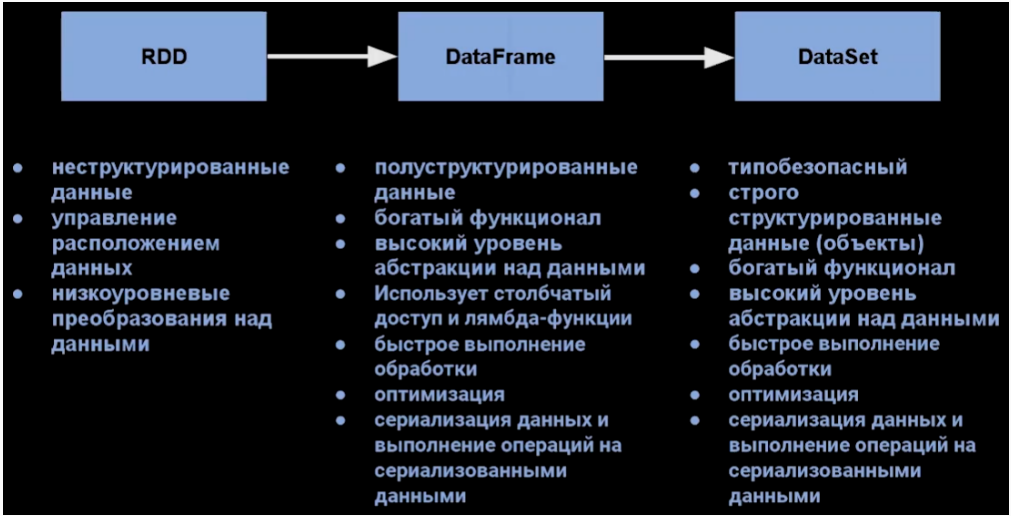


Spark Core

Spark Core

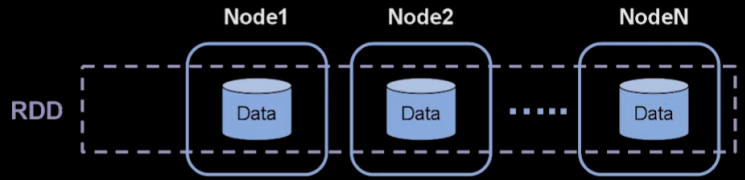

Spark Core
Abstraction
Java
Scala
Python
R
Structured API
Datasets
Dataframe
SQL
Low level API
Distributed variables
RDD

Spark Core


RDD
DataFrame
DataSet
• неструктурированные данные
• управление расположением данных
• низкоуровневые преобразования над данными
• полуструктурированные данные
• богатый функционал
• высокий уровень абстракции над данными
• Использует столбчатый доступ и лямбда-функции
• быстрое выполнение обработки
• оптимизация
• сериализация данных и выполнение операций на сериализованными данными
• типобезопасный
• строго структурированные данные (объекты)
• богатый функционал
• высокий уровень абстракции над данными
• быстрое выполнение обработки
• оптимизация
• сериализация данных и выполнение операций на сериализованными данными

RDD

RDD (Resilient Distributed Datasets) - это отказоустойчивая распределенная структура данных, которая позволяет пользователям явно сохранять промежуточные результаты в памяти, управлять их разбиением для оптимизации размещения данных и манипулировать ими, используя богатый набор операторов.




```
data = [1,2,3,4,5]

rdd = sc.parallelize(data)
```

Dataframe

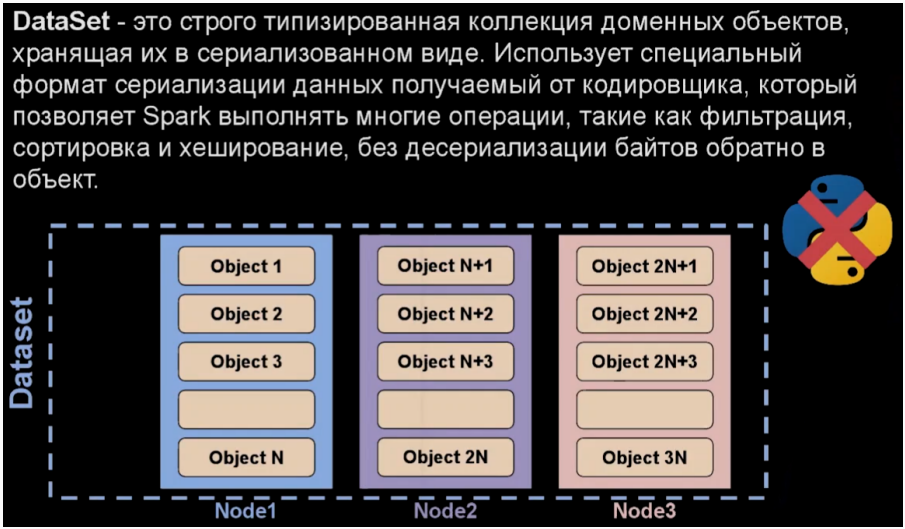
DataFrame - это распределенная коллекция данных, организованная в именованные столбцы. Эта структура концептуально эквивалентна таблице в реляционной базе данных или фрейму данных в R / Python, но с более мощной оптимизацией под капотом.

		Column1	Column2	....	ColumnN
Node1	Row1	V	V	V	V
	Row2	V	V	V	V
	...	V	V	V	V
NodeN	...	V	V	V	V
	...	V	V	V	V
	RowN	V	V	V	V

```
df = sqlContext.createDataFrame(rdd, schema)
```

Dataset

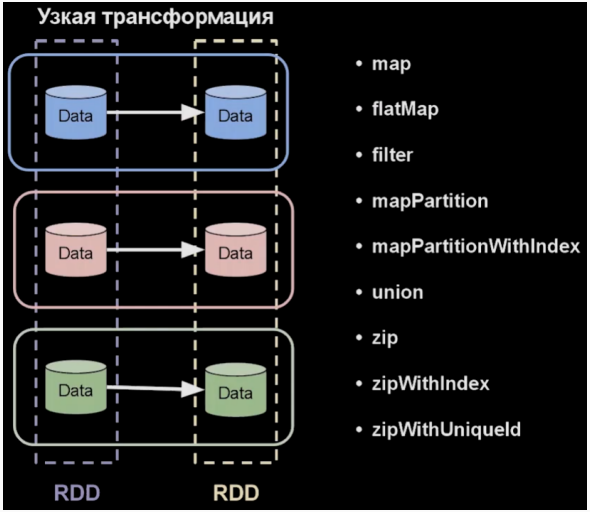
DataSet - это строго типизированная коллекция доменных объектов, хранящая их в сериализованном виде. Использует специальный формат сериализации данных получаемый от кодировщика, который позволяет Spark выполнять многие операции, такие как фильтрация, сортировка и хеширование, без десериализации байтов обратно в объект.



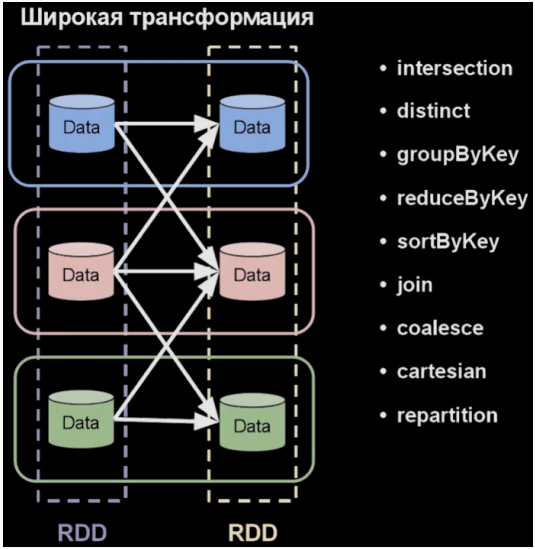
Типы трансформаций и действия

- Узкая трансформация (Narrow Transformation)
- Широкая трансформация (Wide Transformation)
- Действия (Actions)

Узкая трансформация



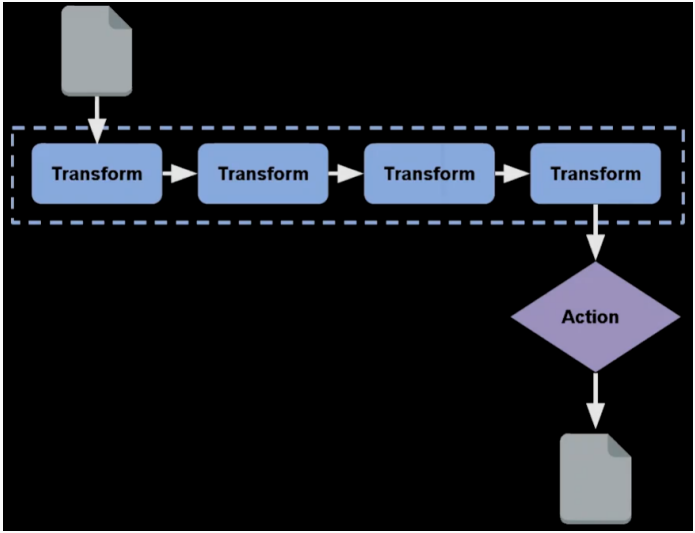
Широкая трансформация



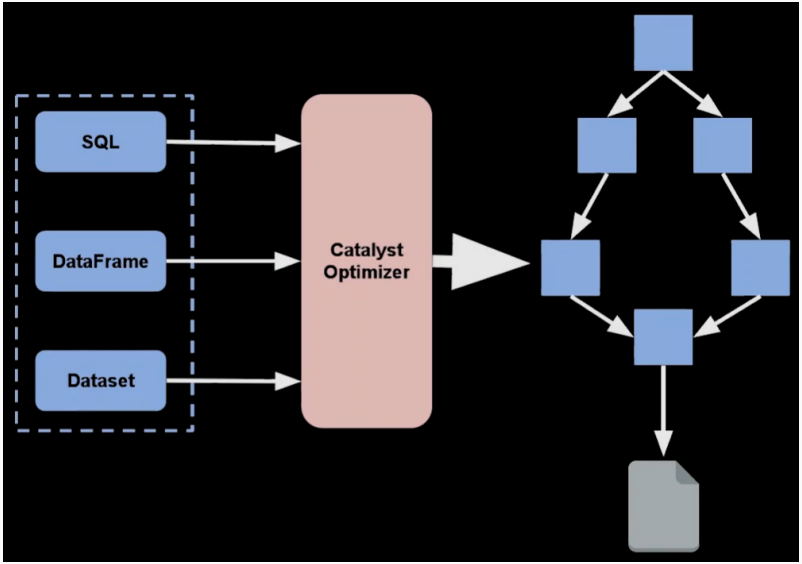
Действия

- count
- collect
- take
- top
- countByValue
- reduce
- fold
- aggregate
- foreach
- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile

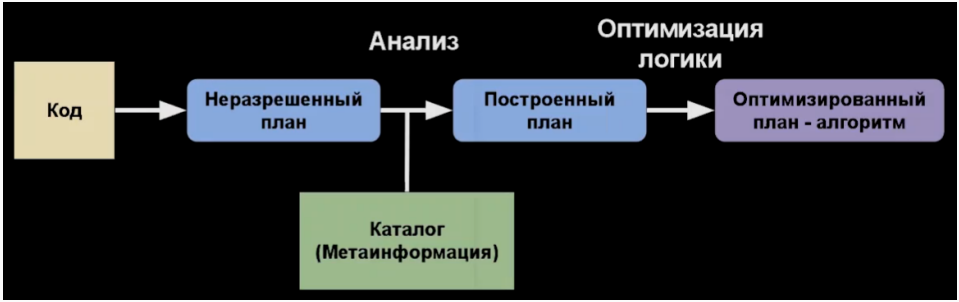
Ленивое выполнение



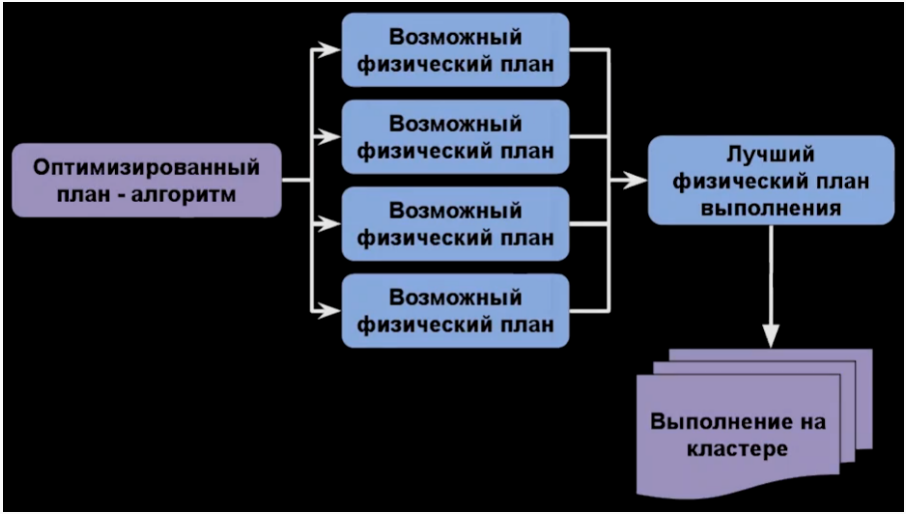
Оптимизация



Логическое планирование



Физическое планирование



DataFrame Spark vs Pandas

```
import pandas as pd

df = pd.DataFrame({
    “first”: range(200),
    “second”: range(200,400)
})

spark_df = spark.createDataFrame(df)

pandas_df = spark_df.toPandas()
```

