SparkSession — управление Apache Spark

Откройте для себя мощный инструмент, который делает работу с большими данными эффективной

ОСНОВЫ

Что такое SparkSession?

SparkSession — это центральный объект для работы с Apache Spark, появившийся в версии 2.0. Он революционизировал способ взаимодействия с фреймворком, объединив возможности нескольких контекстов в одном удобном интерфейсе.

Это единая точка входа для всех операций: создания DataFrame, выполнения SQL-запросов, управления ресурсами кластера и конфигурации приложения.

Унификация

Заменяет SparkContext, SQLContext и HiveContext

Управление

Контроль над всеми аспектами работы Spark

АРХИТЕКТУРА

Архитектура Spark: драйвер и исполнители

Драйвер (Driver)

Главный процесс, который управляет приложением, планирует задачи и координирует работу всех исполнителей в кластере

Исполнители (Executors)

JVM-процессы, распределенные по узлам кластера, которые выполняют задачи и хранят данные в оперативной памяти для быстрого доступа

Роль SparkSession

Работает на стороне драйвера, управляя жизненным циклом приложения, распределением ресурсов и координацией всех компонентов

Архитектурная схема Spark

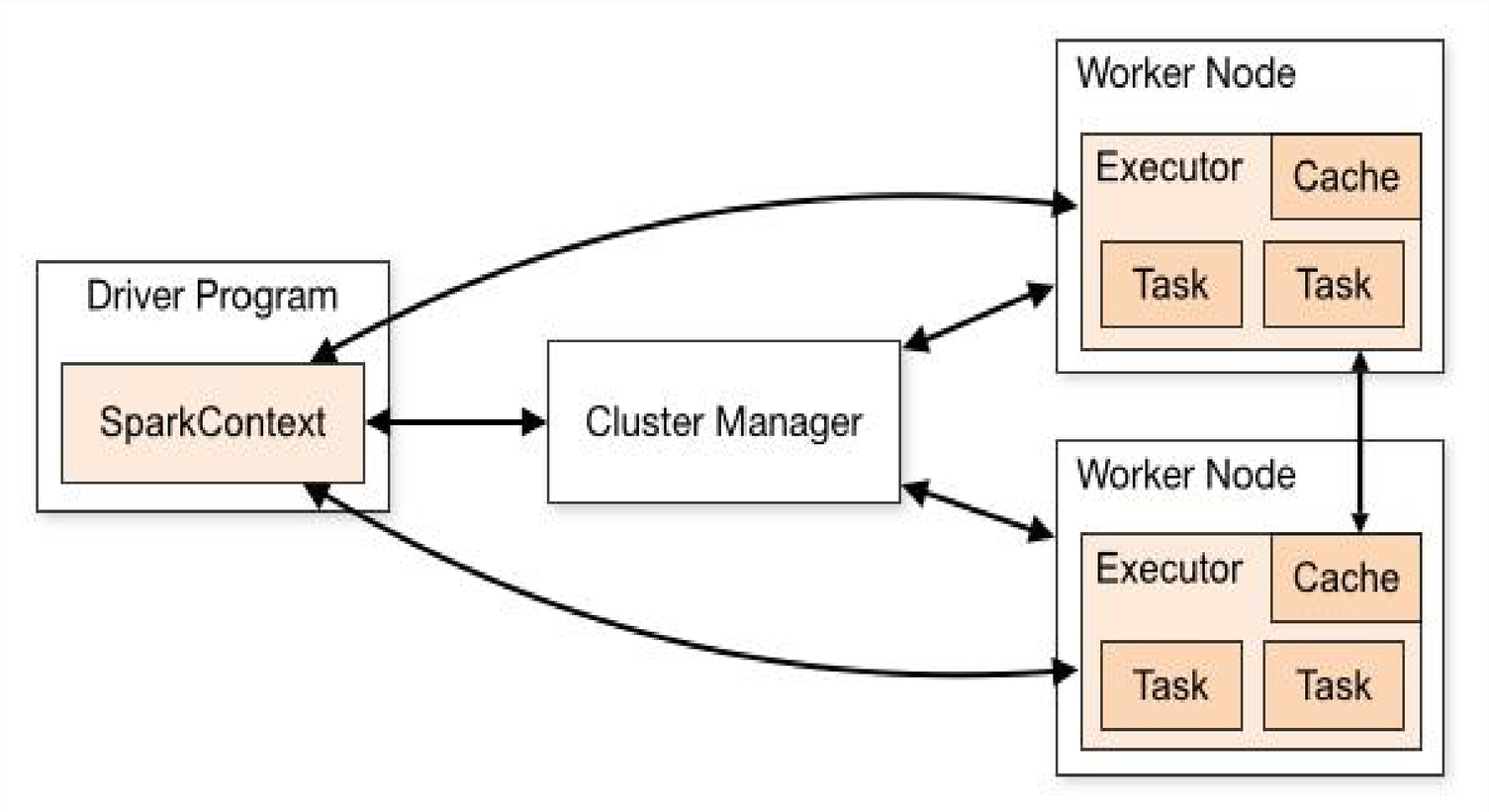

Схема взаимодействия компонентов

Драйвер и SparkSession

- Создание и управление сессией
- Планирование и распределение задач
- Мониторинг выполнения операций

Исполнители в кластере

- Параллельное выполнение задач
- Кэширование данных в памяти
- Отчетность о результатах драйверу

УПРАВЛЕНИЕ РЕСУРСАМИ

Как SparkSession управляет ресурсами кластера?

