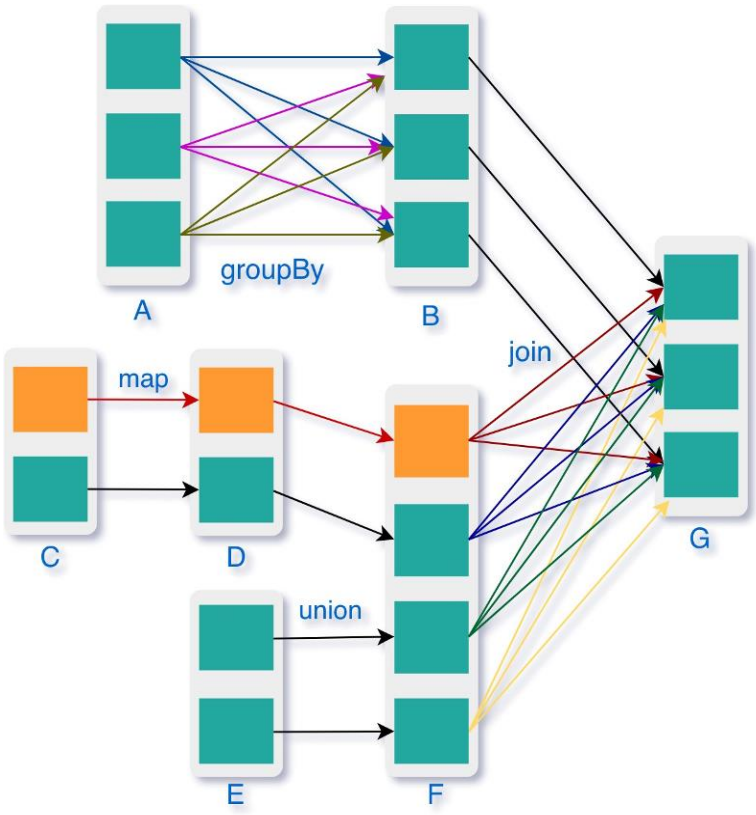
УНИВЕРСИТЕТ ИТМО

Н. А. Бутаков
М. В. Петров
Д. Насонов

# ОБРАБОТКА БОЛЬШИХ ДАННЫХ С APACHE SPARK




Санкт-Петербург
2019

**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**УНИВЕРСИТЕТ ИТМО**

Н. А. Бутаков
М. В. Петров
Д. Насонов

# ОБРАБОТКА БОЛЬШИХ ДАННЫХ С APACHE SPARK

Учебно-методическое пособие

РЕКОМЕНДОВАНО К ИСПОЛЬЗОВАНИЮ В УНИВЕРСИТЕТЕ ИТМО
по направлению подготовки 01.04.02 Прикладная математика и информатика
в качестве учебно-методического пособия для реализации образовательных программ высшего образования магистратуры

УНИВЕРСИТЕТ ИТМО

Санкт-Петербург
2019

Бутаков Н. А., Петров М. В., Насонов Д. Обработка больших данных с Apache Spark– СПб: Университет ИТМО, 2019. – 50 с.

Рецензенты:
Боченина Клавдия Олеговна, кандидат технических наук, старший научный сотрудник Национального Центра Когнитивных Разработок, доцент Института дизайна и урбанистики, Университета ИТМО.

Учебно-методическое пособие содержит теоретический материал и примеры выполнения задач для курса «Введение в технологии обработки больших данных». Пособие составлено с учётом проведения лабораторных работ с помощью фреймворка Apache Spark. Содержание дисциплины охватывает круг вопросов, связанных с организацией построения ETL-конвейеров на основе Spark SQL и DataFrame API для распределенного выполнения на кластерных вычислительных система, включая использование итеративных вычислений, важных для машинного обучения, рассмотрения shuffle механизмов и принципов организации управлением памятью в Spark. В результате освоения дисциплины студенты приобретают способности разработки программ и построения конвейеров обработки различных данных, навыки по работе с распределенными кластерными системами, а также способности к применению машинного обучения на распределенных наборах данных.

УНИВЕРСИТЕТ ИТМО

**Университет ИТМО** – ведущий вуз России в области информационных и фотонных технологий, один из немногих российских вузов, получивших в 2009 году статус национального исследовательского университета. С 2013 года Университет ИТМО – участник программы повышения конкурентоспособности российских университетов среди ведущих мировых научно-образовательных центров, известной как проект «5 в 100». Цель Университета ИТМО – становление исследовательского университета мирового уровня, предпринимательского по типу, ориентированного на интернационализацию всех направлений деятельности.

Содержание

# Введение

Современный мир и информационные системы являются ориентированными на данные, генерирующихся в огромных количествах каждый день. Поэтому для получения действительно ценных результатов в современных реалиях в промышленных, бизнес- или научных задачах очень важно уметь эффективно обрабатывать доступные нам большие данные, пользуясь доступными для этого инструментами. В данном пособии содержится теоретический материал, необходимый для понимания базовых основ того, как работает Apache Spark, а также набор практических примеров, позволяющих начать пользоваться данным фреймворком для решения задач обработки данных. Теоретический материал охватывает вопросы пакетной обработки больших данных и построения ETL-конвейеров на основе Spark SQL и DataFrame API для распределенного выполнения на кластерных вычислительных система, включая использование итеративных вычислений, важных для машинного обучения, рассмотрения shuffle механизмов и принципов организации управлением памятью в Spark. Практическая часть включает как примеры простых скриптов, позволяющие провести обработку с использование встроенных функций и 1-2 трансформаций, что позволит быстро начать новичкам работать с фреймворком, так и примеры комплексных скриптов, включающих конвейеры обработки данных, использование пользовательских функций обработки данных и broadcasting’a. В результате освоения дисциплины студенты приобретают способности разработки программ и построения конвейеров обработки различных данных, навыки по работе с распределенными кластерными системами; а также способности к применению машинного обучения на распределенных наборах данных.

# 1. Архитектура распределенного приложения Spark

Прежде чем мы начнем обсуждать, как устроен Spark, один из самых распространенных и популярных инструментов на текущий момент, нам необходимо ввести понятие кластера и узла, основных вычислительных средств, за счет которых и возможна обработка больших данных.

***Компьютерный кластер*** – это набор слабо или тесно связанных *узлов* (компьютеров), которые работают вместе и с некоторыми ограничениями могут рассматриваться как единая система [1].

***Узел*** – это отдельный компьютер, обладающий определенными вычислительными ресурсами (т.е. ядра CPU, оперативная память RAM) и ресурсами хранения данных – диски (HDD, SSD).

Отдельный узел может эффективно, за разумное время, обработать только определенную часть данных, но для обработки всего массива данных затратит неприемлемое, слишком большое время. Объединение узлов в кластер позволяет “горизонтально” масштабировать имеющуюся вычислительную мощность – т.е. параллельно читать и обрабатывать данные, которые теперь должны быть доступны по частям. Такой доступ к ним обеспечивается за счет распределенных файловых систем и хранилищ, зачастую совмещенных с самим вычислительным кластером – т.е. хранение данных, в идеале, происходит на дисках тех же узлов, что их и обрабатывают. Примерами таких систем являются: HDFS, HBase, ClickHouse, Cassandra и многие другие [2–4]. Не являясь напрямую предметом, рассматриваемым в данном пособии, такие системы могут участвовать в приводимых примерах, не сказываясь принципиально на понимании основного материала.

Однако, кроме очевидной пользы, объединение узлов в кластер и использование его влечет определенные трудности, связанные с его распределенностью:

- кластер имеет более сложную структуру, включая сетевую составляющую, но снижает требования к отдельным узлам, в результате чего, увеличивая производительность по сравнению с одним узлом, требует более сложной организации приложения, обрабатывающего данные;
- средства обработки и памяти больше не могут рассматриваться как однородные;
- требуется учитывать накладные расходы на связь между узлами.

В целом, для эффективного использования кластера требуется обеспечить его представление как единой системы, приблизив, насколько это возможно, работу с ним к работе с единым компьютером. Для достижения такого

представления и облегчения работы с кластером как раз и предназначены фреймворки типа Apache Spark.

Apache Spark может рассматриваться как:

- набор примитивов (т. е. фреймворк) для обслуживания всего цикла обработки данных, который используется для написания высокоуровневой логики обработки данных;
- универсальная платформа, предоставляющая различные средства и поддерживающая различные режимы обработки данных: пакетную обработку, потоковую обработку, sql-запросы, обработку графов;
- кластерное, распределенное приложение (в момент непосредственной обработки данных).

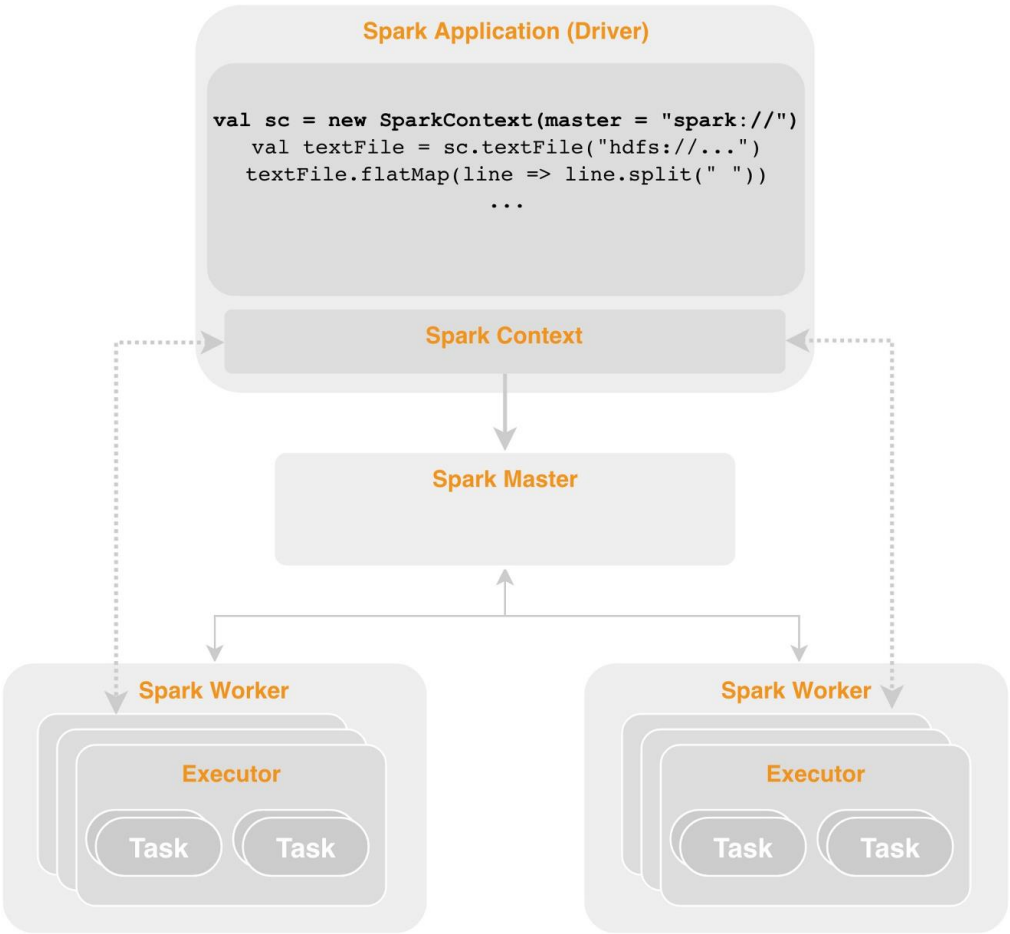



Рисунок 1.1 – Общая архитектура распределенного приложения Spark

Spark, являясь фреймворком для разработки распределенных приложений обработки данных, предполагает фиксированную master-slave [5] архитектуру таких приложений, проиллюстрированную на Рис. 1.1 Основными ее компонентами являются driver и executor.

Driver – компонент, ответственный за:

- отслеживание состояния обработки, генерацию задач и их перезапуск в случае отказов;
- оркестровку вычислений – назначение задач на вычислительные ресурсы (занимаемые с помощью executor’ов) с учетом их использования данных;
- контроль за вычислительными ресурсами, включая их выделение и возвращение менеджеру ресурсов;

Executor – компонент, ответственный за выполнение вычислений над данными на ресурсах вычислительного узла, где он располагается. В его задачи входит:

- контроль за расходом вычислительных ресурсов;
- хранение данных и предоставление доступа к ним, включая загрузку данных с диска или их десериализацию, для выполнения обработки;
- запуск выполнения задач и сохранение их результатов.

В приложении всегда один driver, который может располагаться как внутри кластера, так и снаружи его, и некоторое количество executor’ов (их может и не быть – в этом случае ничего выполняться не будет), располагающихся на узлах кластера и использующих вычислительные ресурсы, которые executor’ы предоставляют. К таким ресурсам относятся ядра процессора (CPU), оперативная память (RAM) и дисковое пространство (HDD).

Следует отметить, что driver обращается к своим executor’ам каждый раз, когда ему нужно запустить или перезапустить задачи, а также принимает и обрабатывает регулярные периодические отчеты от своих executor’ов, реализуемые через архитектурный паттерн heartbeat [6]. Поэтому не рекомендуется размещать driver на узле вне кластера, который имеет малую скорость соединения с кластером, например, на узле, отделенном от кластера сетью Интернет.

Современная обработка данных предполагает совместное использование кластера несколькими приложениями, в том числе несколькими разными приложениями Spark. Отдельные приложения Spark имеют абсолютно независимые наборы executor’ов, и executor не может переходить от одного драйвера к другому. Но executor’ы разных приложений могут располагаться на одних и тех же узлах кластера, если ресурсы узлов это позволяют.

И driver, и executor требуют для своего исполнения JVM [7], однако driver может становится частью других приложений, используясь как программная библиотека – в этом случае запуск приложения Spark происходит программно с помощью создания объекта специального класса SparkContext или SparkSession. Такой способ будет более подробно рассмотрен в следующем разделе. Executor всегда является отдельным JVM процессом, запущенным на узле кластера или в контейнере [8] на узле кластера.

В случае, когда приложение Spark является самостоятельным и имеет свой собственный jar файл с определенным Main классом, его можно запустить с помощью специального скрипта spark-submit из стандартного дистрибутива Spark [9].

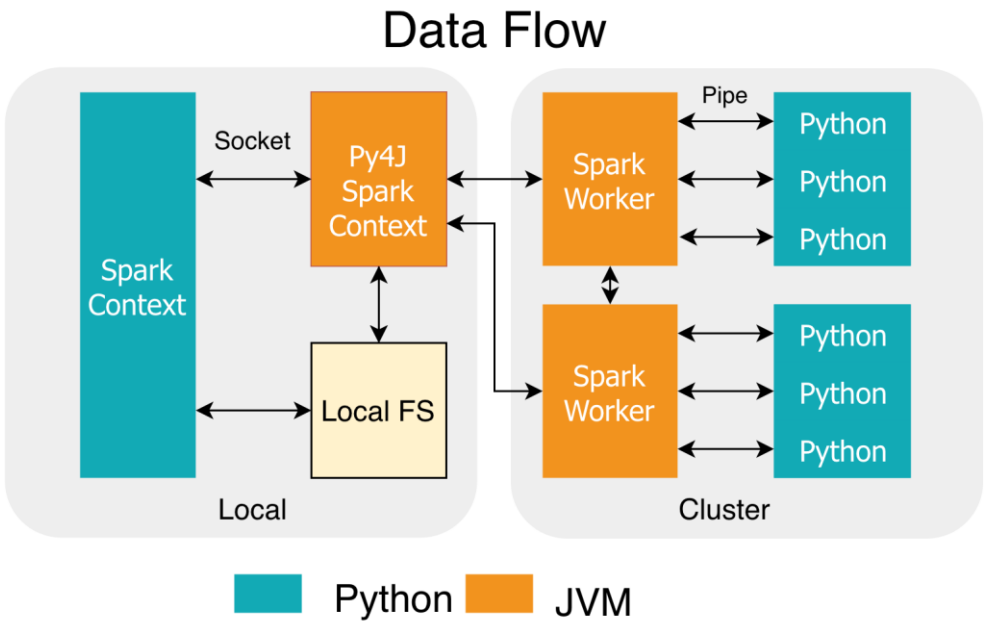



Рисунок 1.2 – Общая архитектура PySpark.

Следует отметить, что Spark – полиглотная технология, поддерживает различные языки программирования, например python и R. Эти языки имеют огромное количество уже готовых алгоритмов, функций, решений, например в ML, и поэтому целесообразно было добавить поддержу сторонних языков программирования.

Рассмотрим PySpark реализацию для языка Python. В рамках этого создаются отдельные процессы python-интерпретаторы. Данные для этих процессов передаются через механизм каналов (Pipe), а пользовательский код сериализуется в pickle формате на стороне python драйвера и затем десериализируется в python worker, после чего выполняется.

Если вы используете PySpark в рамках стандартного API, по скорости он действительно может быть сравним со Scala версией.

