Тема 4. Apache Spark — основная платформа обработки больших данных

Лекция 4.1. Apache Spark: архитектура и преимущества перед MapReduce

Apache Spark создан в 2009 году в AMPLab Калифорнийского университета в Беркли и передан в Apache Software Foundation в 2013 году. Главное техническое решение, отличающее Spark от MapReduce — хранение промежуточных результатов в оперативной памяти (in-memory computing). MapReduce записывает результат каждого шага на диск; Spark держит данные в памяти между операциями. Для итерационных задач (алгоритмы машинного обучения выполняют десятки-сотни итераций) это даёт ускорение в 10–100 раз.

— Batch processing: обработка данных пакетами — быстрее MapReduce благодаря in-memory вычислениям.

— Structured Streaming: потоковая обработка данных в режиме реального времени на базе DataFrame API.

— Spark SQL: аналитика с помощью SQL и DataFrame API с оптимизацией через Catalyst.

— MLlib: библиотека масштабируемых алгоритмов машинного обучения.

— GraphX: обработка графовых данных (PageRank, компоненты связности, кратчайшие пути).

— Унификация в одном фреймворке устраняет необходимость поддерживать отдельные инструменты для каждого типа задач.

Компоненты кластера

— Driver: главная программа. Создаёт SparkSession, описывает логику обработки, строит DAG, координирует выполнение. Запускается на выделенном driver-узле.

— SparkSession (Spark 2.0+): единая точка входа, объединяющая SparkContext, SQLContext и HiveContext. Создаётся один раз при запуске приложения.

— Cluster Manager: управляет ресурсами кластера. Поддерживаемые варианты: встроенный Standalone, YARN (Hadoop), Kubernetes (современный стандарт), Apache Mesos.

— Executor: JVM-процесс на рабочем узле. Выполняет задачи (tasks), хранит кэшированные данные. Каждое Spark-приложение имеет выделенные Executor'ы, изолированные от других приложений.

— Task: минимальная единица параллельной работы. Один task обрабатывает одну партицию данных.

Лекция 4.2. Ленивые вычисления и DAG

Ленивые вычисления (Lazy Evaluation) — ключевой принцип Spark. Когда программист вызывает трансформацию (map, filter, join), Spark не выполняет её немедленно, а добавляет узел в граф вычислений (DAG — Directed Acyclic Graph). Реальные вычисления начинаются только при вызове действия (action: collect, count, save). К этому моменту Spark имеет полную картину вычислительного графа и может применить глобальные оптимизации: объединить несколько фильтров в один проход, переставить операции в более эффективный порядок, выбрать оптимальный алгоритм соединения.

Трансформации (Transformations) и действия (Actions)

— Узкие трансформации (Narrow): каждая партиция входного RDD используется для вычисления ровно одной партиции выходного. Данные не перемещаются между узлами. Примеры: map(), filter(), flatMap(). Быстрые.

— Широкие трансформации (Wide / Shuffle): данные перераспределяются между партициями и узлами. Требуют сортировки и сетевой передачи данных. Примеры: groupByKey(), reduceByKey(), join(), repartition(). Дорогие по времени и ресурсам.

— Действия (Actions): запускают выполнение DAG и возвращают результат в Driver или записывают в хранилище. Примеры: collect(), count(), first(), take(n), saveAsTextFile(), write.parquet().

Лекция 4.3. RDD — Resilient Distributed Dataset

RDD — исходная абстракция данных Spark. Неизменяемая, партиционированная коллекция объектов, распределённых по узлам кластера. «Resilient» (устойчивый) означает: если партиция потеряна из-за сбоя узла, Spark восстанавливает её, повторно применяя записанную последовательность трансформаций (lineage) к исходным данным. Не требует репликации данных для восстановления.