DataFrame Spark vs Pandas

- Работа над PySpark DataFrame выполняется параллельно на разных узлах кластера, в случае с Pandas это невозможно.
- Операции с PySpark DataFrame по своей природе ленивы, в Pandas мы получаем результат, как только применяем какую-либо операцию.
- В PySpark мы не можем изменить DataFrame из-за его неизменного свойства(выполнение операций создает новый экземпляр).
  В Pandas операции изменяют текущий DataFrame.
- Pandas API поддерживает больше операций,
  чем PySpark DataFrame.
- Сложные операции проще выполнять в Pandas,
  чем в Pyspark DataFrame.

Поиск дубликатов

Постановка задачи

Постановка задачи

Нечеткие дубликаты очень имеют похожее, но не обязательно одинаковое содержание

Текст 1

санкт-петербург, 21 ноября - риа новости. полицейские задержали в пятницу подозреваемого по громкому делу о двойных и тройных продажах дольщикам квартир дома на ленинском проспекте, 93 в петербурге, сообщает в понедельник пресс-службы гумвд по сзфо.

Текст 2

с.-петербург, 5 дек - риа новости. полицейские задержали подозреваемого по громкому делу о двойных и тройных продажах дольщикам квартир дома на ленинском проспекте, 93 в петербурге, сообщает в понедельник заместитель пресс-службы гумвд по сзфо.

Постановка задачи

Два варианта постановки задачи:

1. Необходимо проверить, является ли новый документ дубликатом какого-либо документа в БД
   ($O(n)$ операций сравнения при решении «в лоб»)
2. Необходимо проверить, содержатся ли в БД документы, являющиеся дубликатами других документов друга
   ($O(n^2)$ операций сравнения при решении «в лоб»)

Актуальность задачи

1. Улучшение качества индекса и архивов поисковых систем за счет удаления избыточной информации
2. Объединение новостных сообщений в сюжеты на основе сходства этих сообщений по содержанию
3. Фильтрация спама
4. Установление нарушений авторских прав при незаконном копировании информации

Методы поиска нечетких дубликатов

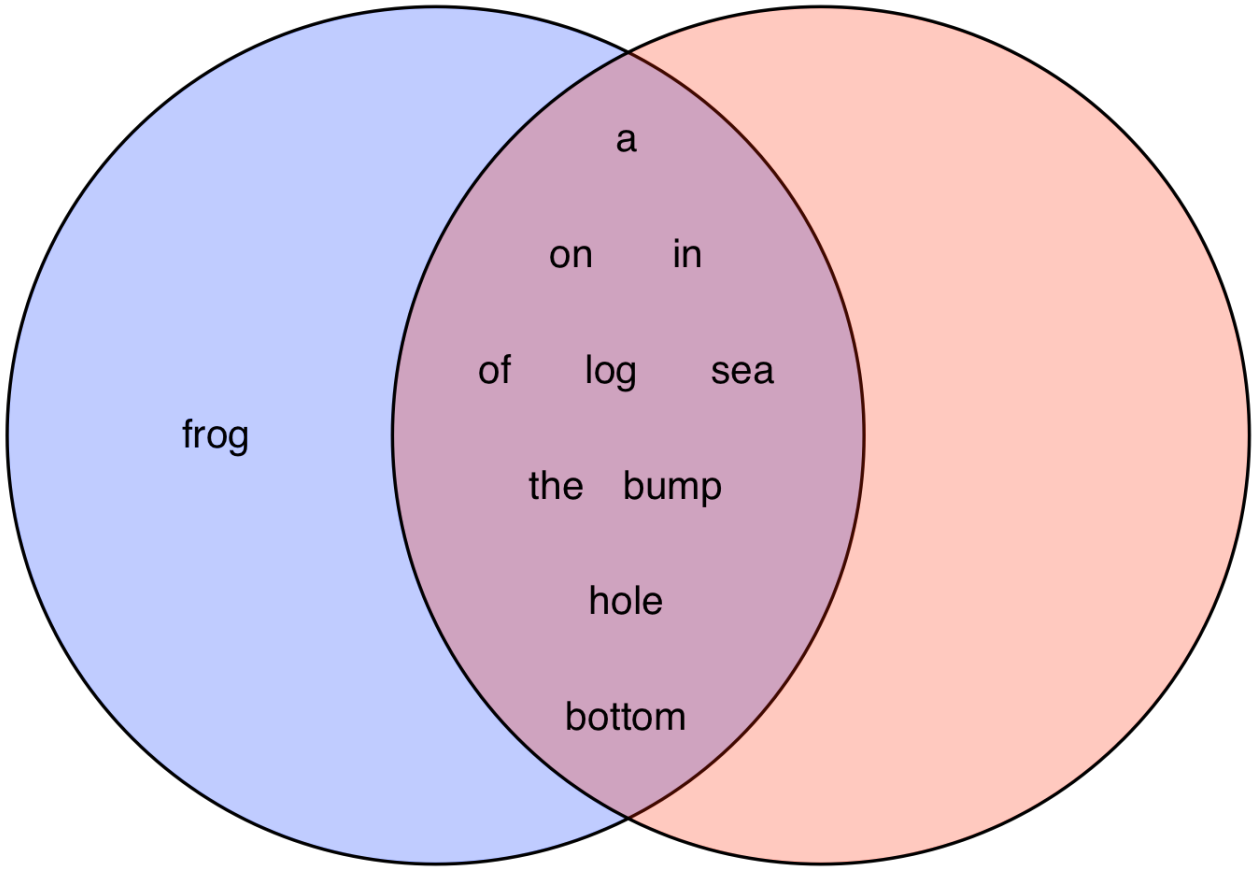
Методы поиска нечетких дубликатов

1. Мешок слов
2. Шинглинг
3. Хэширование
4. MinHash
5. LSH

Мешок слов

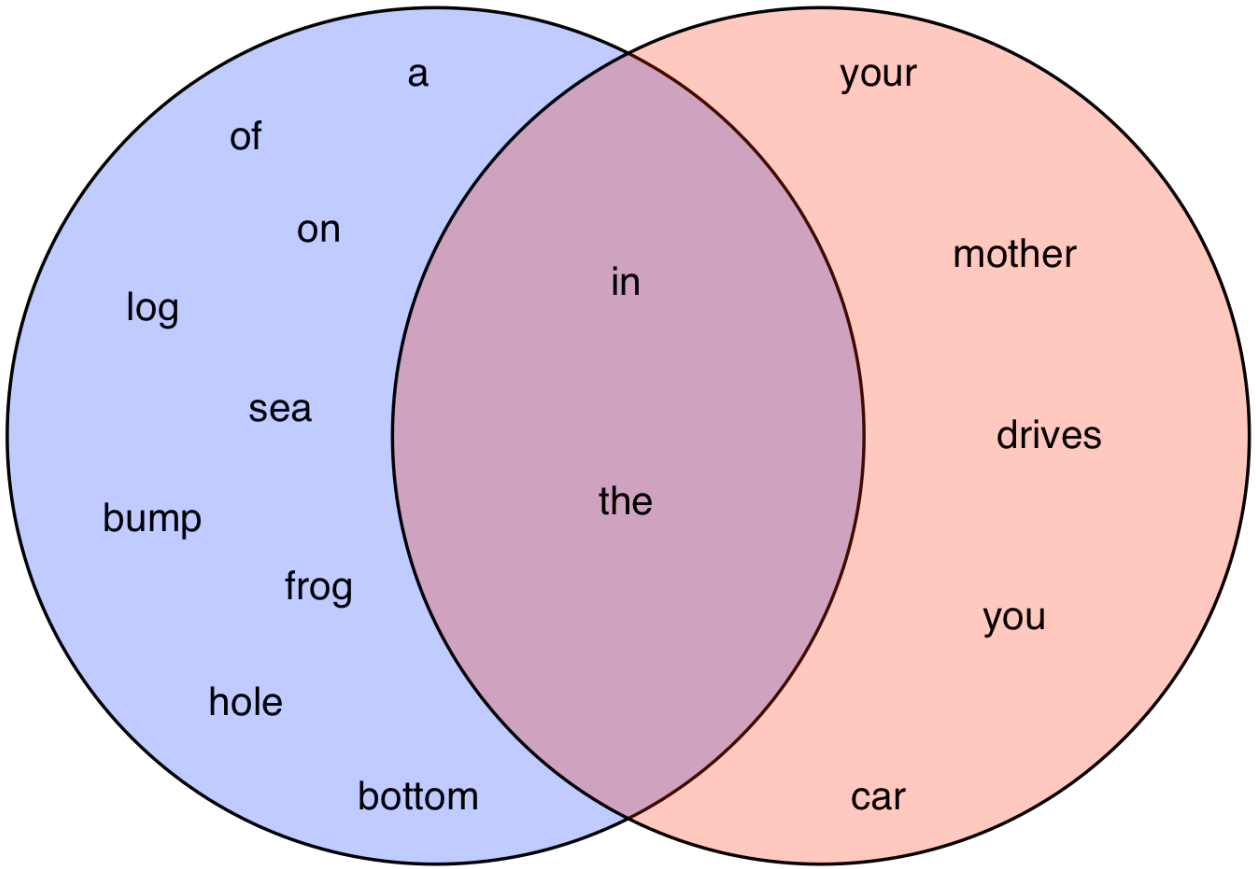
"a bump on the log in the hole in the bottom of the sea"
"a frog on the bump on the log in the hole in the bottom of the sea"