Однако, несмотря на все преимущества использования Python, накладные расходы (из-за необходимости перемещать данные между процессами и тратить ресурсы на интерпретацию) могут быть значимыми при использовании пользовательских функций, в частности, замедление может быть в несколько раз или даже на порядок[10].

# 2. Основные концепции Spark

## 2.1 RDD и граф преобразований

Для представления данных Spark использует концепцию RDD (Resilient Distributed Datasets). RDD – это абстракция набора данных, состоящего из записей одного произвольного типа и разделенного на части, которые размещены по всему кластеру. RDD содержит метаинформацию о наборе данных, но не сами данные, и хранится в driver приложения Spark во время его выполнения и только во время его выполнения. Непосредственно данные, разбитые на блоки, хранятся на executor'ах или во внешнем хранилище (если датасет не был еще загружен для обработки, т.е. так могут быть представлены, условно говоря, входные данные приложения Spark).

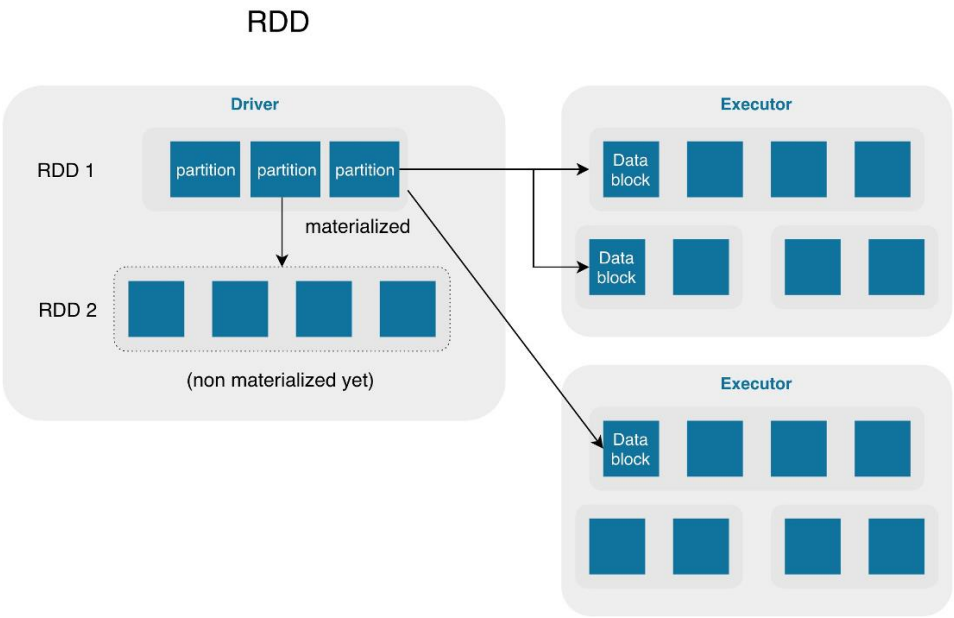



Рисунок 2.1 – RDD в Spark.

На рис. 2.1 приведена иллюстрация, поясняющая выше сказанное. RDD содержит метаинформацию следующего рода:

- набор партиций (partition), каждая из которых имеет свой порядковый идентификатор;
- зависимости на родительские RDD (которые могут быть двух принципиальных типов narrow и wide);
- функция, которая позволяет вычислить партиции текущего RDD, имея родительские наборы данных;
- partitioner – специальный объект, позволяющий определить в какую партицию следует отнести ту или иную запись из RDD (его роль более подробно будет прояснена в разделе 2.4);

– список предпочитаемых локаций для обработки каждой из партиций (обычно этот список содержит узлы, наиболее близкие к месту физического хранения партиции, например, список может содержать адрес DataNode распределенной файловой системы HDFS, которая содержит блок, представляемой партицией RDD).

Следует отметить следующие важные характеристики RDD.

RDD представляют некоторый шаг в обработке данных, но не обязательно ссылаются на конкретные данные, лежащие в хранилище или загруженные в кластер. За счет этого становится возможными представлять этапы обработки, которые случатся *когда-то* в будущем.

Связанные между собой зависимостями отдельные RDD вместе формируют план обработки данных, начинающийся, как правило, с загрузки данных из внешнего хранилища и завершающийся также выгрузкой уже обработанных (возможно многократно) данных во внешнее хранилище. Такой *не материализованный* план (граф определенной формы – ациклический направленный граф или DAG) за счет отсутствия ссылок на конкретные наборы данных в кластере может быть один раз сконструирован и многократно повторно использован, что, например, полезно при итеративных вычислениях алгоритмов машинного обучения. На рис. 2.2 приведен пример такого графа вычислений.

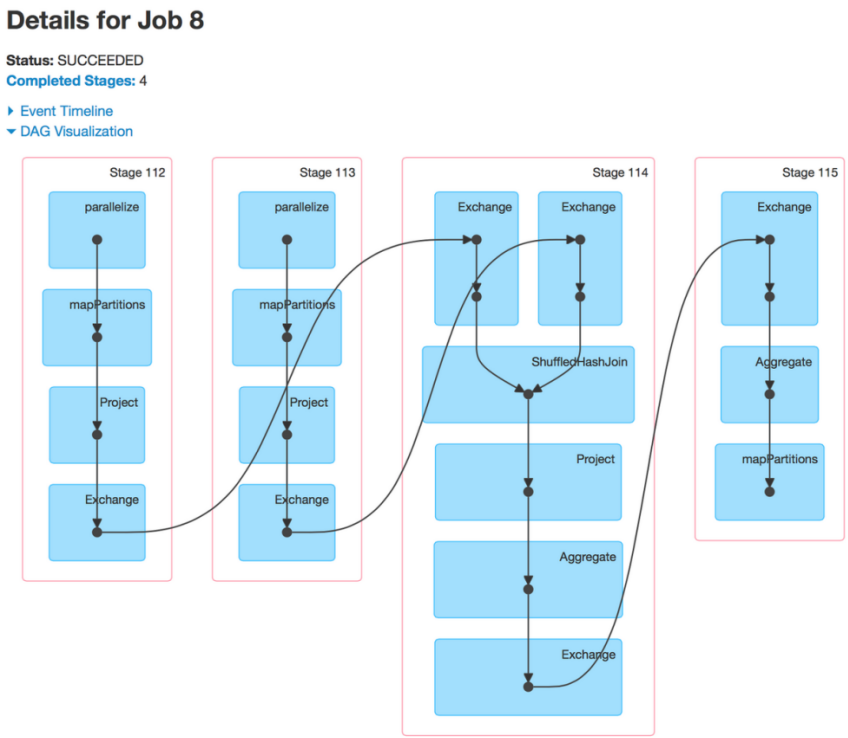



Рисунок 2.2 – Граф обработки данных, составленный из RDD (показаны синим цветом).

Построение плана – т.е. соединение одного RDD с другим с помощью объектов зависимостей – осуществляется с помощью специальных функций, называемых трансформации (transformations). Такая функция берет родительский RDD, генерирует дочерний RDD с определенной функцией обработки, соединяет дочерний RDD с родительским с помощью объекта зависимости и возвращает дочерний RDD для дальнейших манипуляций. Подробный список доступных трансформаций может быть найден [11]. Ниже, в листинге 2.1, приведен пример построения простого графа обработки в виде линейной цепочки действий, представляющих собой классическое приложение [i]WordCount.

Листинг 2.1 – Не материализованное приложение WordCount

```
val conf = new SparkConf()
  .setAppName("WordCount")
  .setMaster("local[*]")

val sc = new SparkContext(conf)

sc.textFile("build.sbt")
  .flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey{case (a,b) => a + b}
```

Для того, чтобы провести реальную обработку, согласно плану, который был построен с помощью RDD, необходимо его материализовать – т.е. отправить на выполнение с помощью специальных операций, называемых действиями (actions).

Листинг 2.2 – Материализация приложения WordCount

```
sc.textFile("build.sbt")
  .flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey{case (a,b) => a + b}
  .saveAsTextFile("/tmp/wc.txt")
```

За счет того, что при материализации мы знаем весь план выполнения обработки, у Spark имеется возможность ее оптимизировать, например, убрав

генерацию лишних промежуточных версий данных. Подробный список доступных действий может быть найден [11]. В листинге 2.2 приведен пример материализации приложения [ii]WordCount.

## 2.2 Основные этапы обработки данных

Все действия над данными, выполняемые фреймворком можно разделить на следующие этапы:

- загрузка данных в кластер из внешнего хранилища, что подразумевает опрос хранилища и определение количества партиций (в том числе преобразование внутреннего представления и партиций с данными в партиции Spark), формирование объектов задач и размещение их на соответствующих уровню локальности executor'ах;
- применение преобразований, определенных пользователем, над данными, что также в процессе требует управления размещением данных в оперативной памяти и на диске;
- применение служебных преобразований, результатами которых являются служебные файлы с данными, такие как shuffle-файлы;
- передача данных между узлами согласно формированию новых партиций – т.е. shuffle-операции. Такие операции могут происходить как в случае mapreduce – подобных вычислений, так и слияния двух датасетов или репартицирования исходного датасета;
- персистинг данных;
- выгрузка данных во внешнее хранилище.

Следует отметить сразу же несколько моментов относительно применения вычислений, как служебных, так и пользовательских. Запуск вычислений (и конструирование результатов или новых RDD) возможен для отдельных партиций – таким образом получают возможность эффективно работать операции head, take и sample. А также возможно частичное повторение вычислений, при котором потерянные партиции в уже материализованных RDD могут восстановлены независимо.

## 2.3 Загрузка данных из внешнего хранилища

Как происходит загрузка данных, а также сколько партиций будет создано в процессе нее, определяется адаптером конкретного хранилища. Примерами таких адаптеров могут служить [12, 13] . В целом, в типичные задачи такого адаптера входит следующее:

- выполнение служебных запросов для определения количества партиций и получения всей необходимой метаинформации (например, схемы таблиц или идентификация схемы с помощью сэмплирования json-документов);
- определение местоположения партиций (что особенно актуально, если кластер хранилища и вычислительный кластер являются одним целым);

– подготовка и оптимизация запросов, специфичных для хранилища, для выборки данных из него, включая передачу необходимых параметров для фильтрации на стороне хранилища (если хранилище позволяет это);
– трансфер данных из источника и их запись во внутреннее представление, которое сможет использовать для вычислений Apache Spark;

При чтении данных из хранилища сначала будет выполнена операция подсчета, что тоже может потребовать достаточно интенсивных вычислений. Особенное внимание тут следует уделять при работе с источниками, где схема данных не известна наперед: json-файлы в hdfs, данные в mongodb и т.п. Загрузка данных также является операцией вычислений над данными и происходит при материализации датасетов.

## 2.4 Изменение размещения данных и количества партиций

В процессе вычислений может возникнуть потребность в изменении размещения данных и / или количества партиций в датасете.

Например, после загрузки данных из файлов в hdfs мы хотим, чтобы все данные, принадлежащие одному и тому же пользователю, попали на один узел, так как все дальнейшие операции будут происходить только в рамках одного пользователя.

Чтобы добиться такого эффекта, мы можем воспользоваться функций repartition с указанием количества партиций или конкретного partitioner. В Spark по умолчанию доступно два основных partitioner, позволяющих добиться этой цели:

– HashPartitioner – размещает запись в партиции в соответствие с хэшом от ключа этой записи (актуально для key-value RDD).
– RangePartitioner – размещает запись в партиции в соответствии с диапазоном, в который попадает ключ данной записи (актуально для key-value RDD). Такой partitioner может быть полезен, например, в ситуации когда нам необходимо агрегировать данные по продажам за отдельные недели – тут RangePartitioner может помочь с размещением всех данных, относящихся к конкретной неделе только в одной партиции, тем самым уберая необходимость в реальной сетевой передачи данных.

В случае если RDD не назначен partitioner, распределение записей в партиции происходит равномерно. Отметим, что в случае DataFrame API репартицирование может быть применено к нескольким колонкам. Также существует возможность дописать свой собственный partitioner, например, для ситуации, когда нам имеет смысл разбивать данные и по идентификатору клиента, и по интервалу совершения операции – таким образом все данные одного пользователя за одну неделю окажутся в одной партиции. При этом, если пользователь имеет очень много активности, его обработку можно будет

распараллелить (так как партиция – это минимальная единица последовательной обработки).

Следует отметить, что размещать таким образом данные можно не только для одного датасета, а сразу для нескольких, например, имеющих один и тот же первичный ключ. В случае если нам понадобится проводить над ними операцию join (см. пример в листинге 3.7), за счет одинаковых partitioner'ов и соответственно одинаковому расположению партиций с теми же ключами на узлах, получится избежать сетевой передачи в операции shuffle (при условии одинакового количества партиций в обоих RDD).

## 2.5 Как происходит вычисление над данными в Spark

Непосредственно запуском вычислений над данными в Spark управляет DAGScheduler, находящийся в драйвере приложения и создаваемых вместе с объектом SparkContext. DAGScheduler отвечает за:

- создание выполняемого графа приложения;
- генерацию задач, назначение и их рассылку на экзекьютеры, а также за мониторинг хода их выполнения и предоставлению пользователю этой информации пользователю;
- отслеживание событий отказа и принятие решения о перезапуске или остановке вычислений, а также об игнорировании определенных экзекьютеров;

Выполняемый граф приложений отличается от графа преобразования данных из-за оптимизаций, применяемых DAGscheduler.

Материализованная форма графа состоит из стадий (stage). Стадия представляет собой последовательность RDD, связанных narrow зависимостями, функции которых объединены в одну единственную функцию обработки (это происходит с помощью так называемого volcano паттерна [14]. Результатом такой стадии является либо сетевой обмен данными между executor'ами и соответственно узлами кластера – т.е. операция shuffle, которая будет рассмотрена более подробно позднее – либо сохранение результатов во внешнее хранилище, либо, в определенных случаях, возвращение данных в driver приложения (см. действие collect).

Объединение функций обработки по narrow зависимостям позволяет не генерировать промежуточных наборов данных и таким образом избежать ненужных накладных расходов на запись на диск данных, их сериализацию / десериализацию (а если бы запись происходила в надежное внешнее хранилище, то еще и накладных расходов на репликацию).

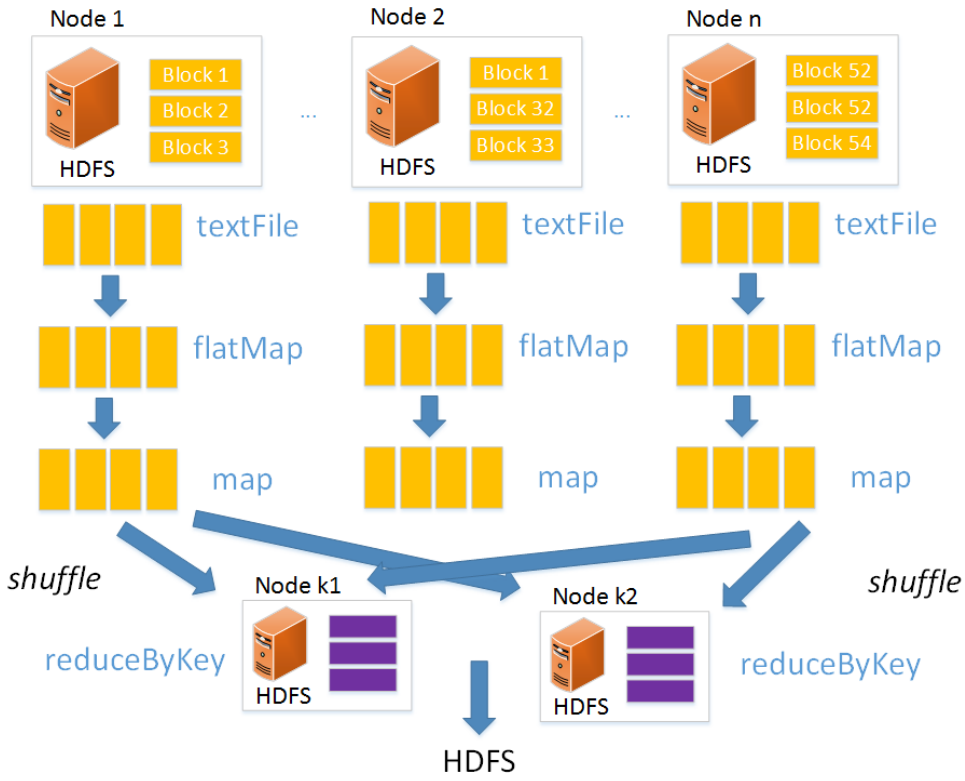
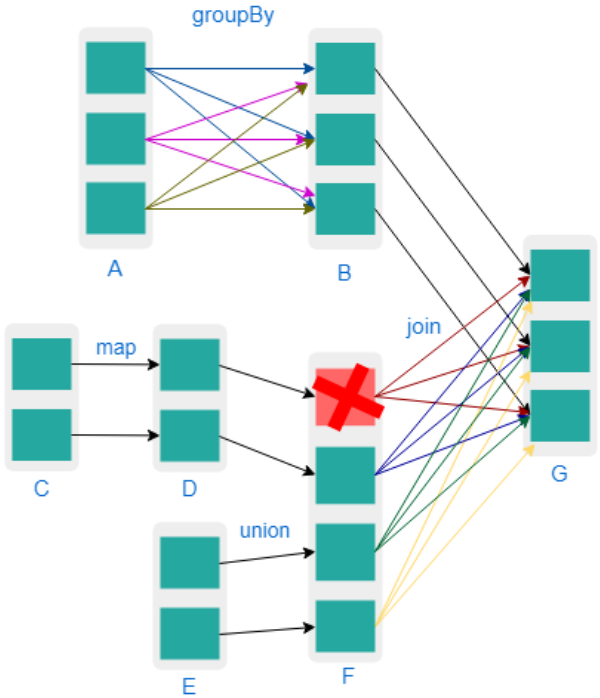



Рисунок 2.3 – Shuffle операция в результате трансформации reduceByKey в приложении WordCount

***Narrow зависимости*** – это зависимости, возникающие при операциях, в которых для получения партиции дочернего датасета нужна только одна партиция родительского датасета. Примерами таких операций служат map, filter, flatMap, mapPartitions – в них одна родительская партиция превращается в одну дочернюю.

