


Основополагающие

☰ Tags	
📎 Files & media	



Apache Spark — это универсальная и высокопроизводительная кластерная вычислительная платформа, которая выполняет вычисления в оперативной памяти. Фраемворк (то есть набор библиотек).

Spark написан на Scala.



Обрати внимание!

Основные две абстракции, которые предоставляет Spark — RDD и shared variables.

▼ Сравнение с Hadoop и MapReduce

Ключевая особенность в скорости вычислений нам дает работа с RAM памятью, но! Spark из-за этого не работает с такими же массивными файлами как Map Reduce.

- Если результирующий набор данных **превышает объем оперативной памяти**, то Spark **не обойдет Map Reduce**. Также если скорость обработки не критична, то можно использовать MR (ибо диск дешевле чем оперативка). По понятным причинам архивные данные и исторические лучше для MR.
- Если фигурирует машинное обучение или графы, то, очевидно, Spark лучше. Также в итерационных алгоритмах спарк получше, не забываем еще про обработку в реальном времени

Spark не предоставляет хранилище, поэтому Spark не может полностью заменить Hadoop.

Параметр	Apache Spark	MapReduce
Тип	Фреймворк для распределённой обработки данных в памяти	Модель программирования для обработки данных
Основная функция	Быстрая обработка больших данных с использованием вычислений в памяти	Распределённая обработка данных с использованием диска
Производительность	Высокая за счёт использования памяти (in-memory processing)	Низкая, так как каждый этап чтения/записи данных происходит на диск
Тип обработки данных	Обработка данных в реальном времени (microbatch) и пакетная обработка	Пакетная обработка данных
Модель программирования	Поддерживает разнообразные API: SQL, Streaming, MLlib, GraphX	Основана на двух этапах: Map и Reduce
Хранение данных	Не хранит данные, интегрируется с HDFS, S3 и другими хранилищами	В основном использует HDFS для хранения данных
Масштабируемость	Очень масштабируемый , поддерживает работу на тысячах узлов	Масштабируем, но медленнее из-за интенсивного использования диска
Поддержка операций	Поддерживает широкий спектр операций: map, filter, join, reduceByKey и др.	Операции ограничены основными функциями Map и Reduce
Управление ресурсами	Работает с YARN, Mesos или Kubernetes для управления ресурсами	Работает с YARN для управления ресурсами в Hadoop
Язык программирования	Поддерживает Java, Scala, Python, R	Обычно используется с Java , также поддерживает Python
Поддержка реального времени	Поддерживает потоковую обработку (Spark Streaming, Structured Streaming)	Поддержка потоковой обработки отсутствует, только пакетная
Устойчивость к сбоям	Встроенная устойчивость через DAG и механизм повторных вычислений (lineage)	Устойчивость к сбоям через репликацию данных в HDFS
Использование памяти	Использует память для хранения промежуточных данных между	Использует диск для хранения промежуточных данных

Параметр	Apache Spark	MapReduce
	задачами	
Назначение	Быстрая обработка больших данных с различными сценариями использования (ML, Graph, Streaming)	Пакетная обработка больших данных с использованием диска

Основные выводы:

Spark активно использует локальные диски на нодах кластера, поэтому он не совсем in-memory.

5-10 мл строчек в таблице - мало. терабайты-петабайты - вот это норм. сотни терабайт точно были

- Так как spark использует DAG, то у нас нормально перераспределяются задачи (например, понимает, что фильтр не требует “map-шафл-редюс-map-шафл- редюс” как в Map Reduce)
- несколько сотни миллионов -миллиардов данных в сутки

▼ Экосистема

1. Spark Core:

- Основной модуль, реализующий ключевые функциональные возможности фреймворка Spark.
- Включает планирование заданий, управление памятью, обработку ошибок и взаимодействие с системами хранения данных (HDFS, S3 и др.).
- Основной API — это **RDD (Resilient Distributed Datasets)**, который представляет собой распределённые коллекции данных, обрабатываемые параллельно.
- Spark Core предоставляет функции управления этими коллекциями: трансформации данных (map, filter и др.), поддержка отказоустойчивости через lineage (линейность) и параллелизм вычислений.

