Apache Spark Архитектура

После этого вы поймёте, почему Spark — это не "быстрый Pandas", а распределённая система со сложной внутренней архитектурой

КОНТЕКСТ

Где мы сейчас?

01

Лекция 1: Основы

Spark — это движок для больших данных (Volume, Velocity, Variety). Мы познакомились с концепцией распределённой обработки.

02

Лекция 2: Практика

Мы запустили PySpark в Docker за 3 минуты. Всё работает локально.

03

Сегодня: Внутренности

Теперь главный вопрос: Как устроена архитектура Spark? Как данные перемещаются? Почему иногда всё тормозит?

Главная идея: Spark — система

Spark — это распределённая система из нескольких "ролей", работающих вместе. Это не просто код, который вы импортируете — это целая инфраструктура.

Аналогия со строительной площадкой:

- **Вы, прораб → план работы**
- **Бригады → выполняют работу**
- **Подрядчик который выделил пять бригад, два экскаватора и склад для цемента → управляет ресурсами**

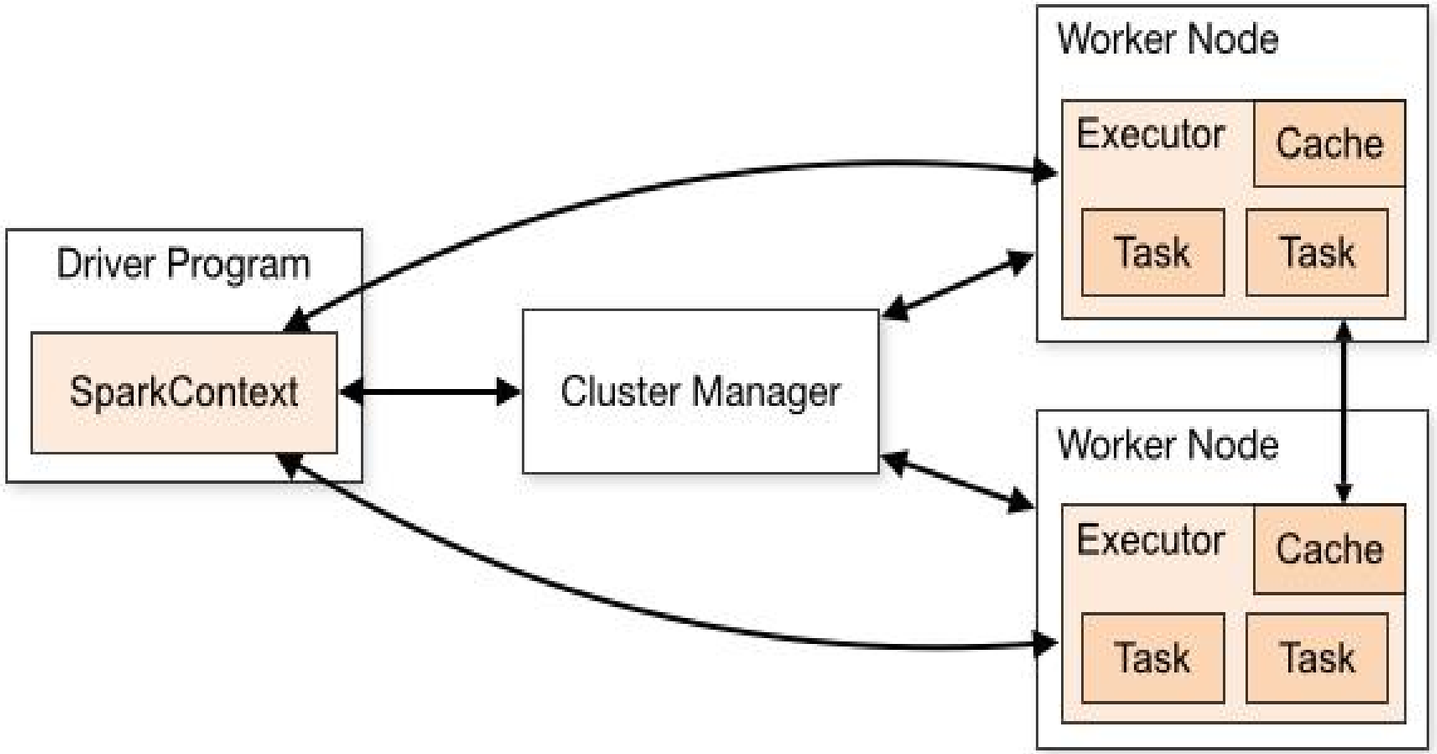
👉Это не метафора — это почти буквально то, как устроен Spark. Каждая роль имеет свою функцию.