***Wide зависимости*** – это зависимости, возникающее при операциях, в которых для получения партиции дочернего датасета нужны несколько или все партиции родительского датасета. Примерами таких операций служат groupByKey, reduceByKey, join, repartition. В примере, данном выше, как раз и используется такая операция для того, чтобы сгруппировать одни и теже слова для подсчета, но это может потребовать сетевой передачи данных, так как одни и те же слова могут встречаться в данных, лежащих на разных узлах. На рис. 2.3 показана такая ситуация.

а)




б)

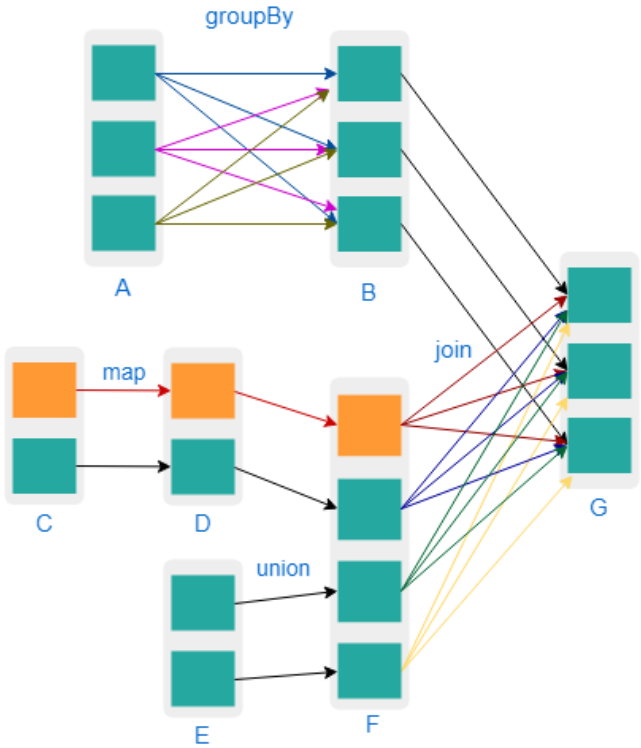



Рисунок 2.4 – Восстановление утраченных партиций по narrow зависимостям за счет частичного пересчета: (а) – потеря партиции в RDD F; (б) – восстановление партиции в RDD F за счет пересчета родителей данной партиции.

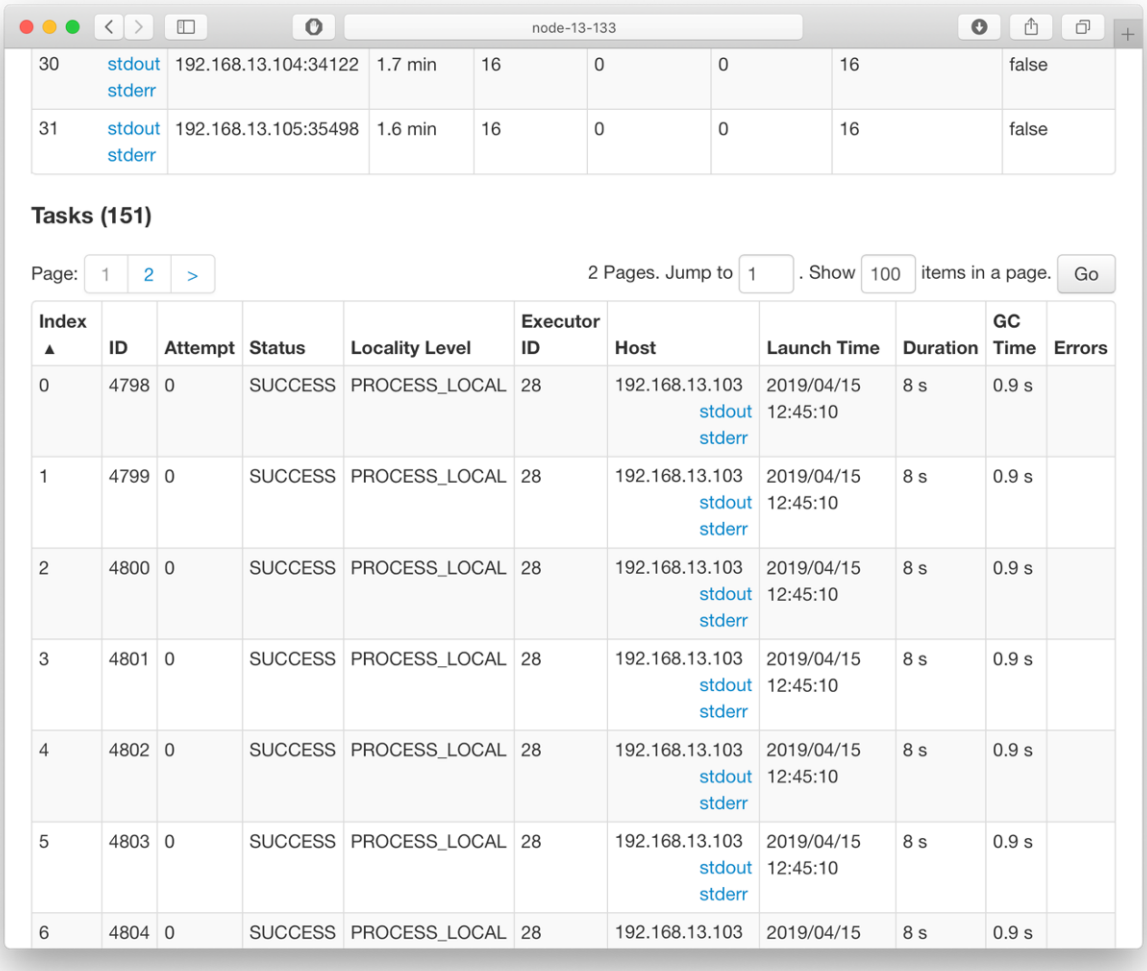



Рисунок 2.5 – Задачи и их статусы в пользовательском интерфейсе приложения Spark

*Narrow зависимости* не только более эффективны с точки зрения производительности, так как не предполагают дорогой сетевой передачи данных (а также подготовки к ней, включающей работу с диском и сериализацию, см. раздел 2.7), но и более надежны в ситуациях утраты части данных, так как в этом случае Spark позволяет пересчитать только утраченные партиции из неутраченных родительских партиций (в случае, если какой-то промежуточный RDD был закэширован) или даже партиций, получаемых из внешнего хранилища в самом начале обработки. На рис. 2.4 проиллюстрировано такое восстановление.

В случае же wide зависимостей одна дочерняя партиция может зависить от всех родительских, что и происходит в случае reduceByKey предыдущего примера. В таком случае для восстановления нескольких утраченных партиций дочернего RDD придется восстанавливать все партиции родительских RDD, если они тоже утрачены, что может быть очень затратно. Чтобы избежать таких

проблем, в случае wide зависимостей рекомендуется использовать checkpointing.

Стадия состоит из задач (task), каждая из которых генерируется на партицию в датасете.

Задачи затем рассылаются по экзекьютерам для их выполнения. Каждая из задач при этом содержит:

- сериализованную функцию (может состоять из последовательного набора заданных пользователем преобразований или сервисных преобразований) – на стороне executor'а она будет десериализованна и ей будет предоставлен итератор к коллекции данных при выполнении;
- информацию о партиции, над данными которой необходимо выполнить вычисления;
- порядковые идентификаторы самой задачи и текущей попытки.

Следует отметить, что задача также может содержать необходимые операции (согласно тому, что реализовано в адаптере к хранилищу) по загрузке данных из хранилища. В этом случае итератор не будет содержать данных, а задача сама соединиться с хранилищем и загрузит данные. Кроме хранилища, источником данных для задач могут являться закэшированные данные, данные shuffle, загруженные с удаленной машины. и данные чекпоинтов.

## 2.6 Ветвление и итеративные вычисления

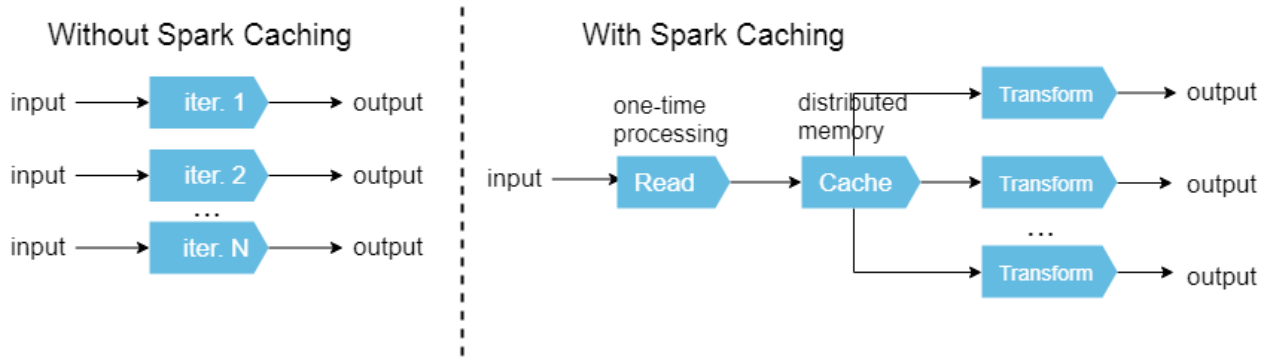



Рисунок 2.6 – Иллюстрация ветвления вычислений с помощью кэширования.

Возможность выполнять итеративные вычисления и обработку данных, повторно использующую те же самые данные, без дорогостоящей записи промежуточных результатов во внешнее хранилище – накладные расходы на сериализацию / десериализацию, запись на диск и чтение с диска, а также репликацию – позволяет существенно ускорить процесс выполнения (особенно по сравнению с классическим фреймворком Hadoop). Повторное использование возможно за счет механизма кэширования (caching / persisting), имеющегося в Spark. Кэширование осуществляется с помощью вызова специальных трансформаций cache() или persists () (в последнюю нужно передавать уровень, на котором будут сохраняться данные: память; диск; одновременно память и

диск; уровни с сериализацией). Трансформации не приводят к немедленной материализации RDD. Вместо этого при первой материализации данные будут сохранены на соответствующем уровне и не будут повторно рассчитаны при следующих обращениях к ним, что проиллюстрировано на рис. 2.6. Пример использования кэширования приведен в листинге 3.5.

## 2.7. Shuffle механизм

Процесс shuffling является одним из фундаментальных, т. к. позволяет изменять размещение данных на узлах, лежа в основе операций объединения (join) и группировки (group by, reduce) данных. Для задач машинного обучения shuffle является неотъемлемой частью, например, для алгоритма Expectation Maximization или word2vec.

На вход shuffle поступает RDD с n партициями, а выходом будет RDD с m партициями. В силу исторических причин мы будем называть входной RDD map-стороной, а выходной – reduce-стороной. В Spark количество выходных партиций не зависит напрямую от количества уникальных ключей в конкретном датасете, а определяется параметром spark.sql.shuffle.partitions, который задается в настройках SparkContext. Каждая запись из партиций первого RDD будет оценена по некоторому ключу, и относительно него будет назначена партиция из выходного RDD, в которую эта запись попадет. Таким образом, все записи с одинаковыми ключами попадут в одну и ту же партицию выходного RDD. На рис 2.7 приведена иллюстрация данного процесса.

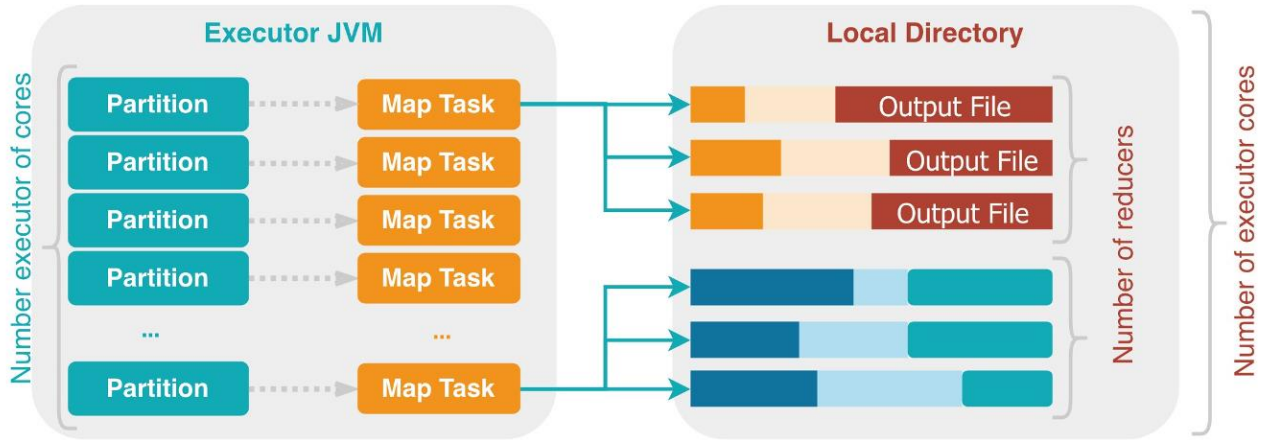



Рисунок 2.7 – Shuffle – процесс изменяющий размещение данных

Для реализации shuffle необходима подготовка данных на map-стороне – группировка входных записей по их ключам, чтобы их затем можно было отослать на узел с нужной партицией.

Существует два основных способа, как это можно сделать. Первый способ (hash shuffle) подразумевает использование отдельного набора файлов для каждой map-задачи. Его оптимизированная версия, используемая в текущих

версиях Spark, подразумевает переиспользование набора файлов для каждого из слотов.

Sort Shuffle используют другую логику обработки. Hash shuffle на выходе производит по одному отдельному файлу для каждой выходной партиции и, как следствие, для каждого из «редьюсеров». С помощью же sort shuffle возможно сократить их количество: выходные файлы упорядочены по id «редьюсера» и и содержат индекс в самом начале, состоящий из указателей позиций (offsets) на начало блока записей с конкретным id. Это позволяет легко получить блок данных, относящихся к «редьюсеру x», просто используя информацию о положении связанного блока данных в файле. Но, конечно, для небольшого количества «редьюсеров» очевидно, что хеширование отдельных файлов будет работать быстрее, чем сортировка, поэтому сортировка в случайном порядке имеет план «отката»: когда количество «редьюсеров» меньше, чем «spark.shuffle. sort.bypassMergeThreshold» (по умолчанию 200), используется hash shuffle (см. рис. 2.8).

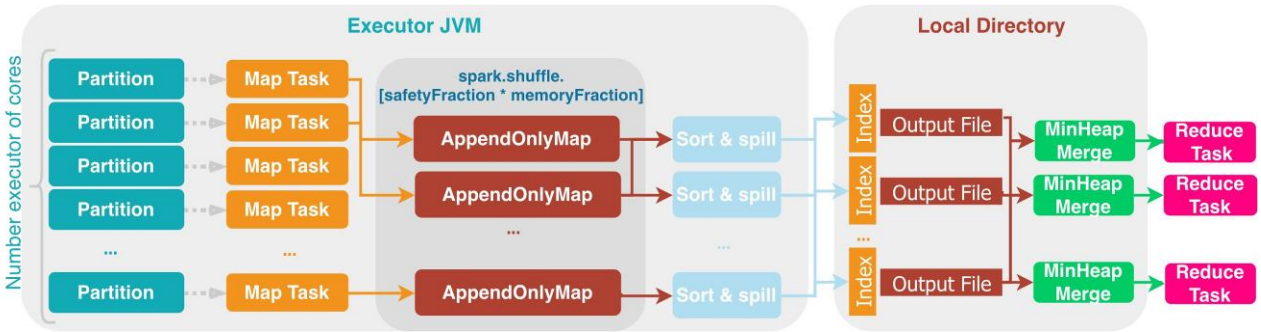



Рисунок 2.8 – Sort Shuffle – процесс, изменяющий размещение данных

## 2.8 Управление памятью в Apache Spark

Модель памяти Spark предполагает наличие 3 основных областей памяти: Reserved Memory, User Memory, Spark Memory. Все они изображены на как на рис. 2.9:

**Зарезервированная память (Reserved Memory)**. Это память, зарезервированная системой, и ее размер жестко закодирован. Начиная с версии Spark 1.6.0, его значение составляет 300 МБ, что означает, что эти 300 МБ RAM не участвуют в вычислениях размера области памяти Spark, и его размер нельзя изменить каким-либо образом без перекомпиляции Spark или установки spark.testing.reservedMemory, что не рекомендуется, так как это параметр тестирования, не предназначенный для использования во время реальных вычислений.