Коэффициент Жаккара: $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$



Мешок слов

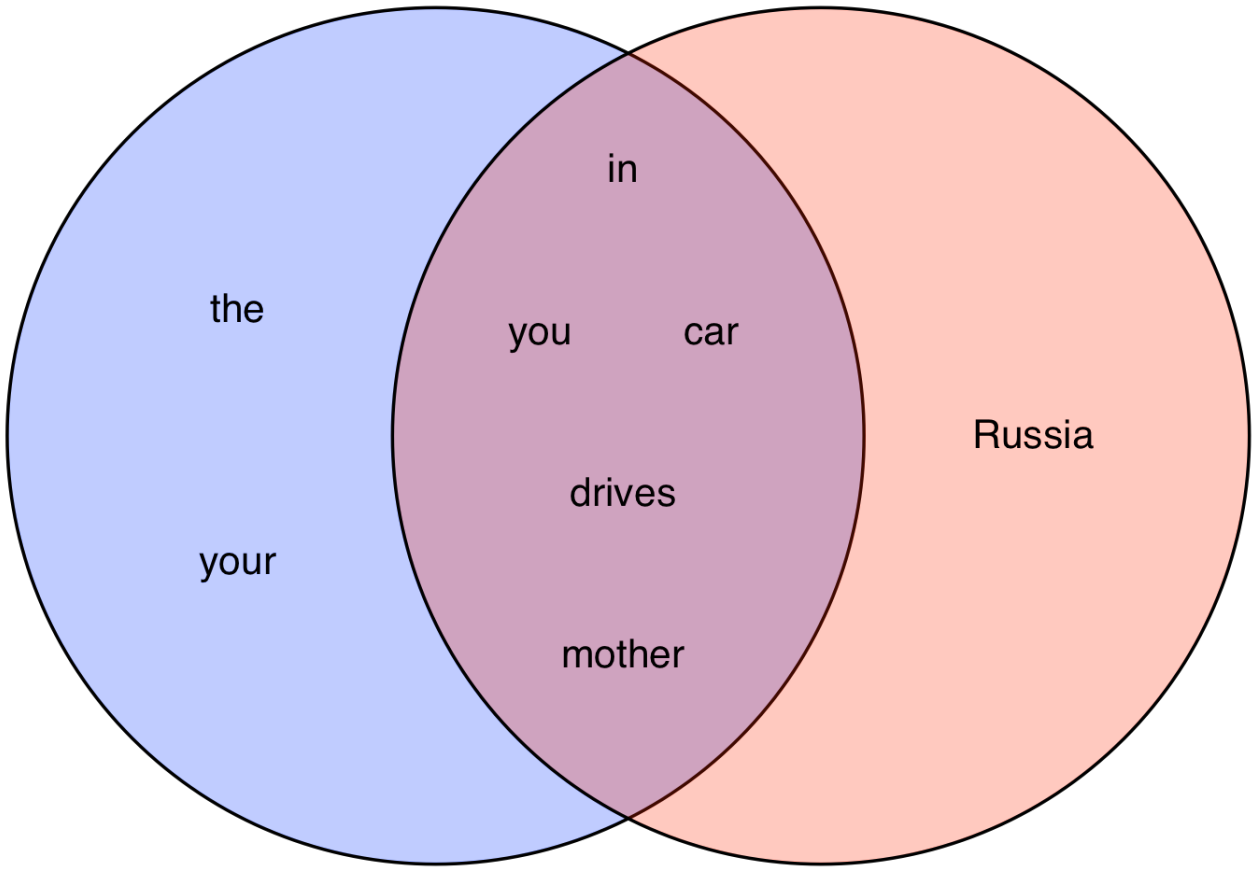
"a frog on the bump on the log in the hole in the bottom of the sea"
"your mother drives you in the car"



Мешок слов

"Your mother drives you in the car"
"In mother Russia, car drives you!"

Проблема: не учитывается контекст слов



Шинглинг

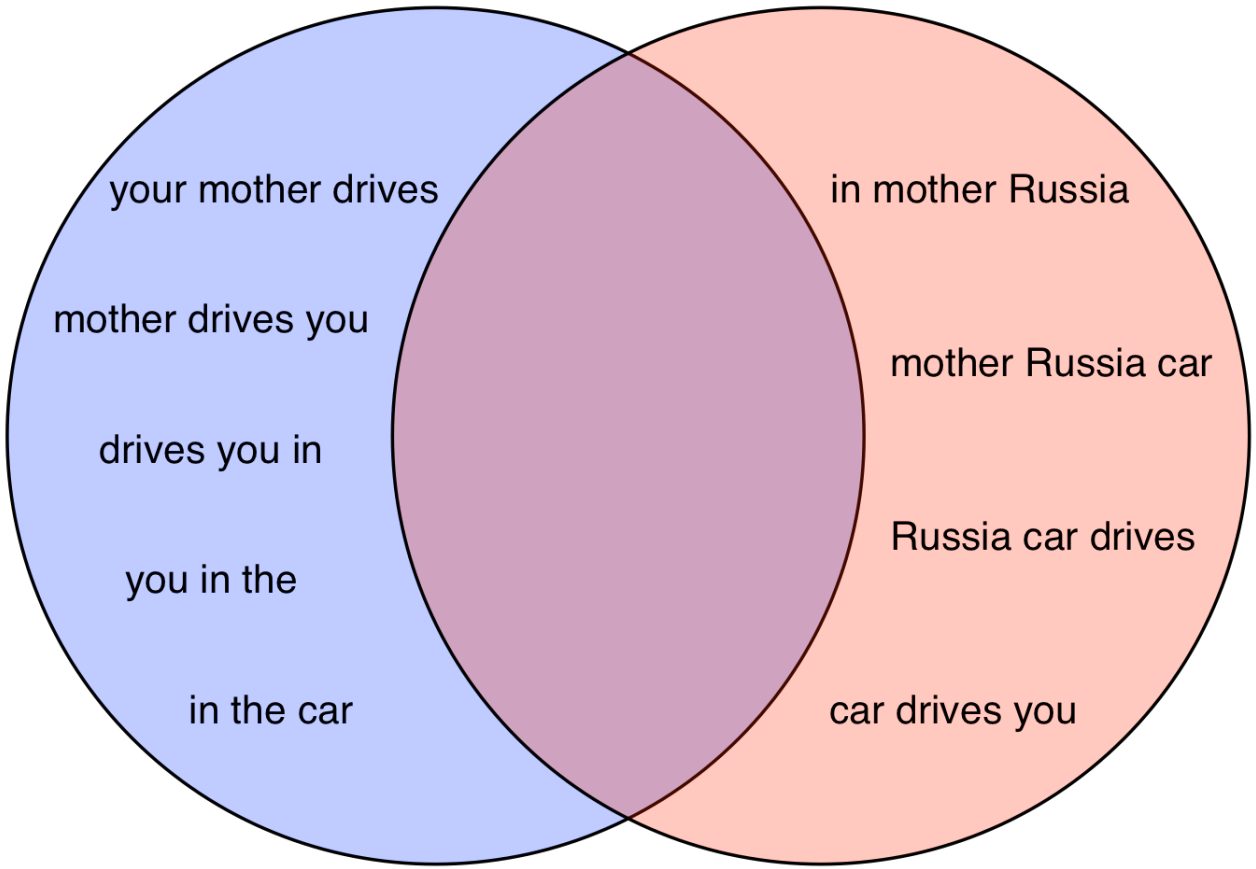
Представим документ в виде множества всевозможных последовательностей фиксированной длины k, состоящих из соседних слов — шинглов (shingles)

"Your mother drives you in the car":

- your mother drives
- mother drives you
- drives you in
- you in the
- in the car

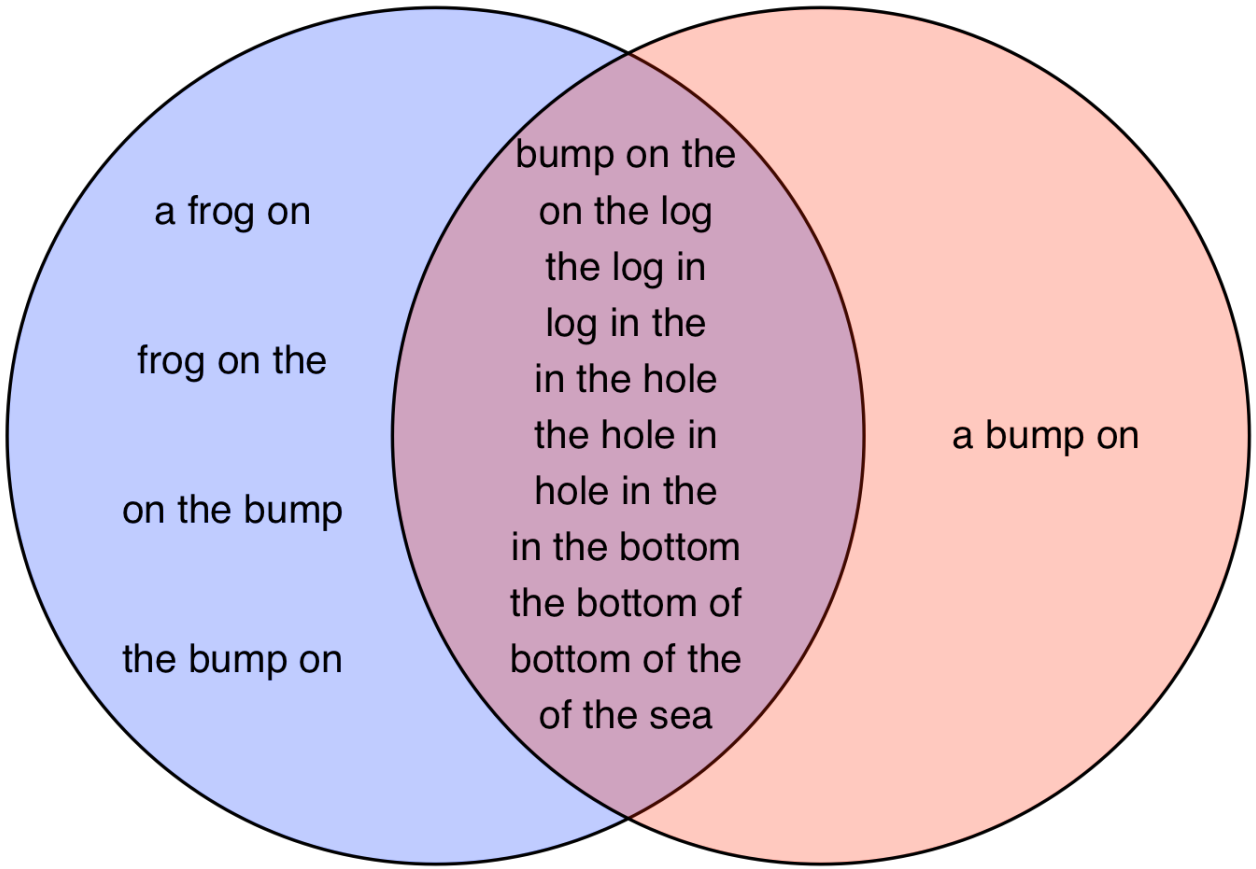
Шинглинг

"Your mother drives you in the car"
"In mother Russia, car drives you!"



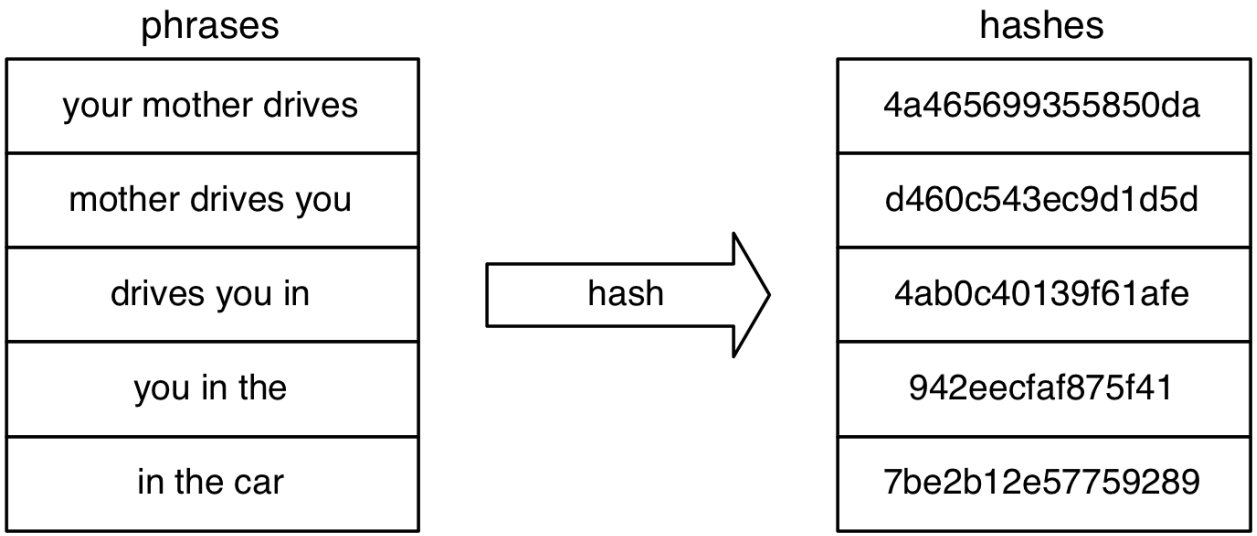
Шинглинг

"a bump on the log in the hole in the bottom of the sea"
"a frog on the bump on the log in the hole in the bottom of the sea"



