МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Воронежский государственный технический университет»
Кафедра систем автоматизированного проектирования
и информационных систем

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по выполнению лабораторных работ
по дисциплине «Технологии обработки больших данных»
для студентов направления 09.03.02 «Информационные системы
и технологии» (профиль «Технологии искусственного
интеллекта») очной формы обучения

Воронеж 2024

ЛАБОРАТОРНЫЕ РАБОТЫ 1-2

ОБРАБОТКА И АНАЛИЗ ДАННЫХ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕК PYTHON

Цель работы: ознакомиться с основными библиотеками Python для анализа данных, получить практические навыки анализа данных с использованием библиотеки Pandas

Библиотеки Python для анализа данных

1. Pandas

Pandas - это пакет Python с открытым исходным кодом, который предоставляет высокоэффективные, простые в использовании структуры данных и инструменты анализа для помеченных данных на языке программирования Python. Pandas расшифровывается как *библиотека анализа данных Python.* Кто-нибудь знал об этом?

Когда использовать? Pandas - это идеальный инструмент для обработки данных. Он предназначен для быстрой и простой обработки данных, чтения, агрегирования и визуализации.

Pandas берет данные в файле CSV или TSV или базу данных SQL и создает объект Python со строками и столбцами, который называется фреймом данных. Фрейм данных очень похож на таблицу в статистическом программном обеспечении, скажем, в Excel или SPSS.

Что можно делать с помощью Pandas?

1. Индексирование, манипулирование, переименование, сортировка, объединение фрейма данных;
2. Обновить, добавить, удалить столбцы из фрейма данных;
3. Восстановить недостающие файлы, обработать недостающие данные или NAN;
4. Построить гистограмму или прямоугольную диаграмму.

Это делает *Pandas* фундаментальной библиотекой в изучении Python для Data Science.

2. NumPy

NumPy - один из самых фундаментальных пакетов в Python - универсальный пакет для обработки массивов. Он предоставляет высокопроизводительные объекты многомерных массивов и инструменты для

работы с массивами. NumPy - это эффективный контейнер универсальных многомерных данных.

Основной объект NumPy - это однородный многомерный массив. Это таблица элементов или чисел одного и того же типа данных, проиндексированная набором натуральных чисел. В NumPy размеры называются *осями,* а число осей называется *рангом.* Класс массива NumPy называется *ndarray,* он же *array*.

Когда использовать? NumPy используется для обработки массивов, в которых хранятся значения одного и того же типа данных. NumPy облегчает математические операции над массивами и их векторизацию. Это значительно повышает производительность и, соответственно, ускоряет время выполнения.

Что можно делать с помощью NumPy?

1. Основные операции с массивами: добавление, умножение, срез, выравнивание, изменение формы, индексирование массивов;
2. Расширенные операции с массивами: стековые массивы, разбиение на секции, широковещательные массивы;
3. Работа с DateTime или линейной алгеброй;
4. Основные нарезки и расширенное индексирование в NumPy Python.

3. SciPy

Библиотека SciPy является одним из ключевых пакетов, которые составляют стек SciPy. Теперь есть разница между SciPy Stack и библиотекой SciPy. *SciPy* основывается на объекте массива NumPy и является частью стека, который включает в себя такие инструменты, как Matplotlib, Pandas и SymPy с дополнительными инструментами.

Библиотека SciPy содержит модули для эффективных математических процедур, таких как линейная алгебра, интерполяция, оптимизация, интеграция и статистика. Основной функционал библиотеки SciPy построен на NumPy и его массивах.

Когда использовать? SciPy использует массивы в качестве базовой структуры данных. Он имеет различные модули для выполнения общих задач научного программирования, таких как линейная алгебра, интеграция, матанализ, обыкновенные дифференциальные уравнения и обработка сигналов.

Что можно делать с помощью SciPy?

1. Математические, научные, инженерные вычисления;
2. Процедуры численной интеграции и оптимизации;
3. Поиск минимумов и максимумов функций;
4. Вычисление интегралов функции;
5. Поддержка специальных функций;
6. Работа с генетическими алгоритмами;
7. Решение обыкновенных дифференциальных уравнений.

4. Matplotlib

Это, несомненно, моя любимая и основная библиотека Python. Вы можете создавать истории с данными, визуализированными с помощью Matplotlib. Еще одна библиотека из стека SciPy - Matplotlib - строит 2D-фигуры.

Когда использовать? Matplotlib - это библиотека Python, предоставляющая API для встраивания графиков в приложения. Очень напоминает MATLAB, встроенный в язык программирования Python.

Что можно делать с помощью Matplotlib?

Гистограммы, столбцовые диаграммы, точечные диаграммы, круговые диаграммы - Matplotlib может отображать широкий спектр визуализаций. Приложив немного усилий, с Matplotlib, вы можете создавать любые визуализации:

1. Линейные диаграммы;
2. Точечные диаграммы;
3. Диаграммы с областями;
4. Столбцовые диаграммы и гистограммы;
5. Круговые диаграммы;
6. Диаграммы «стебель-листья»;
7. Контурные графики;
8. Поля векторов;
9. Спектрограммы.

Matplotlib также облегчает использование меток, сеток, легенд и некоторых других объектов форматирования. В общем, речь идет обо всем, что можно нарисовать!

5. Seaborn

Итак, когда вы читаете официальную документацию по Seaborn, она определяется как библиотека визуализации данных на основе Matplotlib, предоставляющем высокоуровневый интерфейс для изображения интересных и информативных статистических графиков. Проще говоря, seaborn - это расширение Matplotlib с дополнительными возможностями.

Так в чем разница между Matplotlib и Seaborn? Matplotlib используется для основного построения столбцовых, круговых, линейных, точечных диаграмм и пр., в то время как Seaborn предоставляет множество шаблонов визуализации с меньшим количеством синтаксических правил, причем более простых.

Что можно делать с помощью Seaborn?

1. Определять отношения между несколькими переменными (корреляция);
2. Соблюдать качественные переменные для агрегированных статистических данных;

3. Анализировать одномерные или двумерные распределения и сравнивать их между различными подмножествами данных;

4. Построить модели линейной регрессии для зависимых переменных;

5. Обеспечить многоуровневые абстракции, многосюжетные сетки.

Seaborn — это отличный вариант для библиотек визуализации R, таких как *corrplot* и *ggplot.*

6. Scikit Learn

Scikit Learn, представленный миру как проект Google Summer of Code, представляет собой надежную библиотеку машинного обучения для Python. Он включает в себя алгоритмы ML, такие как SVM, random forests, k-means кластеризацию, спектральную кластеризацию, сдвиг среднего значения, перекрестную проверку и многие другие. Даже NumPy, SciPy и связанные с ними научные операции поддерживаются Scikit Learn, при этом Scikit Learn является частью SciPy Stack.

Когда использовать? Scikit-learn предоставляет ряд контролируемых и неконтролируемых алгоритмов обучения через согласованный интерфейс в Python. Scikit learn будет вашим руководством для того, чтобы модели контролируемого обучения, такие как Naive Bayes, группировали непомеченные данные, такие как KMeans.

Что можно делать с помощью Scikit Learn?

1. Классификация: обнаружение спама, распознавание изображений;

2. Кластеризация: воздействия лекарственных препаратов, цена акций;

3. Регрессия: сегментация клиентов, группировка результатов эксперимента;

4. Уменьшение размерности: визуализация, повышенная эффективность;

5. Выбор модели: повышенная точность благодаря настройке параметров;

6. Предварительная обработка: подготовка входных данных в виде текста для обработки с помощью алгоритмов машинного обучения.

Scikit Learn фокусируется на моделировании данных; не манипулировании данными. Для обобщения и манипуляции у нас есть NumPy и Pandas.

7. TensorFlow

TensorFlow - это библиотека AI, которая помогает разработчикам создавать крупномасштабные нейронные сети со многими слоями, используя графики потоков данных. TensorFlow также облегчает построение моделей глубокого обучения, продвигает современную технологию ML / AI и позволяет легко развертывать приложения на базе ML.

Одним из наиболее развитых веб-сайтов среди всех библиотек является TensorFlow. Гиганты, такие как Google, Coca-Cola, Airbnb, Twitter, Intel, DeepMind, все используют TensorFlow!

Когда использовать? TensorFlow достаточно эффективен, когда дело доходит до классификации, восприятия, понимания, обнаружения, прогнозирования и создания данных.

Что можно делать с помощью TensorFlow?

1. Распознавание голоса / звука - IoT, автомобильная промышленность, безопасность, UX/UI, телекоммуникации;
2. Анализ настроений - в основном для CRM или CX;
3. Текстовые Приложения — Обнаружение угроз, Google Translate, Gmail Smart Reply;
4. Распознавание лиц - Facebook's Deep Face, Photo tagging, Smart Unlock;
5. Временной ряд - рекомендации от Amazon, Google и Netflix;
6. Обнаружение видео - обнаружение движения, обнаружение угроз в реальном времени в играх, безопасности, аэропортах.

8. Keras

Keras - это высокоуровневый API TensorFlow для создания и обучения кода глубоких нейронных сетей. Это библиотека нейронных сетей с открытым исходным кодом на Python. С Keras статистическое моделирование, работа с изображениями и текстом намного легче с упрощенным кодированием для глубокого обучения.

В чем разница между Keras и TensorFlow?

Keras - это нейросетевая библиотека, написанная на языке Python, а TensorFlow - это библиотека с открытым исходным кодом для различных задач машинного обучения. TensorFlow предоставляет как высокоуровневые, так и низкоуровневые API, в то время как Keras предоставляет только высокоуровневые API. Keras *создан для Python* и делает его более удобным, модульным и компонуемым, чем TensorFlow.

Что можно делать с помощью Keras?

1. Определить процентную точность;
2. Функция вычисления потерь;
3. Создать пользовательские функциональные слои;
4. Встроенные функции обработки данных и изображений;
5. Функции с повторяющимися блоками кода: глубиной 20, 50, 100 слоев.

9. Statsmodels

Statsmodels - это универсальный пакет Python, который обеспечивает простые вычисления для описательной статистики и оценки и формирования статистических моделей.

Что можно делать с помощью Statsmodels?

1. Линейная регрессия;
2. Корреляция;
3. Метод наименьшего квадрата (OLS);
4. Анализ выживания;
5. Обобщенные линейные модели и байесовская модель;
6. Однофакторный и двухфакторный анализ, проверка гипотез.

10. Plotly

Plotly - это типичная графическая библиотека для Python. Пользователи могут импортировать, копировать, вставлять или передавать данные, которые должны быть проанализированы и визуализированы. Plotly предлагает изолированную версию Python (где вы можете запустить Python, ограниченный в своих возможностях). Теперь осталось понять, что значит ограниченная версия, но я точно знаю, что Plotly облегчает задачу!

Когда использовать? Вы можете использовать Plotly, если хотите создавать и отображать фигуры, обновлять фигуры, наводить курсор на текст для получения подробной информации. Plotly также имеет дополнительную функцию отправки данных на облачные серверы.

Что можно делать с помощью Plotly?

Библиотека графиков Plotly имеет широкий спектр графиков, которые вы можете построить:

1. ***Основные диаграммы:*** линейные, круговые, точечные, пузырьковые, Ганта, санбёрст, древовидные, санкей, графики с областями;
2. ***Статистические стили и стили Seaborn***: ошибки, гистограммы, диаграммы Facet и Trellis, деревообразные графики, графики-скрипки, линии тренда;
3. ***Научные карты:*** контур, троичный сюжет, логарифмический график, поля векторов, ковровый график (Carpet plot), радарчарт, тепловые карты Роза ветров и Полярный сюжет;
4. Финансовые графики;
5. Карты;
6. Субплоты;
7. Трансформации;
8. Взаимодействие Jupyter Widgets.

Plotly это *типичная* библиотека графиков.

Далее рассмотрим подробно библиотеку Pandas.

Обработка данных с использованием библиотеки Pandas

Pandas – это библиотека, которая предоставляет очень удобные с точки зрения использования инструменты для хранения данных и работе с ними. Официальный сайт *pandas* находится по ссылке https://pandas.pydata.org/.

Особенность *pandas* состоит в том, что эта библиотека очень быстрая, гибкая и выразительная. Это важно, т.к. она используется с языком *Python*, который не отличается высокой производительностью. *pandas* прекрасно подходит для работы с одномерными и двумерными таблицами данных, хорошо интегрирован с внешним миром – есть возможность работать с файлами *CSV*, таблицами *Excel*, может стыковаться с языком *R*.

Установка *pandas*

Для проведения научных расчетов, анализа данных или построения моделей в рамках машинно обучения для языка *Python* существуют прекрасное решение – *Anaconda*. *Anaconda* – это пакет, который содержит в себе большой набор различных библиотек, интерпретатор языка *Python* и несколько сред для разработки.

Pandas присутствует в стандартной поставке *Anaconda*. Если же его там нет, то его можно установить отдельно. Для этого стоит воспользоваться пакетным менеджером, который входит в состав *Anaconda*, который называется *conda*. Для его запуска необходимо перейти в каталог *[Anaconda install path]\Scripts* в *Windows*. В операционной системе *Linux*, после установки *Anaconda* менеджер *conda* должен быть доступен везде.

Введите командной строке:

```
>conda install pandas
```

В случае, если требуется конкретная версия *pandas*, то ее можно указать при установке.

```
>conda install pandas=0.13.1
```

При необходимости, можно воспользоваться пакетным менеджером *pip*, входящим в состав дистрибутива *Python*.

```
>pip install pandas
```

Если вы используете *Linux*, то ещё одни способ установить *pandas* – это воспользоваться пакетным менеджером самой операционной системы. Для *Ubuntu* это выглядит так:

```
>sudo apt-get install python-pandas
```

После установки необходимо проверить, что *pandas* установлен и корректно работает. Для этого запустите интерпретатор *Python* и введите в нем следующие команды.

```
>>> import pandas as pd
>>> pd.test()
```

В результате в окне терминала должен появиться следующий текст:

```
Running unit tests for pandas
pandas version 0.18.1
numpy version 1.11.1
pandas is installed in c:\Anaconda3\lib\site-packages\pandas
Python version 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul 5 2016, 11:41:13) [MSC v.1900 64 bit (AMD64)]
nose version 1.3.7
.....

----------------------------------------------------------------------
Ran 11 tests in 0.422s
OK
```

Это будет означать, что *pandas* установлен и его можно использовать.

Основные структуры данных Pandas

Библиотека pandas предоставляет две структуры: *Series* и *DataFrame* для быстрой и удобной работы с данными (на самом деле их три, есть еще одна структура – *Panel,* но в данный момент она находится в статусе *deprecated* и в будущем будет исключена из состава библиотеки *pandas*). *Series* – это маркированная одномерная структура данных, ее можно представить, как таблицу с одной строкой. С *Series* можно работать как с обычным массивом (обращаться по номеру индекса), и как с ассоциированным массивом, когда можно использовать ключ для доступа к элементам данных. *DataFrame* – это двумерная маркированная структура. Идейно она очень похожа на обычную таблицу, что выражается в способе ее создания и работе с ее элементами. *Panel* – про который было сказано, что он вскоре будет исключен из *pandas,* представляет собой трехмерную структуру данных. О *Panel* мы больше говорить не будем. В рамках этой части мы остановимся на вопросах создания и получения доступа к элементам данных структур *Series* и *DataFrame*.

Структура данных *Series*

Для того, чтобы начать работать со структурами данных из *pandas* требуется предварительно импортировать необходимые модули. Убедитесь, что нужные модули установлены на вашем компьютере.