**Пользовательская память (User Memory)**. Это пул памяти, который остается после выделения Spark Memory, и может быть использован для хранения структуры данных пользователя, которые создаются в процессе

вычислений с помощью трансформаций RDD и функций, задаваемых определяемых пользователем (user-defined function, UDF).

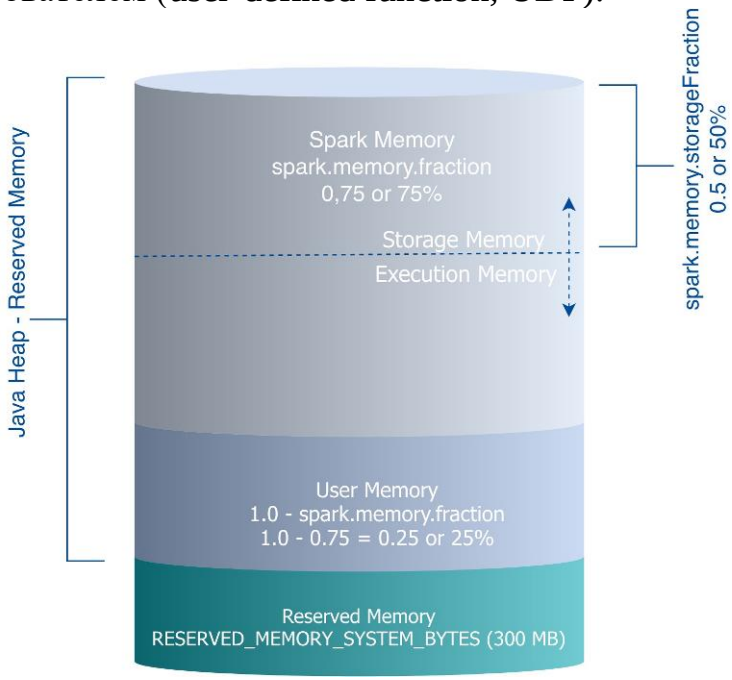



Рисунок 2.9 –Структура оперативной памяти executor'а приложения Spark

Например, вы можете переписать агрегацию Spark, используя хеш-таблицу, и использовать трансформацию mapPartitions для запуска этой агрегации, которая будет потреблять пользовательскую память. В Spark 1.6.0 размер этого пула памяти можно рассчитать следующим образом:

**(Java Heap – Reserved Memory) × (1 – spark.memory.fraction)**

Параметр ***spark.memory.fraction*** по умолчанию равен 0,25.

Например, если Java Heap равен 4 ГБ, то размер пользовательской памяти будет 949 МБ. Стоит отметить, что то, как используется пользовательская память и что там хранится, полностью зависит от самого пользователя. Spark не будет учитывать, соблюдается ли эта границу или нет, что может привести к ошибкам, связанным с перерасходом памяти.

**Память Spark (Spark Memory).** Это пул памяти, управляемый самим Spark. Его размер можно рассчитать следующим образом:

**(Java Heap – Reserved Memory) × spark.memory.fraction**

Например, если размер Java Heap в 4 ГБ, то этот пул будет иметь размер 2847 МБ. Весь этот пул разделен на 2 области - *память хранения и память выполнения (Storage Memory and Execution Memory)*, и граница между ними задается параметром spark.memory.storageFraction, который по умолчанию равен 0.5. Преимущество этой новой схемы управления памятью состоит в том, что эта граница не является статической, и в случае нехватки памяти граница

будет перемещаться, то есть одна область будет расти за счет заимствования пространства у другой.

***Память хранения (Storage memory)***. Этот пул используется как для хранения кэшированных данных Spark, так и для временного развертывания сериализованных данных. Также все Broadcast переменные хранятся там как кешированные блоки с уровнем персистентности MEMORY_AND_DISK по умолчанию.

***Память исполнения (Execution Memory)***. Этот пул используется для хранения объектов, необходимых во время выполнения задач Spark. Например, он используется для хранения промежуточного буфера на map этапе, а также для хранения хеш-таблицы для этапа агрегации по хэшу. Этот пул также поддерживает размещение данных на диске, если недостаточно памяти.

# 3. DataFrame API и Spark SQL

## 3.1 Датафреймы

RDD API, трансформации и действия которого были рассмотрены выше, является достаточно низкоуровневым и используется в основном только для реализации некоторых алгоритмов машинного обучения (например, там, где нужен полный контроль над происходящим и применение встроенного оптимизатора Catalyst не желательно), а также для поддержки унаследованных программ обработки данных (legacy code). Основным API на данный момент (версия Spark 2.4.1) является DataFrame API.

Основные отличия этого API перечислены ниже.

- Представление датасета и его трансформация осуществляется с помощью специальных объектов, называемых датафреймами. Датафрейм – это таблица со столбцами, а не просто набор записей, как это было в случае с RDD.
- Датафреймы с их схемами (метаинформация в виде описания колонок и их типов), доступными на каждом шаге обработки, позволяют реализовать структурированную обработку данных (structured processing). В этом случае ядро Spark имеет доступ к информации не только о типе всей записи, но и о каждом конкретном столбце и может использовать ее, а это, в свою очередь, может позволить оптимизировать работу с ними (см. примеры ниже).
- Формат хранения данных изменен с ориентированного на ряды (row-wise) на колоночно-ориентированный (column-wise). Это позволяет добиться более эффективного хранения данных за счет сжатия по отдельным колонкам, а также более эффективной выборки и обработки данных за счет векторных возможностей современных процессоров (SIMD, наборы инструкций SSE 4.1 и SSE 4.2). Для более подробной информации рекомендуется обратиться к документации проекта Tungsten [14].
- API обработки данных, предоставляемое конечному пользователю, становится SQL-подобным (в том числе возможно использование не только языков программирования Scala, Java, Python, R, но и самого SQL), что имеет ряд преимуществ: оно проще для пользователя и позволяет быстрее ознакомиться с фреймворком; оно увеличивает повторное использование кода за счет функций из стандартной библиотеки Spark SQL (см. модуль [15]) и упрощает написание программ обработки; оно служит повышению эффективности программ, так как функции из стандартной библиотеки оптимизированы для работы с новым представлением данных (в отличие от функций, определяемых пользователем, т.е. user-defined function, UDF) и могут, например, использовать встроенную возможность компиляции для конвейера из применяемых последовательно функций.

– При материализации весь полученный сценарий обработки за счет специального встроенного оптимизатора Catalyst [16] приводится из SQL-подобной формы к исполнимой форме графа, состоящей из стадий и рассмотренной ранее. В процессе этого приведения, детальный анализ которого не входит в задачи данного пособия и более подробное ознакомление с ним остается на усмотрение самого читателя, применяются различные техники оптимизации, использующие информацию о структуре датафреймов и ее изменении, например: удаление лишних проекций и трансформаций (если результирующие столбцы не используются в конечном датафрейме); сокращение столбцов, читаемых с внешних хранилищ (избегается дорогое дисковое чтение) и не использующихся в обработке; осуществление предварительной фильтрации данных на стороне внешнего хранилища, если оно позволяет это (так называемый predicate pushdown), что ведет к уменьшению читаемых с диска данных, уменьшению количества операций сериализации/десериализации, а также уменьшению сетевой передачи данных.

Следует отметить, что DataFrame API не отменяет использование RDD при вычислениях, так как датафреймы транслируются в такую же последовательности RDD, связанных narrow и wide зависимостями.

## 3.2 Начало работы с DataFrame API: SparkSession

Далее мы рассмотрим примеры работы с DataFrame API в рамках построения сценариев обработки данных на Spark. Использование DataFrame API начинается с создания специального объекта SparkSession, служащего входной точкой для дальнейших операций. Подробную информацию об этом объекте можно найти в [17], а пример его создания приведен в листинге 3.1.

Листинг 3.1 – Создание входной точки DataFrame API - объекта SparkSession

```
val spark = SparkSession.builder
  .appName("user-traces")
  .master("local[*]")
  .config("spark.executor.cores", "8")
  .config("spark.executor.instances", "4")
  .config("spark.cores.max", "128")
  .config("spark.executor.memory", "15g")
  .config("spark.sql.shuffle.partitions", "2000")
  .getOrCreate()
```

SparkSession не заменяет, а дополняет собой базовый объект SparkContext. Последний все еще может быть доступен для получения и использования с помощью обращения к соответствующему полю SparkSession: spark.sparkContext. Как и в SQL, DataFrame API позволяет производить

проекции и агрегации над данными с помощью трансформаций с соответствующими названиями. Пример работы с ними приведен в листинге 3.2.

Листинг 3.2 – Пример агрегации данных с помощью DataFrame API с использованием встроенных в Spark SQL агрегационных функций

```
var followersDf=spark.read.parquet("/storage/followers.parquet")

// мы будем использовать только 3 колонки
// это может позволить не читать остальные колонки из хранилища
//если адаптер к хранилищу позволяет произвести такую оптимизацию
followersDf = followersDf.select("profile", "key","follower")

  // группируем по уникальным follower
//  считаем по группам количество
//  выходная схема: follower, following_count
import org.apache.spark.sql.functions.count
val following = followersDf
        .groupBy("follower")
        .agg(count("profile").alias("following_count"))

// пишем обратно в хранилище
following.write.parquet("/storage/counted_followers.parquet")
```

В рассмотренном примере для агрегации данных, а именно подсчета количества фолловеров, используется функция *count* из стандартной библиотеки Spark SQL. Полный список всех доступных функций может быть найден здесь [18].

Как было отмечено выше, информация о структуре результирующего датафрейма может быть получена на каждом шаге обработки с помощью специальных служебных методов: df.schema или df.printSchema(). На рис. 3.1 представлены схемы датафреймов из рассмотренного скрипта.

Важной составляющей обработки с помощью DataFrame API (да и важной составляющей обработки больших данных в целом) является работа с несколькими различными таблицами, принадлежащих одному или нескольким датасетам. В частности, это операции объединения данных и фильтрации одного датафрейма на основе данных из другого датафрейма. Рассмотрим пример из листинга 3.3, посвященный этим двум операциям. Фильтрация здесь осуществляется исключительно за счет использования операции join специального типа. Следует отметить, что результат в случае отсутствия операции distinct был бы тем же самым, но выполнялся бы медленнее, из-за необходимости обработки и передачи лишних данных.

а)

```
followersDf.printSchema()
```

```
root
 |-- follower: long (nullable = true)
 |-- key: string (nullable = true)
 |-- profile: long (nullable = true)
```

б)

```
followingDf.printSchema()
```

```
root
 |-- follower: long (nullable = true)
 |-- following_count: long (nullable = false)
```

Рисунок 3.1 – Схемы датафреймов: (а) followersDf; (б) followingDf

Листинг 3.3 – Работа в DataFrame API с несколькими таблицами: фильтрация одного датафрейма за счет данных из другого с помощью операции join.

```
val traces = spark.read.parquet("/storage/users.parquet")
  .where("uid > 0")

val userWallPosts = spark.read.parquet("/storage/wall_posts.parquet")

//   оставляем только одну колонку, состоящую из уникальных значений
val uids = traces.select("uid").distinct()

// фильтруем первую таблицу (левую) с помощью 2-ой
// используя left_semi join
// только записи левой таблицы, имеющие соответствующие правой таблице
// owner_id останутся
// колонки правой таблицы не будут присутствовать
// в результирующей таблице

userWallPosts.join(
  uids,
  userWallPosts.col("owner_id") === userWallPosts.col("uid"),
  "left_semi"
)
.write.parquet("/storage/filtered_wall_posts.parquet")
```

Обратим внимание на одну особенность, принципиальную для эффективного выполнения данной программы. Для начала получим explain обработки, вызвав соответствующий метод у результирующего датафрейма: userWallPostsDf.explain(). В листинге 3.4 приведен вывод этой команды. В корне дерева физического плана исполнения (а именно его выдает эта команда) можно видеть операцию SortMergeJoin. Эта операция описывает конкретный тип операции join, который будет применен в данном случае. В данном случае операция будет выполнена с помощью Sort Shuffle, рассмотренного выше.

Достоинством такого типа операции join является то, что она может работать с датасетами любого размера и любыми условиями join. Однако это может потребовать передачи данных каждого из двух датафреймов на одни и те же узлы, где они и будут объединены. Определение “совместных” партиций происходит на основе хэширования полей записей, по которым производится объединение.

Листинг 3.4 – Результат команды explain для скрипта из листинга 3.3

```
== Physical Plan ==
SortMergeJoin [owner_id#818L], [uid#0L], LeftSemi
:- Sort [owner_id#818L ASC NULLS FIRST], false, 0
:  +- Exchange hashpartitioning(owner_id#818L, 2000)
:     +- Project [...]
:        +- Filter isnotnull(owner_id#818L)
:                                 +- FileScan parquet […] Batched: false, Format: Parquet, Location: InMemoryFileIndex[/storage/wall_posts.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(owner_id)], ReadSchema:
struct<attachments:array<struct<album:struct<created:bigint,description:string,id:string,owner_id...
+- *Sort [uid#0L ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(uid#0L, 2000)
      +- InMemoryTableScan [uid#0L]
         +- InMemoryRelation [uid#0L], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
            +- *Project [uid#0L]
               +- *Filter (isnotnull(uid#0L) && (uid#0L > 0))
                  +- *FileScan parquet [uid#0L] Batched: true, Format: Parquet, Location: InMemoryFileIndex[/storage/users.parquet, PartitionFilters: [], PushedFilters: [IsNotNull(uid), GreaterThan(uid,0)], ReadSchema: struct<uid:bigint>
```

В случае, если один датасет существенно больше другого, а в приведенном в листинге 3.3 примере именно такой случай, выполнение join может быть оптимизировано за счет применения другого типа этой операции – BroadcastHashJoin. В этом случае меньший датасет (если он достаточно маленький, что определяется параметром spark.sql.autoBroadcastJoinThreshold) будет целиком собран с узлов в драйвер приложения Spark, для него будет построена HashMap структура данных (ключ – кортеж по значениями колонок, по которым происходит объединение, значение – соответствующая запись из малого датасета), и ее копия однократно будет разослана на все имеющиеся executor’ы. Объединенный датафрейм будет получен путем фильтрации большего датасета по ключам из этой HashMap и объединения с записями из нее в случае совпадения ключей. В листинге 3.5 приведена модификация предыдущей программы за счет использования кэширования (функция cache()) и материализации меньшего датасета. Последнее необходимо для того, чтобы Spark оценил размеры обоих датасетов и убедился, что один из них подпадает под ограничение spark.sql.autoBroadcastJoinThreshold, что позволит ему заменить тип join в физическом плане выполнения.

Листинг 3.5 – Изменение типа join за счет получения дополнительной информации через материализацию одного из датафреймов.

```
val uids = traces.select("uid").distinct().cache()
uids.count()

userWallPosts
  .join(uids,
    userWallPosts.col("owner_id") === userWallPosts.col("uid"),
    "left_semi"
  )
  .write.parquet("/storage/filtered_wall_posts.parquet")
```

Физический план для модифицированной программы из листинга 3.5 представлен в листинге 3.6.

Листинг 3.6 – Результат команды explain для скрипта из листинга 3.5

```
== Physical Plan ==
BroadcastHashJoin [owner_id#818L], [uid#0L], LeftSemi, BuildRight
:- Project [attachments#805, can_delete#806L, collected_timestamp#807, comments#808, date#809L, final_po
geo#812, id#813L, is_pinned#814L, key#815, likes#816, marked_as_ads#817L, owner_id#818L, post_sou
reply_owner_id#821L, reply_post_id#822L, reposts#823, signer_id#824L, text#825, is_reposted#826, repost_info#82
:  +- Filter isnotnull(owner_id#818L)
:     +-
FileScan
[attachments#805,can_delete#806L,collected_timestamp#807,comments#808,date#809L,final_post#810L,from_id#81
pinned#814L,key#815,likes#816,marked_as_ads#817L,owner_id#818L,post_source#819,post_type#820,reply_owner_
822L,reposts#823,signer_id#824L,text#825,is_reposted#826,repost_info#827] Batched: false, Format:
InMemoryFileIndex[/storage/wall_posts.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(own
struct<attachments:array<struct<album:struct<created:bigint,description:string,id:string,owner_id...
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, true]))
   +- InMemoryTableScan [uid#0L]
         +- InMemoryRelation [uid#0L], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
               +- *Project [uid#0L]
                  +- *Filter (isnotnull(uid#0L) && (uid#0L > 0))
                     +- *FileScan parquet [uid#0L] Batched: true, Format: Parquet, Location:
InMemoryFileIndex[/storage/users.parquet, PartitionFilters: [], PushedFilters: [IsNotNull(uid), GreaterThan(uid,0)],
ReadSchema: struct<uid:bigint>
```

## 3.3 Использование пользовательских функций (UDF)

Ниже, в листинге 3.7, приведен пример более сложной программы обработки данных, использующей еще 2 принципиальные возможности Spark:

- возможность пользователю писать собственные функции обработки (user – defined functions или UDF) в дополнение к тому, что есть в стандартной библиотеке Spark;
- использование broadcast переменных для работы с крупными объемами данных без замыкания их в UDF, что позволяет избежать расходов на дорогостоящие сериализацию и передачу данных, так как каждая задача обладает собственной копией используемой функции и всех ее данных.