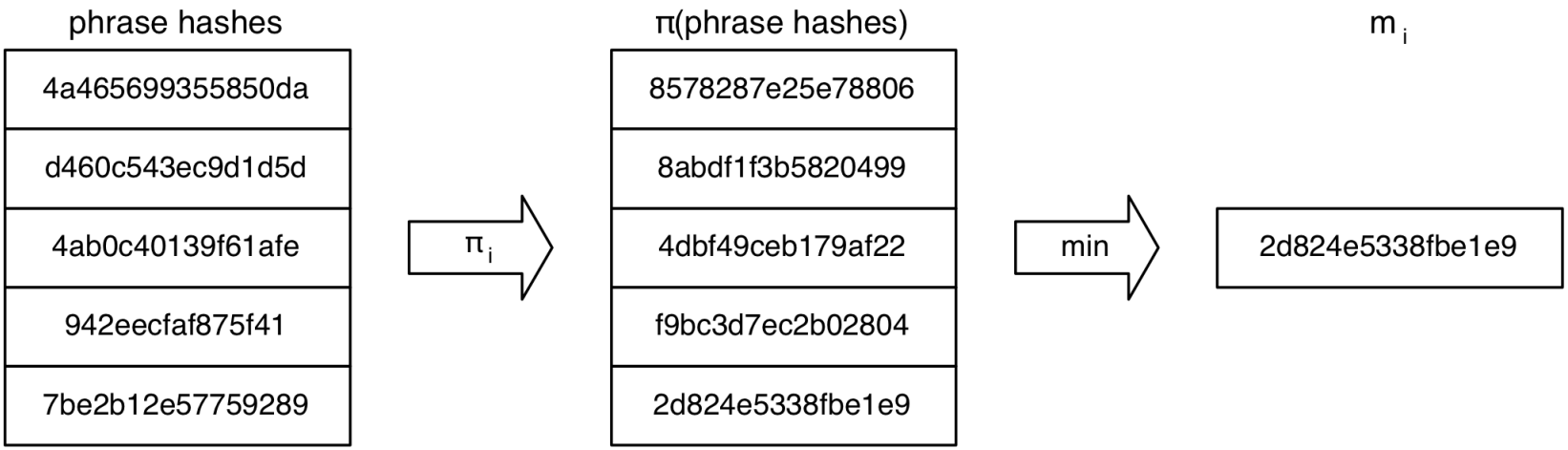
Хэширование

- Проблема шинглинга: приходится хранить документ несколько раз
- Шинглы из k слов: для предложения из n слов храним дополнительно порядка $O(nk)$ слов
- Частичное решение проблемы — хэширование шинглов



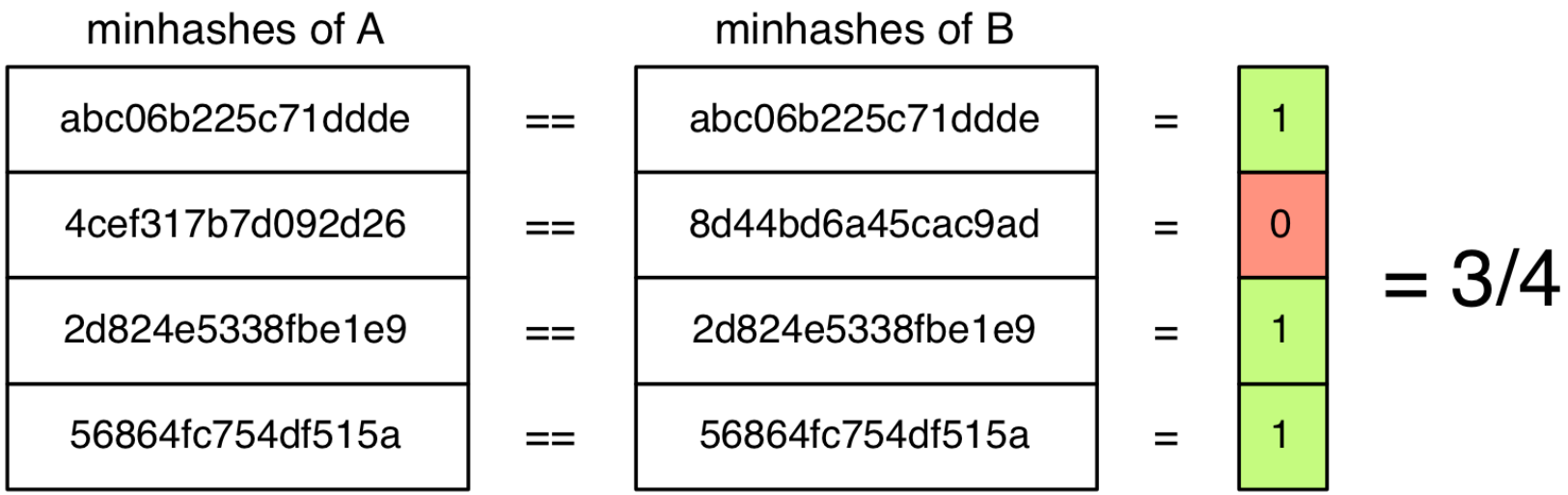
MinHash

- Проблема осталась: для предложения из n слов храним $O(n)$ хэшей (документы могут быть большими)
- Подход MinHash: преобразуем документ в набор хэшей фиксированного размера
- Используем набор из k рандомизированных хэш-функций
- Для каждой хэш-функции π_i $(1 \leqslant i \leqslant k)$ считаем хэши от хэшей шинглов всего документа. Находим минимум m_i



MinHash

- Сигнатура документа — упорядоченный список минимальных хэшей от m_0 до m_{k-1}.
- Метод MinHash позволяет аппроксимировать коэффициент Жаккара как долю совпавших минимальных хэшей

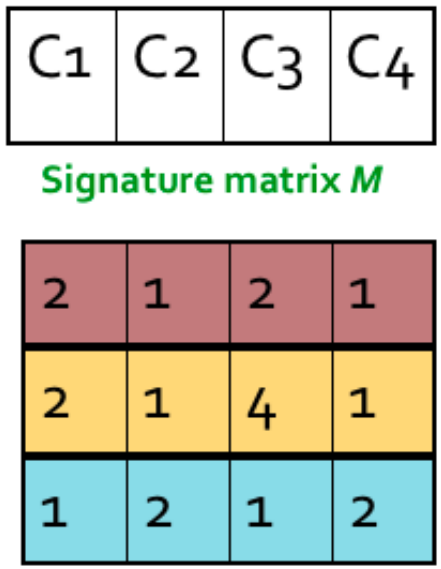


Хранение хэшей — $O(1)$, время сравнения двух документов — $O(1)$

Проблема: сигнатуры не так уж коротки

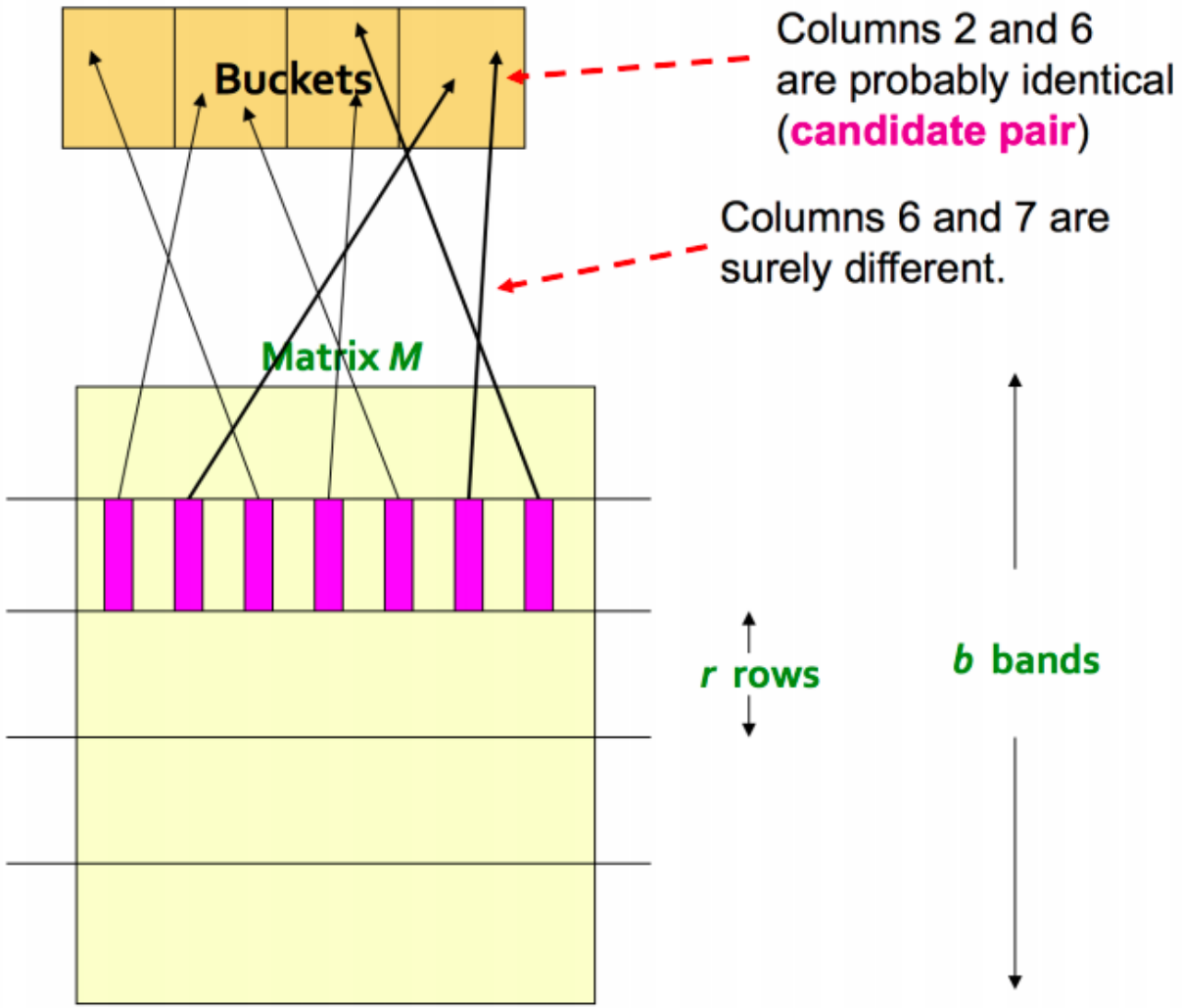
LSH: Locally-Sensitive Hashing

- Хотим найти все такие пары документов, чтобы коэффициент Жаккара между ними был не меньше s
- Ключевая идея: хэшировать все колонки матрицы сигнатур в очень большое количество «корзин» (buckets); документы, попавшие в одну корзину, — такие кандидаты