Помимо самого *pandas* нам понадобится библиотека *numpy*. Наши эксперименты будем проводить с использованием пакета *Anaconda*, в качестве среды разработки советуем взять *Spyder*, который входит в базовую поставку *Anaconda*. Для того, чтобы запустить *Spyder*, перейдите в каталог *Scripts*, который находится в папке с установленной *Anaconda* и запустите *spyder.exe*. Для нас он в первую очередь имеет ценность в том, что в нем есть редактор исходного кода, на случай, если нам понадобится написать довольно большую программу, и интерпретатор для быстрых экспериментов. Если строки кода будут содержать префикс в виде цифры в квадратных скобках, то это означает, что данные команды мы вводим в интерпретаторе, в ином случае, это будет означать, что код написан в редакторе.

Импортируем нужные нам библиотеки.

```
In [1]: import numpy as np
In [2]: import pandas as pd
```

Создать структуру *Series* можно на базе различных типов данных:

- словари *Python*;
- списки *Python*;
- массивы из *numpy: ndarray*;
- скалярные величины.

Конструктор класса *Series* выглядит следующим образом:

```
pandas.Series(data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)
```

data – массив, словарь или скалярное значение, на базе которого будет построен *Series;*

index – список меток, который будет использоваться для доступа к элементам *Series*. Длина списка должна быть равна длине *data*;

dtype – объект *numpy.dtype*, определяющий тип данных;

copy – создает копию массива данных, если параметр равен *True* в ином случае ничего не делает.

В большинстве случаев, при создании *Series,* используют только первые два параметра. Рассмотрим различные варианты как это можно сделать.

Создание *Series* из списка *Python*

Самый простой способ создать *Series* – это передать в качестве единственного параметра в конструктор класса список Python.

```
In [3]: s1 = pd.Series([1, 2, 3, 4, 5])
In [4]: print(s1)
0 1
1 2
2 3
3 4
4 5
dtype: int64
```

В данном примере была создана структура *Series* на базе списка из языка *Python*. Для доступа к элементам *Series*, в данном случае, можно использовать только положительные целые числа – левый столбец чисел, начинающийся с нуля – это как раз и есть индексы элементов структуры, которые представлены в правом столбце.

Можно попробовать использоваться больше возможностей из тех, что предлагает *pandas*, для этого передадим в качестве второго элемента список строк (в нашем случае – это отдельные символы). Такой шаг позволит нам обращаться к элементам структуры *Series* не только по численному индексу, но и по метке, что сделает работу с таким объектом, похожей на работу со словарем.

```
In [5]: s2 = pd.Series([1, 2, 3, 4, 5], ['a', 'b', 'c', 'd', 'e'])
In [6]: print(s2)
a 1
b 2
c 3
d 4
e 5
dtype: int64
```

Обратите внимание на левый столбец, в нем содержатся метки, которые мы передали в качестве *index* параметра при создании структуры. Правый столбец – это по-прежнему элементы нашей структуры.

Создание *Series* из *ndarray* массива из *numpy*

Создадим простой массив из пяти чисел, аналогичный списку из предыдущего раздела. Библиотеки *pandas* и *numpy* должны быть предварительно импортированы.

```
In [3]: ndarr = np.array([1, 2, 3, 4, 5])
In [4]: type(ndarr)
Out[4]: numpy.ndarray
```

Теперь создадим *Series* с буквенными метками.

```
In [5]: s3 = pd.Series(ndarr, ['a', 'b', 'c', 'd', 'e'])
```

```
In [6]: print(s3)
a 1
b 2
c 3
d 4
e 5
dtype: int32
```

Создание *Series* из словаря (*dict*)

Еще один способ создать структуру *Series* – это использовать словарь для одновременного задания меток и значений.

```
In [7]: d = {'a':1, 'b':2, 'c':3}
In [8]: s4 = pd.Series(d)
In [9]: print(s4)
a 1
b 2
c 3
dtype: int64
```

Ключи (*keys*) из словаря *d* станут метками структуры *s4*, а значения (*values*) словаря – значениями в структуре.

Создание *Series* с использованием константы

Рассмотрим еще один способ создания структуры. На этот раз значения в ячейках структуры будут одинаковыми.

```
In [10]: a = 7
In [11]: s5 = pd.Series(a, ['a', 'b', 'c'])
In [12]: print(s5)
a 7
b 7
c 7
dtype: int64
```

В созданной структуре *Series* имеется три элемента с одинаковым содержанием.

Работа с элементами *Series*

В будущем будет написан отдельный урок, посвященный индексации и работе с элементами *Series* и *DataFrame*, сейчас рассмотрим основные подходы.

К элементам *Series* можно обращаться по численному индексу, при таком подходе работа со структурой не отличается от работы со списками в *Python*.

```
In [13]: s6 = pd.Series([1, 2, 3, 4, 5], ['a', 'b', 'c', 'd', 'e'])
In [14]: s6[2]
```

```
Out[14]: 3
```

Можно использовать метку, тогда работа с *Series* будет похожа на работу со словарем (*dict*) в *Python*.

```
In [15]: s6['d']
Out[15]: 4
```

Доступно получение *slice*'ов.

```
In [16]: s6[:2]
Out[16]:
a 1
b 2
dtype: int64
```

В поле для индекса можно поместить условное выражение.

```
In [17]: s6[s6 <= 3]
Out[17]:
a 1
b 2
c 3
dtype: int64
```

Со структурами *Series* можно работать как с векторами: складывать, умножать вектор на число и т.п.

```
In [18]: s7 = pd.Series([10, 20, 30, 40, 50], ['a', 'b', 'c', 'd', 'e'])
In [19]: s6 + s7
Out[19]:
a 11
b 22
c 33
d 44
e 55
dtype: int64

In [20]: s6 * 3
Out[20]:
a 3
b 6
c 9
d 12
```

```
e 15
dtype: int64
```

Структура данных *DataFrame*

Если *Series* представляет собой одномерную структуру, которую для себя можно представить как таблицу с одной строкой, то *DataFrame* – это уже двумерная структура – полноценная таблица с множеством строк и столбцов.

Перед работой с *DataFrame* не забудьте импортировать библиотеку *pandas*.

Конструктор класса *DataFrame* выглядит так:

```
class pandas.DataFrame(data=None, index=None, columns=None, dtype=None, copy=False)
```

data – массив *ndarray*, словарь (*dict*) или другой *DataFrame*;

index – список меток для записей (имена строк таблицы);

columns – список меток для полей (имена столбцов таблицы);

dtype – объект *numpy.dtype*, определяющий тип данных;

copy – создает копию массива данных, если параметр равен *True* в ином случае ничего не делает.

Структуру *DataFrame* можно создать на базе:

- словаря (*dict*) в качестве элементов которого должны выступать: одномерные *ndarray*, списки, другие словари, структуры *Series*;
- двумерные *ndarray*;
- структуры *Series*;
- структурированные *ndarray*;
- другие *DataFrame*.

Рассмотрим на практике различные подходы к созданию *DataFrame'ов*.

Создание *DataFrame* из словаря

В данном случае будет использоваться одномерный словарь, элементами которого будут списки, структуры *Series* и т.д.

Начнем с Series.

```
In [3]: d = {"price":pd.Series([1, 2, 3], index=['v1', 'v2', 'v3']),
   ...: "count": pd.Series([10, 12, 7], index=['v1', 'v2', 'v3'])}
In [4]: df1 = pd.DataFrame(d)

In [5]: print(df1)
   count price
v1    10     1
v2    12     2
v3     7     3

In [6]: print(df1.index)
```

```
Index(['v1', 'v2', 'v3'], dtype='object')

In [7]: print(df1.columns)
Index(['count', 'price'], dtype='object')
```

Теперь построим аналогичный словарь, но на элементах *ndarray*.

```
In [8]: d2 = {"price":np.array([1, 2, 3]),
  ...: "count": np.array([10, 12, 7])}
In [9]: df2 = pd.DataFrame(d2, index=['v1', 'v2', 'v3'])
In [10]: print(df2)
   count price
v1    10     1
v2    12     2
v3     7     3

In [11]: print(df2.index)
Index(['v1', 'v2', 'v3'], dtype='object')

In [12]: print(df2.columns)
Index(['count', 'price'], dtype='object')
```

Как видно – результат аналогичен предыдущему. Вместо *ndarray* можно использовать обычный список из *Python*.

Создание *DataFrame* из списка словарей

До это мы создавали *DataFrame* из словаря, элементами которого были структуры *Series*, списки и массивы, сейчас мы создадим *DataFrame* из списка, элементами которого являются словари.

```
In [13]: d3 = [{"price": 3, "count":8}, {"price": 4, "count": 11}]
In [14]: df3 = pd.DataFrame(d3)
In [15]: print(df3)
  count price
0     8     3
1    11     4

In [16]: print(df3.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2 entries, 0 to 1
Data columns (total 2 columns):
count 2 non-null int64
price 2 non-null int64
dtypes: int64(2)
memory usage: 112.0 bytes
None
```

Создание *DataFrame* из двумерного массива

Создать *DataFrame* можно также и из двумерного массива, в нашем примере это будет *ndarray* из библиотеки *numpy*.

```
In [17]: nda1 = np.array([[1, 2, 3], [10, 20, 30]])
In [18]: df4 = pd.DataFrame(nda1)
In [19]: print(df4)
   0  1  2
0  1  2  3
1 10 20 30
```

Работа с элементами *DataFrame*

Работа с элементами *DataFrame* – доступ к элементам данной структуры – тема достаточно обширная и она будет освещена в одном из ближайших уроков. Сейчас мы рассмотрим наиболее часто используемые способы работы с элементами *DataFrame*.

Основные подходы представлены в таблице ниже.

Операция	Синтаксис	Возвращаемый результат
Выбор столбца	df[col]	*Series*
Выбор строки по метке	df.loc[label]	*Series*
Выбор строки по индексу	df.iloc[loc]	*Series*
Слайс по строкам	df[0:4]	*DataFrame*
Выбор строк, отвечающих условию	df[bool_vec]	*DataFrame*

Теперь посмотрим, как использовать данные операций на практике.

Для начала создадим *DataFrame*.

```
In [3]: d = {"price":np.array([1, 2, 3]),
   ...: "count": np.array([10, 20, 30])}
In [4]: df = pd.DataFrame(d, index=['a', 'b', 'c'])
In [5]: print(df)
  count price
a    10     1
b    20     2
c    30     3
```

Операция: выбор столбца.

```
In [6]: df['count']
Out[6]:
a 10
b 20
c 30
Name: count, dtype: int32
```

Операция: выбор строки по метке.

```
In [7]: df.loc['a']
Out[7]:
count 10
price 1
Name: a, dtype: int32
```

Операция: выбор строки по индексу.

```
In [8]: df.iloc[1]
Out[8]:
count 20
price 2
Name: b, dtype: int32
```

Операция: slice по строкам.

```
In [9]: df[0:2]
Out[9]:
  count price
a    10     1
b    20     2
```

Операция: выбор строк, отвечающих условию.

```
In [10]: df[df['count'] >= 20]
Out[10]:
  count price
b    20     2
c    30     3
```

Доступ к данным в структурах Pandas

При работе со структурами *Series* и *DataFrame* из библиотеки *pandas,* как правило, используют два основных способа получения значений элементов.

Первый способ основан на использовании меток, в этом случае работа ведется через метод *.loc*. Если вы обращаетесь к отсутствующей метке, то будет сгенерировано исключение *KeyError*.

Такой подход позволяет использовать:

- метки в виде отдельных символов [‘*a*’] или чисел [5], числа используются в качестве меток, если при создании структуры не был указан список с метками;
- список меток [‘*a*’, ‘*b*’, ‘*c*’];
- слайс меток [‘*a*’:’*c*’];
- массив логических переменных;
- *callable* функция с одним аргументом.

Второй способ основан на использовании целых чисел для доступа к данных, он предоставляется через метод *.iloc*. При использовании *.iloc*, если вы обращаетесь к несуществующему элементу, то будет сгенерировано исключение *IndexError*. Логика использования *.iloc* очень похожа на работу с *.loc*. При таком подходе можно использовать:

- отдельные целые числа для доступа к элементам структуры;
- массивы целых чисел [0, 1, 2];
- слайсы целых чисел [1:4];
- массивы логических переменных;
- *callable* функцию с одним аргументом.

В зависимости от типа используемой структуры, будет меняться форма *.loc*:

- для *Series*, она выглядит так: *s.loc[indexer]*;
- для *DataFrame* так: *df.loc[row_indexer, column_indexer]*.

Использование различных способов доступа к данным

Создадим объекты типов *Series* и *DataFrame*, которые в дальнейшем будут использованы для экспериментов. Для этого сначала импортируем необходимые библиотеки.

```
In [1]: import pandas as pd
```

Создадим структуру *Series*.

```
In [3]: s = pd.Series([10, 20, 30, 40, 50], ['a', 'b', 'c', 'd', 'e'])

In [4]: s['a']
Out[4]: 10

In [5]: s
Out[5]:
```

```
a 10
b 20
c 30
d 40
e 50
dtype: int64
```

Создадим структуру *DataFrame*.

```
In [6]: d = {"price":[1, 2, 3], "count": [10, 20, 30], "percent": [24, 51, 71]}

In [7]: df = pd.DataFrame(d, index=['a', 'b', 'c'])

In [8]: df
Out[8]:
  count percent price
a    10      24     1
b    20      51     2
c    30      71     3
```

Доступ к данным структуры *Series*

Доступ с использованием меток

При использовании меток для доступа к данным можно применять один из следующих подходов:

- первый, когда вы записываете имя переменной типа *Series* и в квадратных скобках указываете метку, по которой хотите обратиться (пример: *s[‘a’]*);
- второй, когда после имени переменной пишите *.loc* и далее указываете метку в квадратных скобках (пример: *s.loc[‘a’]*).

Обращение по отдельной метке.

Получение элементов с меткой *‘a’*:

```
In [9]: s['a']
Out[9]: 10
```

Обращение по массиву меток.

Получение элементов с метками *‘a’, ‘c’* и *‘e’*:

```
In [10]: s[['a', 'c', 'e']]
Out[10]:
a 10
c 30
e 50
dtype: int64
```

Обращение по слайсу меток.
Получение элементов структуры с метками от *‘a’* до *‘e’*:

```
In [11]: s['a':'e']
Out[11]:
a 10
b 20
c 30
d 40
e 50
dtype: int64
```

Доступ с использованием целочисленных индексов

При работе с целочисленными индексами, индекс можно ставить сразу после имени переменной в квадратных скобках (пример: *s[1]*), или можно воспользоваться *.iloc* (пример: *s.iloc[1]*).

Обращение по отдельному индексу.
Получение элемента с индексом 1:

```
In [12]: s[1]
Out[12]: 20
```

Обращение с использованием списка индексов.
Получение элементов с индексами 1, 2 и 3.

```
In [13]: s[[1, 2 ,3]]
Out[13]:
b 20
c 30
d 40
dtype: int64
```

Обращение по слайсу индексов.
Получение первых трех элементов структуры:

```
In [14]: s[:3]
Out[14]:
a 10
b 20
c 30
dtype: int64
```

Обращение через *callable* функцию

При таком подходе в квадратных скобках указывается не индекс или метка, а функция (как правило, это лямбда функция), которая используется для выборки элементов структуры.

Получение всех элементов, значение которых больше либо равно 30:

```
In [15]: s[lambda x: x>= 30]
Out[15]:
c 30
d 40
e 50
dtype: int64
```

Обращение через логическое выражение

Данный подход похож на использование *callable* функции: в квадратных скобках записывается логическое выражение, согласно которому будет произведен отбор.

Получение всех элементов, значение которых больше 30:

```
In [16]: s[s > 30]
Out[16]:
d 40
e 50
dtype: int64
```

Доступ к данным структуры *DataFrame*

Доступ с использованием меток

Рассмотрим различные варианты использования меток, которые могут являться как именами столбцов таблицы, так и именами строк.

Обращение к конкретному столбцу.

Получение всех элементов столбца *‘count’*:

```
In [17]: df['count']
Out[17]:
a 10
b 20
c 30
Name: count, dtype: int64
```

Обращение с использованием массива столбцов.