2. Spark SQL:

- Пакет для работы со **структурированными данными**, поддерживающий запросы на SQL и Hive Query Language (HQL).

- Позволяет работать с множеством источников данных, таких как **таблицы Hive, Parquet, JSON** и другие форматы.
- Поддерживает смешивание SQL-запросов с программными конструкциями на **Python, Java** и **Scala** в одном приложении, что позволяет комбинировать SQL-запросы с более сложной аналитикой на уровне RDD или **DataFrame**.
- **DataFrame** и **DataSet** (из-за более позднего появления и наличия схемы) являются частью **SparkSQL**.

3. Spark Streaming:

- Компонент для **обработки потоковых данных** в реальном времени.
- Поддерживает источники данных, такие как очереди сообщений (например, Kafka), журналы веб-серверов или обновляемые файлы.
- Обеспечивает **микروпакетную** обработку данных, что позволяет работать с непрерывными потоками событий в реальном времени.

4. MLlib (Spark ML):

- Библиотека для **машинного обучения (Machine Learning)**, входящая в состав Spark.
- Поддерживает широкий спектр алгоритмов: **классификация, регрессия, кластеризация, совместная фильтрация** и другие.
- Включает инструменты для построения и тестирования моделей, а также возможность работы с большими объемами данных благодаря параллельным вычислениям.

5. GraphX:

- Библиотека для **обработки графов** и параллельных вычислений на графах.
- Примеры использования включают анализ социальных сетей (графы друзей), оптимизация маршрутов и другие задачи, связанные с графовыми структурами.

Менеджеры кластеров

Для масштабирования приложений Spark поддерживает несколько менеджеров кластеров:

- **Hadoop YARN** — менеджер ресурсов из экосистемы Hadoop.
- **Apache Mesos** — распределённая система управления ресурсами для работы на кластерах.
- **Standalone Scheduler** — встроенный менеджер кластера в Spark, который подходит для простых сценариев развертывания.

