1. Назначение платформы Hadoop

Hadoop — это платформа с открытым исходным кодом для **распределённого хранения и параллельной обработки очень больших объёмов данных** (от терабайтов до петабайтов) на кластере дешёвых серверов.
Основные цели:

- Масштабируемость: возможность увеличивать объёмы данных простым добавлением узлов.
- Отказоустойчивость: автоматическое восстановление при сбоях узлов.
- Высокая пропускная способность: эффективная обработка больших файлов.
- Работа с неструктурированными и полуструктурированными данными.

2. Основные модули Hadoop и их назначение

Классическая экосистема Hadoop включает **четыре ключевых компонента**:

1) HDFS (Hadoop Distributed File System)

Распределённая файловая система для хранения больших данных. Делит файлы на блоки и распределяет их по узлам кластера с репликацией для надёжности.

2) YARN (Yet Another Resource Negotiator)

Менеджер ресурсов и планировщик задач.
Отвечает за:

- распределение вычислительных ресурсов между приложениями;
- запуск приложений;
- контроль выполнения задач.

3) MapReduce

Модель и фреймворк для выполнения распределённых вычислений.
Обрабатывает данные параллельно по схеме:

- **Map** — преобразование данных;
- **Reduce** — агрегирование результатов.

4) Hadoop Common (или Core)

Набор общих библиотек и утилит, необходимых для работы всех остальных модулей (конфигурации, сериализация, взаимодействие узлов).

3. Область применения платформы Hadoop

Hadoop используют там, где необходимо хранить и обрабатывать **огромные данные**:

- Аналитика больших данных (Big Data Analytics)
- Обработка логов, телеметрии, событий
- Хранилища данных для Data Lake
- Машинное обучение на больших датасетах
- Анализ поведения пользователей (clickstream analysis)

- Аналитика IoT
- Обработка данных социальных сетей
- Научные и статистические вычисления с большими объёмами данных
- ETL для больших объёмов (экстракция, трансформация, загрузка)

4. Назначение HDFS

HDFS (Hadoop Distributed File System) — это распределённая файловая система, разработанная для:

- хранения очень больших файлов (гигабайты–петабайты);
- обеспечения высокой пропускной способности чтения/записи;
- устойчивости к отказам узлов (через репликацию блоков);
- работы на обычных, недорогих серверах;
- эффективного параллельного доступа.

Ключевая идея: *перемещать вычисления к данным, а не данные к вычислениям*.

5. Архитектура HDFS

Архитектура HDFS построена по модели **Master–Slave** и включает три типа узлов:

1. NameNode (основной узел)

- Централизованно управляет файловой системой.
- Хранит метаданные:
  - структуру каталогов и файлов;
  - расположение блоков по DataNode;
  - конфигурацию репликации.
- Не хранит сами данные.
- Является критическим компонентом кластера.

2. DataNode (рабочие узлы)

- Хранят реальные блоки данных.
- Выполняют команды NameNode: создание, удаление, копирование блоков.
- Периодически отправляют отчёты (heartbeats, block reports).

3. (Опционально) Secondary NameNode / Checkpoint Node

- Не является резервным NameNode.
- Периодически сохраняет состояние файловой системы и помогает уменьшить размер журнала изменений.

Принципы работы HDFS

- **Файл делится на блоки (обычно 128–256 МБ)**.
- Блоки распределяются по DataNode с **репликацией (обычно 3 копии)**.

- При сбое узла блоки автоматически восстанавливаются на другие DataNode.
- Высокая пропускная способность обеспечивается параллельным считыванием блоков.

6. Назначение YARN

YARN (Yet Another Resource Negotiator) — это подсистема Hadoop, отвечающая за:

- **управление ресурсами кластера** (CPU, память);
- **планирование и координацию выполнения задач**;
- **запуск распределённых приложений** (MapReduce, Spark, Flink, Tez, Storm и др.);
- **изоляцию и контроль ресурсов для каждого приложения**.

Иными словами, YARN — это «операционная система» Hadoop-кластера.

7. Архитектура YARN

Архитектура YARN построена по принципу *Master–Slave* и включает следующие компоненты:

1. ResourceManager (RM) — управляющий узел

Отвечает за:

- глобальное распределение ресурсов между приложениями;
- выбор узлов для запуска задач;
- запуск ApplicationMaster для каждого приложения.