Листинг 3.7 – Пример обработки данных с использованием UDF и broadcast-переменных (приведено для PySpark, но код для Spark существенно не отличается).

```
  filename = "cleaned-comm-words-all.txt"

# чтение файлов с данными, нужными для обработки
# (данные существенного объема)
  with open(filename, "r", encoding="utf-8") as f:
    phrases = f.readlines()

  phrases = {ph.lower().strip() for ph in phrases}
  phrases = {ph: i for i, ph in enumerate(phrases)}

  iph_to_name = {i: "{}_phrase_{}".format(file_to_method[filename], i)
            for ph, i in phrases.items()}

# создание broadcast переменных
# в этот момент реальные данные рассылаются по executor'ам
  phrasesBcs = spark.sparkContext.broadcast(phrases)
  iphToNameBcs = spark.sparkContext.broadcast(iph_to_name)

# функции обработки данных, определенные пользователем
# пишутся на языках поддерживаемых Spark (Scala, Java, Python, R)
# могут использовать любые библиотеки из экосистемы языка
# функция может использовать больше чем одну колонку
# из DataFrame, к которому она будет применена
# важен порядок в котором ей будут передаваться колонки
# в момент применения
  def correct_text(is_reposted, text, repost_info):
    return repost_info.orig_text if is_reposted else text

# следующие 2 функции используют
# ранее созданные broadcast переменные для обработки
# для этого созданные переменные "замыкаются" в функции обработки
# это возможно благодаря тому, что broadcast переменная не содержит
# сами данные, а только идентификатор,
# по которому их можно будет получить во время выполнения на executor
  def check_appearance(text):
    phrasesDict = phrasesBcs.value
    foundPhrases = [i for ph, i in phrasesDict.items() if ph in text.lower()]-
    return foundPhrases

  def calculate_appearance(found_phrases):
    iph_to_name_dict = iphToNameBcs.value
    calcOfAppearance = {i: 0 for i, name in iph_to_name_dict.items()}
    for ph_in_post in found_phrases:
        for iph in ph_in_post:
            calcOfAppearance[iph] += 1
    return {iph_to_name_dict[iph]: count for iph, count in calcOfAppearance.items()}

# создание UDF-ов путем регистрации их в драйвере
# их сериализованное представление будет рассылаться
# вместе с task'ами на executor'ы
# необходимо указывать возвращаемый тип данных
correctText = udf(correct_text, returnType=StringType())
checkAppearance = udf(check_appearance, returnType=ArrayType(IntegerType()))
calculateAppearance = udf(

    calculate_appearance,
    returnType=MapType(StringType(), IntegerType())
)

# применение UDF - функций для трансформации DataFrame'ов
```

```
userWallCheckedPhrasesDf = userWallPostsDf \
   .select("owner_id", correctText("is_reposted", "text", "repost_info").name("text"))

userWallCheckedPhrasesDf = userWallCheckedPhrasesDf \
   .select("owner_id", checkAppearance("text").name(PH_CHECK_COL))

userWallCheckedPhrasesDf.write.parquet("/storage/wall_phrases.parquet")
```

## 3.4 Пользовательские функции агрегации

В Spark имеется возможность задавать пользовательские функции агрегации. В этом случае необходимо унаследовать объект типа UserDefinedAggregateFunction и реализовать следующие методы:

- bufferSchema
- datatype
- deterministic
- initialize
- update
- merge
- evaluate

После этого нужно будет создать экземпляр объекта пользовательской функции агрегации, зарегистрировать его через spark.udf.register и затем использовать в функции agg из стандартной библиотеки. Пример показан в листинге 3.8.

Листинг 3.8 – Пример обработки данных с использованием UDAF функций.

```
import org.apache.spark.sql.expressions.MutableAggregationBuffer
import org.apache.spark.sql.expressions.UserDefinedAggregateFunction
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._

class GeometricMean extends UserDefinedAggregateFunction {

 // входные типы
 override def inputSchema: org.apache.spark.sql.types.StructType =
   StructType(StructField("value", DoubleType) :: Nil)

 // внутренние типы для агрегации
 override def bufferSchema: StructType = StructType(
   StructField("count", LongType) :: StructField("product", DoubleType) :: Nil
 )
```

```
  // Выходной тип
  override def dataType: DataType = DoubleType

  override def deterministic: Boolean = true

  // начальные значения для буффера.
  override def initialize(buffer: MutableAggregationBuffer) {
    buffer(0) = 0L
    buffer(1) = 1.0
  }

  // обновление буфера
  override def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
    buffer(0) = buffer.getAs[Long](0) + 1
    buffer(1) = buffer.getAs[Double](1) * input.getAs[Double](0)
  }

  // слияние
  override def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
    buffer1(0) = buffer1.getAs[Long](0) + buffer2.getAs[Long](0)
    buffer1(1) = buffer1.getAs[Double](1) * buffer2.getAs[Double](1)
  }

  // окончательное значение, по окончательному значению из буфера
  override def evaluate(buffer: Row): Double = {
    math.pow(buffer.getDouble(1), 1.toDouble / buffer.getLong(0))
  }
}

//    создание инстанса UDAF
val gm = new GeometricMean

//    регистрация UDAF
spark.udf.register("gm", gm)

userWallPosts.groupBy("group_id")
   .agg(gm($"id").as("GeometricMean"))
    .show()
```

# 4. Создание, настройка и запуск Spark проекта

Для работы со Spark потребуется установить и настроить следующее программное обеспечение: JDK, Maven, Intellij Idea. JDK позволяет разработчикам создавать java-программы, которые могут выполняться непосредственно в JVM. Maven – это инструмент для сборки java проектов и управления различными java библиотеками. Intellij Idea – популярная среда разработки программного обеспечения для различных языков программирования, в частности для Java и Scala. В данной главе подробно рассматривается настройка окружения для ОС Windows и ОС Linux, создание и настройка нового Scala проекта, запуск просто приложения spark в локальном режиме.

## 4.1 Настройка окружения

Для начала необходимо скачать JDK 8 (проверьте последнюю поддерживаемую Spark версию) для вашей версии ОС: http://www.oracle.com/technetwork/java/javase/downloads/index.html.

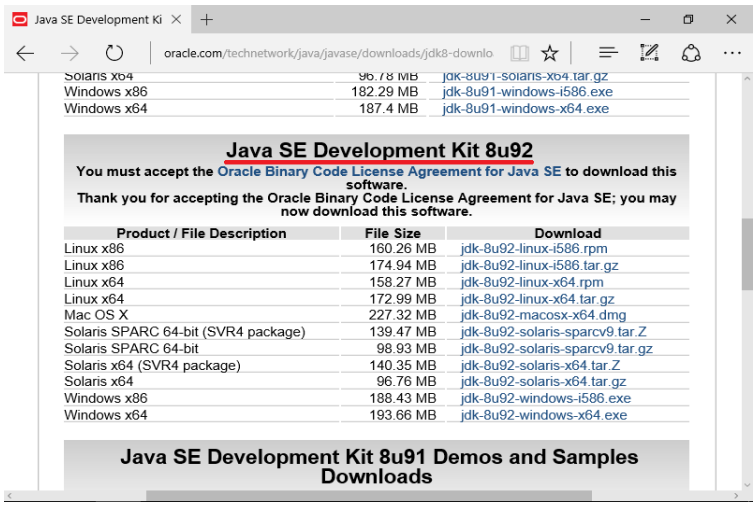



Рисунок 4.1 – Java SE Development Kit.

Затем проследовать инструкциям инсталлятора и использовать настройки по умолчанию. Необходимо также скачать Apache Maven http://maven.apache.org/download.cgi, что изображено на рис. 4.2:

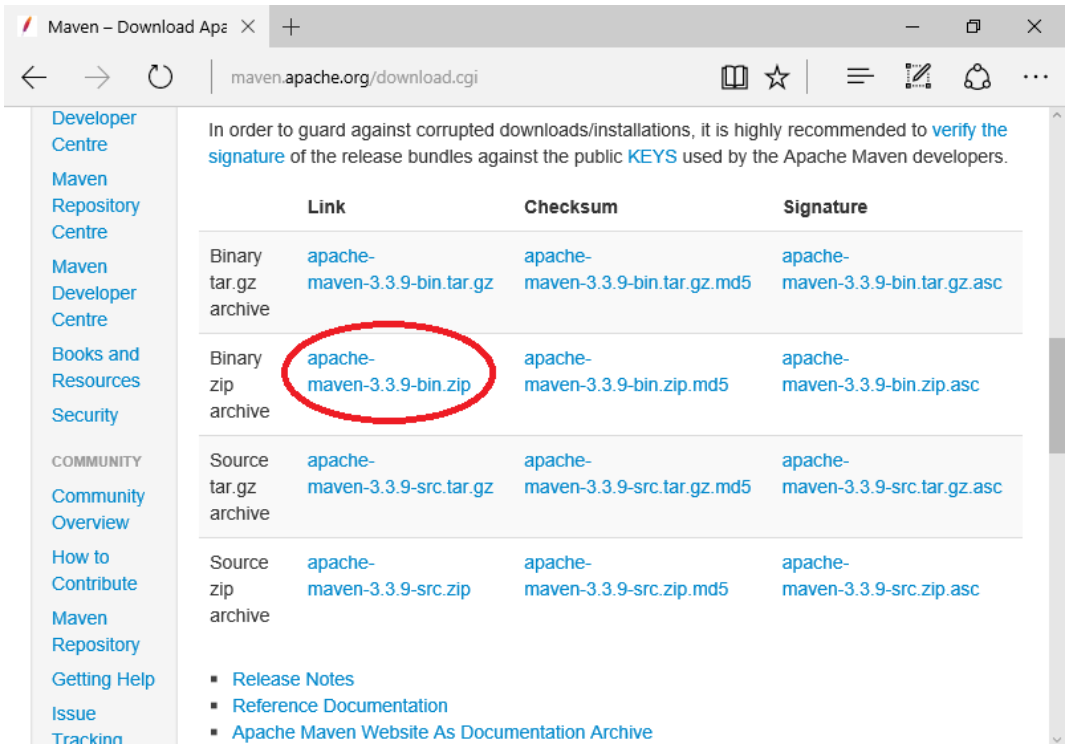



Рисунок 4.2 – Загрузка Maven с главного сайта.

Извлеките файлы из архива в корень диска C: (см. Рис. 4.3).

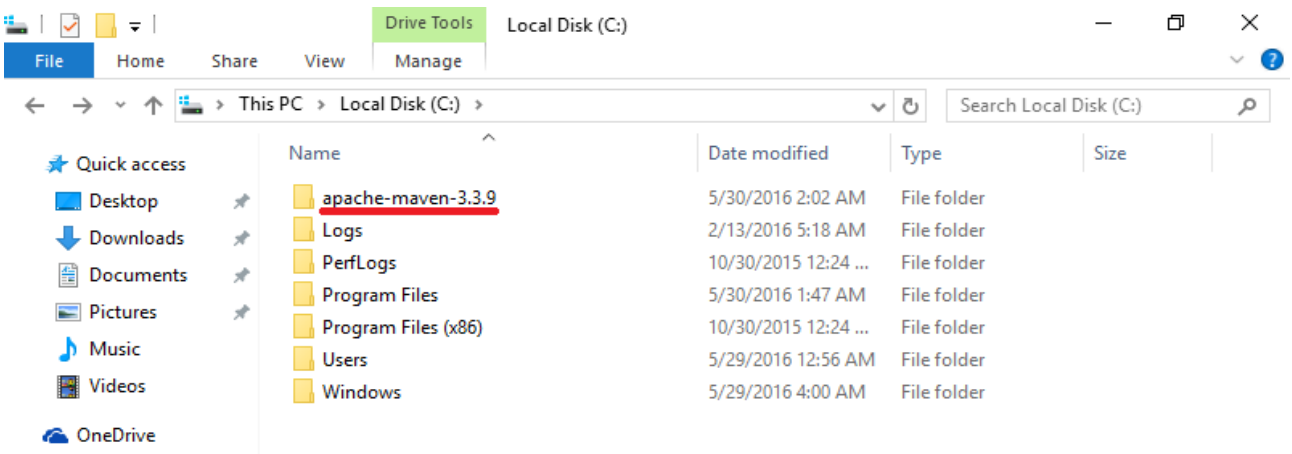



Рисунок 4.3 – Директория с Apache Maven.

В поиске (нажмите клавишу Windows) найдите и выберите: *Система (Панель управления)/ System (Control Panel)* (рис 4.4).

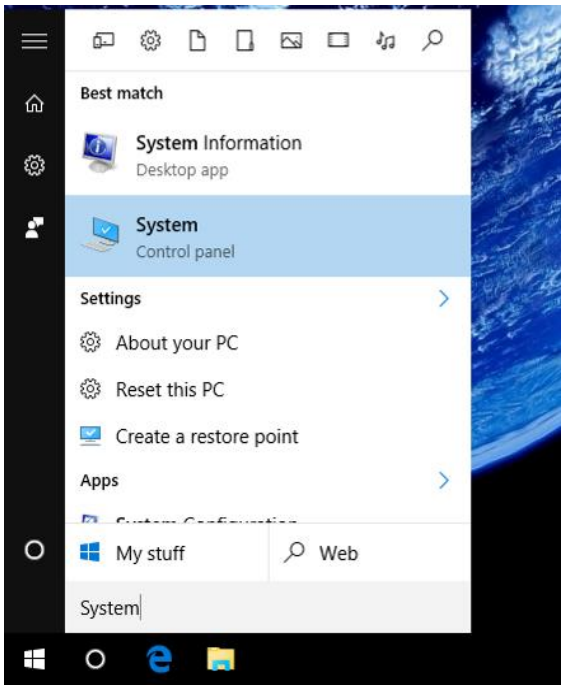



Рисунок 4.4 – Панель управления.

Нажмите *Расширенные настройки системы (Advanced system settings)*, затем *Переменные среды (Environment Variables)*(см. Рис. 4.5).

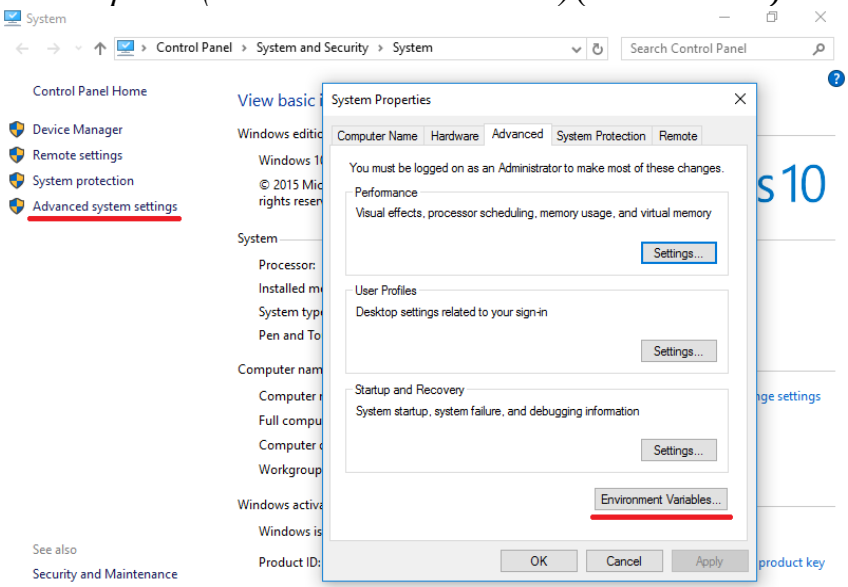



Рисунок 4.5 – Окно системных свойств.

Добавьте две новые системные переменные: JAVA_HOME и M2_HOME нажав *New* в разделе *System Variables* (см. рис. 4.6 и рис. 4.7).

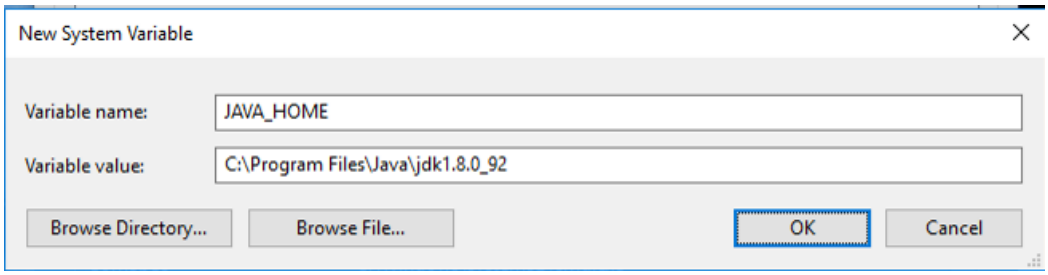



Рисунок 3.1.6 – Создание переменной JAVA_HOME.