Получение элементов столбцов *‘count’* и *‘price’*:

```
In [18]: df[['count','price']]
Out[18]:
   count  price
a     10      1
b     20      2
```

```
c    30    3
```

Обращение по слайсу меток.

Получение элементов с метками от *‘a’* до *‘b’*.

```
In [19]: df['a':'b']
Out[19]:
   count percent price
a    10      24    1
b    20      51    2
```

Обращение через *callable* функцию

Подход в работе с *callable* функцией для *DataFrame* аналогичен тому, что используется для *Series*, только при формировании условий необходимо дополнительно указывать имя столбца.

Получение всех элементов, у которых значение в столбце *‘count’* больше 15:

```
In [20]: df[lambda x: x['count'] > 15]
Out[20]:
  count percent price
b   20      51    2
c   30      71    3
```

Обращение через логическое выражение

При формировании логического выражения необходимо указывать имена столбцов, также как и при работе с *callable* функциями, по которым будет производиться выборка.

Получить все элементы, у которых *‘price’* больше либо равен 2.

```
In [21]: df[df['price'] >= 2]
Out[21]:
  count percent price
b   20      51    2
c   30      71    3
```

Использование атрибутов для доступа к данным

Для доступа к данным можно использовать атрибуты структур, в качестве которых выступают метки.

Начнем со структуры *Series*.

Воспользуемся уже знакомой нам структурой.

```
In [3]: s = pd.Series([10, 20, 30, 40, 50], ['a', 'b', 'c', 'd', 'e'])
```

Для доступа к элементу через атрибут необходимо указать его через точку после имени переменной.

```
In [4]: s.a
Out[4]: 10

In [5]: s.c
Out[5]: 30
```

Т.к. структура *s* имеет метки *‘a’, ‘b’, ‘c’, ‘d’, ‘e’*, то для доступа к элементу с меткой *‘a’* мы может использовать синтаксис *s.a*.

Этот же подход можно применить для переменной типа *DataFrame*.

```
In [6]: d = {"price":[1, 2, 3], "count": [10, 20, 30], "percent": [24, 51, 71]}
In [6]: df = pd.DataFrame(d, index=['a', 'b', 'c'])
```

Получим доступ к столбцу *‘price’*.

```
In [7]: df.price
Out[7]:
a 1
b 2
c 3
Name: price, dtype: int64
```

Получение случайного набора из структур *pandas*

Библиотека *pandas* предоставляет возможность получить случайный набор данных из уже существующей структуры. Такой функционал предоставляет как *Series* так и *DataFrame*. У данных структур есть метод *sample()*, предоставляющий случайную подвыборку.

Начнем наше знакомство с этим методом на примере структуры *Series*.

```
In [3]: s = pd.Series([10, 20, 30, 40, 50], ['a', 'b', 'c', 'd', 'e'])
```

Для того, чтобы выбрать случайным образом элемент из *Series* воспользуйтесь следующим синтаксисом.

```
In [4]: s.sample()
Out[4]:
c 30
dtype: int64
```

Можно сделать выборку из нескольких элементов, для этого нужно передать нужное количество через параметр *n*.

```
In [5]: s.sample(n=3)
Out[5]:
e    50
a    10
b    20
dtype: int64
```

Также есть возможность указать долю от общего числа объектов в структуре, используя параметр *frac*.

```
In [6]: s.sample(frac=0.3)
Out[6]:
a 10
c 30
dtype: int64
```

Очень интересной особенностью является то, что мы можем передать вектор весов, длина которого должна быть равна количеству элементов в структуре. Сумма весов должна быть равна единице, вес, в данном случае, это вероятность появления элемента в выборке.

В нашей тестовой структуре пять элементов, сформируем вектор весов для нее и сделаем выборку из трех элементов.

```
In [7]: w = [0.1, 0.2, 0.5, 0.1, 0.1]

In [8]: s.sample(n = 3, weights=w)
Out[8]:
c    30
e    50
a    10
dtype: int64
```

Данный функционал также доступен и для структуры *DataFrame*.

```
In [9]: d = {"price":[1, 2, 3, 5, 6], "count": [10, 20, 30, 40, 50], "percent": [24, 51, 71, 25, 42]}

In [10]: df = pd.DataFrame(d)

In [11]: df.sample()
Out[11]:
  count percent price
0    10      24     1
```

При работе с *DataFrame* можно указать ось.

```
df.sample(axis=1)
```

```
Out[12]:
   count
0     10
1     20
2     30
3     40
4     50
In [13]: df.sample(n=2, axis=1)
Out[13]:
  count price
0    10     1
1    20     2
2    30     3
3    40     5
4    50     6
In [14]: df.sample(n=2)
Out[14]:
  count percent price
4    50      42     6
3    40      25     5
```

Добавление элементов в структуры

Увеличение размера структуры – т.е. добавление новых, дополнительных, элементов – это довольно распространенная задача. В *pandas* она решается очень просто. И самый быстрый способ понять, как это делать – попробовать реализовать эту задачу на практике.

Добавление нового элемента в структуру *Series*.

```
In [3]: s = pd.Series([10, 20, 30, 40, 50], ['a', 'b', 'c', 'd', 'e'])
In [4]: s
Out[4]:
a 10
b 20
c 30
d 40
e 50
dtype: int64
In [5]: s['f'] = 60

In [6]: s
Out[6]:
a 10
b 20
c 30
d 40
```

```
e 50
f 60
dtype: int64
```

Добавление нового элемента в структуру *DataFrame*.

```
In [3]: d = {"price":[1, 2, 3, 5, 6], "count": [10, 20, 30, 40, 50], "percent": [24, 51, 71, 25, 42]}

In [4]: df
Out[4]:
  count percent price
0    10      24     1
1    20      51     2
2    30      71     3
3    40      25     5
4    50      42     6

In [5]: df['value'] = [3, 14, 7, 91, 5]

In [6]: df
Out[6]:
  count percent price value
0    10      24     1     3
1    20      51     2    14
2    30      71     3     7
3    40      25     5    91
4    50      42     6     5
```

Индексация с использованием логических выражений

На практике очень часто приходится получать определенную подвыборку из существующего набора данных. Например, получить все товары, скидка на которые больше пяти процентов, или выбрать из базы информацию о сотрудниках мужского пола старше 30 лет. Это очень похоже на процесс фильтрации при работе с таблицами или получение выборки из базы данных. Похожий функционал реализован в *pandas* и мы уже касались этого вопроса, когда рассматривали различные подходы к индексации.

Условное выражение записывается вместо индекса в квадратных скобках при обращении к элементам структуры.

При работе с *Series* возможны следующие варианты использования.

```
In [3]: s = pd.Series([10, 20, 30, 40, 50, 10, 10], ['a', 'b', 'c', 'd', 'e', 'f', 'g'])
In [4]: s[s>30]
Out[4]:
d 40
```

```
e 50
dtype: int64

In [5]: s[s==10]
Out[5]:
a 10
f 10
g 10
dtype: int64

In [6]: s[(s>=30) & (s<50)]
Out[6]:
c 30
d 40
dtype: int64
```

При работе с *DataFrame* необходимо указывать столбец по которому будет производиться фильтрация (выборка).

```
In [3]: d = {"price":[1, 2, 3, 5, 6], "count": [10, 20, 30, 40, 50], "percent": [24, 51, 71, 25, 42],
   ...: "cat":["A", "B", "A", "A", "C"]}

In [4]: df = pd.DataFrame(d)

In [5]: df
Out[5]:
  cat count percent price
0   A    10     24     1
1   B    20     51     2
2   A    30     71     3
3   A    40     25     5
4   C    50     42     6

In [6]: df[df["price"] > 3]
Out[6]:
  cat count percent price
3   A    40     25     5
4   C    50     42     6
```

В качестве логического выражения можно использовать довольно сложные конструкции с использованием *map*, *filter*, лямбда-выражений и т.п.

```
In [7]: fn = df["cat"].map(lambda x: x == "A")
```

```
In [8]: df[fn]
Out[8]:
  cat  count  percent  price
0   A     10       24      1
2   A     30       71      3
3   A     40       25      5
```

Использование *isin* для работы с данными в *pandas*

По структурам данных *pandas* можно строить массивы с данными типа *boolean*, по которому можно проверить наличие или отсутствие того или иного элемента. Проще всего это показать на примере.

```
In [3]: s = pd.Series([10, 20, 30, 40, 50, 10, 10], ['a', 'b', 'c', 'd', 'e', 'f', 'g'])

In [4]: s.isin([10, 20])
Out[4]:
a True
b True
c False
d False
e False
f True
g True
dtype: bool
```

Работа с *DataFrame* аналогична работе со структурой *Series*.

```
In [3]: df = pd.DataFrame({"price":[1, 2, 3, 5, 6], "count": [10, 20, 30, 40, 50], "percent": [24, 51, 71, 25, 42]})

In [4]: df.isin([1, 3, 25, 30, 10])
Out[4]:
  count percent price
0  True   False  True
1 False   False False
2  True   False  True
3 False    True False
4 False   False False
```

Работа с пропусками в данных

Создадим структуру *DataFrame*, которая будет содержать пропуски. Для этого импортируем необходимые нам библиотеки.

```
In [1]: import pandas as pd
```

```
In [2]: from io import StringIO
```

После этого создадим объект в формате *csv*. *CSV* – это один из наиболее простых и распространенных форматов хранения данных, в котором элементы отделяются друг от друга запятыми, более подробно о нем можете прочитать здесь.

```
In [3]: data = 'price,count,percent\n1,10,\n2,20,51\n3,30,'
In [4]: df = pd.read_csv(StringIO(data))
```

Полученный объект *df* – это *DataFrame* с пропусками.

```
In [5]: df
Out[5]:
  price count percent
0     1    10     NaN
1     2    20    51.0
2     3    30     NaN
```

В нашем примере, у объектов с индексами 0 и 2 отсутствуют данные в поле *percent*. Отсутствующие данные помечаются как *NaN*. Добавим к существующей структуре еще один объект (запись), у которого будет отсутствовать значение в поле *count*.

```
In [6]: df.loc[3] = {'price':4, 'count':None, 'percent':26.3}

In [7]: df
Out[7]:
  price count percent
0   1.0  10.0     NaN
1   2.0  20.0    51.0
2   3.0  30.0     NaN
3   4.0   NaN    26.3
```

Для начала обратимся к методам из библиотеки *pandas*, которые позволяют быстро определить наличие элементов *NaN* в структурах. Если таблица небольшая, то можно использовать библиотечный метод *isnull*. Выглядит это так.

```
In [8]: pd.isnull(df)
Out[8]:
  price count percent
0 False False    True
1 False False   False
2 False False    True
3 False True    False
```

Таким образом мы получаем таблицу того же размера, но на месте реальных данных в ней находятся логические переменные, которые принимают значение *False*, если значение поля у объекта есть, или *True*, если значение в данном поле – это *NaN*. В дополнение к этому можно посмотреть подробную информацию об объекте, для этого можно воспользоваться методом *info()*.

```
In [9]: df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4 entries, 0 to 3
Data columns (total 3 columns):
price 4 non-null float64
count 3 non-null float64
percent 2 non-null float64
dtypes: float64(3)
memory usage: 128.0 bytes
```

В нашем примере видно, что объект *df* имеет три столбца (*count, percent* и *price*), при этом в столбце *price* все объекты значимы – не *NaN*, в столбце *count* – один *NaN* объект, в поле percent – два *NaN* объекта. Можно воспользоваться следующим подходом для получения количества NaN элементов в записях.

```
In [10]: df.isnull().sum()
Out[10]:
price 0
count 1
percent 2
dtype: int64
```

Замена отсутствующих данных

Отсутствующие данные объектов можно заменить на конкретные числовые значения, для этого можно использовать метод *fillna()*. Для экспериментов будем использовать структуру *df*, созданную в предыдущем разделе.

```
In [11]: df.isnull().sum()
Out[11]:
price 0
count 1
percent 2
dtype: int64

In [12]: df
Out[12]:
  price count percent
```

```
0   1.0  10.0    NaN
1   2.0  20.0   51.0
2   3.0  30.0    NaN
3   4.0   NaN   26.3

In [13]: df.fillna(0)
Out[13]:
  price count percent
0   1.0  10.0     0.0
1   2.0  20.0    51.0
2   3.0  30.0     0.0
3   4.0   0.0    26.3
```

Этот метод не изменяет текущую структуру, он возвращает структуру *DataFrame*, созданную на базе существующей, с заменой *NaN* значений на те, что переданы в метод в качестве аргумента. Данные можно заполнить средним значением по столбцу.

```
In [14]: df.fillna(df.mean())
Out[14]:
  price count percent
0   1.0  10.0   38.65
1   2.0  20.0   51.00
2   3.0  30.0   38.65
3   4.0  20.0   26.30
```

В зависимости от задачи используется тот или иной метод заполнения отсутствующих элементов, это может быть нулевое значение, математическое ожидание, медиана и т.п. Для замены *NaN* элементов на конкретные значения, можно использовать интерполяцию, которая реализована в методе *interpolate()*, алгоритм интерполяции задается через аргументы метода.

Удаление объектов/столбцов с отсутствующими данными

Довольно часто используемый подход при работе с отсутствующими данными – это удаление записей (строк) или полей (столбцов), в которых встречаются пропуски. Для того, чтобы удалить все объекты, которые содержат значения *NaN* воспользуйтесь методом *dropna()* без аргументов.

```
In [15]: df.dropna()
Out[15]:
  price count percent
1   2.0  20.0   51.0
```

Вместо записей, можно удалить поля, для этого нужно вызвать метод *dropna* с аргументом *axis=1*.

```
In [16]: df.dropna()
```

```
Out[16]:
  price count percent
1   2.0  20.0    51.0
In [17]: df.dropna(axis=1)
Out[17]:
  price
0   1.0
1   2.0
2   3.0
3   4.0
```

pandas позволяет задать порог на количество не-*NaN* элементов. В приведенном ниже примере будут удалены все столбцы, в которых количество не-*NaN* элементов меньше трех.

```
In [18]: df.dropna(axis = 1, thresh=3)
Out[18]:
  price count
0   1.0  10.0
1   2.0  20.0
2   3.0  30.0
3   4.0   NaN
```

Лабораторные задания

1 Ознакомиться с библиотеками Python для анализа данных

2 Изучить принципы обработки данных с использованием библиотеки Pandas

3 Получить у преподавателя индивидуальное задание на обработку данных с использованием Pandas.

4 Оформить отчёт

ЛАБОРАТОРНАЯ РАБОТА № 3.
ПЛАТФОРМА HADOOP И ЕЁ КОМПОНЕНТЫ

Цель работы

Цель лабораторной работы заключается в закреплении теоретических основ курса « Технологии обработки больших данных» и получении первоначальных навыков настройки и использования системы Hadoop.

Общие сведения

Hadoop — популярная программная платформа (software framework) построения распределенных приложений для массово-параллельной обработки (massive parallel processing, MPP) данных в рамках вычислительной парадигмы MapReduce.

Hadoop считается одним из основополагающих решений в области «больших данных» (big data). Вокруг Hadoop образовалась целая экосистема из связанных проектов и технологий.

Архитектура Hadoop

Hadoop состоит из четырёх модулей:

- связующее программное обеспечение Hadoop Common;
- распределённая файловая система HDFS;
- система для планирования заданий и управления кластером;
- платформа программирования и выполнения распределённых MapReduce вычислений Hadoop MapReduce.

В Hadoop Common входят библиотеки управления файловыми системами, поддерживаемыми Hadoop, и сценарии создания необходимой инфраструктуры и управления распределённой обработкой.