Состоит из двух подсистем:

- **Scheduler** — планировщик ресурсов без учёта логики выполнения задач;
- **Application Manager** — запускает ApplicationMaster.

2. NodeManager (NM) — рабочий узел

Отвечает за:

- мониторинг ресурсов узла (CPU, RAM);
- запуск контейнеров (Containers);
- отчётность для ResourceManager (heartbeats).

3. ApplicationMaster (AM) — управляющий процесс каждого приложения

Для каждого приложения создаётся свой AM, который:

- запрашивает ресурсы у ResourceManager;
- распределяет задачи по NodeManager;
- следит за выполнением задач;
- обрабатывает сбои.

4. Контейнеры (Containers)

Универсальная сущность YARN, представляющая выделенный набор ресурсов:

- память;
- CPU;
- среду выполнения (runtime).

Каждая задача приложения выполняется в контейнере.

8. Принципы работы Hadoop YARN

1. **Клиент отправляет приложение** в кластер.
2. **ResourceManager выделяет первый контейнер** для запуска **ApplicationMaster**.
3. **ApplicationMaster регистрируется** в ResourceManager.
4. ApplicationMaster **запрашивает ресурсы** для выполнения задач.
5. ResourceManager **решает, где можно запустить контейнеры** (с учётом доступных ресурсов).
6. NodeManager **создаёт контейнеры** и запускает задачи.
7. ApplicationMaster **отслеживает выполнение задач**, повторно запускает упавшие.
8. После завершения работы:
    - Закрываются контейнеры
    - ApplicationMaster отключается
    - Ресурсы освобождаются

Ключевые идеи:
✓ Разделение вычислений и планирования
✓ Гибкость для разных фреймворков (не только MapReduce)
✓ Эффективное распределение ресурсов

9. ПО, необходимое для установки Hadoop

Типичный набор:

1. **Linux** (Ubuntu, Debian, CentOS, Rocky Linux и т.д.)
2. **Java (OpenJDK 8 или 11)** — необходимо для запуска Hadoop
3. **SSH** — для удалённого управления узлами кластера
4. **Hadoop** (дистрибутив Apache Hadoop)
5. **Python** (необязательно, но используется многими инструментами экосистемы)
6. (Опционально) **SSH-ключи** для работы без пароля между узлами кластера
7. (Опционально) ПО для распределённых систем:
    - Zookeeper (для HBase, Kafka, YARN HA)
    - Maven (если требуется сборка)

10. Основные этапы установки Hadoop

1. Подготовка системы

- Установка ОС Linux
- Настройка hostname и /etc/hosts
- Установка SSH и настройка входа по ключу без пароля

2. Установка Java

- Установка OpenJDK
- Настройка JAVA_HOME

3. Установка Hadoop

- Распаковка дистрибутива
- Настройка переменных среды (HADOOP_HOME, PATH)

4. Конфигурация Hadoop

Настройка основных файлов:

- **core-site.xml** — базовые настройки Hadoop
- **hdfs-site.xml** — параметры файловой системы HDFS (репликация, пути)
- **mapred-site.xml** — конфигурация MapReduce
- **yarn-site.xml** — настройки YARN

5. Форматирование NameNode

hdfs namenode -format

6. Запуск сервисов

- запуск HDFS:
- start-dfs.sh
- запуск YARN:
- start-yarn.sh

7. Проверка работы

- просмотр веб-интерфейсов:
    - NameNode: http://localhost:9870
    - ResourceManager: http://localhost:8088
- тестовый запуск MapReduce:
- hadoop jar hadoop-mapreduce-examples.jar wordcount input output

11. Команды для работы с HDFS и их назначение

HDFS CLI использует синтаксис:

hdfs dfs <команда> <аргументы>

Вот основные команды:

1) hdfs dfs -ls <путь>

Вывод списка файлов и каталогов в HDFS.

2) hdfs dfs -mkdir <путь>

Создание каталога в HDFS.

3) hdfs dfs -put <локальный_файл> <hdfs_путь>

Копирование файла из локальной файловой системы в HDFS.

4) hdfs dfs -get <hdfs_файл> <локальный_путь>

Скачивание файла из HDFS в локальную файловую систему.

5) hdfs dfs -cat <hdfs_файл>

Просмотр содержимого файла в HDFS.

6) hdfs dfs -rm <путь>

Удаление файла.

7) hdfs dfs -rm -r <каталог>

Удаление каталога и всех вложенных файлов.