Столбцы — документы, строки - мин. хэши

LSH: Locally-Sensitive Hashing



$b(r)$ — параметры, которые надо настраивать

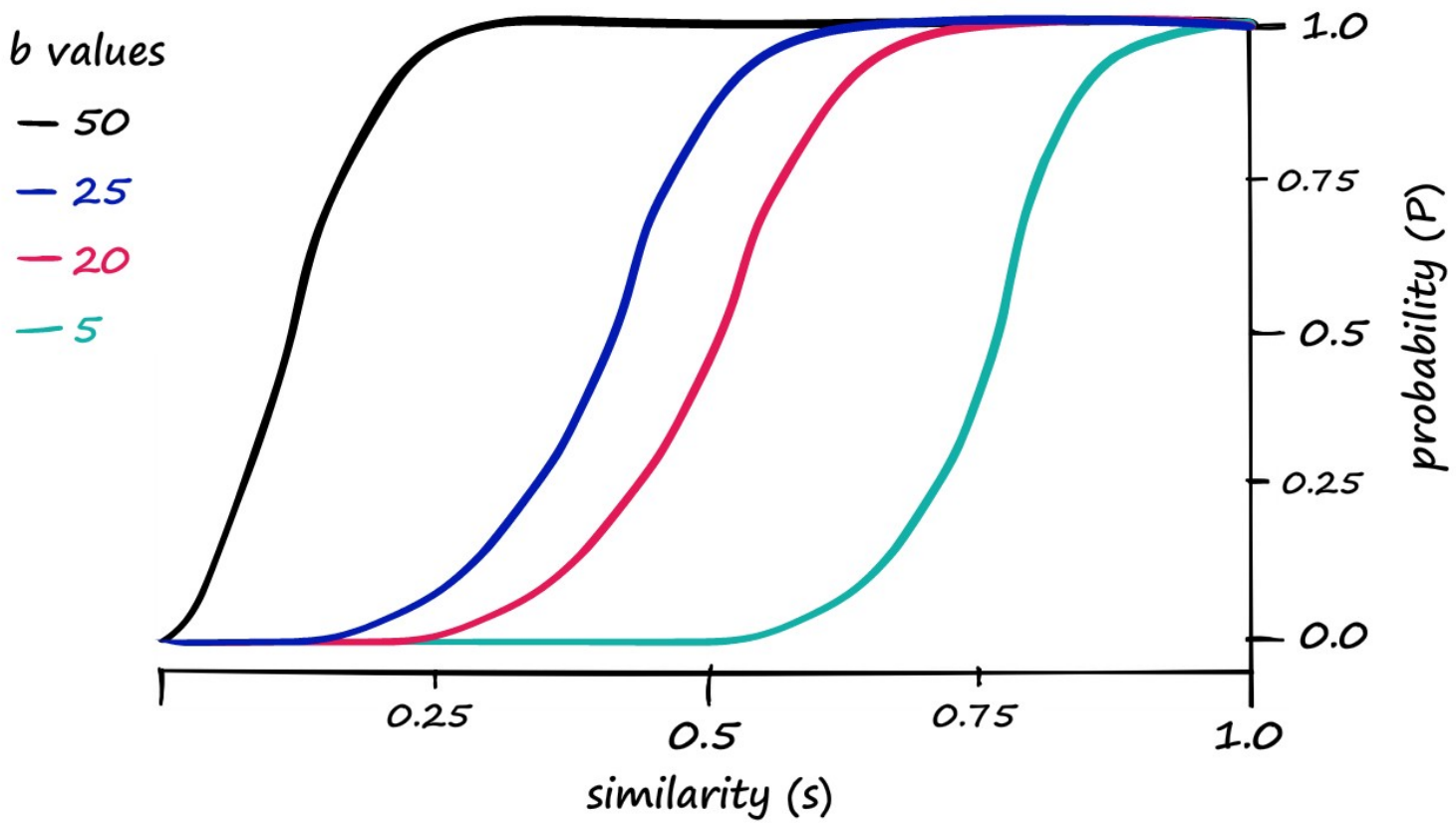
LSH: Locally-Sensitive Hashing

Вероятность того, что пара документов будет идентифицирована как дубликаты с учетом показателя сходства s, количества полос b и количества строк r в каждой полосе:

$$P = 1 - (1 - s^r)^b$$

Для 128 хэш-функций, $b = 16$, $s = 0.9$ получим p=0.99987745

LSH: Locally-Sensitive Hashing



$(r = 100/b)$

Сильный сдвиг влево увеличивает FP,
сильный сдвиг вправо увеличивает FN

Практическая реализация

Библиотеки

1. datasketch (https://github.com/ekzhu/datasketch)
2. simhash (https://github.com/1e0ng/simhash) — альтернатива MinHash
3. множество кастомных реализаций алгоритмов (Google)

MinHash token filter

edit

Uses the MinHash technique to produce a signature for a token stream. You can use MinHash signatures to estimate the similarity of documents. See Using the `min_hash` token filter for similarity search.

The `min_hash` filter performs the following operations on a token stream in order:

1. Hashes each token in the stream.
2. Assigns the hashes to buckets, keeping only the smallest hashes of each bucket.
3. Outputs the smallest hash from each bucket as a token stream.

This filter uses Lucene's MinHashFilter.

Configurable parameters

edit

`bucket_count`
(Optional, integer) Number of buckets to which hashes are assigned. Defaults to `512`.

`hash_count`
(Optional, integer) Number of ways to hash each token in the stream. Defaults to `1`.

`hash_set_size`
(Optional, integer) Number of hashes to keep from each bucket. Defaults to `1`.

Hashes are retained by ascending size, starting with the bucket's smallest hash first.

`with_rotation`
(Optional, Boolean) If `true`, the filter fills empty buckets with the value of the first non-empty bucket to its circular right if the `hash_set_size` is `1`. If the `bucket_count` argument is greater than `1`, this parameter defaults to `true`. Otherwise, this parameter defaults to `false`.

https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-minhash-tokenfilter.html
https://www.youtube.com/watch?v=X4H5ccw9RM4

- Сохраняем в таблице БД b хэшей
- Для поиска дубликатов делаем запрос вида

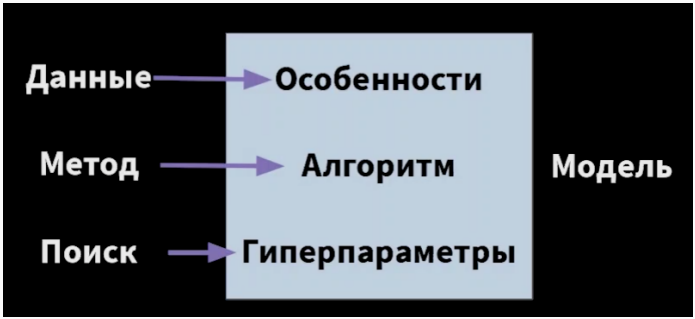
```
SELECT array_agg(doc_id) AS ids
FROM documents
GROUP BY hash_i
HAVING COUNT(*) > 1
```

Лекция 6.
Введение в Spark ML

Распределенные модели машинного обучения

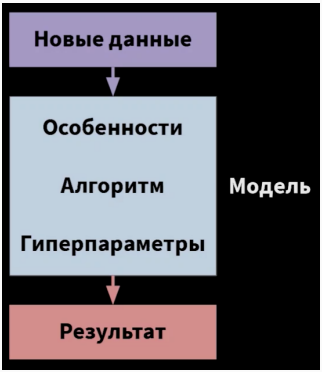
Модели машинного обучения

Любую модель машинного обучения можно представить как чёрный ящик, который состоит из трёх компонентов:



Модели машинного обучения

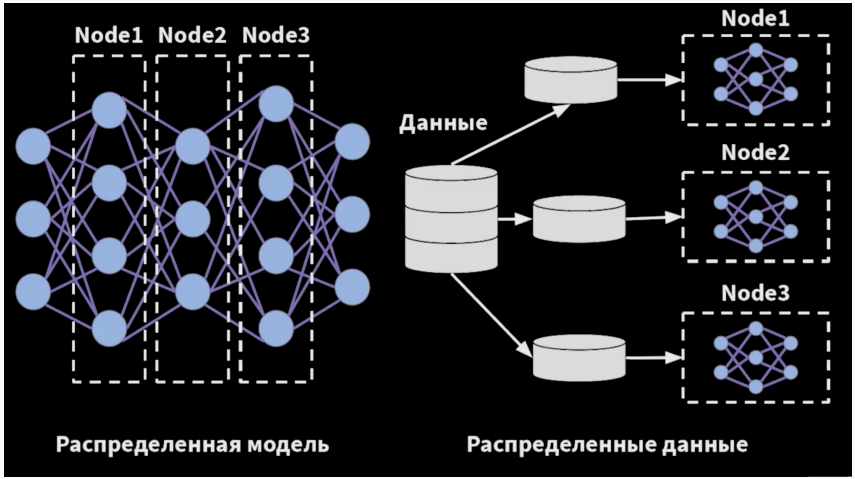
После того, как модель обучена, можно её применять на каких-то ранее неизвестных данных для того, чтобы получать новые результаты и в дальнейшем на них реагировать:



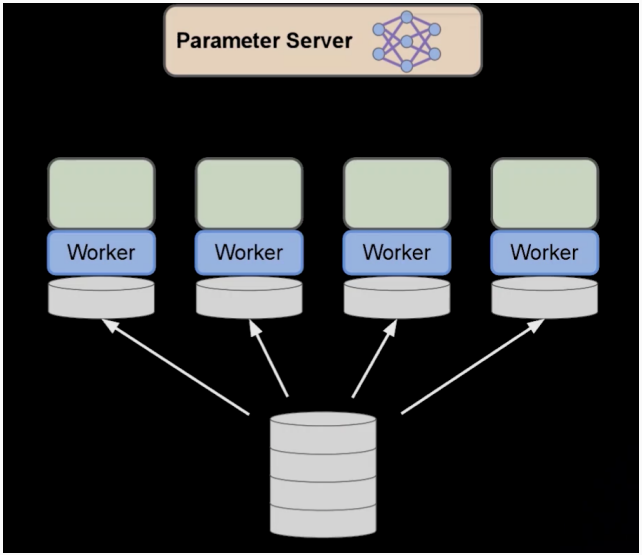
Абстрактный Pipeline машинного обучения

Протухание модели — ситуация, когда модель уже не работает эффективно на тех данных, которые поступают. Имеются какие-то достаточно сильные изменения в данных, которые делают модель более неприспособленной к решению поставленных задач

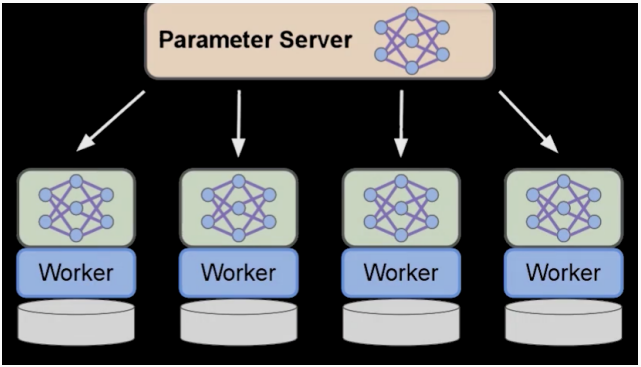
Распределенное машинное обучение



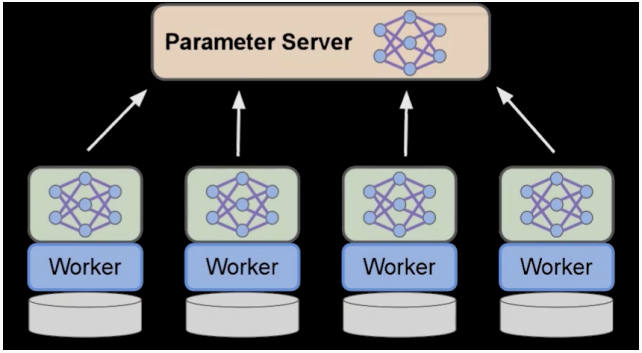
Распределенные данные



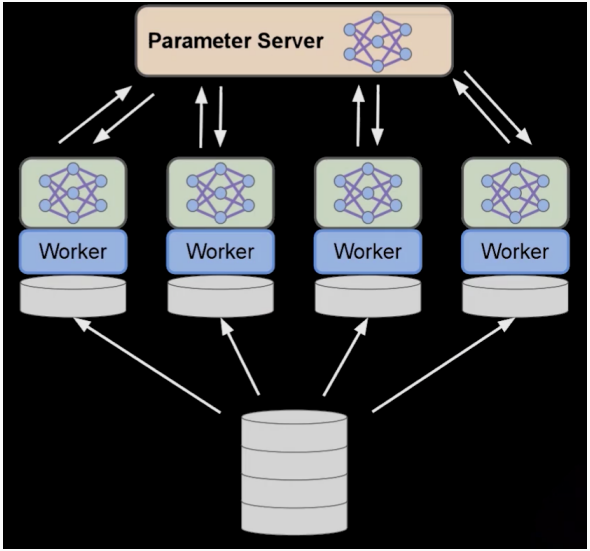
Распределенные данные



Распределенные данные



Распределенные данные



Компоненты и модели Spark ML

Spark ML

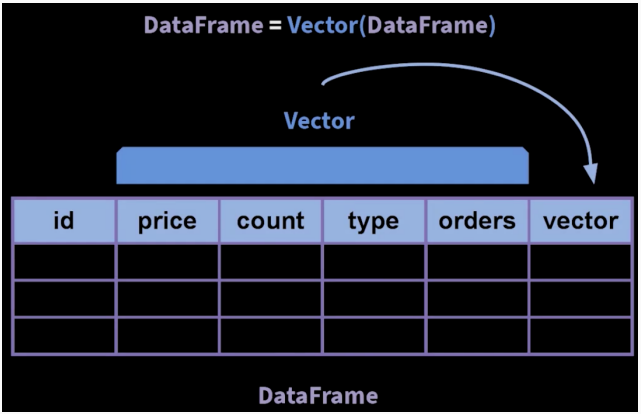
Spark ML (MLlib) — библиотека машинного обучения Spark. Её цель — сделать практическое машинное обучение масштабируемым и простым