HDFS (Hadoop Distributed File System) — файловая система, предназначенная для хранения файлов больших размеров, поблочно распределённых между узлами вычислительного кластера. Все блоки в HDFS (кроме последнего блока файла) имеют одинаковый размер, и каждый блок может быть размещён на нескольких узлах, размер блока и коэффициент репликации (количество узлов, на которых должен быть размещён каждый блок) определяются в настройках на уровне файла. Благодаря репликации обеспечивается устойчивость распределённой системы к отказам отдельных узлов. Файлы в HDFS могут быть записаны лишь однажды (модификация не поддерживается), а запись в файл в одно время может вести только один процесс. Организация файлов в пространстве имён — традиционная иерархическая: есть корневой каталог, поддерживается вложение каталогов, в одном каталоге могут располагаться и файлы, и другие каталоги.

Развёртывание экземпляра HDFS предусматривает наличие центрального узла имён (name node), хранящего метаданные файловой системы и метаинформацию о распределении блоков, и серии узлов данных (data node), непосредственно хранящих блоки файлов. Узел имён отвечает за обработку операций уровня файлов и каталогов — открытие и закрытие файлов, манипуляция с каталогами, узлы данных непосредственно отрабатывают операции по записи и чтению данных. Узел имён и узлы данных снабжаются веб-серверами, отображающими текущий статус узлов и позволяющими просматривать содержимое файловой системы.

HDFS является неотъемлемой частью проекта, однако, Hadoop поддерживает работу и с другими распределёнными файловыми системами без использования HDFS, поддержка Amazon S3 и CloudStore реализована в основном дистрибутиве. С другой стороны, HDFS может использоваться не только для запуска MapReduce-заданий, но и как распределённая файловая система общего назначения, в частности, поверх неё реализована распределённая NoSQL-СУБД HBase, в её среде работает масштабируемая система машинного обучения Apache Mahout.

YARN (англ. Yet Another Resource Negotiator — «ещё один ресурсный посредник») — модуль, появившийся с версией 2.0 Hadoop, отвечающий за управление ресурсами кластеров и планирование заданий. Если в предыдущих выпусках эта функция была интегрирована в модуль MapReduce, где была реализована единым компонентом (JobTracker), то в YARN функционирует логически самостоятельный демон — планировщик ресурсов (ResourceManager), абстрагирующий все вычислительные ресурсы кластера и управляющий их предоставлением приложениям распределённой обработки. Работать под управлением YARN могут как MapReduce-программы, так и любые другие распределённые приложения, поддерживающие соответствующие программные интерфейсы. YARN обеспечивает возможность параллельного выполнения нескольких различных задач в рамках кластера и их изоляцию (по принципам мультиарендности). Разработчику распределённого приложения необходимо реализовать специальный класс управления приложением (ApplicationMaster), который отвечает за координацию заданий в рамках тех ресурсов, которые предоставит планировщик ресурсов; планировщик ресурсов же отвечает за создание экземпляров класса управления приложением и взаимодействия с ним через соответствующий сетевой протокол.

Hadoop MapReduce — программный каркас для программирования распределённых вычислений в рамках парадигмы MapReduce. Разработчику приложения для Hadoop MapReduce необходимо реализовать базовый обработчик, который на каждом вычислительном узле кластера обеспечит преобразование исходных пар «ключ — значение» в промежуточный набор пар «ключ — значение» (класс, реализующий интерфейс Mapper, назван по функции высшего порядка Map), и обработчик, сводящий промежуточный набор пар в окончательный, сокращённый набор (свёртку, класс, реализующий интерфейс Reducer). Каркас передаёт на вход свёртки отсортированные выводы

от базовых обработчиков, сведе́ние состоит из трёх фаз — shuffle (тасовка, выделение нужной секции вывода), sort (сортировка, группировка по ключам выводов от распределителей — досортировка, требующаяся в случае, когда разные атомарные обработчики возвращают наборы с одинаковыми ключами, при этом правила сортировки на этой фазе могут быть заданы программно и использовать какие-либо особенности внутренней структуры ключей) и собственно reduce (свёртка списка) — получения результирующего набора. Для некоторых видов обработки свёртка не требуется, и каркас возвращает в этом случае набор отсортированных пар, полученных базовыми обработчиками.

Hadoop MapReduce позволяет создавать задания как с базовыми обработчиками, так и со свёртками, написанными без использования Java: утилиты Hadoop streaming позволяют использовать в качестве базовых обработчиков и свёрток любой исполняемый файл, работающий со стандартным вводом-выводом операционной системы (например, утилиты командной оболочки UNIX), есть также SWIG-совместимый прикладной интерфейс программирования Hadoop pipes на C++. Также, в состав дистрибутивов Hadoop входят реализации различных конкретных базовых обработчиков и свёрток, наиболее типично используемых в распределённой обработке.

Введение в MapReduce

Парадигма MapReduce была предложена Google в статье Джефри Дина и Санжая Чемавата «MapReduce: упрощенная обработка данных на больших кластерах» (Jeffrey Dean and Sanjay Ghemawat MapReduce: Simplied Data Processing on Large Clusters). Данная статья носит довольно поверхностный характер. Ее недосказанности компенсирует работа Ральфа Ламмеля из Исследовательского центра Microsoft в Редмонде «И снова модель программирования MapReduce компании Google» (Ralf Lammel Google's MapReduce Programming Model – Revisited). Во введении Ламмель заявляет, что он проанализировал статью Дина и Чемавата, которую он неоднократно и с почтением называет судьбоносной, используя метод, который называется «обратной инженерией» (reverse engineering). То есть, он постарался раскрыть смысл того, что же в ней на самом деле представлено. Ламмель увязывает MapReduce с функциональным программированием, языками Лисп и Haskell, что естественным образом вводит читателя в контекст.

MapReduce – это скорее инфраструктурное решение, способное эффективно использовать сегодня кластерные, а в будущем многоядерные архитектуры.

Большинство современных параллельных компьютеров принадлежит к категории «много потоков команд, много потоков данных» (Multiple Program Multiple Data, MPMD). Каждый узел выполняет фрагмент общего задания, работая со своими собственными данными, а в дополнение существует сложная система обмена сообщениями, обеспечивающая согласование совместной работы. Такой вид параллелизма раньше называют параллелизмом на уровне

задач (task level parallelism), но иногда его еще называют функциональным параллелизмом или параллелизмом по управлению. Его реализация сводится к распределению заданий по узлам и обеспечению синхронности происходящих процессов.

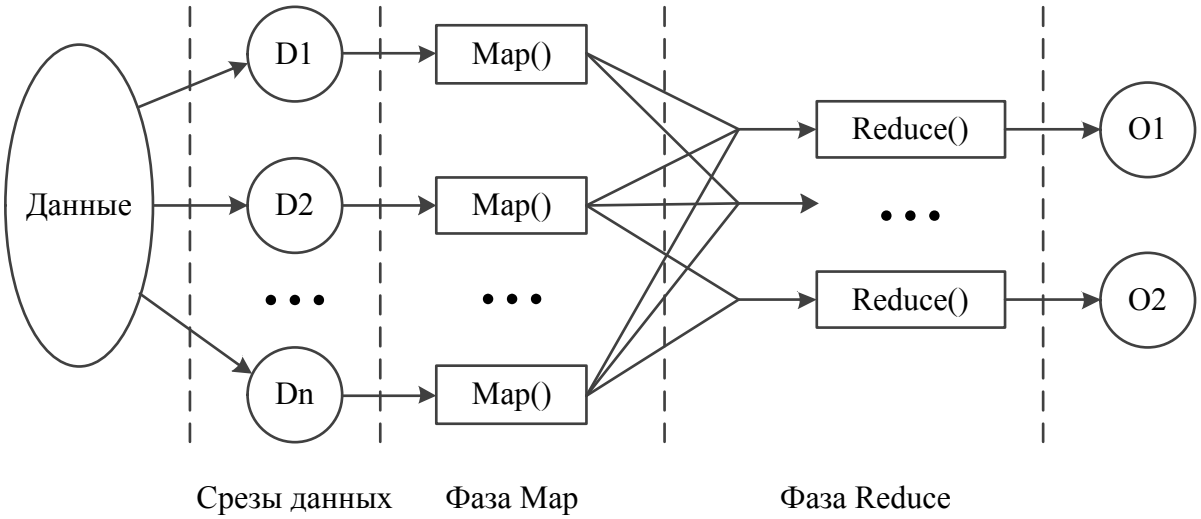
Альтернативный тип параллелизма называют параллелизмом данных (data parallelism), который попадает в категорию «один поток команд, много потоков данных» (Single Program Multiple Data, SPMD). Реализация SPMD предполагает, что сначала данные должны быть каким-то образом распределены между процессорами, обработаны, а затем собраны. Эту совокупность операций можно назвать map-reduce, как принято в функциональном программировании, хотя точнее было бы split-aggregate, то есть разбиение и сборка, но привилось первое. Возможны различные способы реализации SPMD.

Реализация SPMD требует выделения ведущей части кода, ее называют Master или Manager (далее менеджер), и подчиненных ей частей Worker (далее исполнитель). Менеджер «раздает» задания исполнителям и потом их собирает. До появления MapReduce эта модель рассматривалась как малоперспективная из-за наличия «бутылочного горла» на тракте обмена между множеством исполнителей и одним менеджером. Создание MapReduce разрешило эту проблему и открыло возможность для обработки огромных массивов данных с использованием архитектуры SPMD.

Допустим, что следует решить простейшую задачу: обработать массив, разбив его на подмассивы. В таком случае работа менеджера сводится к тому, что он делит этот массив на части, посылает каждому исполнителю положенный ему подмассив, а затем получает результаты и объединяет их (рисунок 1). В функцию исполнителя входит получение данных, обработка и возврат результатов менеджеру. Распределение нагрузок может быть статическим (static load balancing) или динамическим (dynamic load balancing). Парадигма создана по мотивам комбинации map и reduce в функциональном языке программирования Липс. В нем map использует в качестве входных параметров функцию и набор значений, применяя эту функцию по отношению к каждому из значений, а reduce комбинирует полученные результаты.

Канонический пример приложения, написанного с помощью MapReduce это процесс, подсчитывающий, сколько раз различные слова встречаются в наборе документов:

```
// Функция, используемая исполнителями на Map-фазе
// для обработки пар ключ-значение из входного потока
void map(String name, String document):
    // Входные данные:
    //   name - название документа
    //   document - содержимое документа
    for each word w in document:
        EmitIntermediate(w, "1");
```



~ Упрощенная схема потоков данных в парадигме MapReduce

```
// Функция, используемая исполнителями на Reduce-фазе
// для обрабоки пар ключ-значение, полученных на Map-шаге
void reduce(String word, Iterator partialCounts):
    // Входные данные:
    //   word - слово
    //   partialCounts - список группированных промежуточных результатов.
    //       Количество записей в partialCounts и есть требуемое значение
    int result = 0;
    for each v in partialCounts:
        result += parseInt(v);
    Emit(AsString(result));
```

В этом коде на map-фазе каждый документ разбивается на слова, и возвращаются пары, где ключом является само слово, а значением — «1». Если в документе одно и то же слово встречается несколько раз, то в результате предварительной обработки этого документа будет столько же этих пар, сколько раз встретилось это слово.

Далее выполняется объединение всех пар с одинаковым ключом, и они передаются на вход функции reduce, которой остается сложить их, чтобы получить общее количество вхождений данного слова во все документы.

Созданная в Google конструкция MapReduce делает примерно то же, но по отношению к гигантским объемам данных. В этом случае map – это функция запроса от пользователя, помещенная в библиотеку MapReduce. Ее работа сводится к выбору входной пары, ее обработке и формированию результата в виде значения и некоторого промежуточного ключа, служащего указателем для reduce. Конструкция MapReduce собирает вместе все значения с одинаковыми промежуточными ключами и передает их в функцию reduce, также написанную пользователем. Эта функция воспринимает промежуточные значения, каким-то образом их собирает и воспроизводит результат, скорее всего выраженный меньшим количеством значений, чем входное множество.

Теперь свяжем функциональную идею MapReduce со схемой SPMD. Вызовы map распределяются между множеством машин путем деления входного потока данных на M срезов (splits или shard), каждый срез

обрабатывается на отдельной машине. Вызовы reduce распределены на R частей, количество которых определяется пользователем.

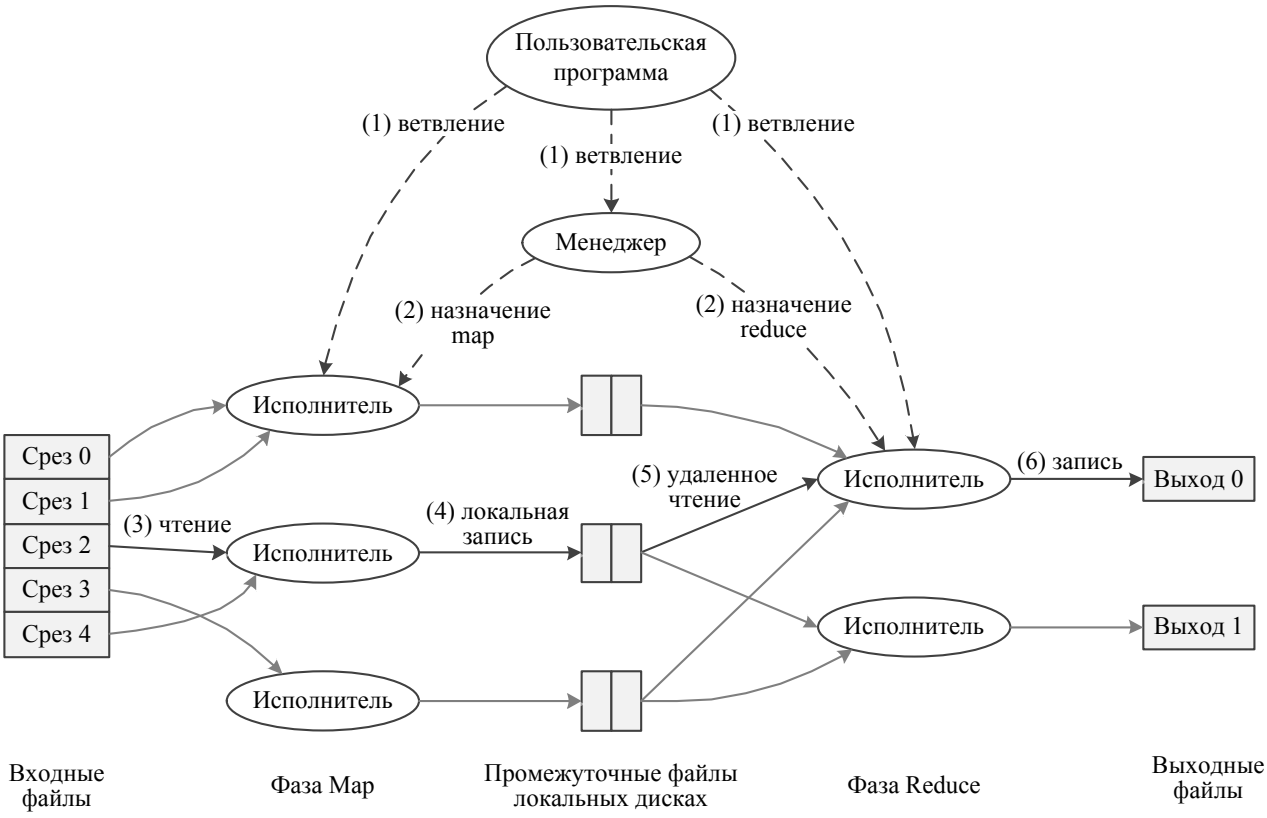
При вызове функции из библиотеки MapReduce выполняется примерно такая последовательность операций (рисунок 2):

1) входные файлы разбиваются на срезы размером от 16 до 64 Мбайт каждый, и на кластере запускаются копии программы. Одна из них менеджер, а остальные – исполнители. Всего создается M задач map и R задач reduce. Поиском свободных узлов и назначением на них задач занимается менеджер;

2) в процессе исполнения исполнители, назначенные на выполнение задачи map, считывают содержимое соответствующих срезов, осуществляют их грамматический разбор, выделяют отдельные пары «ключ и соответствующее ему значение», а потом передают эти пары в обрабатывающую их функцию map. Промежуточное значение в виде идентификатора и значения буферизуется в памяти;

3) периодически буферизованные пары сбрасываются на локальный диск, разделенный на R областей. Расположение этих пар передается менеджеру, который отвечает за дальнейшую передачу этих адресов исполнителям, выполняющим reduce;

4) исполнители, выполняющие задачу reduce, ждут сообщения от менеджера о местоположении промежуточных пар. По его получении они, используя процедуры удаленного вызова, считывают буферизованные данные с локальных дисков тех исполнителей, которые выполняют map. Загрузив все промежуточные данные, исполнитель сортирует их по промежуточным ключам и, если есть необходимость, группирует;



Ход выполнения вызова MapReduce

5) исполнитель обрабатывает данные по промежуточным ключам и передает их в соответствующую функцию reduce для вывода результатов;

6) когда все задачи map и reduce завершаются, конструкция MapReduce возобновляет работу вызывающей программы и та продолжает выполнять пользовательский код.