8) hdfs dfs -copyFromLocal <файл> <путь>

Аналог -put, копирование из локальной в HDFS.

9) hdfs dfs -copyToLocal <файл> <путь>

Аналог -get, копирование из HDFS в локальную.

10) hdfs dfs -du -h <путь>

Показать объём данных, занимаемый файлами и каталогами.

11) hdfs dfsadmin -report

Показать состояние кластера HDFS, список DataNode.

12. Назначение HDFS API

HDFS API — это программный интерфейс, который позволяет приложениям взаимодействовать с распределённой файловой системой HDFS.

Назначение:

- чтение и запись файлов в HDFS из Java-программ;
- создание, удаление, переименование файлов и каталогов через код;
- управление правами доступа;
- получение метаданных файлов;
- поддержка потокового ввода/вывода (FSDataInputStream, FSDataOutputStream).

HDFS API используется Hadoop-приложениями, MapReduce, Spark, Hive, HBase и сторонними системами для работы с файлами в распределённой среде.

13. Определение MapReduce и назначение

MapReduce — это:

распределённая вычислительная модель и фреймворк Hadoop, который позволяет обрабатывать большие объёмы данных параллельно на множестве узлов кластера.

Назначение:

- выполнение сложных вычислений на больших данных;
- параллельная обработка распределённых данных;
- автоматическое распределение задач по узлам;
- упрощение разработки (разработчик пишет только функции Map и Reduce).

MapReduce скрывает детали параллелизма, отказоустойчивости и передачи данных между узлами.

14. Преимущества MapReduce перед другими моделями

1. Масштабируемость

Легко обрабатывает петабайты данных, распределяя работу по кластеру.

2. Отказоустойчивость

При сбое задачи автоматически перезапускаются на другом узле.

3. Простота разработки

Разработчик пишет только две функции (Map и Reduce), всё остальное делает Hadoop.

4. Локальность данных

Вычисления происходят *там, где находятся данные*, минимизируя сетевые передачи.

5. Высокая пропускная способность

Эффективен для пакетной обработки больших файлов.

6. Чёткое разделение этапов

Удобен для задач, которые можно выразить в виде "разбить → преобразовать → агрегировать".

15. Основные стадии решения MapReduce задачи

MapReduce состоит из **трёх ключевых этапов** (формально пяти, если учитывать внутренние стадии):

1. Map Stage (отображение)

- Чтение входных данных (input splits).
- Применение функции map(key, value).
- Генерация промежуточных пар <key, value>.

2. Shuffle & Sort Stage (перемешивание и сортировка)

Hadoop автоматически выполняет:

- сортировку данных по ключам;
- передачу промежуточных данных от узлов Map к узлам Reduce;

- группировку значений по ключам.

Это «сердце» MapReduce.

3. Reduce Stage (свертка)

- Получение отсортированных данных.
- Применение функции reduce(key, list_of_values).
- Генерация итоговых результатов.

Расширённый детальный список стадий:

1. **Input Splitting** — разбиение входных данных на части.
2. **Mapping** — выполнение map-задач.
3. **Partitioning** — разделение по разделам (partition).
4. **Shuffle and Sort** — пересылка и сортировка.
5. **Reducing** — выполнение reduce-задач и запись результата.

16. Опишите стадию Map

Map — первая стадия выполнения задачи MapReduce.
Её назначение — **преобразовать входные данные в набор промежуточных пар (key, value)**.

Как работает стадия Map:

1. Входные данные разбиваются на части (input splits).
2. Каждый узел выполняет свою часть с помощью функции:
3. map(key, value) → list<key, value>
4. Функция Map читает данные построчно или по записям и генерирует пары.
5. Промежуточные данные сохраняются локально до передачи на Reduce.

Пример: подсчёт слов.
Map получает строку текста и создаёт пары:

("слово", 1)

17. Опишите стадию Shuffle

Shuffle — это стадия между Map и Reduce, отвечающая за **пересылку, распределение, сортировку и группировку промежуточных данных**.

Что делает Shuffle:

1. **Пересылает промежуточные пары между узлами Map и Reduce.**
2. **Сортирует** данные по ключам.
3. **Группирует значения** с одинаковым ключом.
4. Передаёт сгруппированные данные в Reduce.

Игровое сравнение:
Shuffle — это «сортировочный центр», который получает миллионы пар от всех Map-задач и распределяет их по Reduce-задачам.