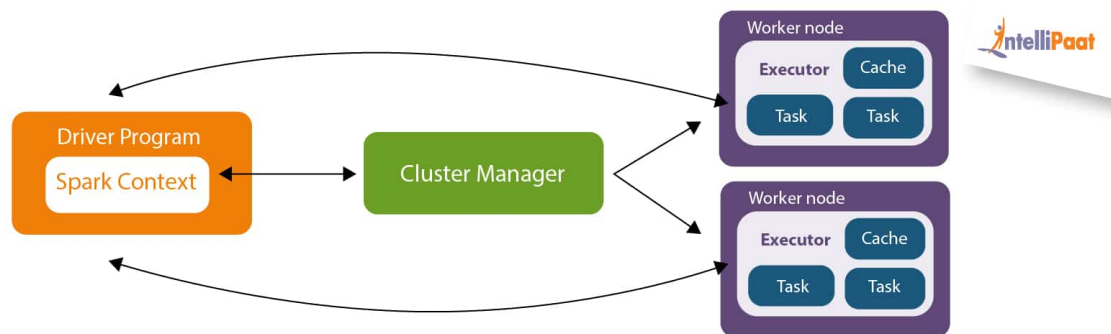
▼ Spark API

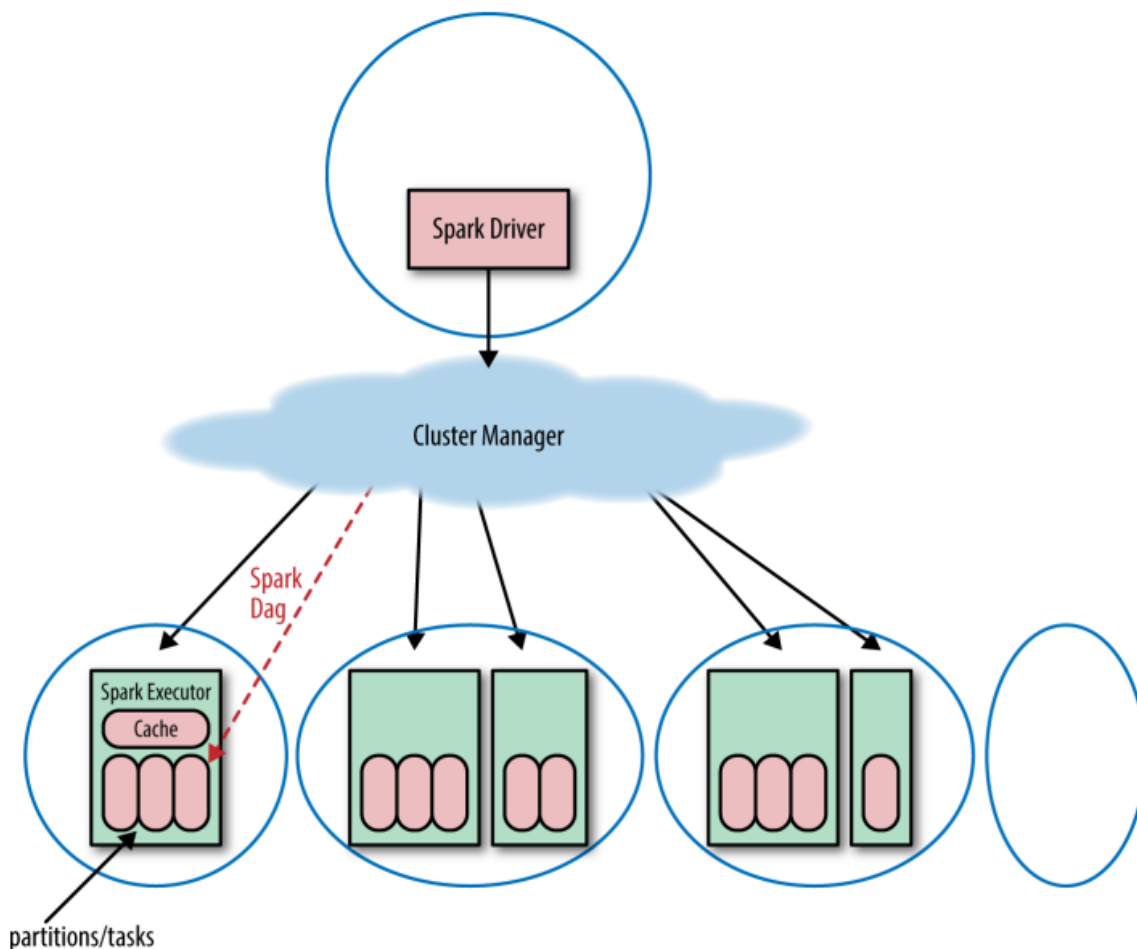
- Java
- Scala
- R
- Python
- SQL

Когда пишем код на **R** или **Python**, то Spark транслирует этот код для того, чтобы он мог запускаться на JVM's ноде (executor).

▼ Архитектура

Основные компоненты: драйвер (внутри SparkContext/SparkSession), менеджер кластера, и рабочие узлы (worker)





В распределенном режиме Spark использует архитектуру **ведущий/ ведомый (master/slave)** с одним центральным координатором (драйвером) и множеством распределенных **рабочих узлов** (исполнителями).

▼ Program Driver



Драйвер Spark — процесс Java, запускает `main()` - функцию. Отдельная физическая машина.

- **Преобразует пользовательскую программу в задания.** Драйвер преобразует пользовательскую программу и делит ее на **задачи (tasks)**, принимает входные данные, а также создает RDD, планирует задачи (tasks scheduling).