Одно из важнейших преимуществ этой, замысловатой на первый взгляд, конструкции состоит в том, что она позволяет надежно работать на платформах с низкими показателями надежности. Для обнаружения потенциальных сбоев менеджер периодически опрашивает исполнителей, и, если какой-то из них задерживается с ответом сверх заданного норматива, менеджер считает его дефектным и передает исполнение на свободные узлы. Различие между задачами map и reduce в данном случае состоит в том, что map хранит промежуточные результаты на своем диске, и выход из строя такого узла приводит к их потере и требует повторного запуска, а reduce хранит свои данные в глобальном хранилище.

Порядок выполнения работы

1 Установка операционной системы

В качестве операционной системы для нашего кластера будем использовать Ubuntu Server 16.04.3 LTS.

Все узлы будут работать на VirtualBox. Выставим следующие системные настройки для виртуальной машины: 10 GB пространства для жёсткого диска, два ядра и 1024 Мб памяти. Виртуальную машину можно оснастить двумя сетевыми адаптерами: один NAT, а другой для внутренней сети.

После того, как была скачена и установлена операционная система, необходимо обновиться и установить ssh и rsync:

```
sudo apt-get update && sudo apt-get upgrade
sudo apt-get install ssh
sudo apt-get install rsync
```

Для редактирования файлов с консоли будем использовать редактор nano. Для его установки введем команду:

```
sudo apt-get install nano
```

Для запуска:

```
nano файл
```

или если нужно редактировать системные файлы (с root правами), то

```
sudo nano файл
```

Для удобства также можно поставит оболочку Midnight Commander

```
sudo apt-get install mc
```

для ее запуска

```
mc
```

или если хотите редактировать системные файлы (с root правами), то

```
sudo mc
```

Изменим имя узла на master в файле /etc/hostname.

2 Установка Java

Далее необходимо установить OpenJDK 8 версии:

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

3 Создание отдельной учетной записи для запуска Hadoop

Мы будем использовать выделенную учетную запись для запуска Hadoop. Это не обязательно, но рекомендуется. Также предоставим новому пользователю права sudo, чтобы облегчить себе жизнь в будущем.

```
sudo addgroup hadoop
sudo adduser --ingroup hadoop hduser
sudo usermod -aG sudo hduser
```

Во время создания нового пользователя, необходимо будет ввести ему пароль.

4 Настройка статического IP адреса

Для дальнейшей работы нам потребуются IP-адреса серверов. Для того чтобы узнать IP-адрес, можно воспользоваться командой

```
ifconfig
```

Вместо использования динамически выделенных адресов более удобным может оказаться использование статических адресов. Для настройки статического IP-адреса замените в файле /etc/network/interfaces для соответствующего интерфейса «dhcp» на «static» и укажите значения адреса, маски сети, шлюза и адрес DNS сервера для соответствия требованиям вашей сети:

```
auto enp0s3
iface enp0s3 inet static
  address 192.168.0.1
  netmask 255.255.255.0
  gateway 192.168.0.254
  dns-nameservers 192.168.0.254
```

В приведенных далее примерах для серверов используются адреса вида 192.168.0.X.

5 Настройка доменного имени узла

Нам необходимо, чтобы все узлы могли легко обращаться друг к другу. В большом кластере желательно использовать dns сервер, но для нашей маленькой конфигурации подойдет файл /etc/hosts. В нем мы будем описывать соответствие ip-адреса узла к его имени в сети. Для одного узла ваш файл должен выглядеть примерно так:

```
127.0.0.1       localhost

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters

192.168.0.1 master
```

6 Настройка SSH

Для управления узлами кластера hadoop необходим доступ по ssh. Для созданного пользователя hduser предоставить доступ к master.
Для начала необходимо сгенерировать новый ssh ключ:

```
ssh-keygen -t rsa -P ""
```

Следующим шагом необходимо добавить созданный ключ в список авторизованных:

```
cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
```

Проверяем работоспособность, подключившись к себе:

```
ssh master
```

Чтобы вернуться в локальную сессию, просто наберите:

```
exit
```

7 Отключение IPv6

Если не отключить IPv6, то в последствии можно получить много проблем.
Для отключения IPv6 в Ubuntu 12.04 / 12.10 / 13.04 нужно отредактировать файл sysctl.conf:

```
sudo nano /etc/sysctl.conf
```

Добавляем следующие параметры:

```
# IPv6
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1
```

Сохраняем и перезагружаем операционную систему командой

```
sudo reboot
```

8 Установка Apache Hadoop

Скачаем необходимые файлы. Актуальные версии фреймворка располагаются по адресу: http://www.apache.org/dyn/closer.cgi/hadoop/common. Воспользуемся стабильной версией 2.7.4.

Создадим папку downloads в корневом каталоге и скачаем последнюю версию:

```
sudo mkdir /downloads
cd /downloads
sudo wget http://apache-mirror.rbc.ru/pub/apache/hadoop/common/stable/hadoop-2.7.4.tar.gz
```

Распакуем содержимое пакета в /usr/local/, переименуем папку и выдадим пользователю hduser права создателя:

```
sudo mv /downloads/hadoop-2.7.4.tar.gz /usr/local/
cd /usr/local/
sudo tar xzf hadoop-2.7.4.tar.gz
sudo mv hadoop-2.7.4 hadoop
sudo chown -R hduser:hadoop hadoop
```

9 Обновление $HOME/.bashrc

Для удобства, добавим в .bashrc список переменных:

```
#Hadoop variables
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64
export HADOOP_INSTALL=/usr/local/hadoop
export PATH=$PATH:$HADOOP_INSTALL/bin
export PATH=$PATH:$HADOOP_INSTALL/sbin
export HADOOP_MAPRED_HOME=$HADOOP_INSTALL
export HADOOP_COMMON_HOME=$HADOOP_INSTALL
export HADOOP_HDFS_HOME=$HADOOP_INSTALL
export YARN_HOME=$HADOOP_INSTALL
```

На этом шаге заканчиваются предварительные подготовки.

10 Настройка Apache Hadoop

Все последующая работа будет вестись из папки /usr/local/hadoop. Откроем etc/hadoop/hadoop-env.sh и зададим JAVA_HOME.

```
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64
```

Опишем, какие у нас будут узлы в кластере в файле etc/hadoop/slaves

```
master
```

Этот файл может располагаться только на главном узле. Все новые узлы необходимо описывать здесь.

Основные настройки системы располагаются в etc/hadoop/core-site.xml:

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://master:9000</value>
    </property>
</configuration>
```

Настройки HDFS лежат в etc/hadoop/hdfs-site.xml:

```
<configuration>
    <property>
```

```
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>file:/usr/local/hadoop/tmp/hdfs/namenode</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>file:/usr/local/hadoop/tmp/hdfs/datanode</value>
    </property>
</configuration>
```

Здесь параметр dfs.replication задает количество реплик, которые будут хранится на файловой системе. По умолчанию его значение равно 3. Оно не может быть больше, чем количество узлов в кластере.

Параметры dfs.namenode.name.dir и dfs.datanode.data.dir задают пути, где будут физически располагаться данные и информация в HDFS. Необходимо заранее создать папку tmp.

Сообщим нашему кластеру, что мы желаем использовать YARN. Для этого изменим etc/hadoop/mapred-site.xml:

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
    <property>
        <name>mapreduce.job.reduces</name>
        <value>1</value>
    </property>
    <property>
        <name>mapreduce.task.io.sort.mb</name>
        <value>16</value>
    </property>
    <property>
        <name>mapreduce.map.memory.mb</name>
        <value>256</value>
    </property>
    <property>
        <name>mapreduce.map.cpu.vcores</name>
        <value>1</value>
    </property>
    <property>
        <name>mapreduce.reduce.memory.mb</name>
        <value>256</value>
    </property>
    <property>
        <name>mapreduce.reduce.cpu.vcores</name>
        <value>1</value>
    </property>
    <property>
        <name>mapreduce.job.heap.memory-mb.ratio</name>
        <value>0.8</value>
    </property>
```

```
    <property>
        <name>mapreduce.map.java.opts</name>
        <value>-Djava.net.preferIPv4Stack=true -Xmx52428800</value>
    </property>
    <property>
        <name>mapreduce.reduce.java.opts</name>
        <value>-Djava.net.preferIPv4Stack=true -Xmx52428800</value>
    </property>
</configuration>
```

Все настройки по работе YARN описываются в файле etc/hadoop/yarn-site.xml:

```
<configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
    <property>
        <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
        <value>org.apache.hadoop.mapred.ShuffleHandler</value>
    </property>
    <property>
        <name>yarn.resourcemanager.scheduler.address</name>
        <value>master:8030</value>
    </property>
    <property>
        <name>yarn.resourcemanager.address</name>
        <value>master:8032</value>
    </property>
    <property>
        <name>yarn.resourcemanager.webapp.address</name>
        <value>master:8088</value>
    </property>
    <property>
        <name>yarn.resourcemanager.resource-tracker.address</name>
        <value>master:8031</value>
    </property>
    <property>
        <name>yarn.resourcemanager.admin.address</name>
        <value>master:8033</value>
    </property>
    <property>
        <name>yarn.nodemanager.resource.cpu-vcores</name>
        <value>1</value>
    </property>
    <property>
        <name>yarn.scheduler.minimum-allocation-vcores</name>
        <value>1</value>
    </property>
    <property>
        <name>yarn.scheduler.increment-allocation-vcores</name>
        <value>1</value>
    </property>
    <property>
        <name>yarn.scheduler.maximum-allocation-vcores</name>
        <value>2</value>
```

```
    </property>
    <property>
        <name>yarn.nodemanager.resource.memory-mb</name>
        <value>2060</value>
    </property>
    <property>
        <name>yarn.scheduler.minimum-allocation-mb</name>
        <value>1</value>
    </property>
    <property>
        <name>yarn.scheduler.increment-allocation-mb</name>
        <value>512</value>
    </property>
    <property>
        <name>yarn.scheduler.maximum-allocation-mb</name>
        <value>2316</value>
    </property>
    <property>
        <name>yarn.nodemanager.vmem-check-enabled</name>
        <value>false</value>
    </property>
</configuration>
```

Настройки resourcemanager нужны для того, чтобы все узлы кластера можно было видеть в панели управления.

Сменим пользователя на hduser:

```
su hduser
```

Отформатируем HDFS:

```
/usr/local/hadoop/bin/hdfs namenode –format
```

Запустим hadoop службы:

```
/usr/local/hadoop/sbin/start-dfs.sh
/usr/local/hadoop/sbin/start-yarn.sh
```

Необходимо убедиться, что запущены следующие java-процессы:

```
hduser@master:/usr/local/hadoop$ jps
4868 SecondaryNameNode
5243 NodeManager
5035 ResourceManager
4409 NameNode
4622 DataNode
5517 Jps
```

Теперь у нас есть готовый образ, который послужит основой для создания кластера.

Далее можно создать требуемое количество копий нашего образа.

На копиях необходимо настроить сеть, сгенерировать новые MAC-адреса для сетевых интерфейсов, выдать им необходимые IP-адреса и поправить файл /etc/hosts на всех узлах кластера так, чтобы в нем были прописаны все соответствия. Например:

```
127.0.0.1       localhost
```

```
192.168.0.1     master
192.168.0.2     slave1
```

Заменим имя нового узла на slave1, для этого внесем изменения в файл /etc/hostname.

Сгенерируем на узле новые SSH-ключи и добавим их все в список авторизованных на узле master.

На каждом узле кластера изменим значения параметра dfs.replication в /usr/local/hadoop/etc/hadoop/hdfs-site.xml. Например, выставим везде значение 2.

```
<configuration>
   <property>
      <name>dfs.replication</name>
      <value>2</value>
   </property>
</configuration>
```

Добавим на узле master новый узел в файл /usr/local/hadoop/etc/hadoop/slaves:

```
master
slave1
```

11 Создание на главном узле файл подкачки

Создадим на главном узле папку, в которую попозже мы подмонтируем файл подкачки:

```
sudo mkdir /media/swap
```

Создаем файл подкачки

```
sudo dd if=/dev/zero of=/media/swap/swapfile.img bs=2048 count=1M
```

Выставляем нужные права на файл:

```
sudo chmod 600 /media/swap/swapfile.img
```

Создаем swap

```
sudo mkswap /media/swap/swapfile.img
```

Добавляем swap в fstab. Это нужно сделать чтобы каждый раз при старте ОС, автоматически монтировался файл подкачки, который мы создали, для этого открываем файл /etc/fstab в редакторе:

```
sudo nano /etc/fstab
```

и добавляем в файл:

```
# mount swap image
/media/swap/swapfile.img swap swap sw 0 0
```

Активируем (включаем) наш swap

```
sudo swapon /media/swap/swapfile.img
```

Убедимся, что swap нормально работает. Для этого выполним:

```
cat /proc/swaps
```

12 Запуск Hadoop

Когда все настройки прописаны, то на главном узле можно запустить наш кластер.

```
/usr/local/hadoop/sbin/start-dfs.sh
/usr/local/hadoop/sbin/start-yarn.sh
```

На slave-узле должны запуститься следующие процессы:

```
hduser@slave1:/usr/local/hadoop$ jps
1748 Jps
1664 NodeManager
1448 DataNode
```

Теперь у нас есть свой мини-кластер. Посмотреть состояние нод кластера можно по адресу http://*master*:8088/cluster/nodes.

Давайте запустим задачу Word Count. Для этого нам потребуется загрузить в HDFS несколько текстовых файлов. Для примера, возьмём книги в формате txt с сайта Free ebooks — Project Gutenberg.

```
cd /home/hduser
mkdir books
cd books
wget http://www.gutenberg.org/files/20417/20417.txt
wget http://www.gutenberg.org/files/5000/5000-8.txt
wget http://www.gutenberg.org/files/4300/4300-0.txt
wget http://www.gutenberg.org/files/972/972.txt
```

Перенесем наши файлы в HDFS:

```
cd /usr/local/hadoop
bin/hdfs dfs -mkdir /in
bin/hdfs dfs -copyFromLocal /home/hduser/books/* /in
bin/hdfs dfs -ls /in
```

Запустим Word Count:

```
/usr/local/hadoop/bin/hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoo
p-mapreduce-examples-2.7.4.jar wordcount /in /out
```

Отслеживать работу можно через консоль, а можно через веб-интерфейс ResourceManager'a по адресу http://*master*:8088/cluster/apps/.

По завершению работы, результат будет располагаться в папке /out в HDFS.

Для того, чтобы скачать его на локальную файловую систему выполним:

```
/usr/local/hadoop/bin/hdfs dfs -copyToLocal /out /home/hduser/
```

Теперь в директории /home/hduser/out можно увидеть результаты выполнения задачи.