Три ключевые роли в Spark

Роль	Что делает	Где живёт
Driver	Запускает программу, строит план выполнения, собирает результат	Это главный, кординирующий процесс вашей программы, где бы вы её ни запустили
Executor	Выполняет задачи, хранит данные в памяти, обрабатывает партиции	На worker-нодах (в кластере или в том же Docker)
Cluster Manager	Выделяет ресурсы: CPU, RAM, ноды	YARN, Kubernetes, Standalone — или эмуляция в local[*]

💡 Даже в Docker'е (local[*]) все три роли существуют — просто они работают на одной физической машине! Это важно понимать для отладки.

Архитектурная схема Spark



Как данные попадают в executors?

Когда вы читаете файл, Spark автоматически разбивает его на партиции, и каждая партиция обрабатывается отдельным executor'ом. Это фундаментальный принцип параллелизма.

1

Чтение файла

spark.read.csv("data.csv", header=True)

2

Разбиение на партиции

Spark автоматически создаёт 4 партиции (или больше, в зависимости от размера)

3

Распределение

4 партиции → 4 задачи → 4 executor'а работают параллельно

4

Параллельная обработка

Каждый executor обрабатывает свою часть данных независимо

Это и есть параллелизм. Партиция = минимальная единица работы. Больше партиций = больше параллелизма, но и больше overhead на координацию.

DAG и Stage’ы: как Spark превращает ваш код в план

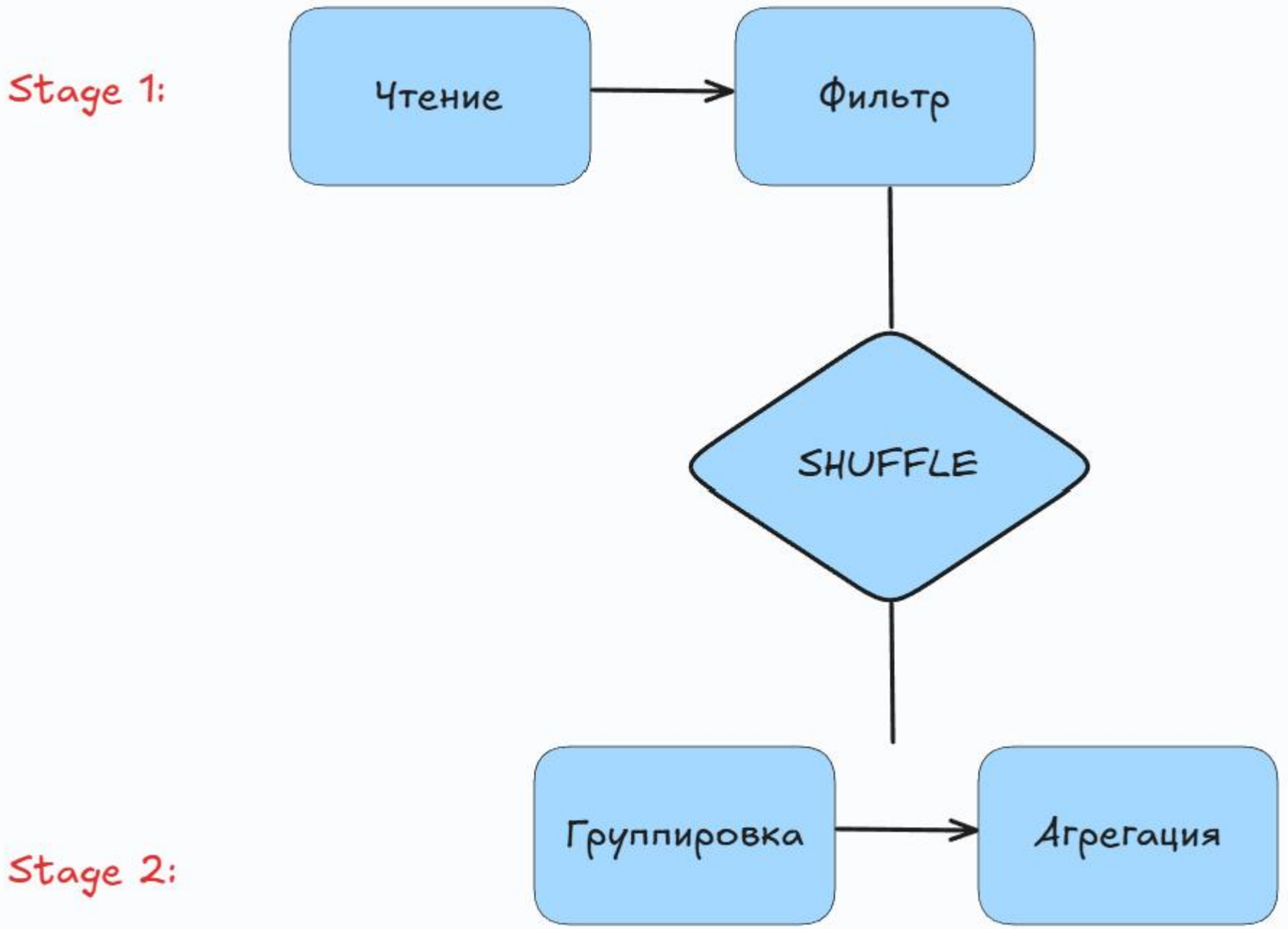
Когда вы пишете цепочку трансформаций:

```
df = spark.read.csv(“employess.csv”)

df.filter(col(“age”)>30)
.groupBy(“city”)
.agg(avg(“salary”))
```