Это ключевой этап, который обеспечивает корректность Reduce.

18. Опишите стадию Reduce

Reduce — заключительная стадия MapReduce, где выполняется **агрегация данных по ключам**.

Принцип работы:

Reduce получает:

(key, [value1, value2, ...])

Применяет функцию:

reduce(key, list_of_values) → result

Результаты записываются в выходной файл (обычно один файл на Reducer).

Пример (подсчёт слов):
Для ключа "слово" Reduce получит список [1, 1, 1, 1]
и вернёт:

("слово", 4)

19. Назначение операции Combine

Combiner — это *локальный мини-Reduce*, который работает на узлах Map **до Shuffle**.

Назначение Combiner:

- уменьшить объём передаваемых данных на стадии Shuffle;
- локально агрегировать результаты Map, например:
- ("слово", 1), ("слово", 1), ("слово", 1)

превращаются в:

("слово", 3)

Combiner:

- работает *не всегда*, а когда Hadoop решит что это возможно;
- не гарантируется к исполнению;
- используется только для операций, где агрегирование — ассоциативное и коммутативное (например, сумма, максимум, минимум).

По сути: **Combiner = оптимизация**.

20. Перечислите области применения MapReduce модели

MapReduce эффективно работает для задач **пакетной обработки больших объёмов данных**.

Основные области применения:

1. **Обработка логов** (web-server logs, clickstream, telemetry).

2. **Аналитика больших данных** (Big Data Batch Processing).
3. **ETL-процессы** (извлечение–трансформация–загрузка).
4. **Обработка текстов** (поиск, индексация, подсчёт частот).
5. **Анализ социальных сетей** (графовые расчёты на больших данных).
6. **Web Indexing** — индексирование интернета (Google начинал именно с MapReduce).
7. **Обработка данных IoT** (сбор и анализ больших массивов телеметрии).
8. **Геномика, биоинформатика** — вычислительные задачи на больших последовательностях.
9. **Научные вычисления** (параллельные вычисления над большими наборами данных).
10. **Машинное обучение** (только batch-алгоритмы, например k-means, PageRank).

21. Перечислите виды MapReduce задач

Существуют несколько типов (категорий) MapReduce задач:

1. **Map-Only задачи**
2. **MapReduce задачи (Map + Reduce)**
3. **Chain MapReduce (цепочки задач)**
4. **Reduce-Side Join задачи**
5. **Map-Side Join задачи**
6. **MapReduce задачи с Combiner**
7. **Iterative MapReduce задачи** (для ML алгоритмов)

Но обычно выделяют три ключевых вида:

- **Map-Only**
- **Classic MapReduce (Map → Shuffle → Reduce)**
- **Chain MapReduce (несколько MapReduce подряд)**
- **ReduceJoin (Reduce-Side Join)**

22. Опишите группу MapOnly задач и их назначение

Map-Only — это задачи, которые **не требуют фазы Reduce**, то есть нет необходимости в агрегировании, группировке или соединении данных.

Назначение:

Используются, когда достаточно только преобразования входных данных.

Особенности MapOnly:

- нет Shuffle;
- нет Reduce;
- Map создает итоговый результат напрямую;

- работает быстрее стандартного MapReduce, так как пропускает тяжёлую стадию Shuffle.

Примеры MapOnly задач:

- Конвертация форматов файлов
- Фильтрация строк/записей
- Очистка данных (data cleaning)
- Локальные преобразования без агрегации
- Copy/Transform операций

Пример: оставить только строки, содержащие слово “ERROR”.

23. Опишите цепочку MapReduce задач

Цепочка MapReduce (Chain MapReduce) — это выполнение **нескольких MapReduce задач последовательно**, когда выход одной задачи служит входом следующей.

Название: **Pipeline MapReduce** или **MR → MR → MR...**

Назначение:

- построение сложных аналитических процессов;
- выполнение пошаговой обработки данных;
- выполнение сложных алгоритмов машинного обучения или графовых вычислений.

Пример цепочки:

1. MR1: Очистка данных
2. MR2: Группировка
3. MR3: Финальная агрегация
4. MR4: Построение индекса

Этот подход часто используется в:

- лог-аналитике;
- построении поисковых индексов;
- больших ETL-процессах;
- MapReduce-алгоритмах PageRank, K-means, Connected Components.