13 Дополнительные команды

Удаление директории из HDFS:

```
/usr/local/hadoop/bin/hdfs dfs –rm -r /out
```

Отмена режима HDFS только для чтения, возникшего из-за сбоя:

```
/usr/local/hadoop/bin/hdfs dfsadmin -safemode leave
```

Остановка Yarn:

```
/usr/local/hadoop/sbin/stop-yarn.sh
```

Остановка DFS:

```
/usr/local/hadoop/sbin/stop-dfs.sh
```

ЛАБОРАТОРНАЯ РАБОТА 4.
РАСПРЕДЕЛЁННАЯ ФАЙЛОВАЯ СИСТЕМА HDFS

Цель работы

Изучение основных операций для работы с распределенной файловой системой HDFS.

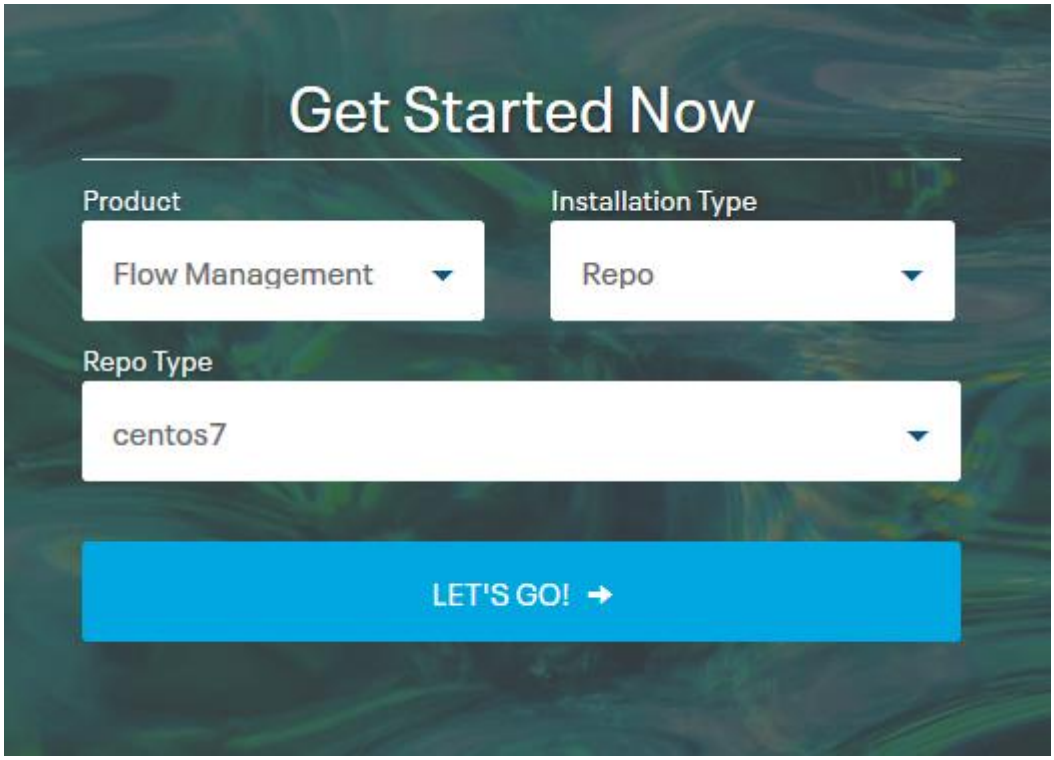
Задачи работы

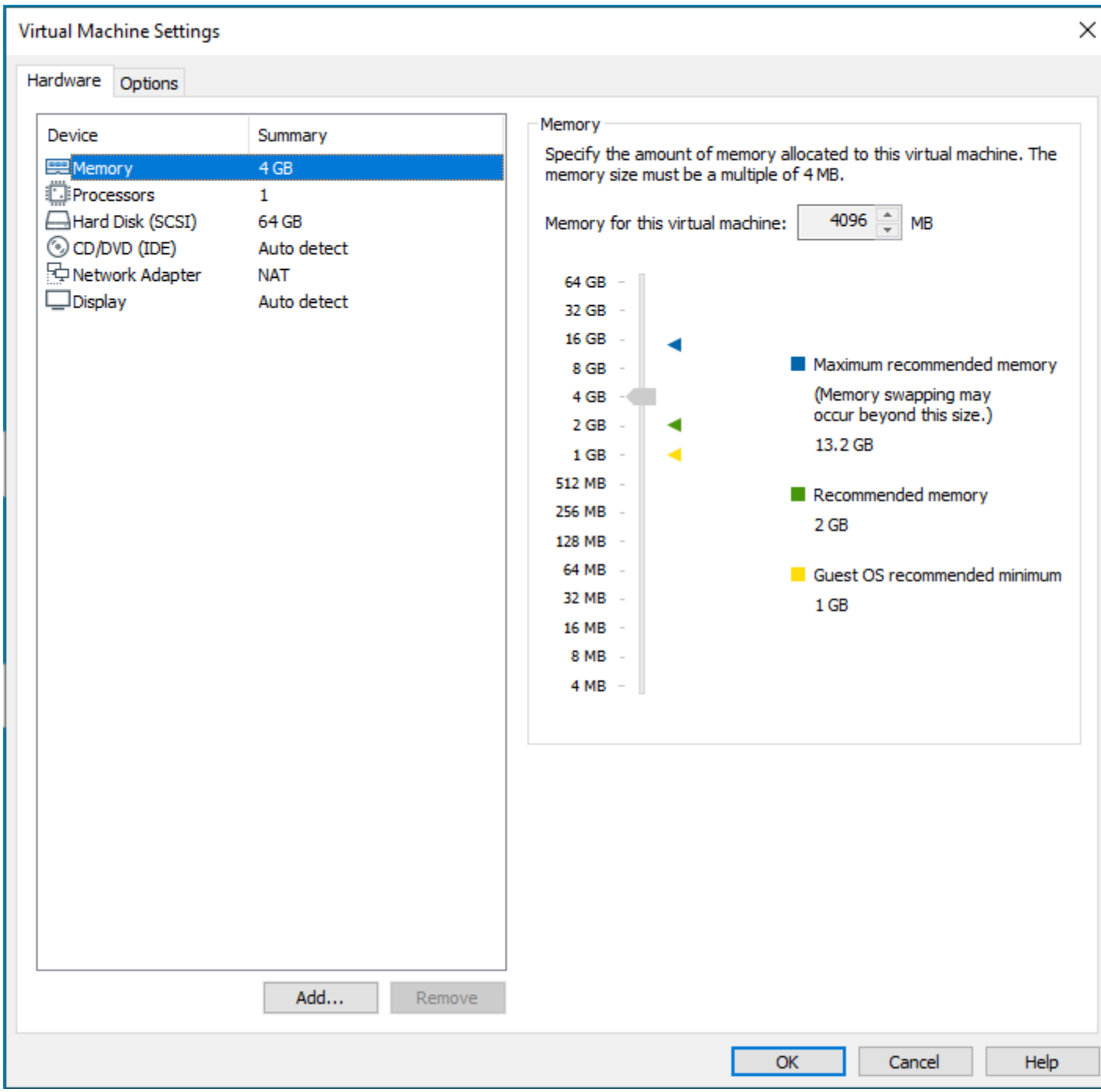
- подготовка окружения
- запуск shell-клиента
- изучение основных shell-команд

Подготовка рабочего окружения

Для выполнения задания необходимо подготовить виртуальное окружение:

- настроить окружения для запуска виртуальных машин VMware
- скачать и развернуть виртуальную машину от компании Cloudera по ссылке https://www.cloudera.com/downloads/
- ИЛИ сразу скачать готовый образ для VMWare





Задание на лабораторную работу

Запуск shell-клиента HDFS в терминале

Нативный shell-клиент для HDFS позволяет работать с файловой системой через командную строку. Для использования shell-клиента через командную строку, необходимо в терминале запустить утилиту **hdfs**:

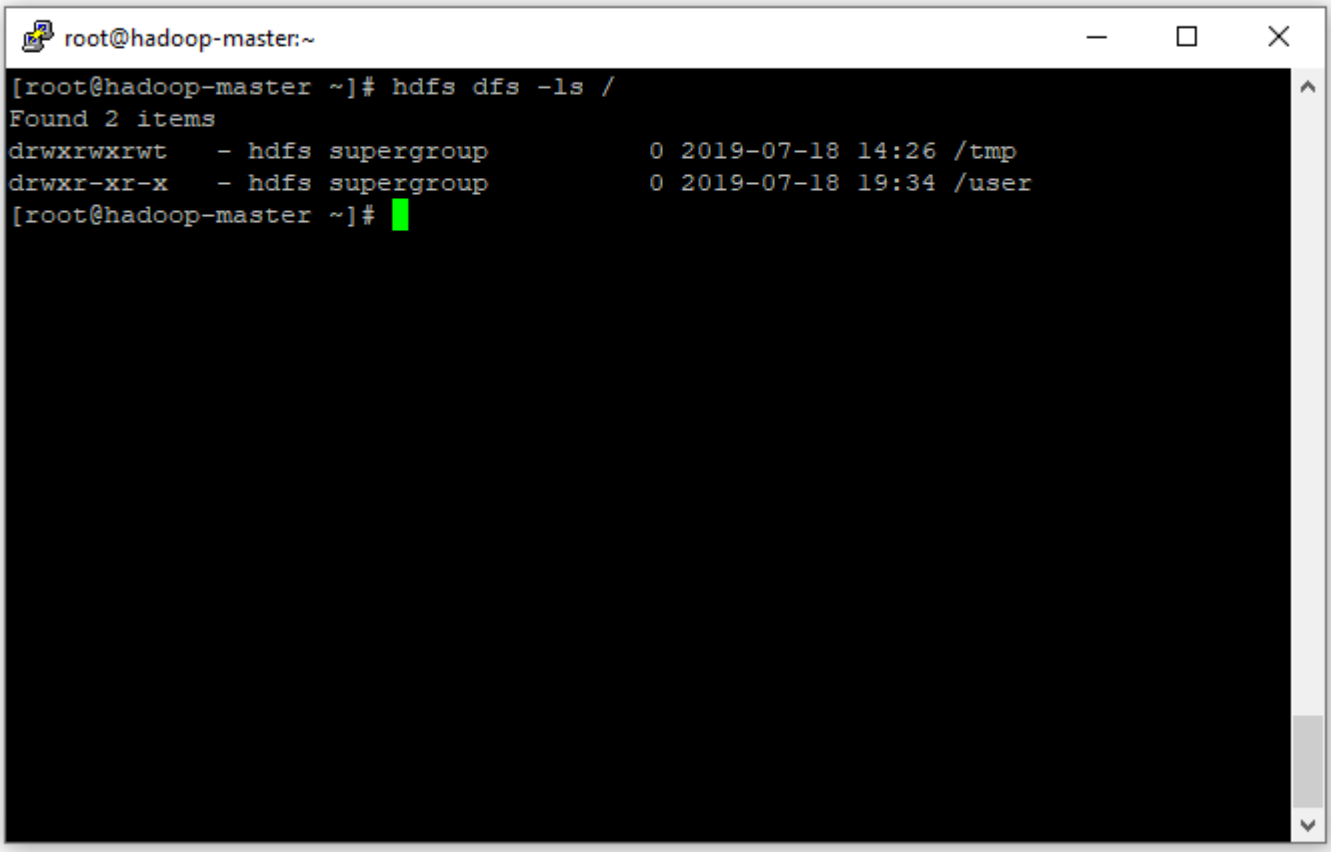
hdfs dfs -<command> -<option> <URI>
где

- **hdfs** -- утилита работы с HDFS;

- **dfs** -- специальный параметр, который обозначает, что мы будем работать с непосредственно с распределенной файловой системой. Могут быть указаны другие параметры;
- **<command>** -- команда, которую мы хотим применить к файловой системе;
- **<command>** -- опции команды (могут отсутствовать);
- **<URI>** -- путь в виде URI-схемы. **URI** (Uniform Resource Identifier) — унифицированный (единообразный) идентификатор ресурса. символьная строка, позволяющая идентифицировать какой-либо ресурс: документ, изображение, файл, службу, ящик электронной почты и т. д. Прежде всего, речь идёт о ресурсах сети Интернет и Всемирной паутины. URI предоставляет простой и расширяемый способ идентификации ресурсов. Расширяемость URI означает, что уже существуют несколько схем идентификации внутри URI, и ещё больше будет создано в будущем.

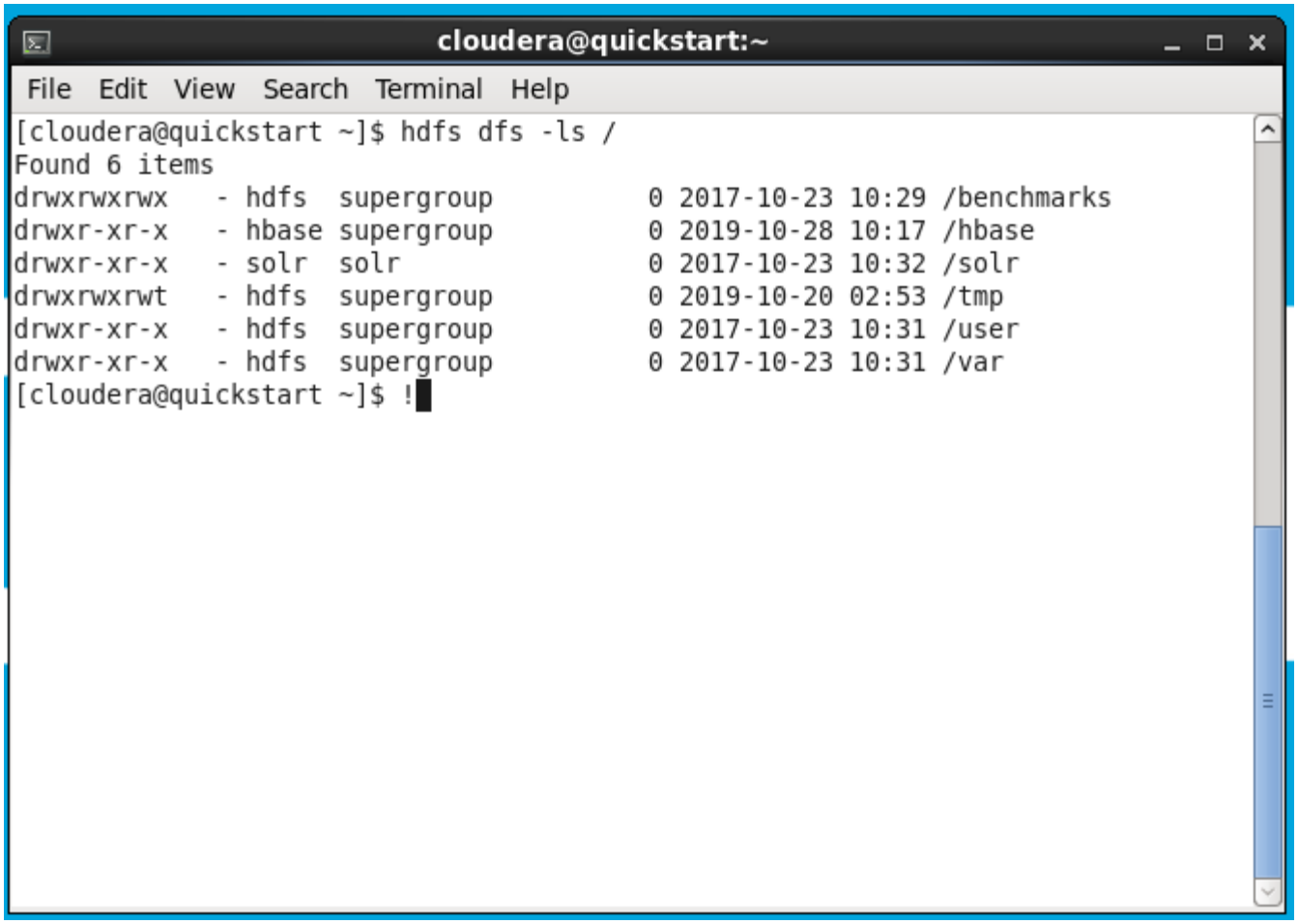
Пример команды для просмотра содержимого (листинг) корневой директории:

hdfs dfs -ls /
Запуск
терминала



Запуск на виртуальной

машине



Задание пути в виде URI-схемы

hdfs://localhost:8020/user/home

Название части URI	Фрагмент	Описание
scheme	hdfs://	схема
authority	localhost:8020	часть авторизации -- название сервера, порт, на котором находится Namenode, может быть указано имя пользователя
HDFS path	/user/home	путь в распределенной или локальной файловой системе