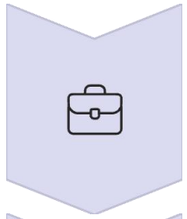
Spark не выполняет это построчно. Он сначала строит план выполнения, как повар сначала читает весь рецепт, прежде чем начать готовить.

Этот план называется DAG — Directed Acyclic Graph. Простыми словами: граф операций без циклов.



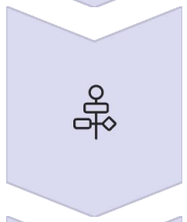
Job, Stage, Task — три уровня выполнения в Spark

Spark выполняет ваш код в три уровня иерархии. Понимание этой структуры критично для оптимизации и отладки.



Job - задача выполнения всего DAG

Запускается при action: count(), show(), write(). Один action = один Job. Это самый верхний уровень выполнения.



Stage - этап вычислений, без обмена данными между нодами

Участок без shuffle. Job разбивается на Stage'ы. Граница между Stage'ами — это shuffle.

Task - задача выполнения Stage на конкретном куске данных

Единица работы над одной партицией. Каждый Stage состоит из Task'ов. Количество Task'ов = количество партиций.

Почему shuffle — это "дорого"?

Shuffle — это момент истины в Spark.
Именно здесь чаще всего возникают проблемы с производительностью.

Запись на диск

Данные сериализуются и записываются на диск (spill) для передачи другим executor'ам.

Передача по сети

Данные передаются между машинами через сеть, это медленно, особенно для больших объёмов.

Создание новых партиций

После shuffle создаются новые партиции с перераспределёнными данными.

reduceByKey vs groupByKey

```
("Moscow", 100_000)
("SPB",     80_000)
("Moscow", 120_000)
("SPB",     90_000)
```

```
("Moscow", 220_000)
("SPB",    170_000)
```

groupByKey()

```
rdd = sc.parallelize([
    ("Moscow", 100_000),
    ("SPB",     80_000),
    ("Moscow", 120_000),
    ("SPB",     90_000)
])

# Группируем ВСЕ значения по ключу
grouped = rdd.groupByKey()  # → ("Moscow", [100_000, 120_000])

# Потом суммируем
result = grouped.mapValues(sum)
```

reduceByKey()

```
rdd = sc.parallelize([
    ("Moscow", 100_000),
    ("SPB",     80_000),
    ("Moscow", 120_000),
    ("SPB",     90_000)
])

# Сразу агрегируем
result = rdd.reduceByKey(lambda a, b: a + b)
```

Три уровня Spark API

Spark предоставляет три уровня API для работы с данными. Каждый уровень решает свои задачи и имеет свою область применения.

RDD API

Низкоуровневый, полный контроль (редко используется сегодня)

DataFrame / Dataset API

Высокоуровневый, оптимизированный, схема данных (основа 95% реальных проектов)

Structured Streaming

Потоковая обработка на основе DataFrame API (логи, события, транзакции в реальном времени)

КЛЮЧЕВЫЕ ВЫВОДЫ

Итог — 3 главные мысли

1

Spark — это система из ролей

Driver координирует, Executors работают, Cluster Manager выделяет ресурсы. Это не библиотека — это распределённая архитектура.

2

Данные разбиты на партиции

Партиция — это основа параллелизма. Одна партиция = одна задача = одно ядро CPU. Управление партициями критично для производительности.

3

Shuffle = дорого

Минимизируйте shuffle: фильтруйте данные до join, используйте reduceByKey вместо groupByKey, контролируйте размер партиций.

Подписывайтесь!

В следующей лекции:

SparkContext и SparkSession: как управлять Spark’ом программно

Если вам понравилось — заходите на

www.youtube.com/@marat_notes

https://vkvideo.ru/@club231048746

Все примеры кода и материалы доступны на GitHub:

github.com/MaratNotes/marat_notes

Перейти на YouTube

Открыть GitHub