01

Инициализация сессии

Запускает драйвер и выделяет исполнителей через менеджер кластера (YARN, Mesos, Kubernetes)

02

Настройка параметров

Определяет количество ядер CPU, объем оперативной памяти для драйвера и каждого исполнителя

03

Контроль параллелизма

Оптимизирует распределение задач по исполнителям для максимально эффективного использования ресурсов

Важно: Правильная настройка ресурсов критична для производительности. Недостаточное выделение памяти может привести к ошибкам, а избыточное — к неэффективному использованию кластера.

ПРАКТИКА

Пример создания SparkSession на Python (PySpark)

```
from pyspark.sql import SparkSession

# Создание SparkSession
spark = SparkSession.builder \
    .appName("MyApp") \
    .master("local[*]") \
    .config("spark.sql.shuffle.partitions", "8") \
    .getOrCreate()

# Работа с данными
df = spark.read.csv("data.csv", header=True)
df.show()

# Остановка сессии
spark.stop()
```

Ключевые параметры

appName

Название приложения для идентификации в UI кластера

master

Режим работы: local, yarn, mesos или kubernetes

config

Настройки памяти и ядер для оптимальной производительности

Совет: Всегда вызывайте `spark.stop()` для корректного освобождения ресурсов кластера после завершения работы.

ОСНОВЫ

Что такое партиция?

Партиция = кусок данных для одного таска, выполняемого на одном ядре исполнителя

Свойства:

- Физически — файл или группа строк на диске/в памяти
- Логически — единица параллелизма (1 партиция → 1 таск → 1 ядро)
- Независимо — обрабатываются параллельно без блокировок
- Неизменяемо внутри стадии — не разбивается «на лету»
- Изменяемо между стадиями — AQE может объединить после шаффла

Два типа:

- Партиции чтения — при spark.read.csv(): **maxPartitionBytes**
- Партиции шаффла — при groupBy()/join(): **shuffle.partitions**

ПРОИЗВОДИТЕЛЬНОСТЬ

Оптимальный размер партиции

Размер партиции	Проблема	Последствие
< 1 МБ	Оверхед на создание таска, сериализацию, планирование	2000 тасков × 0.5 мс = 1 сек оверхеда на 140 МБ данных
1–10 МБ	Приемлемо для локального режима	Небольшой оверхед, но допустимо
10–100 МБ	ОПТИМАЛЬНО	Минимум оверхеда + хороший параллелизм
100–500 МБ	Риск дисбаланса при косых данных	Одно ядро работает дольше: простой остальных
> 500 МБ	Бутылочное горлышко	Одно ядро грузит гигабайт: падение скорости в 5–10×

ПРОИЗВОДИТЕЛЬНОСТЬ

Почему число партиций влияет на скорость?

8 партиций	2000 партиций
Оптимально (8 партиций для 4-ядерного ноутбука)	Слишком много партиций (2000 для 140 МБ)
8 тасков × 0.5 мс = 4 мс оверхеда	2000 тасков × 0.5 мс на создание = 1000 мс оверхеда
Каждый таск читает 17.5 МБ — эффективное использование диска и кэша	Каждый таск читает 70 КБ — дисковые операции на максимум
4 ядра — 2 партиции на ядро — без простоя	4 ядра — 500 волн по 4 таска — 499 переключений контекста
Итог: 0.05 сек на оркестрацию, 2.0 сек на данные	Итог: 2.5–3.0 сек на оркестрацию, 0.8 сек на данные

Важно: Правило: партиции ≈ ядра × 2–3

Как работает автоматическое уменьшение партиций

Когда меняется число партиций при включённом AQE?

01

Планирование

Задаём 2000 партиций для шаффла

02

Шаффл (запись)

Физически создаём 2000 файлов по 70 КБ (140 МБ / 2000)

Каждый исполнитель пишет 2000 файлов локально

03

Анализ статистики (между стадиями) ← КЛЮЧЕВОЙ МОМЕНТ

Драйвер измеряет: «2000 × 70 КБ = 140 МБ, но после агрегации осталось 1000 строк (~20 КБ)»

Решение: «Объединить 2000 → 1 партицию»

04

Агрегация

Запускаем 1 таск вместо 2000

- Изменение происходит ПОСЛЕ шаффла, ДО агрегации
- Физически: 2000 файлов — локальное объединение в памяти — 1 таск
- Нет нового шаффла — только изменение плана выполнения

Три правила перед запуском

1

Не верьте дефолту 200

Ставьте ядра × 2–3 (для ноутбука: 4–8)

2

Всегда вызывайте .stop() между сессиями

Иначе конфиг «застрянет» из предыдущей сессии

3

Проверяйте финальное число партиций

result.rdd.getNumPartitions()

⚠ **• Занижать партиции опаснее, чем завышать:**

• Завысили — AQE спасёт (объединит мелкие)

• Занизили — будете ждать час (увеличить не может)

Подписывайтесь!

В следующей лекции:

CSV vs Parquet: Почему формат хранения может влиять сильнее кода

Если вам понравилось – заходите на телеграм:

t.me/marat_notes

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