- Local Указываем file:// -- схема, а путь -- /to/path/dir. Аналогична команде ls /to/path/dir локальной файловой системы

hdfs dfs -ls file:///to/path/dir

- HDFS Просмотр файлов в распределенной файловой системе HDFS

hdfs dfs -ls hdfs://localhost/to/path/dir
где hdfs:// -- схема, a localhost/to/path/dir -- расположение namenode, а далее путь к файлам в HDFS. На сервере:
hdfs dfs -ls hdfs://hadoop-master.vtizi.ru/
В виртуальной машине:

hdfs dfs -ls hdfs://localhost:8020/

- fs.default.name=hdfs://localhost Чтобы постоянно не указывать схему и сервер при работе с одной и той же HDFS применяется упрощенный способ доступа к файлам HDFS без схемы и части авторизации. В конфигурации указан специальный параметр, который автоматически будет подставляться в схему

hdfs dfs -ls /to/path/dir

Команды Shell

В HDFS клиенте реализованы стандартные команды для работы с файловой системой. Называются похоже на команды bash(локальной файловой системы). Команды:

- похожие на команды Linux: cat, rm, ls, du, ...;
- специфичные для HDFS операции: setrep -- смена фактора репликации;

1. Команда help
получить список всех команд в HDFS-клиенте

hdfs dfs -help
показать информацию по команде:

hdfs dfs -help <commande_name>

2. Команда ls
ls -- самая популярная команда, выводит листинг директории и статистику файлов. Если запустим команду для определенной директории, то увидим ее содержимое:

- сколько элементов в данной директории items
- файлы, которые содержатся в директории
- статистика -- указание прав доступа и типа файла (поддиректории), фактор репликации (у директории -- нет), принадлежность файла (пользователь и группа), время создания. Параметры команды:
- -r -- вывести статистику по поддиректориям Пример команды:

hdfs dfs -ls -r /

3. Команда mkdir
Создает директорию

hdfs dfs -mkdir /data/new_path

4. Команда чтения файлов cat
Выводит источник в стандартный поток вывода -- stdout.

- вывести весь файл hdfs dfs -cat /dir/file.txt -- для больших файлов очень долго и не понятно для чего
- перенаправление потока вывода через каналы **pipe** в утилиты *less, head, tail*
- получить первые 100 строк из файла hdfs dfs -cat /dir/file.txt | head -n 100

5. Команда чтения файлов text
Аналог команды cat, но разархивирует архивы:
hdfs dfs -cat /dir/file.gz -- непонятный текст
hdfs dfs -text /dir/file.gz -- понятный текст
Если файл не заархивирован, то работа команд text и cat будет одинакова.

6. Команда чтения файлов tail
Выводит последние строчки файла:

hdfs dfs -cat /dir/file.txt | tail -- плохо
hdfs dfs -tail /dir/file.txt -- хорошо
В первом случае читается весь файл и только потом от него отрезаются последние строчки для вывода. Для большого файла может потребоваться на чтение много времени -- прочитать все блоки файла со всех серверов. Только с последнего блока вывести последние строки. Во втором случае нет лишних действий, будут выведены последние строчки из последнего блока.

Полезна для просмотра, например, логов.

7. Копирование данных в shell
cp -- копирование файла из одного места в другое. Годится только для небольших файлов!
hdfs dfs -cp /dir/file1 /otherDir/file2
Копирует файл из одной директории в другую. Т.к. файл состоит из блоков, то каждый блок копируется на новое место. Если файл большой и состоит из многих блоков, то копирование такого файла будет выполняться долго. Разумно копировать небольшие файлы, иначе для больших файлов потребуется много времени.

distcp -- копирование больших файлов или множества файлов.

hdfs distcp /dir/file1 /otherDir/file2
это команда не часть команды dfs. При использовании команды distcp блоки файла копируются параллельно и независимо друг от друга. Файл копируется быстро. Для больших файлов. mv -- перемещение файла из одного места в другое
hdfs dfs -mv/dir/file1 /otherDir
Физически перемещение данных не происходит. Указание Namenode, что файл теперь будет храниться в другой директории.

put, copyFromLocal -- копирование локального файла в директорию HDFS
hdfs dfs -put localfile /dir/file
get, copyToLocal -- копирование файла из HDFS в локальную файловую систему
hdfs dfs -get /dir/file localfile

8. Команда удаления в shell
rm -- удаляет файл в корзину. При случайном удалении большого файла его очень трудно восстановить. Поэтому файл сначала помещается в корзину. Корзина -- специальная для каждого пользователя директория. Файлы из нее можно удалить или скопировать на прежнее место. Автоматически очистка корзины задана в параметрах HDFS. Если нужно освободить место в HDFS, то можно удалить файл из корзины.
hdfs dfs -rm /dir/file -- удалить файл в корзину
hdfs dfs -rm -skipTrash /dir/file -- удалить файл без помещения в корзину
hdfs dfs -rm -r /dir -- удалить рекурсивно всю директорию (поддиректории)

9. Команда выведения статистики в shell
du -- размер файла или директории в байтах du -h -- размер файла или директории в удобно-читаемом формате
hdfs dfs -du -h /dir
В HDFS файлы большие, поэтому удобнее использовать ключ -h.

Права доступа в HDFS

- ограничения на уровне файла/директории -- сходство с моделью прав в POSIX -- Read (r), Write (w), Execute (x) -- Разделяется на пользователя, группу и всех остальных
- Права пользователя определяются исходя из прав той ОС, где он запускает клиентское приложение.

У файлов есть владелец и группа, которой он принадлежит.

Задание на лабораторную работу

1. Развернуть виртупальное окружение

2. Вывести с помощью команды help описание основных команды shell-клиента
3. Просмотреть корневую директорию HDFS пользователя vtizi: /user/vtizi или на виртуальной машине cloudera:/home/cloudera
4. Создать в HDFS в директории /user/vtizi поддиректории fio или на виртуальной машине /home/cloudera
5. Создать в локальной файловой системе случайный текстовый файл размером 10 Mb с именем, образованным вашими инициалами

base64 /dev/urandom | head -c 10000000 > file.txt

6. Заархивировать созданный текстовый файл

gzip -c file.txt > file.gz

7. Скопировать текстовый файл и архив в директорию /user/vtizi/fio HDFS или в директорию /home/cloudera/fio виртуальной машины
8. Просмотреть файл и архив с помощью утилит cat, text в комбинации с каналами и утилитами head, tail -- привести не менее 3 вариантов команд и просмотра файла.
9. Создать копию файла file.txt вида date_file.txt, где в начале имени файла-копии указана текущая дата. Вывести листинг
10. Вывести статистику по директории /user/vtizi/fio или /home/cloudera/fio виртуальной машины
11. Удалить поддиректорию /fio со всем содержимым

ЛАБОРАТОРНАЯ РАБОТА № 5
МОДЕЛЬ ВЫЧИСЛЕНИЙ MapReduce

Целью лабораторной работы является знакомство с особенностями реализации вычислительной парадигмы MapReduce с использованием как средств Hadoop, так и языка Python.

Платформа MapReduce

В простейшем приближении приложение платформы MapReduce состоит как минимум из трех частей: функция map, функция reduce и главная функция, которая объединяет управление заданиями и файловым вводом/выводом. В этом отношении Hadoop содержит большое количество интерфейсов и абстрактных классов, предоставляя в распоряжение разработчиков приложений Hadoop большой набор инструментов, начиная с отладчиков, и заканчивая средствами измерения производительности.

MapReduce представляет собой программную платформу (framework) для параллельной обработки больших объемов данных. MapReduce содержит элементы функционального программирования, формально унаследованные от функций `map` и `reduce`, характерных для языков функционального программирования. Процесс MapReduce состоит из двух операций, которые могут, в свою очередь, состоять из множества экземпляров (множество операций map, множество операций reduce). Функция Map получает на вход набор данных и преобразует его в список пар ключ/значение – одна пара для каждого элемента входящего набора. Функция Reduce получает на вход данный список и "сворачивает" пары ключ/значение, группируя их по ключам (в результате получается одна пара ключ/значение для каждого ключа).

Ниже приведен пример, помогающий понять, что это означает. Предположим, что у вас имеется следующий входящий набор данных: `one small step for man, one giant leap for mankind`. В результате выполнения над ним функции Map мы получим следующий список пар ключ/значение:

```
1 (one, 1)  (small, 1)  (step, 1)  (for, 1)  (man, 1)
2 (one, 1)  (giant, 1)  (leap, 1)  (for, 1)  (mankind, 1)
```

Теперь выполним над этим списком функцию Reduce, в результате чего получим следующий набор пар ключ/значение:

```
1 (one, 2)    (small, 1)  (step, 1)     (for, 2)  (man, 1)
2 (giant, 1)  (leap, 1)   (mankind, 1)
```

В результате мы получили счетчик слов во входящем наборе, который очевидно является полезным для процесса индексации. Представьте теперь, что ваш входящий набор на самом деле представляет собой два входящих набора данных: `one small step for man` (первый набор) и `one giant leap for mankind` (второй набор). Вы можете применить функции Map и Reduce к каждому из этих наборов, а затем еще раз выполнить для двух полученных списков ключ/значение функцию Reduce, получив тот же самый результат, что и в вышеприведенном примере. Другими словами, вы можете распараллелить операции над входящим набором и получить тот же результат, но только намного быстрее. В этом и заключается мощь процесса MapReduce – в его основе лежит возможность распараллеливания на любое количество систем. На рисунке 1 эта идея изображена в терминах сегментации и итерирования. Как эта функциональность реализована в Hadoop? Приложение MapReduce запускается по требованию клиента на единственном управляющем узле, который называется JobTracker.

Как и узел NameNode, это единственный узел данного типа в кластере Hadoop, а его задачей является управление приложениями MapReduce. Когда приложение запущено, ему предоставляются входные и выходные директории, содержащиеся в файловой системе HDFS. Узел JobTracker использует информацию о файловых блоках (количество блоков и их месторасположение), чтобы решить, сколько подчиненных задач необходимо создать на узлах типа TaskTracker.

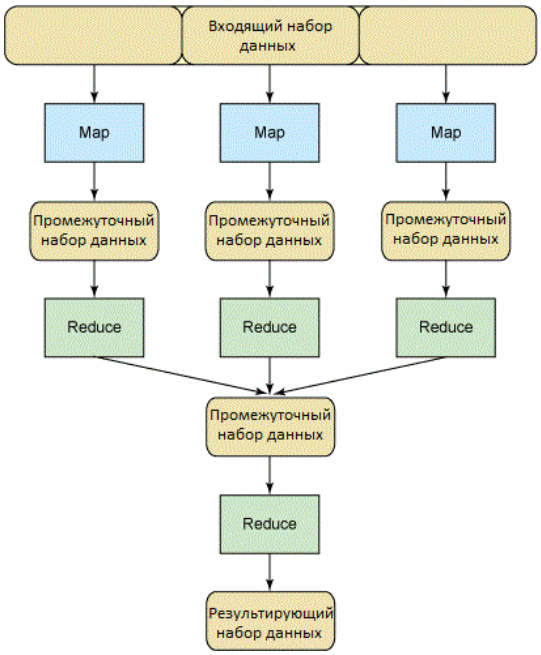


Рис. 1. Концептуальная схема процесса MapReduce

Приложение MapReduce копируется на каждый узел, содержащий входные файловые блоки. Для каждого файлового блока заданного узла создается отдельная подчиненная задача. Каждый узел TaskTracker докладывает о статусе работы и о завершении задачи узлу JobTracker. На рис. 2 показан пример распределенной работы в кластере.

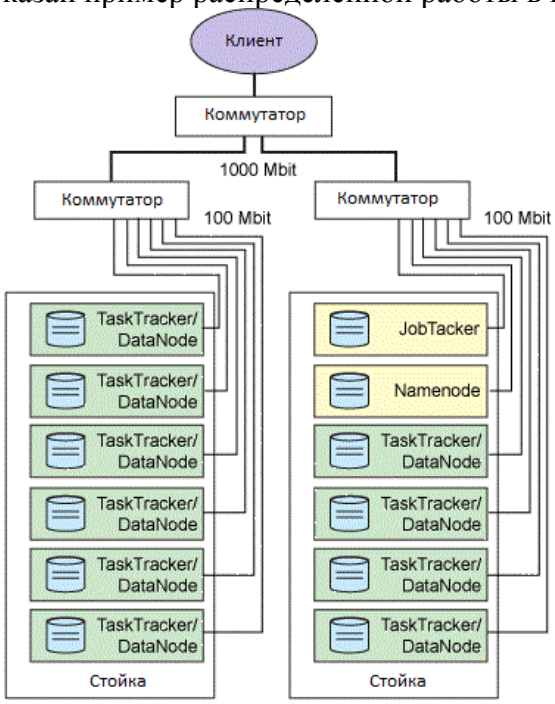


Рисунок 2. Пример кластера Hadoop, показывающий физическое распределение задач обработки и хранения

Это является важным аспектом работы Hadoop, поскольку вместо переноса устройств хранения в зону обработки данных, Hadoop переносит задачи обработки в зону хранения данных. Такое масштабирование задач обработки на количество узлов в кластере способствует эффективной обработке данных.

Перечень необходимого оборудования для выполнения лабораторной работы

Компьютер (например, Intel(R) Core(TM) 15-2320 CPU @ 3.00GHz 3.00 GHz) со следующим программным обеспечением: Программный продукт виртуализации Oracle VM

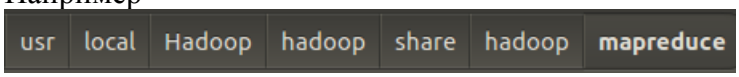
VirtualBox, операционная система Ubuntu Linux, свободно распространяемый набор утилит, библиотек и фреймворк для разработки и выполнения распределённых программ Hadoop.

Содержание работы

Часть 1. Запуск примеров MapReduce.

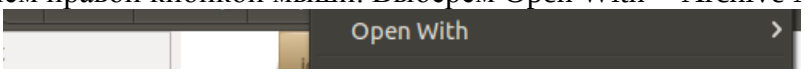
Для запуска примера (ов) перейдём в Наутилусе (или в терминале) по адресу, где лежит Hadoop (разархивированный).

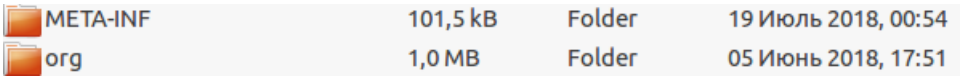
Например



По этому адресу в Наутилусе Вы найдёте файл . Это и есть файл примеров.

Нажмём на нём правой кнопкой мыши. Выберем Open With ->Archive Manager.





Посмотрим содержимое папки **org(/apache/hadoop/examples).** Как видим здесь имеется и пример подсчёта слов (WordCount) **hadoop-mapreduce-examples-2.7.7.jar** (см.выше).