- Spark < 2.0
- MLlib — RDD
- Spark $\geqslant 2.0$
- ML — DataFrame

Компоненты Spark ML

- **DataFrame** — ML API использует DataFrame из Spark SQL в качестве набора данных ML
- **Transformer** — это алгоритм, который может преобразовывать один DataFrame в другой DataFrame
- **Estimator** — это алгоритм, который может выполнить преобразование из DataFrame в Transformer
- **Pipeline** — конвейер, объединяющий любое количество Transformer'ов и Estimator'ов, для создания процесса машинного обучения
- **Parameter** — общий интерфейса для Transformer и Estimator

Spark ML: DataFrame и Vector



Векторизация преобразует некоторый DataFrame в новый, например, добавляя туда некоторую новую колонку, где уже хранится результат сформированного вектора по тем данным, которые определили

Spark ML: Transformer

Transformer — это алгоритм, который может преобразовывать один DataFrame в другой DataFrame

```
new_df = Transformer.transform(df)
```

Цели применения трансформеров:

- **feature** — использование для создания новых признаков из переданного набора данных
- **model** — использование для прогнозирования метки для каждого вектора объекта

Spark ML: Estimator

Estimator — это алгоритм, который может выполнить преобразование из DataFrame в Transformer

```
transformer = Estimator.fit(Df)
```

В самом Spark ML достаточно большое количество Estimator'ов

Одним из таких Estimator'ов является, например, логистическая регрессия

```
log_reg_model = LogisticRegression.fit(df)
```

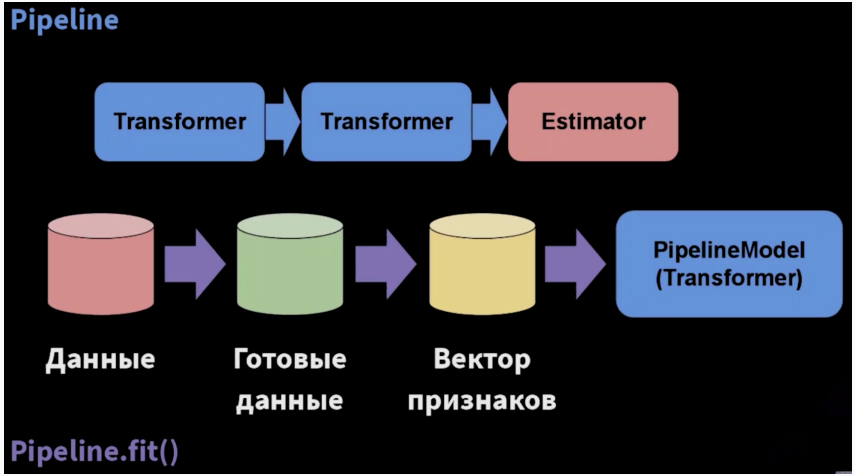
Spark ML: Pipeline

Pipeline — конвейер, объединяющий любое количество Transformer'ов и Estimator'ов, для создания процесса машинного обучения

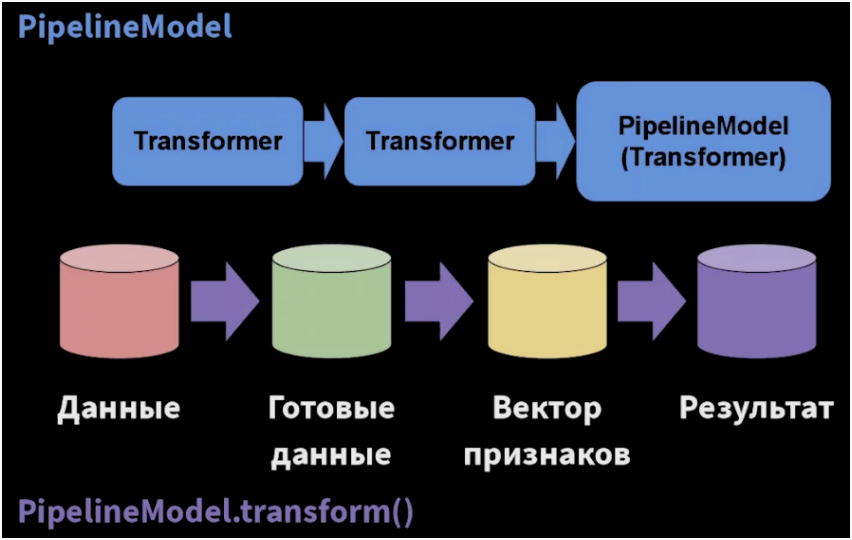
- Задаётся в виде последовательности этапов, и каждый этап — это Transformer или Estimator
- Любой Transformer, созданный в результате работы Estimator'а, автоматически становится частью Pipeline
- Все компоненты являются Stateless, т.е. не хранят состояние, т.к. Spark сильно завязан на «ленивые вычисления»

Обучение моделей в Spark ML

Spark ML: обучение



Spark ML: обучение



Spark ML: модели

Spark ML поддерживает достаточно широкий класс моделей, которые можно применять. Поддерживаемые классы моделей:

- Регрессия
- Кластеризация
- Классификация
- Деревья решений
- Ансамбли деревьев (случайный лес, бустинг)
- Глубокое обучение

Spark ML: глубокое обучение

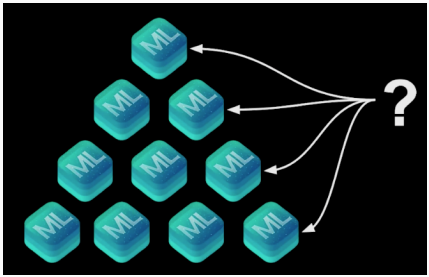
Среди интеграций Spark ML для глубокого обучения стоит отметить 4 фреймворка:

- BigDL
- TensorFlowOnSpark
- DeepLearning4j
- Deep Learning Pipelines (от DataBricks)

Поставка моделей

Управление моделями

При работе с моделями (распределёнными или нет), так или иначе придётся столкнуться с некоторыми проблемами, которые будут требовать управления моделями. Например, может быть достаточно много моделей и необходимо знать, какая из этих моделей лучше, какая наиболее оптимальна, а какая была результатом какого-то эксперимента, который провалился

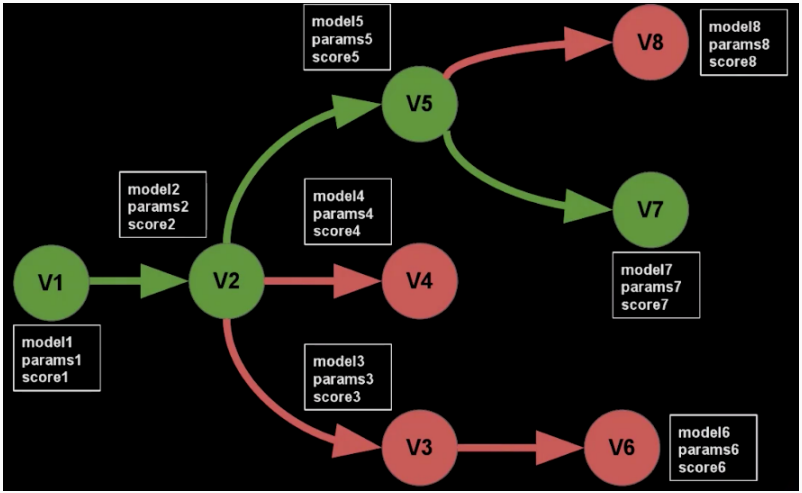


Управление моделями

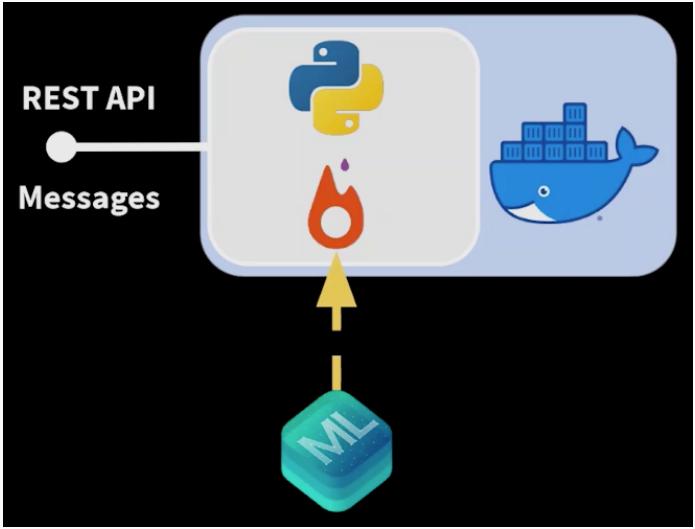
Model	Version	Author	Train Dataset	Score
dynamic-price	1.0.1	John.Doe@gmail.com	dataset_v1	0.865434
dynamic-price	1.2.0-SNAPSHOT	John.Doe@gmail.com	dataset_v1	0.923544
dynamic-price	1.1.6	John.Doe@gmail.com	dataset_v2	0.883454
product-classifier	5.2.34	Arthur.Bron@gmail.com	dataset_v10	0.905345
product-classifier	4.3.4	Kate.Tash@gmail.com	dataset_v6	0.875445
product-classifier	2.36.4	Kate.Tash@gmail.com	dataset_v3	0.834345
product-classifier	1.3.6	Arthur.Bron@gmail.com	dataset_v1	0.769654

Sacred

Управление моделями



Поставка моделей: микросервис



Поставка моделей: микросервис

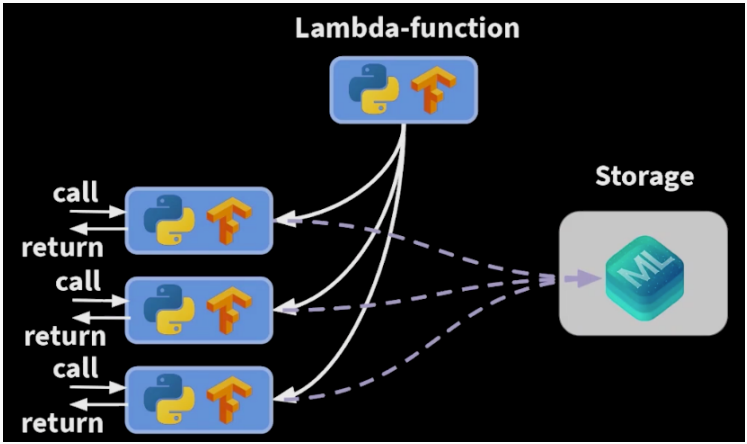
Плюсы:

- Масштабируемое решение: можно увеличивать кол-во сервисов в окружении, чтобы справляться с более высокой нагрузкой
- Возможность переиспользования, т.е. в случае необходимости достаточно легко подключиться к какому-то реализованному REST API для того, чтобы было можно быстро использовать модель
- Проще интеграция с Backend, если это необходимо

Минусы:

- Не для сложных моделей, поскольку сложная архитектура будет требовать больших вычислительных мощностей, что может пагубно сказаться на сервисе
- Такой подход требует постоянную активность от микросервиса, поскольку неизвестно, когда совершится обращение для выполнения модели

Поставка моделей: лямбда-функция



Поставка моделей: лямбда-функция

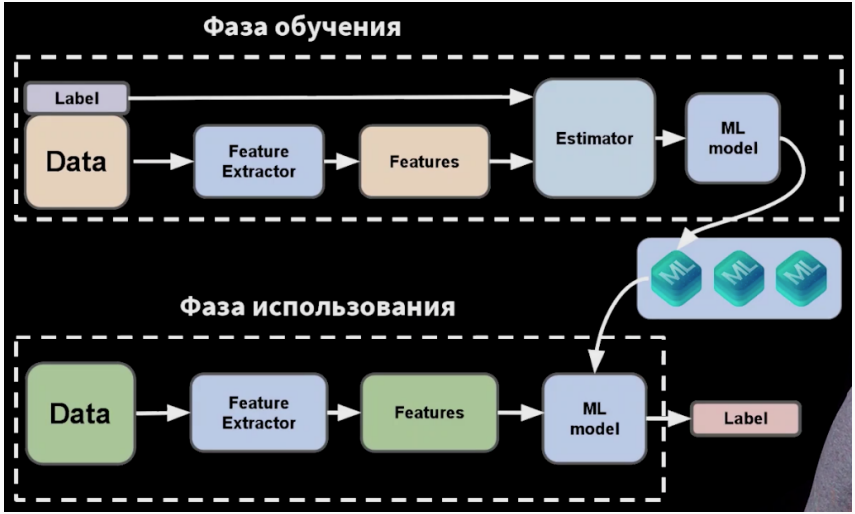
Плюсы:

- Масштабируемое решение. Поскольку функции не хранят состояние, то можно создавать столько экземпляров функции, сколько нужно для организации параллельных вычислений результатов модели.
- Возможность переиспользования, т.к. любое приложение, находящееся в облаке, может спокойно выполнять функцию и получать результат
- Можно добиться высокой производительности вычислений, за счёт комбинирования функций и параллельного их исполнения.