Кэширование и уровни хранения

rdd.cache() эквивалентно rdd.persist(StorageLevel.MEMORY_ONLY). Сохраняет партиции в памяти Executor'ов. При повторном обращении к RDD данные берутся из кэша, а не вычисляются заново. Для итерационных алгоритмов ML это сокращает время выполнения на порядок. Уровни хранения: MEMORY_ONLY (только RAM, самый быстрый), MEMORY_AND_DISK (переполнение на диск), DISK_ONLY (только диск), OFF_HEAP (память вне JVM heap — снижает паузы garbage collector).

Лекция 4.4. DataFrame API и оптимизатор Catalyst

DataFrame — высокоуровневая абстракция над RDD, представляющая данные в виде таблицы с именованными столбцами и типами (IntegerType, StringType, TimestampType и другие). В отличие от RDD, DataFrame не требует явного указания типов в Python/Scala — Spark выводит их из схемы. Ключевое преимущество: все операции над DataFrame проходят через оптимизатор Catalyst.

Catalyst Optimizer

Catalyst принимает логический план запроса (что нужно сделать) и преобразует его в физический план (как именно выполнить). Стадии: разбор (parsing) → анализ (analysis, проверка схемы) → логическая оптимизация (pushdown предикатов, объединение проекций, упрощение выражений) → генерация физического плана (выбор алгоритма JOIN: broadcast hash join, sort-merge join, shuffle hash join) → генерация кода (Tungsten Engine генерирует байт-код JVM под конкретный запрос).

Основные операции DataFrame

— select(), selectExpr() — проекция столбцов, вычисление выражений.

— filter() / where() — фильтрация строк.

— groupBy(...).agg(sum, count, avg, max, min, collect_list) — агрегация.

— join(other, condition, joinType) — типы: inner, left, right, full, cross, semi, anti.

— withColumn(name, expr) — добавление или изменение столбца.

— Window functions: rank(), dense_rank(), row_number(), lag(), lead(), rolling window aggregate — аналитические функции без группировки.

— dropDuplicates(subset) — удаление дубликатов по указанным столбцам.

Лекция 4.5. Spark SQL

Spark SQL интегрирует SQL непосредственно в Spark-приложение. DataFrame регистрируется как временное представление (createOrReplaceTempView), после чего к нему применяются стандартные SQL-запросы. Результат — DataFrame, который можно продолжать трансформировать. Один и тот же оптимизатор Catalyst обрабатывает как программный API, так и SQL: нет разницы в производительности между df.filter("age > 18") и SQL WHERE age > 18.

Чтение и запись данных

— spark.read.parquet(path) — Parquet, предпочтительный формат для аналитики.

— spark.read.csv(path, header=True, inferSchema=True) — CSV с автоматическим определением типов.

— spark.read.json(path) — JSON, включая многострочный формат.

— spark.read.jdbc(url, table, properties) — чтение из реляционных СУБД через JDBC.

— spark.read.format("delta").load(path) — Delta Lake с поддержкой ACID-транзакций.

— df.write.mode("overwrite").partitionBy("date").parquet(path) — запись с партиционированием.

Лекция 4.6. Structured Streaming: потоковая обработка

Structured Streaming (появился в Spark 2.0) представляет поток данных как бесконечно пополняемую таблицу. Каждая новая порция данных (микро-батч) добавляется как новые строки. Запрос к потоку записывается так же, как запрос к статической таблице — на DataFrame API или SQL. Это принципиально отличает Structured Streaming от старого DStream API (Spark 1.x), где разработчик явно управлял временными окнами.

Ключевые концепции

— Event time vs Processing time: event time — время наступления события (метка в данных); processing time — время поступления данных в систему. Structured Streaming поддерживает агрегацию по event time, что корректно при запоздалых данных.

— Watermarking: граница допустимого запаздывания данных. Spark не ждёт данные позже watermark и высвобождает занимаемую ими память состояния.

— Output modes: Append (только новые строки), Update (изменённые строки), Complete (полная таблица результатов при каждом триггере).

— Источники: Apache Kafka (основной production-источник), Amazon Kinesis, файловая система (новые файлы в директории), сокеты (для тестирования).

— Checkpointing: сохранение состояния потока в надёжное хранилище для восстановления после сбоя.