- **Строит DAG.** Программа Spark неявно создает **логический ориентированный ациклический граф (Directed Acyclic Graph, DAG)** операций. В процессе работы драйвер преобразует этот **логический граф в фактический план выполнения**.
- **Анализирует, распределяет и планирует работу среди исполнителей.** На основе составленного плана выполнения драйвер Spark **координирует передачу отдельных заданий исполнителям**. Когда исполнители запускаются, они **регистрируют** себя в драйвере, благодаря чему драйвер имеет полное представление об имеющихся в его распоряжении исполнителях.
- **Следит за местоположением кэшированных данных.** Он использует эту информацию, когда принимает решение о выполнении следующих заданий, использующих эти данные.
- **Создает SparkSession.**
- **Хостинг веб UI .**

Примерный поток выполнения драйвера:

- **Программа пользователя запускается на драйвере** — драйвер управляет координацией выполнения программы.
- **Драйвер вычисляет физический план выполнения программы** — Spark строит план выполнения, который включает разделение данных на задания (jobs), этапы (stages) и задачи (tasks), которые будут распределены по узлам кластера.
- **Драйвер подключается к менеджеру кластера и запрашивает доступные исполнители** — менеджер кластера (например, YARN или Standalone Manager) предоставляет исполнителей (executors), которые будут выполнять задачи.
- **Драйвер отправляет исполнителям запрос на выполнение задач** — после получения ресурсов драйвер распределяет задачи (tasks) между исполнителями.
- **Исполнители выполняют задачи и отправляют результаты обратно драйверу** — каждый исполнитель обрабатывает свою часть данных и отправляет промежуточные или финальные результаты обратно на драйвер.
- **Когда все результаты собраны, SparkContext явно закрывается, и программа драйвера завершается** — это завершающий шаг, когда драйвер

завершает работу после получения всех результатов и выполнения всех операций.

▼ Executor



Исполнители в Spark — это рабочие процессы, ответственные за выполнение отдельных заданий. Исполнители запускаются один раз в начале приложения Spark. **Обычно на одной ноде может быть несколько исполнителей.**

Отвечают за:

1. **Выполняют задания**, переданные приложением, и возвращают результаты драйверу.
2. Обеспечивают **сохранение в памяти наборов RDD, кэшированных пользовательскими программами**, через службу Block Manager, действующую внутри каждого исполнителя. Так как наборы RDD кэшируются непосредственно внутри исполнителей, задания могут манипулировать кэшированными данными.

У каждого executor'а свой собственный java-процесс.

Heartbeat

У исполнителей (executors) есть механизм "**сердцебиения**" — конфигурационный параметр `spark.executor.heartbeatInterval`, который задает частоту, с которой они отправляют сигналы драйверу через равные промежутки времени, чтобы сообщить, что они работают. Если сигнал не поступает, исполнитель считается "зафейленным", и драйвер переназначает задачу другому исполнителю. По умолчанию интервал равен 10 секундам, и он должен быть меньше, чем значение параметра `spark.network.timeout`.

Содержит: ID исполнителя, метрики по задачам, выполняющимся на исполнителе (время выполнения, сборка мусора (GC), время процессора, размер результата и т.д.) и ID менеджера блоков исполнителя.

Сообщение затем поступает драйверу через метод

`org.apache.spark.HeartbeatReceiver#receiveAndReply(context: RpcCallContext)`. Драйвер:

- обновляет время последнего сообщения от данного исполнителя;
- проверяет, знает ли он о блок-менеджере исполнителя;
- обновляет метрики по задачам.

После обработки сигнала "сердцебиения" драйвер готовит ответ, который содержит только одну информацию — **флаг (boolean)**, указывающий, должен ли исполнитель зарегистрировать свой блок-менеджер со всеми уже сгенерированными блоками.

Исполнитель считается "мертвым", если к моменту проверки последнее сообщение "сердцебиения" старше, чем значение тайм-аута, указанное в параметре

`spark.network.timeout`.

При удалении драйвер сообщает планировщику задач (Task Scheduler) об утрате исполнителя. Затем планировщик обрабатывает потерю задач, выполнявшихся на этом исполнителе. Драйвер также передает информацию планировщику DAG (DAG Scheduler), который удаляет все следы (например, блоки shuffle) потерянного исполнителя. Более того, драйвер запрашивает у **SparkContext** замену потерянного исполнителя через используемого кластерного менеджера. Однако эта операция не гарантирует появления нового исполнителя, так как его могут "украсть" другие приложения.