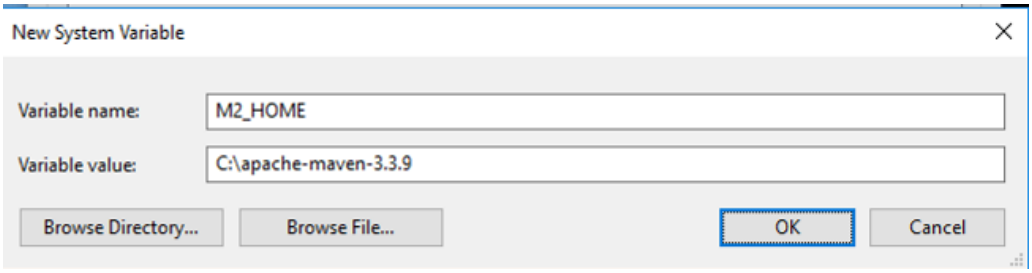



Рисунок 4.7 – Создание переменной M2_HOME.

После этого найдите переменную Path (см. Рис. 4.8) и нажмите Edit.

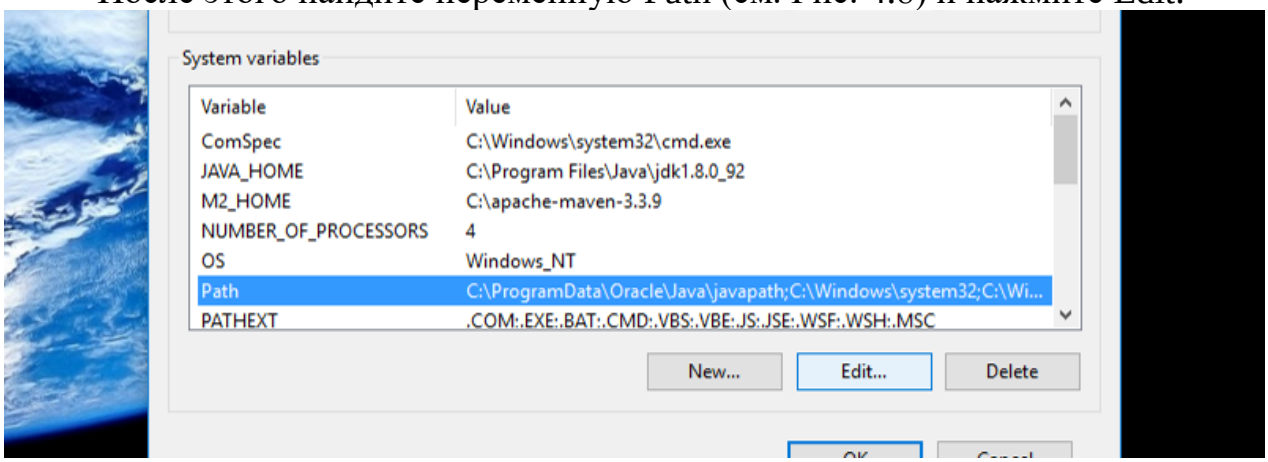



Рисунок 4.8 – Настройка переменной PATH

Добавьте две следующие строки: %JAVA_HOME%\bin и %M2_HOME%\bin, нажав *New* и заполнив пробелы (см. рис. 4.9).

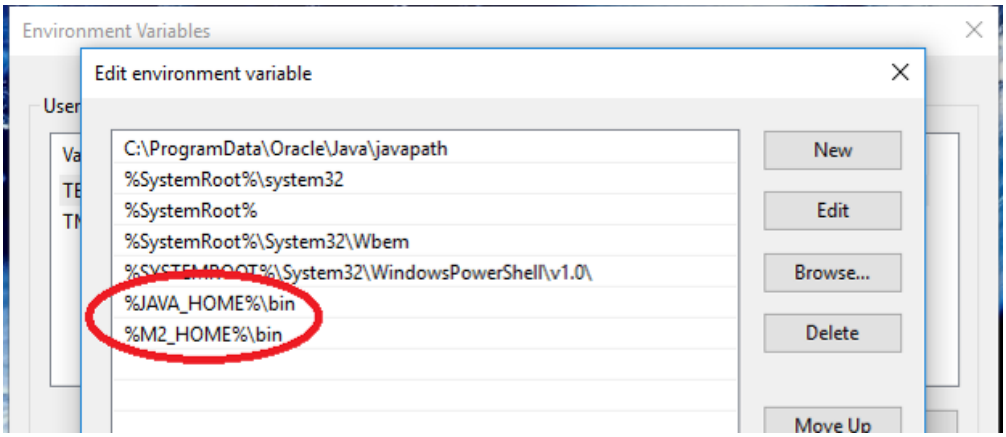



Рисунок 4.9 – Настройка переменных окружения.

Закройте все окна, нажав ОК.

Откройте командную строку, нажав клавишу Windows, и введите cmd (см. Рис. 4.10). Попробуйте выполнить следующие команды в открывшейся консоли: java и mvn. Команды должны быть доступны.

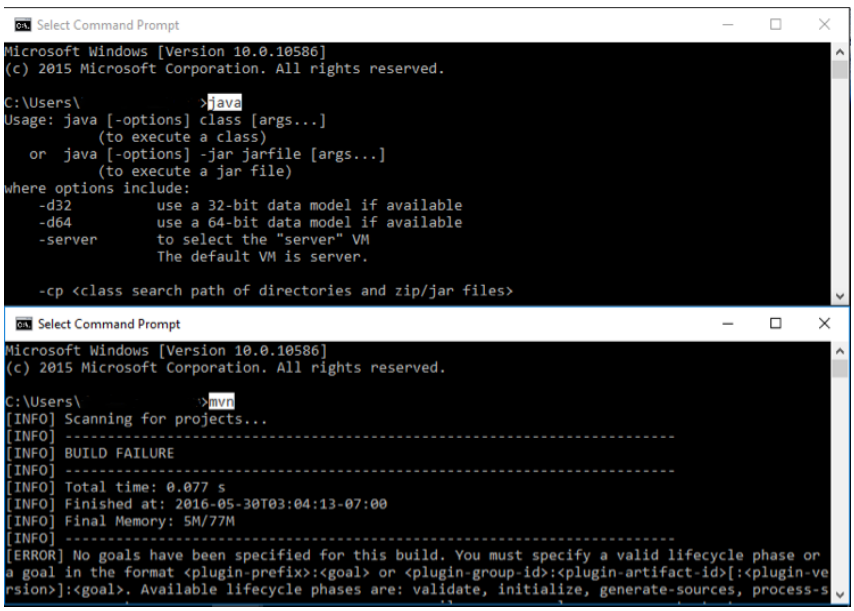



Рисунок 4.10 – Проверка команд java и mvn в терминале

Скачайте и установите IntelliJ IDEA Community Edition, если она не установлена: http://www.jetbrains.com/idea/download. Используйте настройки по умолчанию (см. Рис. 4.11).

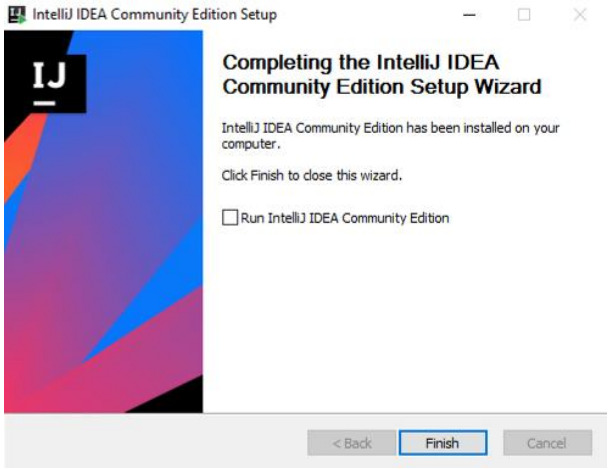



Рисунок 4.11 – IntelliJ IDEA Окно установки.

Также необходимо установить Scala плагин для IntelliJ IDEA. (рис. 4.12).

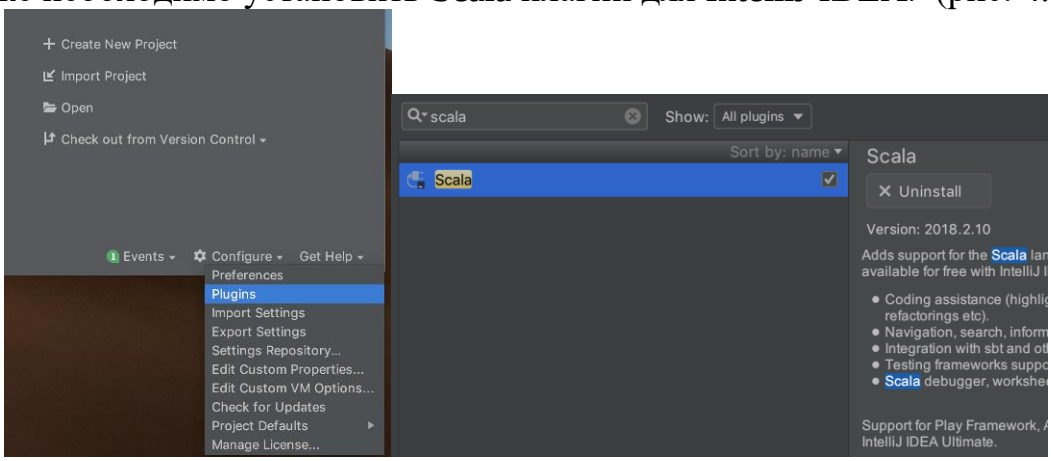



Рисунок 4.12 – IntelliJ IDEA, установка Scala плагина.

Для операционной системы **Centos/Fedora** используйте следующие команды для установки JDK и maven:

Листинг 4.1 – установка JDK и maven в **Centos/Fedora**

```
sudo yum install java-1.8.0-openjdk-devel
sudo yum install maven
```

Для операционной системы **Ubuntu/Debian** используйте следующие команды для установки java и maven:

Листинг 4.2 – установка JDK и maven в **Ubuntu/Debian**

```
sudo apt-get install openjdk-8-jdk
sudo apt install maven
```

Загрузка IntelliJ Idea Community, распаковка и запуск показаны ниже.

Листинг 4.3 – загрузка, распаковка и запуск IntelliJ Idea Community в Linux

```
wget https://download.jetbrains.com/idea/ideaIC-2019.1.tar.gz
tar -zxvf ideaIC*
cd ideaIC* && sh bin/idea.sh
```

## 4.2 Создание нового проекта

Далее будет подробно рассмотрено создание и настройка scala проекта, так как по нашему опыту здесь у большинства новичков появляются проблемы. Будьте внимательны.

Если все прошло успешно, запустите IntelliJ IDEA и нажмите кнопку *Create New Project*.


Рисунок 4.13 – IntelliJ IDEA приветственное окно.

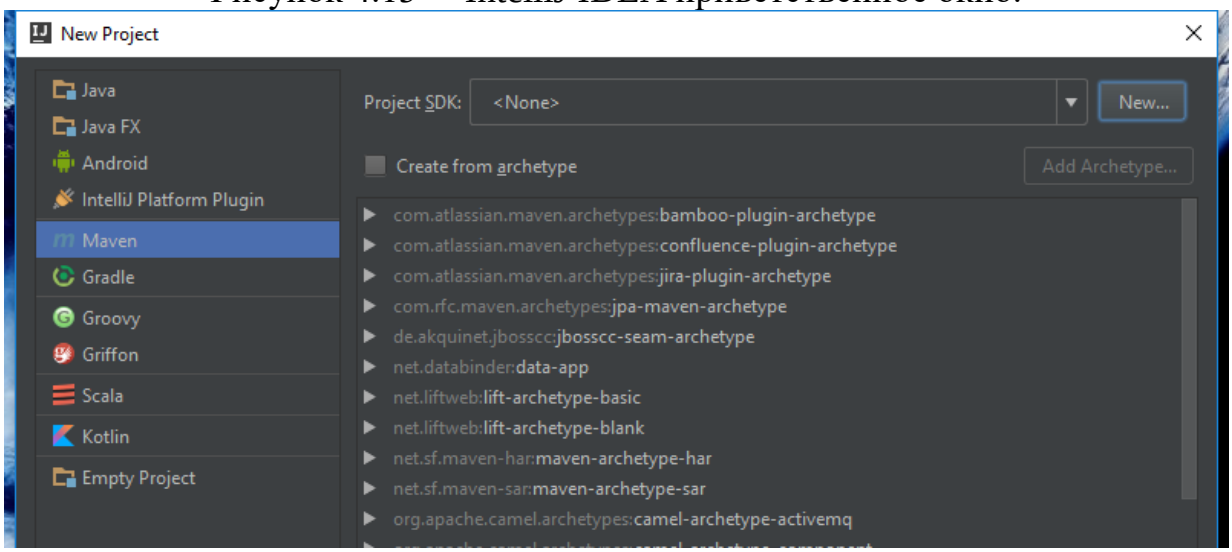



Рисунок 4.14 – Создание нового проекта.

На левой панели выберите Maven. На правой панели нажмите кнопку *Создать* (см. рис. 4.14). Укажите домашний каталог JDK (см. Рис. 4.15). На разных ОС это действие может отличаться.

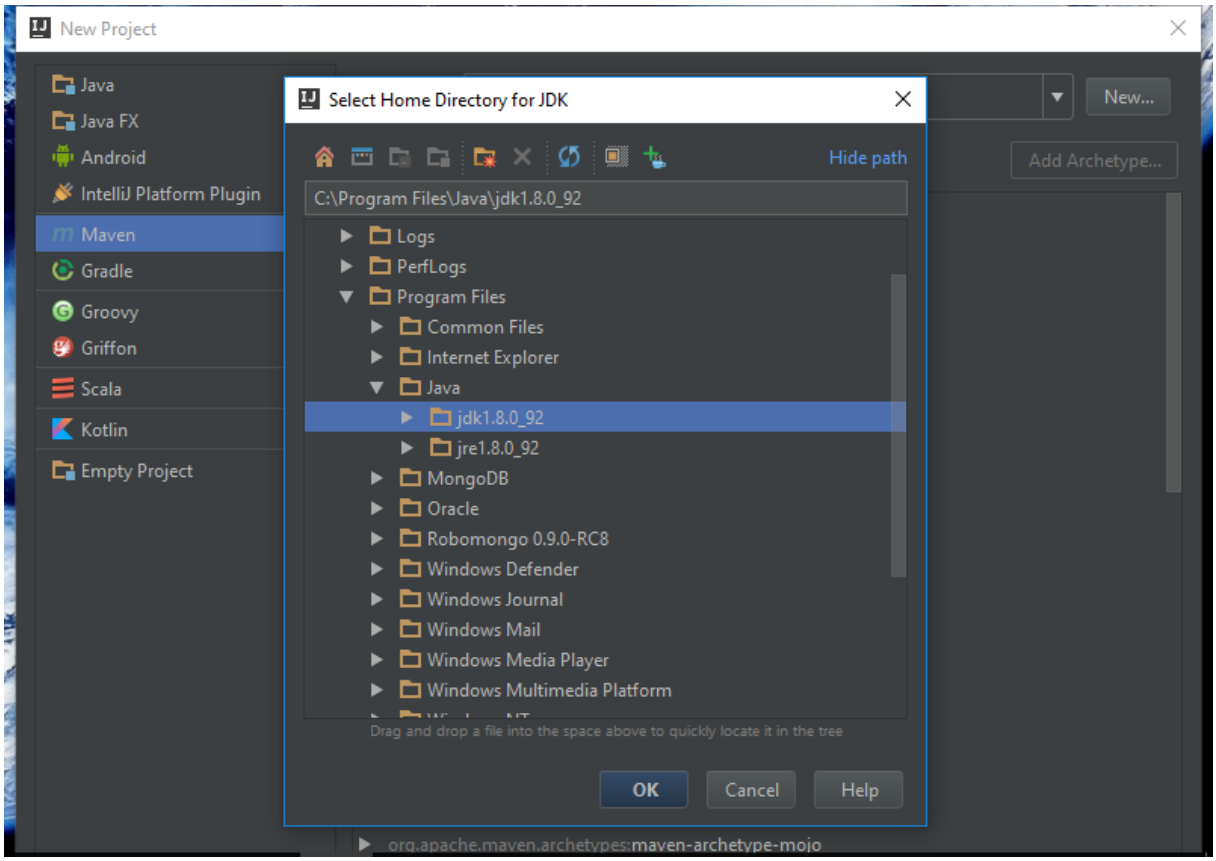



Рисунок 4.15 – Выбор JDK.

Нажмите *OK*, затем *Далее*. Выполните некоторые дополнительные настройки проекта (см. рис. 4.16 и рис. 4.17), если они нужны.