24. Опишите группу ReduceJoin задач и их назначение

Reduce-Side Join (ReduceJoin) — это вид MapReduce задач, в которых **соединение (JOIN) двух или более наборов данных выполняется на стадии Reduce**.

Назначение:

Применяется, когда:

- данные большие и не помещаются в память;
- необходимо объединить данные по ключу;

- невозможно выполнить Map-Side Join (например, нет сортировки или требования по распределению данных).

Как работает ReduceJoin:

1. Map:

- Каждая запись помечается, из какого набора данных она пришла.
- Map генерирует пары вида:
- (ключ_соединения, (метка, данные))

2. Shuffle:

- Все записи с одинаковым ключом попадают на один Reduce.

3. Reduce:

- Сгруппированные данные объединяются по ключу.
- Выполняется логика JOIN: inner join, outer join, left/right join.

Пример:

Соединение:

- таблицы пользователей
- таблицы покупок по user_id

25. Перечислите особенности модели MapReduce

Основные особенности:

1. Параллелизм

Задача автоматически делится на множество подзадач, которые выполняются параллельно на разных узлах.

2. Масштабируемость

Обрабатывает огромные объёмы данных путём добавления узлов (horizontal scaling).

3. Отказоустойчивость

- задачи автоматически перезапускаются при сбоях;
- данные реплицированы в HDFS.

4. Работа по принципу “вычисления рядом с данными”

Map-узлы запускаются на тех узлах, где находятся данные → минимизация сетевого трафика.

5. Чёткое разделение стадий

Map → Shuffle → Reduce.

6. Batch-модель

MapReduce — модель пакетной обработки, не подходит для real-time систем.

7. Требовательность к диску

MapReduce активно использует дисковые операции (shuffle, сортировка, запись результатов).

8. Простота программирования

Разработчик определяет только две функции: Map и Reduce.

9. Подходит для ассоциативных и коммутативных операций

Таких как сумма, счёт, минимум/максимум, агрегации.

10. Ограничения

- плохо подходит для итеративных алгоритмов;
- высокая латентность;
- неэффективно для маленьких задач.

26. Опишите структуру входных и выходных данных классической MapReduce задачи

Классическая MapReduce задача работает с данными в виде пар:

Входные данные для Map:

- разбиваются на **Input Splits**;
- каждая запись представлена как:
- (input_key, input_value)

Типичные варианты:

- **TextInputFormat**:
- (номер_строки, строка_текста)
- **KeyValueTextInputFormat**:
- (ключ, значение)
- **SequenceFileInputFormat**:
  бинарные пары (Writable, Writable)

Выходные данные стадии Map:

Map выводит произвольные пары:

(intermediate_key, intermediate_value)

Эти данные идут в Shuffle.

Вход на Reduce:

После Shuffle данные на Reducer приходят в виде:

(reduce_key, [value1, value2, value3...])

Выходные данные Reduce (финальный результат):

(output_key, output_value)

Формат задаётся через OutputFormat (обычно TextOutputFormat).

27. Приведите пример запуска MapReduce задачи

Пример запуска стандартного примера **WordCount**:

hadoop jar hadoop-mapreduce-examples.jar wordcount /input /output

Где:

- hadoop — командный интерфейс Hadoop
- jar hadoop-mapreduce-examples.jar — файл с MapReduce задачами
- wordcount — имя задачи
- /input — каталог в HDFS с входными данными
- /output — каталог для результата (должен отсутствовать)

28. Опишите способы создания в Java-приложениях mapper- и reducer-классов

В Java MapReduce создаётся через *наследование и переопределение методов*.

1. Создание Mapper-класса

Нужно наследовать:

```
public class MyMapper extends Mapper<KeyIn, ValueIn, KeyOut, ValueOut> {
   @Override
   protected void map(KeyIn key, ValueIn value, Context context)
         throws IOException, InterruptedException {

      // логика преобразования данных
      context.write(keyOut, valueOut);
   }
}
```

Пример WordCount:

```
public class TokenizerMapper
   extends Mapper<Object, Text, Text, IntWritable> {

   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();

   public void map(Object key, Text value, Context context)
         throws IOException, InterruptedException {

      StringTokenizer itr = new StringTokenizer(value.toString());
```

```
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, one);
        }
    }
}
```