Исполнители отправляют сообщения "сердцебиения" с фиксированным интервалом, который задается параметром конфигурации

`spark.executor.heartbeatInterval`. Логически, это значение должно быть меньше, чем параметр `spark.network.timeout`. Как показано в тесте *"the job" should "never start if the heartbeat interval is greater than the network timeout"*, задача никогда не начнется при неправильной настройке этих параметров.

Иногда сообщения, отправленные исполнителем, не могут быть доставлены драйверу. В этом случае исполнитель увеличивает внутренний счетчик неудачных отправок. При каждой неудаче исполнитель пытается отправить сообщение еще раз. Он прекращает попытки, когда счетчик достигает значения, заданного параметром `spark.executor.heartbeat.maxFailures`. В этот момент исполнитель считает, что с драйвером произошла проблема, и завершает свою работу.

Исполнитель находится на рабочем узле (worker node), который может содержать несколько исполнителей.

▼ Cluster manager



Диспетчер кластера — наш посредник, подключаемый компонент Spark.

- Принимает запрос драйвера и подбирает ресурсы (executor) в зависимости от того, где расположены наши данные.
- Управляет ресурсами и нодами, отслеживает статус выполнения
- Благодаря такому посреднику мы можем запускать на нашем кластере несколько Spark-приложений.

Выглядит примерно так: Сначала программа-драйвер отправляет запрос (ping) менеджеру кластера. Менеджер кластера запускает определенное количество исполнителей Spark (JVM, показанные как черные коробки на диаграмме) на рабочих узлах кластера (показанные как синие круги).

Виды:

- **Standalone** - стандартный, работает на архитектуре master/worker. Максимум один исполнитель на каждой worker node. По дефолту, спарк старается использовать максимальное количество разных узлов. If for example the cluster has 4 workers and we configured the use of 4 cores and 4GB RAM, Spark will take 1 core and 1GB RAM of each node to execute the program.
- **Hadoop YARN** — менеджер кластера, происходящий от Hadoop и основанный на HDFS. Благодаря этому он может быстрее считывать данные, хранящиеся в HDFS.
- **Apache Mesos;**

Также если мы пишем код не на Java или Scala, то Spark транслирует этот код в такой, который бы запускался в JVM.

▼ Спекулятивное выполнение задач (Speculative Execution)

Спекулятивное выполнение задач запускается, когда **какая-либо задача выполняется значительно медленнее, чем остальные**. Это позволяет повысить

производительность, параллельно запуская "резервные" копии медленных задач. По умолчанию эта опция **выключена**, и за её включение отвечает параметр

`spark.speculation`.

Основные настройки:

- **spark.speculation.interval** — как часто Spark будет проверять задачи на возможность спекулятивного выполнения (по умолчанию 100 миллисекунд).
- **spark.speculation.multiplier** — во сколько раз задача должна выполняться медленнее медианного времени выполнения задач, чтобы считаться кандидатом для спекулятивного выполнения (по умолчанию 1.5).
- **spark.speculation.quantile** — процент задач, которые должны быть завершены в рамках текущего этапа (stage), чтобы спекулятивное выполнение стало возможным (по умолчанию 0.75, что эквивалентно 75%).

Все это контролируется через **TaskManager**, который следит за выполнением задач. При этом спекулятивная задача не может выполняться на одной и той же ноте больше одного раза.

Когда одна из спекулятивных задач успешно завершается, остальные её копии автоматически **отменяются** (происходит их *умышленное завершение* — *killed intentionally*)

✓ **Преимущества спекулятивного исполнения:**

- **Ускорение выполнения:** В ситуациях с "медленными" задачами, запуск дубликатов помогает быстрее завершить выполнение всего этапа и приложения в целом.
- **Устойчивость к неравномерной загрузке:** Спекулятивное исполнение защищает от проблем с производительностью отдельных узлов.
- **Минимизация влияния "stragglers":** Даже если отдельные задачи сталкиваются с проблемами производительности, остальная часть кластера может продолжить работу без простоев.