Минусы:

- Зависимость от облачных решений, которые используются
- Не для сложных моделей: не очень эффективен для сложных моделей

Поставка моделей: распределенная модель



Database Management Systems

ElasticSearch

Search in a nutshell

1. Take a query string
2. Match it against a document collection
   - Perform full text search, handle synonyms
3. Calculate a set of relevant results
   - Score documents by *relevance*
4. Display a sorted list

ElasticSearch

- Real-time distributed search and analytics engine
- Scalable and efficient data exploration
  - Full-text search
    - Highlighted search snippets, search as you type, did-you-mean, more-like-this
  - Structured search
  - Analytics
    - Real-time query answers on mixed data types (e.g., text and structured data)

- Popular examples
  - *GitHub* uses ElasticSearch to query 130+B lines of code.
  - *Wikipedia* provides full-text search with highlighted snippets
  - *StackOverflow* combines both full-text and geolocation queries for recommending related questions and answers

ElasticSearch

- Document-oriented (JSON) search engine
  - Complex data structures that may contain dates, geo locations, text, other objects, arrays of values
- Built on Lucene search engine library
  - Documents are indexed and searchable
- Highly available and horizontally scalable

ElasticSearch strong points

- The **horizontal scaling** capabilities of ElasticSearch make it suitable for a great variety of applications
- Its **RESTful API** allows programmers to write most of the operations in any programming language
- **JSON-based** interactions make it machine- and human-friendly
- Any modification to stored (indexed) documents is recorded in transaction logs that are replicated in multiple nodes to **avoid data loss**

Data representation

Parallel to relational representation

ElasticSearch: `field`  **SQL:** `column`

- Data is stored in *named entries* belonging to a variety of data types
- SQL calls such an entry a `column` while in ElasticSearch it is called `field`

In ElasticSearch (similarly to other NoSQL databases) a field can contain *multiple* values of the same type (list of values)

Parallel to relational representation

ElasticSearch: `document` ⟺ **SQL:** `row`

- Data objects are represented as rows (SQL) or documents (ElasticSearch)
- Columns and fields are part of a row (SQL) or a document (ElasticSearch)

Row format in SQL is **strict** and follows a predefined schema.
Documents are more **flexible** and can contain a variety of fields (they do not follow a strict schema)

Parallel to relational representation

ElasticSearch: `index` ⟺ **SQL:** `table`

An `index` is like a `table` in a relational database

ElasticSearch: `cluster` ⟺ **SQL:** `database`

In Elasticsearch the indices are grouped in a cluster.

Recap

ES	cluster	index	document	field
SQL	database	table	row	column

The *index* term

The *index* term is overloaded

- *(noun)* An index stores a collection of documents
- *(verb)* To index a document means to insert a document in an index
  - If the document already exists, it is replaced

The *index* term

The *index* term is overloaded

- *(inverted index)* Additional structure that accelerates data retrieval
  - Similar to a traditional relational index
  - *Every field* in a document is indexed in ES
    - All inverted indices are used during search
  - Non-indexed fields (if any) are not searchable

Document

- A document is the top-level (root) object serialized into JSON
- It is uniquely identified by the pair
  - *Index:* where the document (object) is stored
  - *Id:* the identifier of the document
    - Can be provided or uniquely generated by ES

Querying and searching

Searching

Search options

- Structured query on specific fields, possibly sorted
  - similar to SQL query
- Full-text query
  - finds all documents matching the search keywords
  - returns them sorted by *relevance*
- Combination of the two

Key concepts

- Mapping
  - How the data in each field is interpreted
  - ES dinamically generates a mapping by "guessing" data types
    - E.g. it may recognize a `date` type
- Analysis
  - How full text is processed to make it searchable
- Query DSL (Domain Specific Language)
  - Elastic Search query language

Exact values

- Traditional data types (e.g., integer, float, date, but also string)
  - A value must *match exactly* the query
    - Similar to SQL
  - Examples: date, user ID, but also exact strings such as username or email address
- Question answered

  "Does this document match the query?"

Full text

- Textual data, usually written in some human language
  - Typical search is within the textual field
  - Examples: text of a tweet, body of an email
- Question answered

  "*How well* does this document match the query?"
  - Notion of *relevance* of a document for a query

Full text

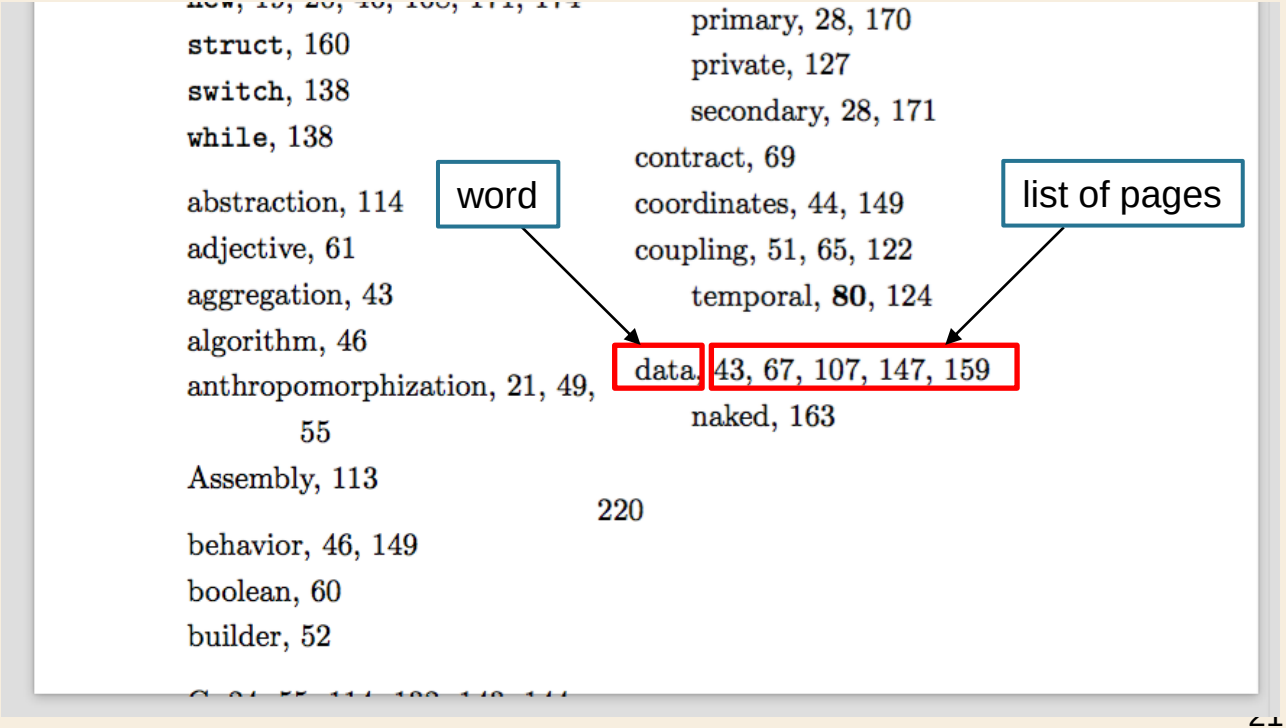
- For full text queries, it is also important understanding the intent
  - abbreviations
    - e.g., USA vs United States of America
  - singulars/plurals, verb conjugation
    - e.g., cat vs cats, does vs did vs to do
  - synonyms
    - e.g., game vs competition
  - order of words building a context
    - e.g., fox news hunting vs fox hunting news

Indexing full text

- ES builds an *inverted index* on every full-text field
  - Designed for fast full-text search
- Inverted index
  - List of all the unique words that appear in any document in the collection
  - For each word: list of the documents in which it appears

Indexing full text

- Similar to analytic indices in books



Analysis

1. Tokenization of a block of text into individual terms suitable for an inverted index
2. Normalization into a standard form to improve their retrieval (or recall) in queries
   - Terms are not exactly the same, but similar enough to be still relevant
     - Lowercase vs uppercase
     - Stemming, i.e., reduction to the root form
       - e.g., cats vs cat
     - Synonym management

- For searching, both indexed text and query string must be analyzed in the same way

Analyzer

- Analyzers provide the following functions
  - Character filters: cleans the string before tokenization
    - e.g., converts & to and
  - Tokenizers: split the string into individual words
    - e.g., by considering white spaces or punctuation as separators
  - Token filters: operate on single terms
    - change terms (e.g., to lowercase)
    - remove (e.g., stopwords)
    - add terms (e.g., synonyms)
- Built-in analyzers are provided by ES

Filter vs Query

- A *filter* is used for fields containing exact values
  - It provides a boolean *matches/does not match* answer for every document
- A *query* is (typically) used for full-text search
  - It also asks the question: How *well* does this document match?
    - calculates how *relevant* each document is to the query
    - assigns it a relevance _score, which is later used to sort matching documents by relevance
  - The concept of relevance is well suited to full-text search
    - there is seldom a completely "correct" answer

Filter vs Query

- Filter execution is more efficient
- Filters are typically used to reduce the number of documents that have to be examined by a query
- Hint
  - use query clauses for *full-text* search or for any condition that should affect the *relevance score*
  - use filter clauses for everything else

ElasticSearch Query

- Expressed in Query DSL
- Submitted as formatted JSON in the body of an HTTP request
- Example: empty query
  - returns all documents in all indices

```
POST /_search
{}
```

- Search on a specific index

```
POST index1/_search
{}
```

Query DSL

- The top level field in an ElasticSearch query is always **`query`**
  - the query type is specified one level below

```
POST departments/_search
{
   "query": {
        "match" : { "name" : "John" }
   }
}
```

Query DSL

- The top level field in an ElasticSearch query is always **`query`**
  - the query type is specified one level below

```
POST departments/_search
{
    "query": {
        "match" : { "name" : "John" }
    }
}
```

  - the query operates on the department index
    - specified in the URI
  - it performs the **`search`** operation

Query DSL

- Query

  Find all the documents in the department index that have a field name containing the term John in it.

- Query type: match query

```
POST departments/_search
{
   “query”: {
      “match” : { “name” : ”John” }
   }
}
```

Compound queries

Compound queries are complex queries specifying multiple matching criteria

```
POST departments/_search
{
   "query": {
      "bool": {
         "should": [
            {"match": {"name": "John"}},
            {"match": {"name": "Mark"}}
         ],
            "minimum_should_match":1,
         "must":{
            {"match": {"title": "developer"}}
         }
         "must_not":{
            {"match": {“lastname": “Smith"}}
         }
      }
   }
}
```

bool specifies the compound query

Compound queries

```
POST departments/_search
{
   "query": {
      "bool": {
         "should": [
            {"match": {"name": "John"}},
            {"match": {"name": "Mark"}}
         ],
            "minimum_should_match":1,
         "must":{
            {"match": {"title": "developer"}}
         }
         "must_not":{
            {"match": {"lastname": "Smith"}}
         }
      }
   }
}
```

should specifies the OR condition

must corresponds to the AND condition

must_not specifies the NOT condition

The match query

- Can be used for both full-text and exact queries
- On a full-text field
  - it analyzes the query string with the correct analyzer before executing the search
  - it returns a relevance score `_score` for the search
- On an exact field or a `not_analyzed` string field
  - it searches the exact value
  - it returns a relevance score `_score` of 1
- When a bool query is specified on full-text fields
  - It combines the `_score` from each must or should clause that matches

Query DSL

- It is possible to specify multiple indices to be searched in the query URI

```
POST rooms,students/_search {...}
```

- When a number of documents can be returned as query result, by default the top 10 relevant results are returned

Earlier versions of ElasticSearch include index types that have been deprecated since version `7.0`.