2. Создание Reducer-класса

Нужно наследовать:

```
public class MyReducer extends Reducer<KeyIn, ValueIn, KeyOut, ValueOut> {
    @Override
    protected void reduce(KeyIn key, Iterable<ValueIn> values, Context context)
            throws IOException, InterruptedException {

        // логика агрегации
        context.write(keyOut, valueOut);
    }
}
```

Пример WordCount:

```
public class IntSumReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {

        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
}
```

3. Настройка Job в main():

```
Job job = Job.getInstance(conf, "word count");
```

```
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setReducerClass(IntSumReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
job.waitForCompletion(true);
```

29. Дайте определение Apache Spark

Apache Spark — это высокопроизводительный распределённый фреймворк для обработки больших данных, который выполняет вычисления **в памяти (in-memory)** и поддерживает:

- batch-обработку;
- потоковую обработку (streaming);
- SQL-запросы;
- машинное обучение (MLlib);
- графовые вычисления (GraphX).

Spark быстрее MapReduce в десятки раз благодаря хранению промежуточных данных в памяти.

30. Раскройте преимущества Apache Spark перед Hadoop MapReduce

Ключевые преимущества Apache Spark:

1. Скорость (в 10–100 раз быстрее MapReduce)

- Spark хранит промежуточные данные в оперативной памяти (in-memory);
- MapReduce постоянно пишет на диск на каждом этапе.

2. Универсальность

Spark содержит богатую экосистему:

- Spark SQL
- Spark Streaming
- MLlib
- GraphX
- DataFrames / Datasets
- Structured Streaming

MapReduce = только batch-обработка.

3. Поддержка итеративных и интерактивных задач

Итеративные алгоритмы (PageRank, ML) работают очень быстро, так как в Spark данные кэшируются в памяти.

MapReduce → медленный из-за постоянного Shuffle.

4. Простота разработки

Spark использует:

- Python
- Scala
- Java
- R

A API — декларативный, высокоуровневый → меньше кода и проще отладка.

MapReduce требует большого количества шаблонного кода на Java.

5. Поддержка real-time потоков

Spark Structured Streaming → обработка данных в реальном времени.

MapReduce не поддерживает real-time.

6. Расширенная оптимизация

Spark использует:

- Catalyst Optimizer
- Tungsten Execution Engine

MapReduce не имеет подобных оптимизаторов.

7. Лучшая работа со сложными данными и SQL

Spark SQL → мощный движок запросов с оптимизациями и DataFrames.

MapReduce → ручная работа с текстом и ключами.

31. Перечислите модули, из которых состоит Apache Spark

Apache Spark включает несколько модулей, обеспечивающих обработку данных различных типов:

1. **Spark Core**
   Основной движок, работа с RDD, управление задачами, распределёнными вычислениями.
2. **Spark SQL**
   Работа с SQL-запросами, DataFrame, Dataset, каталожный оптимизатор (Catalyst).
3. **Spark Streaming**
   Обработка потоков данных (микробатчи).
4. **Structured Streaming**
   Современный движок потоковой обработки в режиме реального времени.
5. **MLlib**
   Библиотека машинного обучения.

6. **GraphX**
   Графовые вычисления, алгоритмы графов (PageRank и др.).
7. **SparkR**
   API для языка R.
8. **PySpark**
   API для Python.

32. Опишите основные этапы установки Apache Spark

1. Установка Java (OpenJDK 8/11)

Spark работает поверх JVM.

2. Установка Hadoop (необязательно, но желательно)

Spark может работать:

- в режиме Standalone,
- поверх Hadoop YARN,
- через Kubernetes.

Для работы с HDFS — нужен Hadoop client.

3. Загрузка Spark

Скачать сборку Spark с сайта Apache (обычно версия spark-<версия>-bin-hadoopX.tgz).

4. Распаковка и настройка

Распаковать архив:

tar -xzf spark-*.tgz

mv spark-* /usr/local/spark

Добавить переменные среды:

export SPARK_HOME=/usr/local/spark

export PATH=$SPARK_HOME/bin:$PATH

5. Настройка конфигурационных файлов (опционально)

В каталоге conf/:

- spark-env.sh
- spark-defaults.conf

6. Запуск демонов Standalone (опционально)

sbin/start-master.sh

sbin/start-slave.sh spark://<master-host>:7077

33. Приведите команду для запуска Spark Shell

Для языка **Scala**:

spark-shell

Для **Python (PySpark)**:

pyspark

Для **R**:

sparkR

34. Дайте определение RDD

RDD (Resilient Distributed Dataset) — это *устойчивый распределённый набор данных* в Apache Spark, представляющий:

- неизменяемую (immutable),
- распределённую коллекцию объектов,
- разделённую на партиции,
- обрабатываемую параллельно на разных узлах кластера.