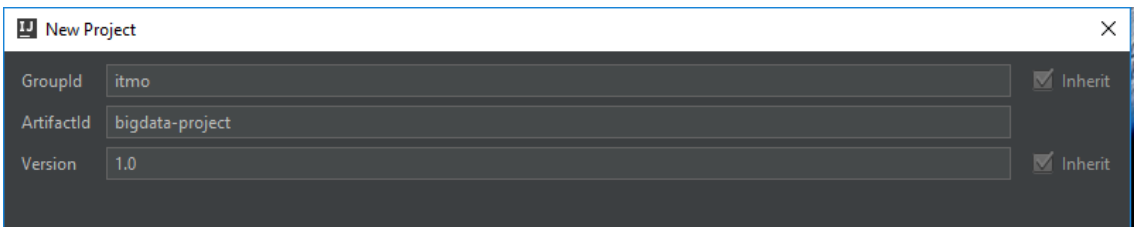



Рисунок 4.16 – Дополнительные настройки проекта.

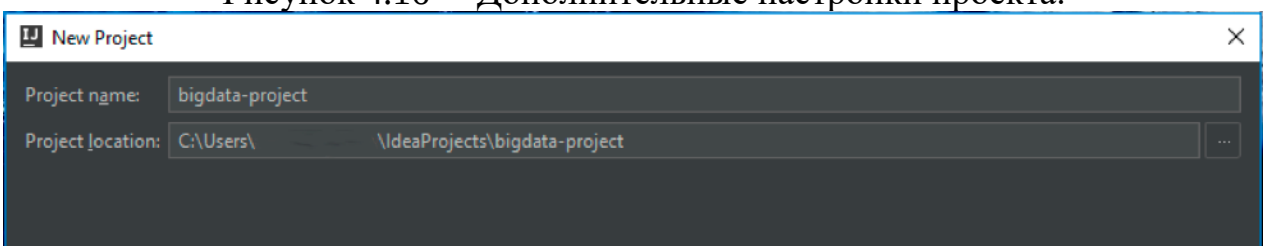



Рисунок 4.17 – Дополнительные настройки проекта.

Нажмите *Finish*. В открывшемся уведомлении нажмите *Enable Auto-Import* (рис. 4.18).

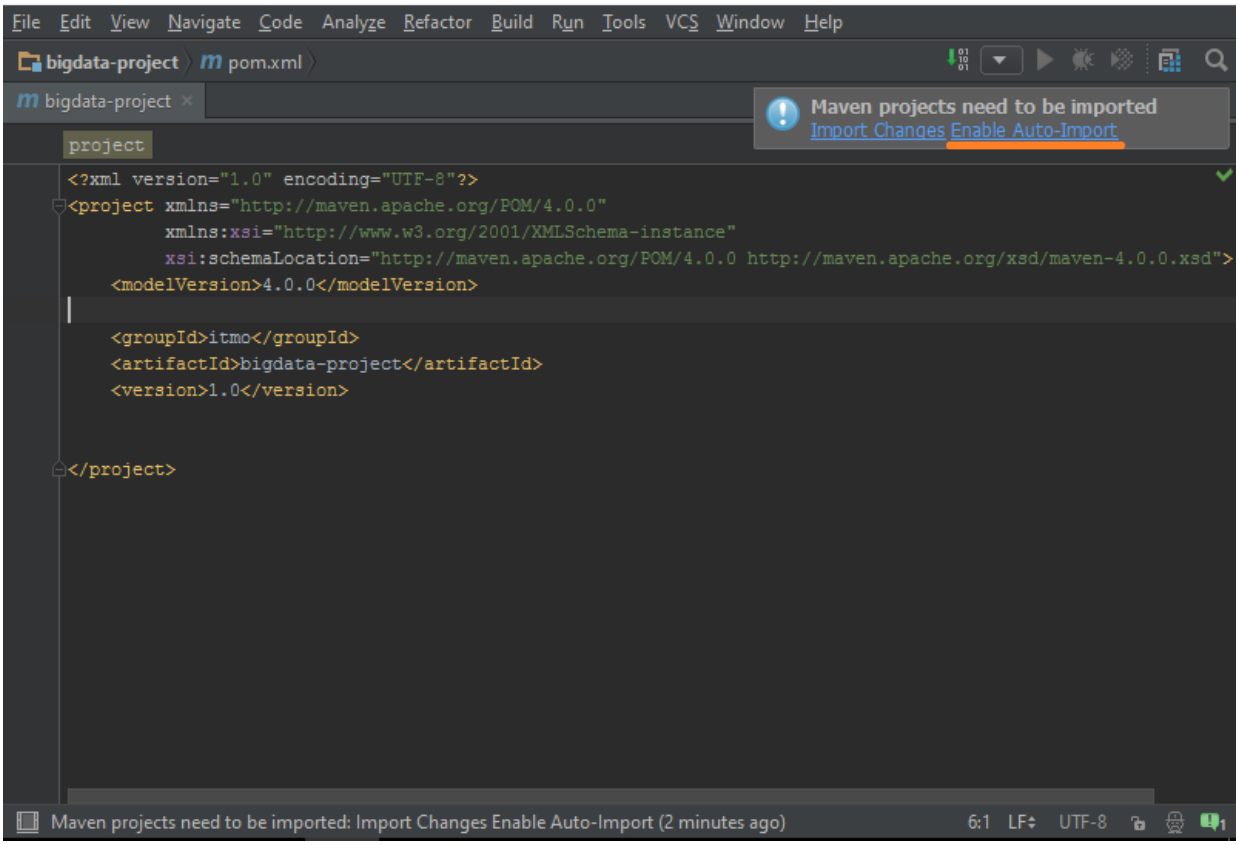



Рисунок 4.18 – Автоматическое импортирование maven зависимостей.

Создайте папку scala в каталоге src/main (см. рис. 4.19). Чтобы сделать это, щелкните правой кнопкой мыши на папке main, затем выберите *New* и *Directory*. Напишите имя scala и нажмите кнопку *OK*.

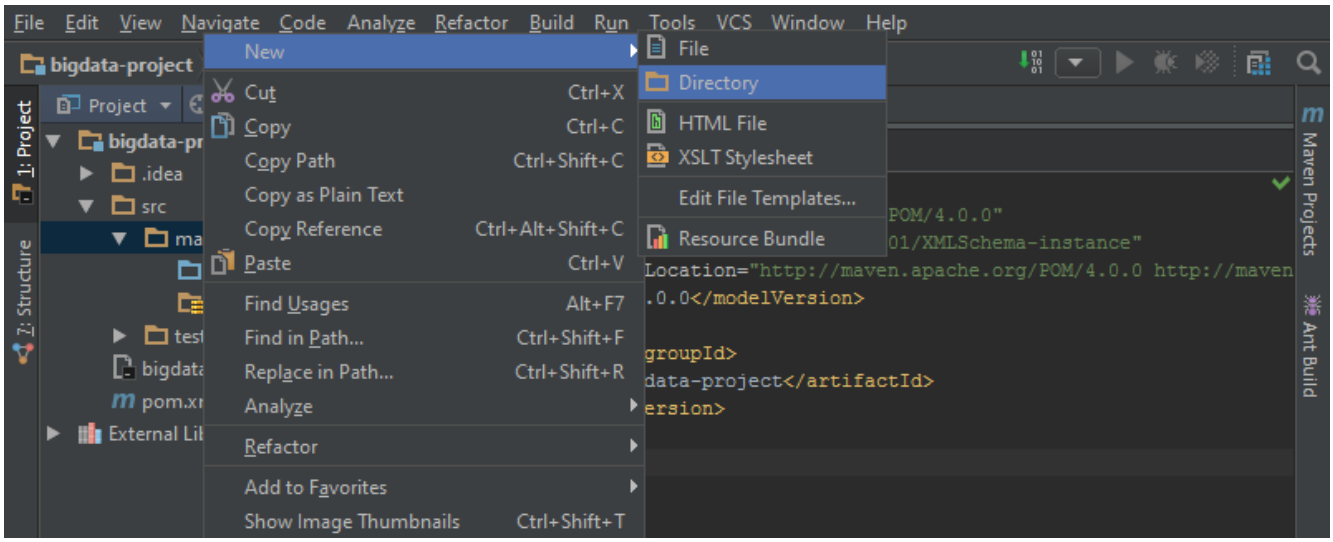



Рисунок 4.19 – Создание новой папки.

Затем щелкните правой кнопкой мыши папку Scala, выберите *Mark Directory As* и *Sources Root* (см. Рис. 4.20).

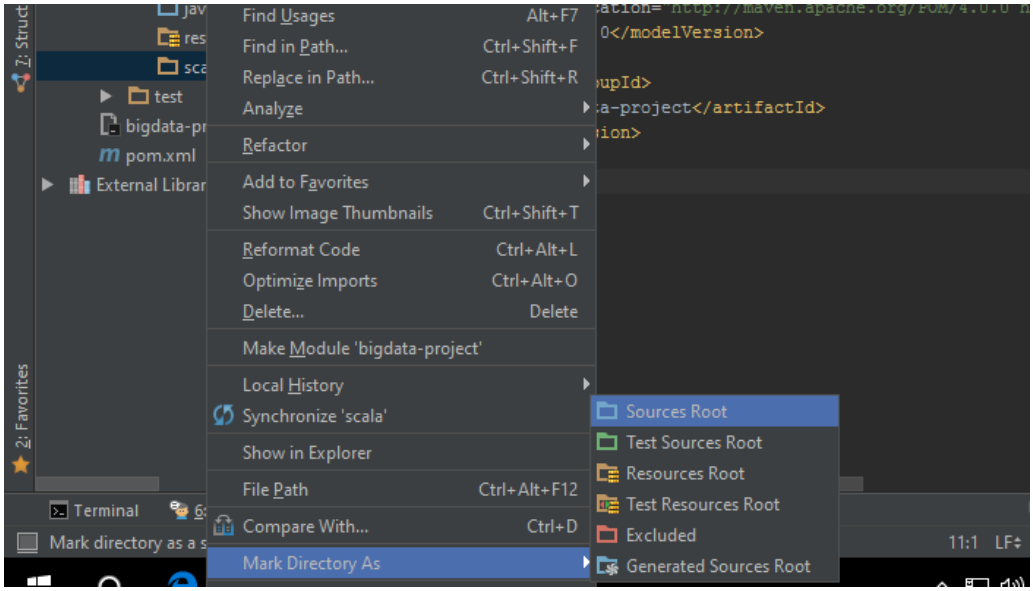



Рисунок 4.20 – Настройка директории с исходными файлами.

Добавьте следующий код в файл pom.xml (см. листинг. 4.4)

Листинг 4.4 – maven зависимость для scala

```
<dependencies>
  <dependency>
    <groupId>org.scala-lang</groupId>
    <artifactId>scala-library</artifactId>
    <version>2.11.7</version>
  </dependency>
</dependencies>
```

Последнюю версию библиотек в репозитории Maven можно посмотреть здесь: http://mvnrepository.com/. Например, последнюю версию библиотеки Scala мы можем найти здесь: http://mvnrepository.com/artifact/org.scala-lang/scala-library.

Создайте новый файл app.scala в каталоге src/main/scala, откройте его и нажмите кнопку *Setup Scala SDK* (см. рис. 4.21), затем нажмите *Create*. В итоге это позволит использовать библиотеку scala в нашем проекте. Тем не менее, это применимо только для настройки проекта в IntelliJ IDEA. Речь идет не о компиляции и запуске кода из командной строки с использованием Maven.

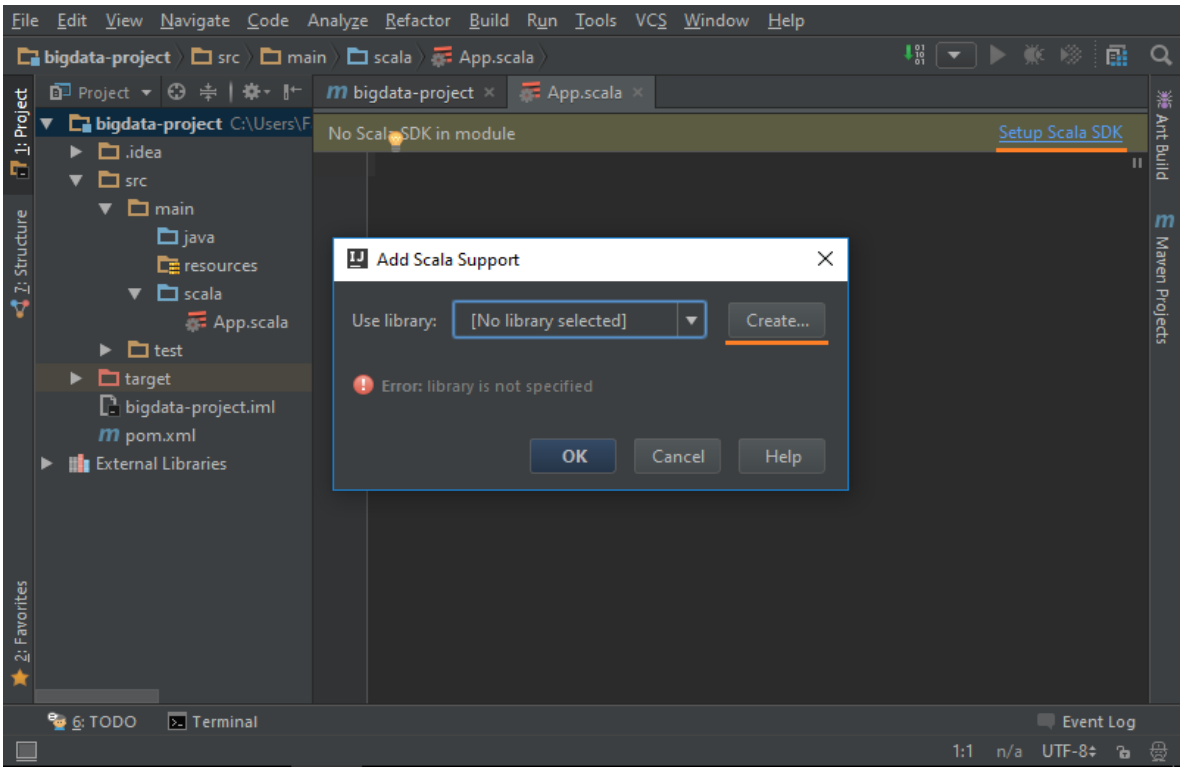



Рисунок 4.21 – Выбор scala для проекта Intellij Idea.

После этого нажмите *Download*, выберите версию scala и нажмите *OK*.

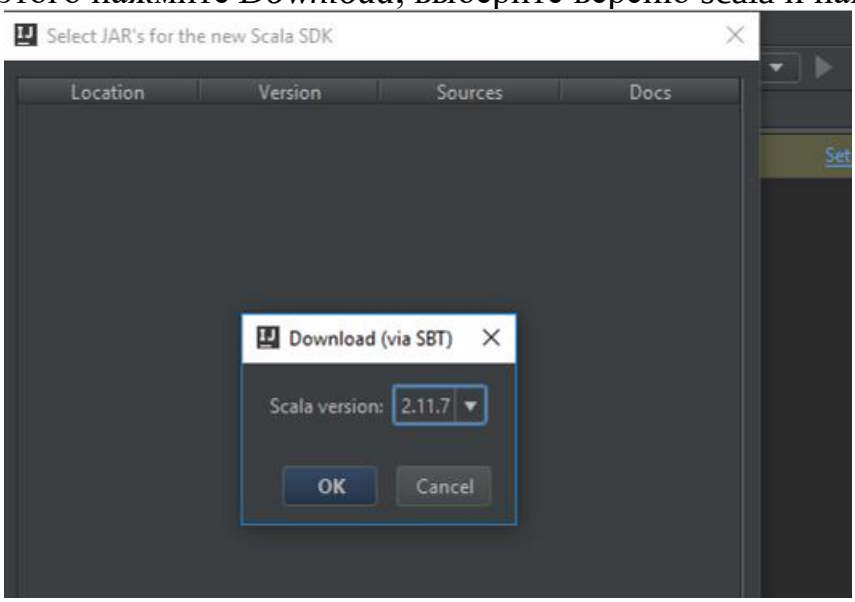



Рисунок 4.22 – Выбор scala для проекта Intellij Idea.

Будьте терпеливы, так как этот процесс может занять много времени, если у вас слабое интернет-соединение. После загрузки нажмите *OK* (см. рис. 4.23).

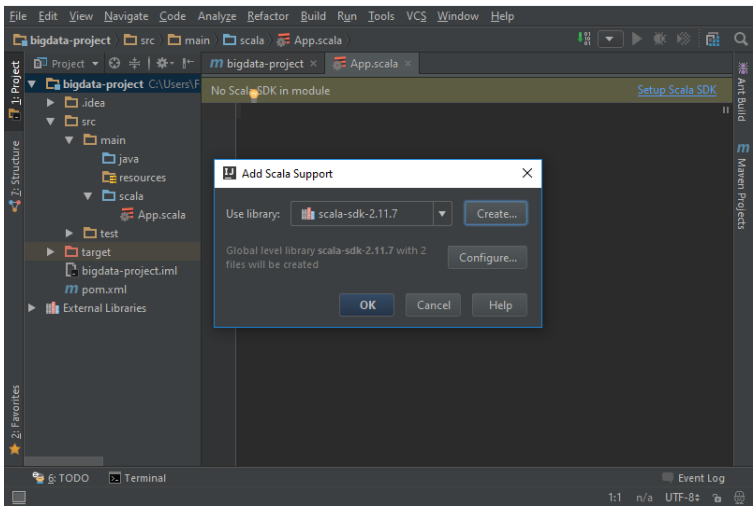



Рисунок 4.23 – Выбор scala для проекта Intellij Idea.