✗ **Недостатки спекулятивного исполнения:**

- **Дополнительные ресурсы:** Запуск дубликатов задач потребляет больше ресурсов (памяти и процессорного времени), что может снизить общую

производительность кластера, особенно если ресурсов недостаточно.

- **Неэффективность для небольших кластеров:** В кластерах с небольшим количеством узлов запуск дубликатов может не привести к значительным улучшениям, так как нет свободных ресурсов для выполнения копий задач.
- **Риски для задач с побочными эффектами:** Если задачи имеют побочные эффекты (например, запись данных во внешние системы), то дубликаты могут привести к некорректным результатам, так как задача может быть выполнена несколько раз.

SparkSession SparkContext

SparkSession и SparkContext



Обрати внимание

SparkContext

определяет, сколько ресурсов выделяется каждому исполнителю (executor). SparkContext — это как набор параметров конфигурации для выполнения заданий Spark. Эти параметры доступны через объект **SparkConf**, который используется для создания **SparkContext**.



SparkContext — это основной объект, который позволяет взаимодействовать с кластером Spark и управляет распределением задач на исполняющие узлы (executors). В соответствии с документацией, в каждой JVM может быть активен только один **SparkContext**. Если нужно создать новый, нужно явно вызвать `stop()` на текущем **SparkContext**, иначе Spark выбросит ошибку. Если все же требуется несколько контекстов, можно установить параметр конфигурации `spark.driver.allowMultipleContexts` в `true`, но это не рекомендуется.



SparkSession — это новая, улучшенная версия **SparkContext**, которая была введена в Apache Spark начиная с версии 2.0. Она инкапсулирует **SparkContext** и другие контексты API (например, для работы с SQL и Streaming). **SparkSession** является унифицированным интерфейсом для работы с Spark и используется для упрощения создания приложений на основе Spark.

Множественные контексты (SparkContext) могут привести к проблемам. Например, несколько **SparkContext** в одной JVM не гарантируют корректное выполнение пайплайна, и управление процессом становится сложнее. Наш workflow перестает быть изолированным, поэтому сбой на одном из контекстов способен повлиять на остальные и даже сломать нашу JVM.

Преимущество SparkSession — благодаря инкапсуляции, можно создавать несколько экземпляров **SparkSession**, каждый из которых будет "оберткой" над одним **SparkContext**. Это решает проблему с несколькими контекстами и позволяет задавать различные конфигурации для каждой сессии, сохраняя изоляцию и упрощая управление пайплайнами.

▼ Вопросы по теме “Основы Apache Spark”

Spark и его архитектура:

1. Что такое **Apache Spark**?
2. В чем ключевые отличия между **Apache Spark** и **MapReduce**?
3. Какие API поддерживает **Spark**?
4. Какие основные компоненты входят в архитектуру Spark?
5. Какую роль выполняет **драйвер Spark** в архитектуре?
6. Что такое **RDD**, и как Spark их использует?

SparkSession и SparkContext:

1. Что такое **SparkContext** и какова его основная роль?
2. В чем разница между **SparkContext** и **SparkSession**?

3. Почему не рекомендуется использовать несколько **SparkContext** в одном приложении?
4. Какие преимущества дает использование нескольких **SparkSession** над одним **SparkContext**?

Поток выполнения драйвера:

1. Каков порядок выполнения программы драйвера в Spark?
2. Что делает драйвер, когда задачи распределяются на исполнителей?
3. Как драйвер определяет, что программа завершена?

Executor и Heartbeat:

1. Что такое **executor** и какие задачи он выполняет в Spark?
2. Что такое механизм "сердцебиения" (**heartbeat**) и для чего он используется?
3. Что произойдет, если от исполнителя не поступает сигнал "сердцебиения" в течение заданного времени?

Спекулятивное выполнение:

1. Что такое **спекулятивное выполнение задач** в Spark?
2. Какие настройки отвечают за запуск спекулятивного выполнения задач?
3. В каких случаях целесообразно использовать спекулятивное выполнение?

Управление кластером:

1. Какую роль выполняет **менеджер кластера** в Spark?
2. Какие типы кластерных менеджеров поддерживает Spark?