Свойства RDD:

- **immutable** — данные нельзя изменить, только создать новые RDD;
- **lazy evaluation** — вычисляется только при вызове action;
- **fault-tolerant** — Spark восстанавливает RDD по lineage (истории преобразований);
- **distributed** — автоматически разделяется между узлами кластера.

RDD — основа Spark Core.

35. Перечислите основные типы операций над RDD

Операции RDD делятся на **две большие категории**:

1. Transformations (трансформации)

Возвращают *новый RDD*.
Выполняются лениво (lazy).

Примеры:

- map()
- flatMap()
- filter()
- groupByKey()
- reduceByKey()
- mapValues()
- distinct()
- union()
- join()
- sortBy()

2. Actions (действия)

Запускают выполнение вычислений и возвращают результат.

Примеры:

- collect()
- count()
- take(n)
- reduce()
- first()
- saveAsTextFile()
- foreach()
- countByKey()

36. Перечислите основные трансформации, производимые над RDD

Трансформации (Transformations) — создают новый RDD. Вычисляются лениво.

Основные трансформации:

Элементарные операции

- **map(func)** — применяет функцию к каждому элементу.
- **flatMap(func)** — как map, но возвращает множество элементов на один входной.
- **filter(func)** — фильтрация элементов.

Агрегации и группировки

- **groupByKey()** — группировка значений по ключу.
- **reduceByKey(func)** — агрегация значений по ключу с функцией reduce.
- **aggregateByKey(zero, seq, comb)** — кастомная агрегация по ключам.
- **combineByKey()** — низкоуровневая комбинация значений по ключу.

Сортировки и множества

- **sortBy(func)** — сортировка.
- **distinct()** — удаление дубликатов.
- **union(otherRDD)** — объединение.
- **intersection(otherRDD)** — пересечение.
- **subtract(otherRDD)** — разность.

Работа с парами ключ–значение (Pair RDD)

- **mapValues(func)** — применяет функцию только к значениям.
- **flatMapValues(func)**
- **join(otherRDD)** — соединение по ключам.
- **leftOuterJoin()**, **rightOuterJoin()**, **fullOuterJoin()**

Работа с распределением

- **coalesce(n)** — уменьшить количество партиций.
- **repartition(n)** — перераспределить данные (shuffle).
- **partitionBy(partitioner)** — кастомное распределение.

37. Перечислите основные действия, выполняемые над RDD

Действия (Actions) — запускают вычисления и возвращают результат:

Работа с элементами

- **collect()** — возвращает все элементы в драйвер.
- **take(n)** — возвращает первые n элементов.
- **takeSample()** — случайная выборка.
- **first()** — первый элемент.
- **top(n)** — n наибольших элементов.

Агрегации

- **count()** — число элементов.
- **countByKey()** — число элементов по ключам.
- **reduce(func)** — объединение всех элементов через функцию.
- **fold(zero)(func)** — reduce с начальным значением.
- **aggregate(zero)(seq, comb)** — пользовательская агрегация.

Сохранение данных

- **saveAsTextFile(path)** — сохранение в текстовый файл.
- **saveAsSequenceFile(path)**
- **saveAsObjectFile(path)**

Побочные действия

- **foreach(func)** — применяет функцию к каждому элементу (на исполнителях).

38. Приведите команды для загрузки данных в RDD

Загрузка выполняется через **SparkContext**.

Загрузка текстовых файлов

```
val rdd = sc.textFile("hdfs://path/to/file")
```

или локальный файл:

```
val rdd = sc.textFile("file:///home/user/data.txt")
```

Загрузка каталога

```
val rdd = sc.textFile("hdfs://path/to/directory")
```

Загрузка из списка

```
val rdd = sc.parallelize(List(1, 2, 3, 4))
```

Загрузка бинарных файлов

```
val rdd = sc.binaryFiles("hdfs://path/*.bin")
```

Загрузка SequenceFile

```
val rdd = sc.sequenceFile[String, Int]("hdfs://path/to/seqfile")
```

Загрузка wholeTextFiles (каждый файл — строка)

```
val rdd = sc.wholeTextFiles("hdfs://path/")
```