## 4.3 Запуск Scala проекта в Intellij Idea

Добавьте следующий код в файл app.scala.

Листинг 4.5 – первая scala программа

```
object App {
  def main(args: Array[String]) {
    println("Hello")
  }
}
```

Наберите *Ctrl + Shift + F10* или нажмите *Run* (см. рис. 4.25) и выберите App.

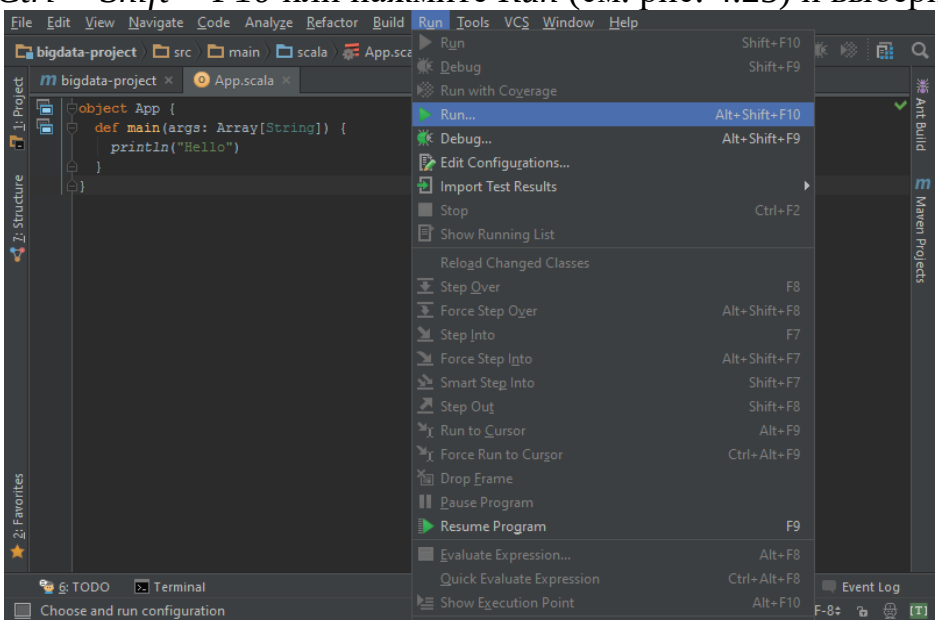



Рисунок 4.24– Запуск проекта в Intellij Idea.

Вы также можете использовать следующую кнопку (см. рис. 4.25).

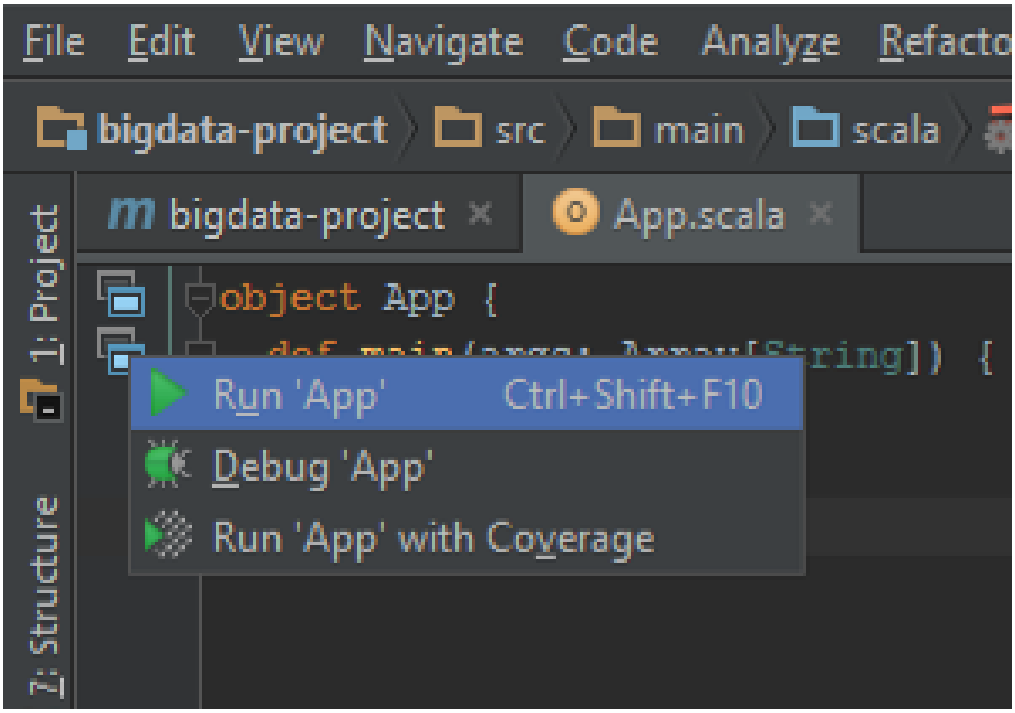



Рисунок 4.25– Запуск проекта в Intellij Idea.

Итак, код работает (см. рис. 4.26) и это означает, что настройка выполнена.

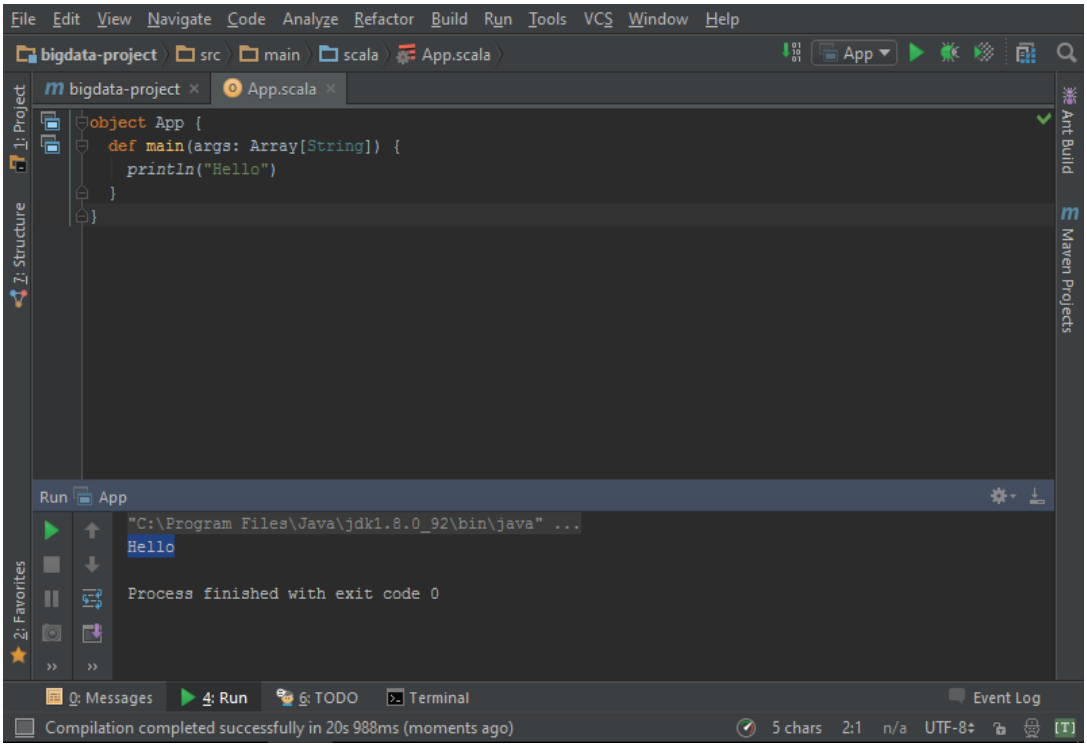



Рисунок 4.26 – Запуск проекта в Intellij Idea.

## 4.4 Первое Spark приложение

Добавьте следующие зависимости в раздел dependencies в файле pom.xml.

Листинг 4.6 – maven зависимости для модуля spark core и модуля spark sql

```
<dependency>
   <groupId>org.apache.spark</groupId>
   <artifactId>spark-core_2.11</artifactId>
   <version>2.4.1</version>
</dependency>
<dependency>
   <groupId>org.apache.spark</groupId>
   <artifactId>spark-sql_2.11</artifactId>
   <version>2.4.1</version>
</dependency>
```

Затем замените существующий код в файле App.scala следующим.

Листинг 4.7 – первое spark приложение

```
import org.apache.spark.{SparkConf, SparkContext}

object App {
  def main(args: Array[String]) {

    // Configuration for a Spark application.
    val conf = new SparkConf().setAppName("Test").setMaster("local[*]")

    // Main entry point for Spark functionality.
    val sc = new SparkContext(conf)

    // Read the input file and create an RDD.
    val rdd = sc.textFile("C:\\Users\\.       \\IdeaProjects\\bigdata-project\\src\\main\\scala\\App.scala")

    // View the content of the RDD.
    rdd.foreach(line => println(line))
  }
}
```

Запустите проект (см. рис. 4.27). Возможно, вы увидите ошибку.

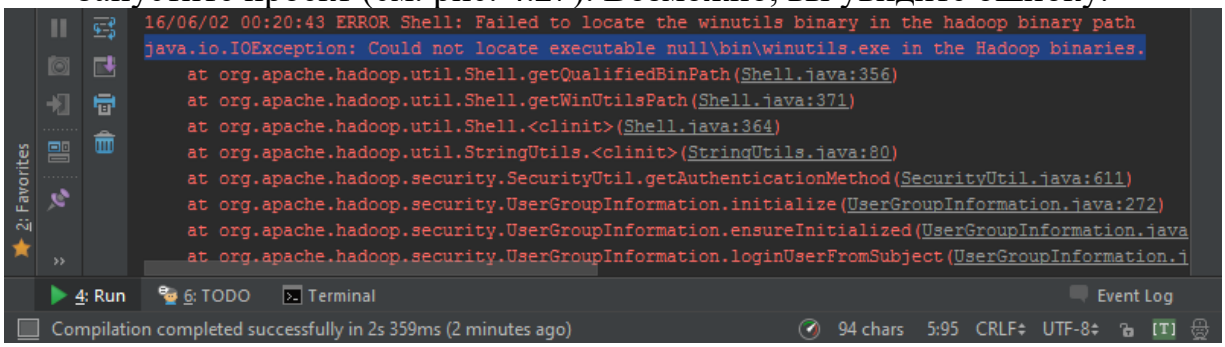



Рисунок 4.27 – Ошибка в windows.

Эта проблема появляется только в ОС Windows. Нужно скачать файл по ссылке http://public-repo-1.hortonworks.com/hdp-win-alpha/winutils.exe и поместить его в папку C:\hadoop\bin, а также добавить следующий код.

Листинг 4.7 – Установка системного свойства HADOOP_HOME.

```
System.setProperty("hadoop.home.dir", "C:\\hadoop")
```

Перезапустите проект (см. рис. 4.28).

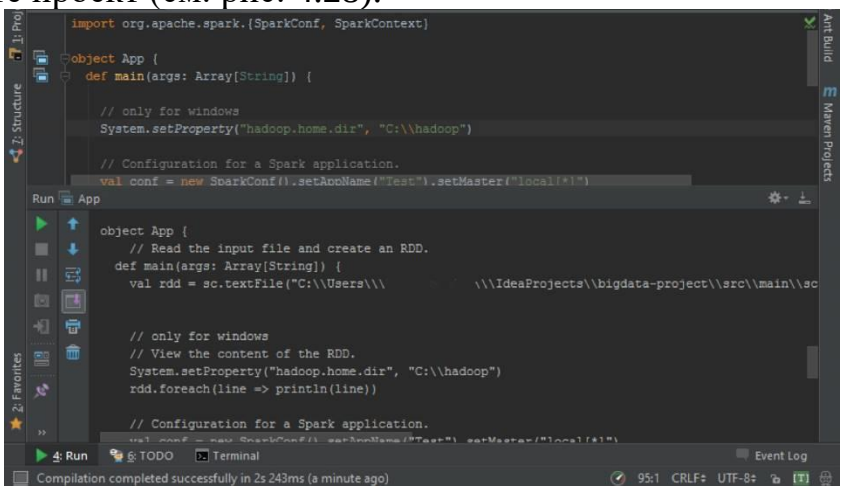



Рисунок 4.28 – Успешный запуск Spark проекта в IntelliJ Idea.

Мы видим, что код был выполнен, причем параллельно, так как после строки “only for windows” должна идти пустая строка, но это не так.

# Заключение

В пособии были рассмотрены теоретические и практические основы работы с фреймворком Apache Spark. Теоретический материал, рассчитанный на людей с минимальным опытом в программировании, позволяет получить представление о том, как устроена пакетная обработка больших данных и на каких принципах строятся ETL-конвейеры на основе Spark SQL и DataFrame API для распределенного выполнения на кластерных вычислительных системах.

Важной составляющей является рассмотрение того, как можно реализовать итеративные вычисления, необходимые для реализации алгоритмов машинного обучения, а также рассмотрение механизма операции shuffle и принципов организации памяти в Spark.

Практические примеры включают в себя: построение простых конвейеров обработки, целиком ориентированных на работу со стандартной библиотекой; оптимизацию выполнения за счет кэширования и изменение типа операции join; написание пользовательских функций обработки данных и их использование совместно с broadcast-операцией; написание пользовательских агрегационных функций (UDAF). Таким образом, приведенные в пособии приемы работы с большими данными позволяют начать применять фреймворк Apache Spark для решения актуальных практических задач.

# Список литературы

1. Computer cluster [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Computer_cluster.

2. Apache Cassandra [Электронный ресурс]. URL: http://cassandra.apache.org/ (дата обращения: 15.04.2019).

3. Distributed - Документация ClickHouse [Электронный ресурс]. URL: https://clickhouse.yandex/docs/ru/operations/table_engines/distributed/ (дата обращения: 15.04.2019).

4. HDFS Architecture Guide [Электронный ресурс]. URL: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html (дата обращения: 15.04.2019).

5. Ведущий — ведомый — Википедия [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Ведущий_—_ведомый (дата обращения: 15.04.2019).

6. Heartbeat-сообщение — Википедия [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Heartbeat-сообщение (дата обращения: 15.04.2019).

7. Java Virtual Machine — Википедия [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Java_Virtual_Machine (дата обращения: 15.04.2019).

8. Enterprise Application Container Platform | Docker [Электронный ресурс]. URL: https://www.docker.com/ (дата обращения: 15.04.2019).

9. Submitting Applications - Spark 2.4.1 Documentation [Электронный ресурс]. URL: https://spark.apache.org/docs/latest/submitting-applications.html (дата обращения: 15.04.2019).

10. Python vs. Scala для Apache Spark — ожидаемый benchmark с неожиданным результатом / Блог компании Одноклассники / Хабр [Электронный ресурс]. URL: https://habr.com/ru/company/odnoklassniki/blog/443324/ (дата обращения: 15.04.2019).

11. RDD Programming Guide - Spark 2.4.1 Documentation [Электронный ресурс].

URL: https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations (дата обращения: 15.04.2019).
12. Clickhouse Spark Connector [Электронный ресурс]. URL: https://github.com/DmitryBe/spark-clickhouse.
13. Spark Cassandra Connector [Электронный ресурс]. URL: https://github.com/datastax/spark-cassandra-connector.
14. Spark 2.x - 2nd generation Tungsten Engine | spark-notes [Электронный ресурс]. URL: https://spoddutur.github.io/spark-notes/second_generation_tungsten_engine.html (дата обращения: 15.04.2019).
15. Getting Started - Spark 2.4.1 Documentation [Электронный ресурс]. URL: https://spark.apache.org/docs/latest/sql-getting-started.html (дата обращения: 15.04.2019).
16. Deep Dive into Spark SQL's Catalyst Optimizer - The Databricks Blog [Электронный ресурс]. URL: https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html (дата обращения: 15.04.2019).
17. SparkSession — The Entry Point to Spark SQL · The Internals of Spark SQL [Электронный ресурс]. URL: https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-SparkSession.html (дата обращения: 15.04.2019).
18. Functions - Spark SQL, Built-in Functions [Электронный ресурс]. URL: https://spark.apache.org/docs/latest/api/sql/ (дата обращения: 15.04.2019).

Бутаков Николай Алексеевич
Петров Максим Владимирович
Насонов Денис

Обработка больших данных с Apache Spark
учебно-методическое пособие

В авторской редакции
Редакционно-издательский отдел Университета ИТМО
Зав. РИО Н. Ф. Гусарова
Подписано к печати
Заказ №
Отпечатано на ризографе

**Редакционно-издательский отдел**
**Университета ИТМО**
197101, Санкт-Петербург, Кронверский пр., 49