Data modifications

Insert

Insert of a new single document is performed by means of a POST operation

- Name of index
- JSON document to be indexed

```
POST /index_name/<id>
{
    JSON document
}
```

`index_name`: name of the index in which the document should be inserted

`<id>`: optional parameter that associate the document with a specific identifier

- If the ID is not provided, ElasticSearch creates a unique identifier for the document (e.g., `W0tpsmIBdwcYyG50zbta`)

Update

- Documents in ES are immutable
  - To update a document, it is reindexed
- When a document is updated, ES
  1. Retrieves the old document
  2. Modifies the retrieved copy
  3. Deletes the old document
  4. Indexes the new document (the copy)
- Internally, the old version of the document is not deleted immediately
  - It is not accessible
  - Deleted documents are cleaned in background

Update

The update of a document is performed using a POST request

- Name of the index
- Unique ID of the document
- the fields to be updated and the associated new values

```
PUT index_name/123/_update
{
  "color" : "red",
}
```

This update request modifies the document with ID=123 setting the value of the "color" field to "red".

Delete

The deletion of a document is performed using a DELETE request

- Name of the index
- Unique ID of the document

```
DELETE index_name/id
```

This operation removes a JSON document from the specified index

- Document removal is not immediate

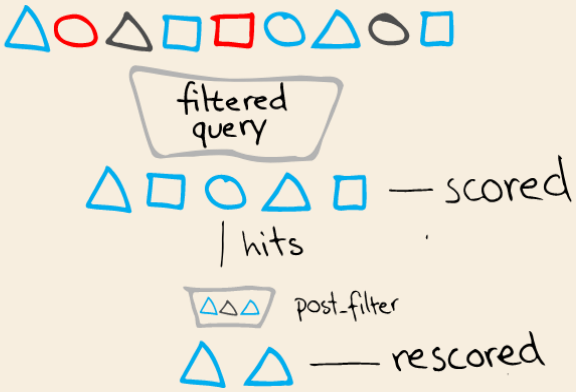
Results scoring

Relevance

- In ElasticSearch, relevance is represented by a value
  - floating-point number
  - computed for each document matching the query
  - stored as **`_score`** for each document in the search result
  - higher **`_score`** values correspond to more relevant documents
- Sorting by relevance is performed by considering the **`_score`** variable
  - by default, documents in a query result are sorted by descending value of the **`_score`** field

ElasticSearch Scoring

1. Compute matching results for the query
   - Compute relevance score for all documents in the query result
2. Select top relevance documents (hits)
   - Default is 10 hits (documents)
3. (optional) Rescore documents
   - more computationally expensive algorithm



Relevance score computation

- Need to compute the similarity between
  - The query
  - Each document
- Each document may contain a (different) subset of the query terms

Relevance score computation

1. Select documents matching the query
   - Boolean model
   - Fast computation
2. Evaluate the importance (weight) of each term in a document with respect to the query
   - Term importance evaluated with TF/IDF (Term Frequency/Inverse Document Frequency) score
   - Document and query are represented in vector form (Vector Space Model)
3. Evaluate the similarity of the vector representation of the query and the document

TF/IDF scoring function

The TF/IDF scoring function takes into account

Term frequency

- How often does the term appear in the field? The more often, the *more* relevant.

Inverse document frequency

- How often does each term appear in the index? The more often, the *less* relevant.

Field-length norm

- How long is the field? The longer it is, the less likely it is that words in the field will be relevant.

Term frequency

- Term frequency is defined by

$$\text{Tf(t in d)} = \sqrt{frequency}$$

  - frequency is the number of times the term t appears in document d

Inverse document frequency

- Inverse document frequency is defined by

$$Idf(t) = 1 + \log (numDocs/(docFreq + 1))$$

  - numDocs is the number of documents in the index
  - docFreq is the number of documents that contain the term

Field-length norm

- Field-length norm is defined by

$$\text{norm(d)} = 1 / \sqrt{numTerms}$$

  - numTerms is the number of terms in the field

TF/IDF scoring function

- The scoring function is based on a combination of the three factors
  - Term frequency
  - Inverse document frequency
  - Field-length norm
- They are calculated and stored at index time
- They are used to calculate the weight of a single term in a document
  - Other methods can be used
- Queries usually contain more than one term
  - Need a way to combine multiple terms

Vector Space Model

- It represents both query and document as (term) vectors
- It provides a way to compare a multi-term query against a document
- A query (or document) is represented as a vector
  - The vector size is the number of terms in the query
  - Each vector element is the weight of one term, calculated with TF/IDF scoring
- Vectors can be compared by measuring the angle between them

Measuring similarity

- Vectors can be compared by measuring the angle between them
  - Cosine similarity
- The angle between a document vector and a query vector is used to compute the similarity between a document and a query
  - It assigns to the document its relevance score for the query

VSM Example

Consider the query
- happy hippopotamus

Considering the TF-IDF heuristics
- happy is a common word and should have low weight (e.g., 2)
- hippopotamus is uncommon and should have a higher weight (e.g., 5)

The 2-dimensional vector associated to the query is

```
[2, 5]
```

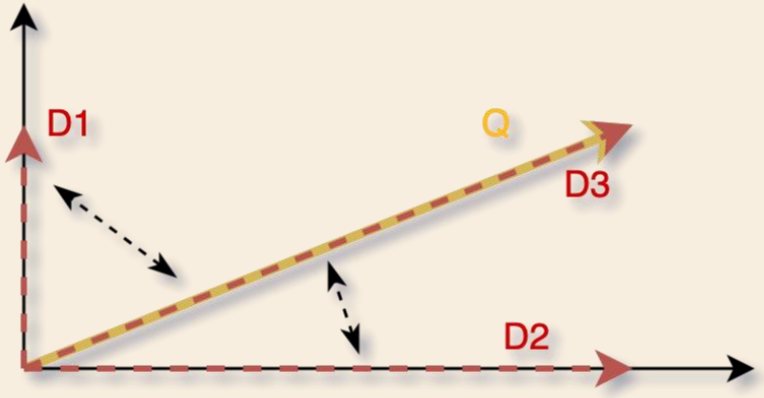
VSM Example

Consider the three documents:

- I am *happy* in summer.
- After Christmas I'm a *hippopotamus*.
- The *happy hippopotamus* helped Harry.

It is possible to create a vector for each document

- Document 1: `(happy, ____________) -> [2,0]`
- Document 2: `( ___ , hippopotamus) -> [0,5]`
- Document 3: `(happy, hippopotamus) -> [2,5]`

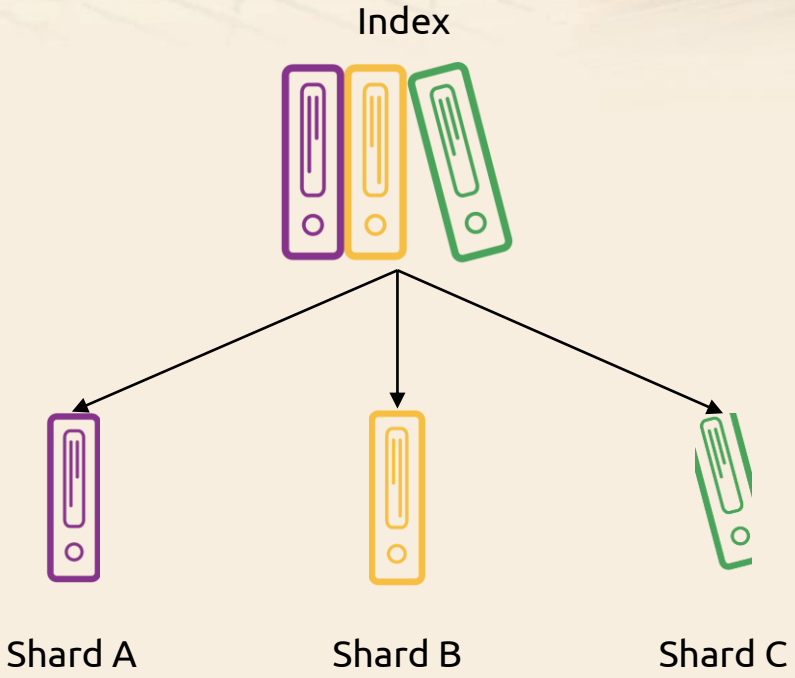


Horizontal scalability

Sharding

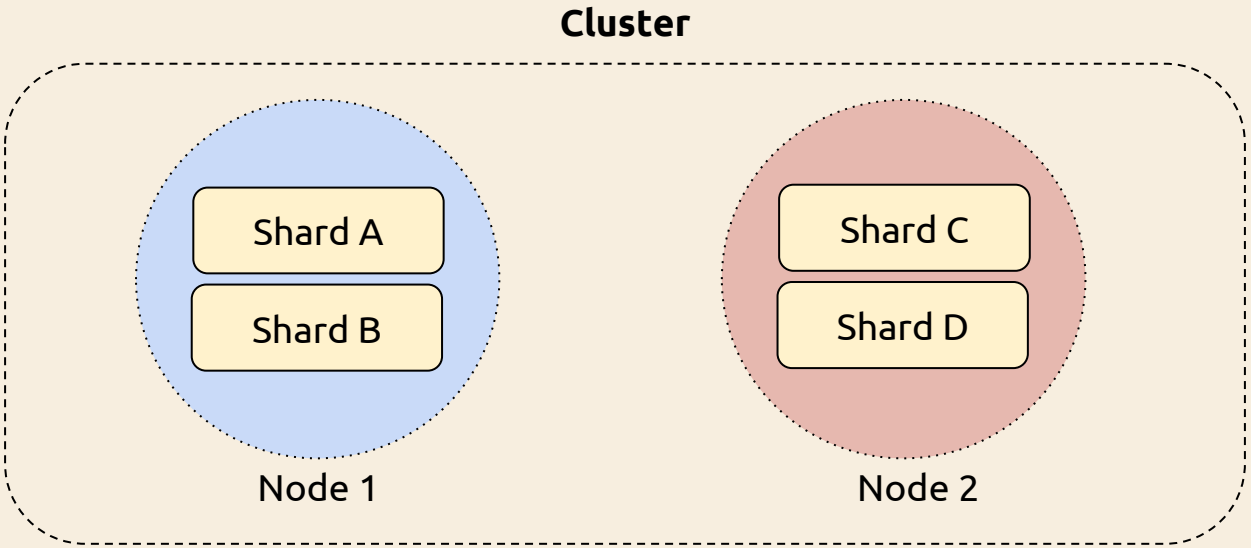
- Sharding is a technique to divide an index in smaller partitions
  - Each partition is a shard
- Each document belongs to a single shard
  - Each shard is an instance of a Lucene index
- When data is written to a shard
  1. It is periodically (every 1 second) written into a new immutable Lucene segment on disk
  2. It becomes available for querying
- Shards are the elementary units in which data is distributed on nodes in a cluster

Shards



Clusters

- A *cluster* is a collection of multiple machines (nodes in the cluster)
- Shards can be stored in any node within the cluster



Sharding

Why is sharding important?

- It allows splitting data in smaller chunks, and thereby scaling on large volumes of data
  - Data may be distributed across multiple nodes within a cluster
  - Shards can be stored on smaller disks
    - E.g., it is possible to store 1TB of data even without a single node with that disk capacity
- Operations can be distributed across multiple nodes and thereby parallelized
  - Performance is increased, because multiple machines can potentially work on the same query.
- Shards may be replicated on different nodes to increase availability

Document versioning

Optimistic concurrency control

- ElasticSearch uses optimistic concurrency control
  - It assumes that conflicts are unlikely to happen
  - However, if the underlying data has been modified between reading and writing, the update will fail
- Different from ACID transactions that need locking
- The process is "simple" for centralized data management

Modification propoagation

- ElasticSearch data may be distributed on different nodes in a cluster
  - Shards may be replicated on different nodes (replica shards)
- When documents are created, updated, or deleted, the new version of the document has to be replicated to other nodes in the cluster
  - The primary copy is always written first
  - The replication requests are sent in parallel and may arrive at their destination *out of sequence*

Document versions

- Elasticsearch needs a way of ensuring that an older version of a document never overwrites a newer version
  - Every document has a `_version` number that is incremented whenever a document is changed
- Elasticsearch uses this `_version` number to ensure that changes are applied in the correct order
  - if an older version of a document arrives after a new version, it can be ignored
  - the `_version` number is used to ensure that conflicting changes made by applications do not result in data loss

Document versions

- APIs that update or delete a document accept a version parameter
  - can apply optimistic concurrency control only when needed