Если мы напишем в терминале строчку

```
kos@kos:/usr/local/Hadoop/hadoop/share/hadoop/mapreduce$ hadoop jar hadoop-mapre
duce-examples-2.7.7.jar
```

То мы увидим список примеров mapreduce например, значения pi, количества слов в заданном списке файлов и т.д.

```
kos@kos:/usr/local/Hadoop/hadoop/share/hadoop/mapreduce$ hadoop jar hadoop-mapre
duce-examples-2.7.7.jar
An example program must be given as the first argument.
Valid program names are:
  aggregatewordcount: An Aggregate based map/reduce program that counts the word
s in the input files.
  aggregatewordhist: An Aggregate based map/reduce program that computes the his
togram of the words in the input files.
  bbp: A map/reduce program that uses Bailey-Borwein-Plouffe to compute exact di
gits of Pi.
  dbcount: An example job that count the pageview counts from a database.
  distbbp: A map/reduce program that uses a BBP-type formula to compute exact bi
ts of Pi.
  grep: A map/reduce program that counts the matches of a regex in the input.
  join: A job that effects a join over sorted, equally partitioned datasets
  multifilewc: A job that counts words from several files.
  pentomino: A map/reduce tile laying program to find solutions to pentomino pro
blems.
  pi: A map/reduce program that estimates Pi using a quasi-Monte Carlo method.
  randomtextwriter: A map/reduce program that writes 10GB of random textual data
 per node.
```

Давайте создадим входной каталог, в который мы добавим несколько файлов, и наше требование – подсчитать общее количество слов в этих файлах. Чтобы вычислить общее количество слов, нам не нужно писать наш MapReduce, при условии, что файл .jar содержит реализацию для подсчета слов. Вы можете попробовать другие примеры, используя тот же файл .jar; просто выполните следующие команды, чтобы проверить поддерживаемые функциональные программы MapReduce с помощью файла hadoop-mapreduce-examples-2.7.7.jar.

Наберем в терминале

```
kos@kos:/usr/local/Hadoop/hadoop/share/hadoop/mapreduce$ hadoop jar hadoop-mapre
duce-examples-2.7.7.jar wordcount
```

Получим

```
Usage: wordcount <in> [<in>...] <out>
kos@kos:/usr/local/Hadoop/hadoop/share/hadoop/mapreduce$
```

Это предупреждение о том, что необходимо создать входной и выходной каталоги

Шаг1

Создайте временные файлы содержимого во входном каталоге. Вы можете создать этот входной каталог в любом месте, где бы вы хотели работать.

$ mkdir ~/input (или так)

```
sudo mkdir input
```

Далее переместим текстовые файлы, которые находятся в текущем каталоге в каталог input

```
kos@kos:/usr/local/Hadoop/hadoop/share/hadoop/mapreduce$ sudo cp $HADOOP_HOME/*.
txt input
```

Посмотрите содержимое каталога input

```
kos@kos:/usr/local/Hadoop/hadoop/share/hadoop/mapreduce$ ls -l input
total 108
-rw-r--r-- 1 root root 86424 ноя 29 20:17 LICENSE.txt
-rw-r--r-- 1 root root 14978 ноя 29 20:17 NOTICE.txt
-rw-r--r-- 1 root root  1366 ноя 29 20:17 README.txt
kos@kos:/usr/local/Hadoop/hadoop/share/hadoop/mapreduce$
```

Эти файлы были скопированы из домашнего каталога установки Hadoop. Для вашего эксперимента вы можете иметь разные и большие наборы файлов.

Шаг 2

Давайте запустим процесс Hadoop, чтобы подсчитать общее количество слов во всех файлах, доступных во входном каталоге, следующим образом:

```
kos@kos:/usr/local/Hadoop/hadoop/share/hadoop/mapreduce$ hadoop jar $HADOOP_HOME
/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.7.jar wordcount input outp
ut
```

Результат будет примерно такой

```
19/11/29 20:25:01 INFO Configuration.deprecation: session.id is deprecated. Inst
ead, use dfs.metrics.session-id
19/11/29 20:25:01 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName
=JobTracker, sessionId=
19/11/29 20:25:01 INFO input.FileInputFormat: Total input paths to process : 3
19/11/29 20:25:01 INFO mapreduce.JobSubmitter: number of splits:3
19/11/29 20:25:02 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_lo
cal714831642_0001
19/11/29 20:25:02 INFO mapreduce.Job: The url to track the job: http://localhost
:8080/
19/11/29 20:25:02 INFO mapred.LocalJobRunner: OutputCommitter set in config null
19/11/29 20:25:02 INFO mapreduce.Job: Running job: job_local714831642_0001
19/11/29 20:25:03 INFO output.FileOutputCommitter: File Output Committer Algorit
hm version is 1
19/11/29 20:25:03 INFO mapred.LocalJobRunner: OutputCommitter is org.apache.hado
op.mapreduce.lib.output.FileOutputCommitter
19/11/29 20:25:03 ERROR output.FileOutputCommitter: Mkdirs failed to create file
```

Шаг 3

Шаг 2 выполнит необходимую обработку и сохранит вывод в файле output / part-r00000, который можно проверить с помощью –

$cat output/*

Он перечислит все слова вместе с их общим количеством, доступным во всех файлах, доступных во входном каталоге.

"AS 4
"Contribution" 1
"Contributor" 1
"Derivative 1
"Legal 1

"License" 1
"License"); 1
"Licensor" 1
"NOTICE” 1
…..

Часть 2.Python & Hadoop. В примере ниже hadoop установлен в папку: /usr/local/hadoop. Рассмотрим простой пример использования Python в Hadoop. Для начала создадим несколько файлов - например, в директории: `/home/hduser/python/wordcount/`

Создать файл **gedit имя_файла.py**

При этом необходимо сделать полный доступ к файлам в наутилусе, для того, чтобы была возможность их сохранения!!! sudo apt-get install –y nautilus-admin 1) mapper.py

```python
#!/usr/bin/env python

import sys

# читаем из стандартного входа
for line in sys.stdin: # для каждой посткпающейе строки
    # удаляем пробелы в начале и конце строки
    line = line.strip()
    # разбиваем строчку на слова
    words = line.split()
    # наращиваем счётчики
    for word in words:
        #  пишем результат в стандартный поток вывода
        # - выход мэппера будем входом
        #   редуктора reducer.py
        #
        # разделям табом слово и назначаем ему число вхождений 1
        print '%s\t%s' % (word, 1)
```

2) reducer.py

```python
#!/usr/bin/env python

from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

# input comes from STDIN
for line in sys.stdin:
    # удаляем проблемы в начале и конце строчки
    line = line.strip()

    # разбиваем каждую по символу таба
    # чтобы получить ключ и значение (число вхождений)
    word, count = line.split('\t', 1)

    # пытаемся перевести строку в число (число вхождений)
    try:
        count = int(count)
    except ValueError:
        # если перевести не получилось
        # то просто игнорируем эту строку и
        continue # продолжаем выполнение

# Следующий блок отработает только потому,
# что хадуп  сначала отсортирует значения по ключу
#  а только потом пошлёт их нашему редуктору
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # записывает результат в стандартный поток вывода
            # опять же разделяя значения табом.
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

# не забудем напечатать и последнее слово (если оно есть)
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

Сделаем **оба эти файла исполняемыми**

Пусть у нас есть файл с содержимым типа (запустите Наутилус с правами суперпользователя):

```
#!/bin/bash
sudo nautilus /var/www/
```

1. Запустите текстовый редактор.

2. Последовательно запишите команды, располагая каждую команду на отдельной строке.

3. Сохраните этот файл, сделайте его исполняемым, применив команду:

```
chmod +x имя_файла.
```

Например:

```
chmod +x runme
```

1

Как запустить в командной строке

Для запуска в командной строке просто перейдите в папку с файлом и наберите команду вида:

```
./runme
```

И протестируем их:

```
echo "foo foo quux labs foo bar quux" | /home/hduser/python/wordcount/mapper.py
```

мы должны получить ответ в виде:

```
foo 1
foo 1
quux    1
labs    1
foo 1
bar 1
quux    1
```

Теперь выполним такую команду (одна строка):

```
echo "foo foo quux labs foo bar quux" | /home/hduser/python/wordcount/mapper.py | sort -k1,1 | /home/hduser/python/wordcount/reducer.py
```

В примере выше мы перенаправляем потоки ввода-вывода - и после выхода мэппера сортируем строки, а затем уже отправляем их редуктору.

Получим ответ:

```
bar 1
foo 3
labs    1
quux    2
```

Лабораторное задание

Выполнить рассмотренные выше примеры

ЛАБОРАТОРНАЯ РАБОТА № 6.
РАСПРЕДЕЛЁННЫЕ ВЫЧИСЛЕНИЯ НА ПЛАТФОРМЕ Apache Spark

Цель работы

Цель лабораторной работы заключается в изучении платформы Apache Spark и получении практических навыков навыков настройки и использования Apache Spark .

Архитектура распределенного приложения Spark

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

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 . 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).

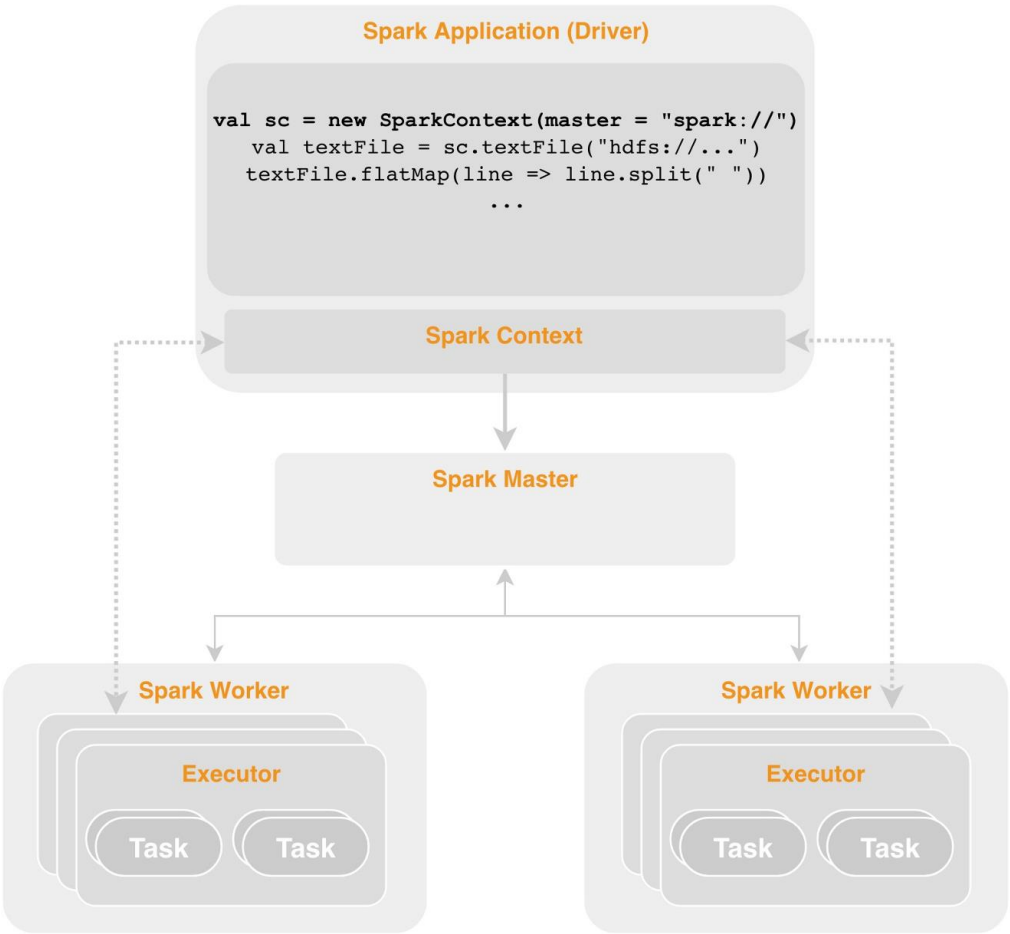


Рисунок 3 – Общая архитектура распределенного приложения Spark

Spark, являясь фреймворком для разработки распределенных приложений обработки данных, предполагает фиксированную master-slave [5] архитектуру таких приложений, проиллюстрированную на Рис. 3. 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. 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, 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 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.

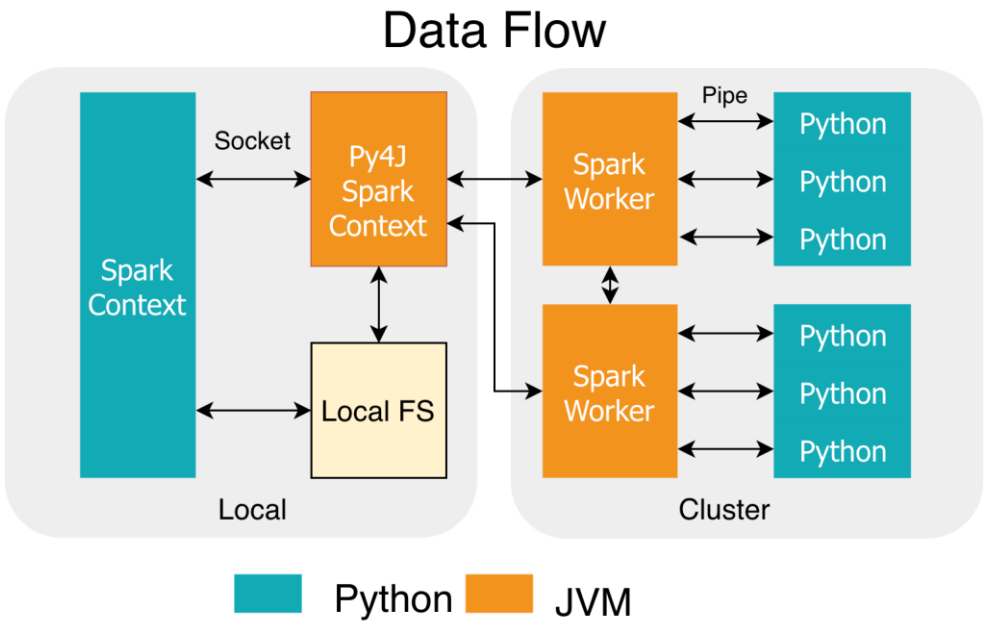


Рисунок 4 – Общая архитектура PySpark.

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.

Основные концепции Spark

· 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).

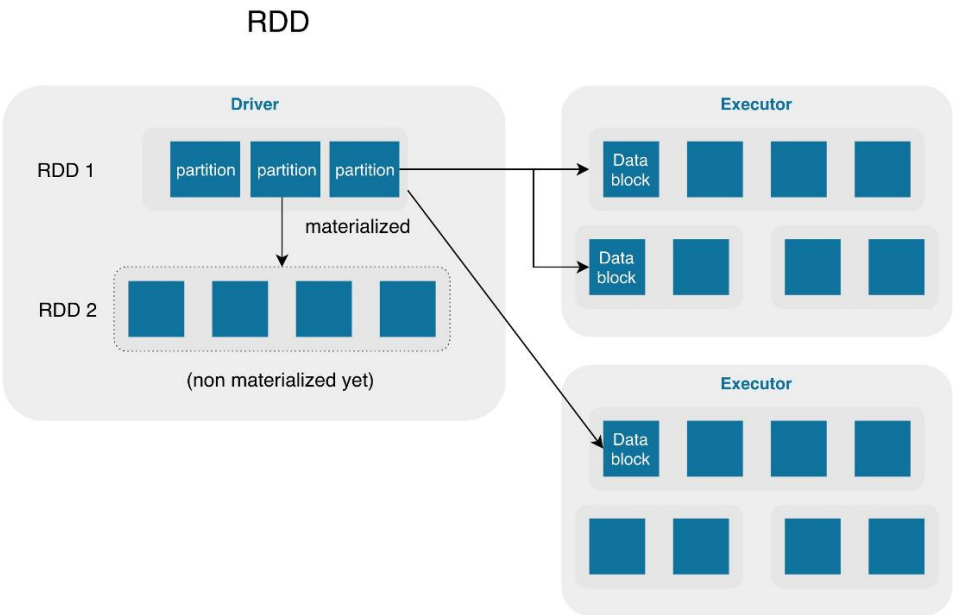


Рисунок 5 – RDD в Spark.

На рис. 5 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 далее);

– 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. 6 приведен пример такого графа вычислений.

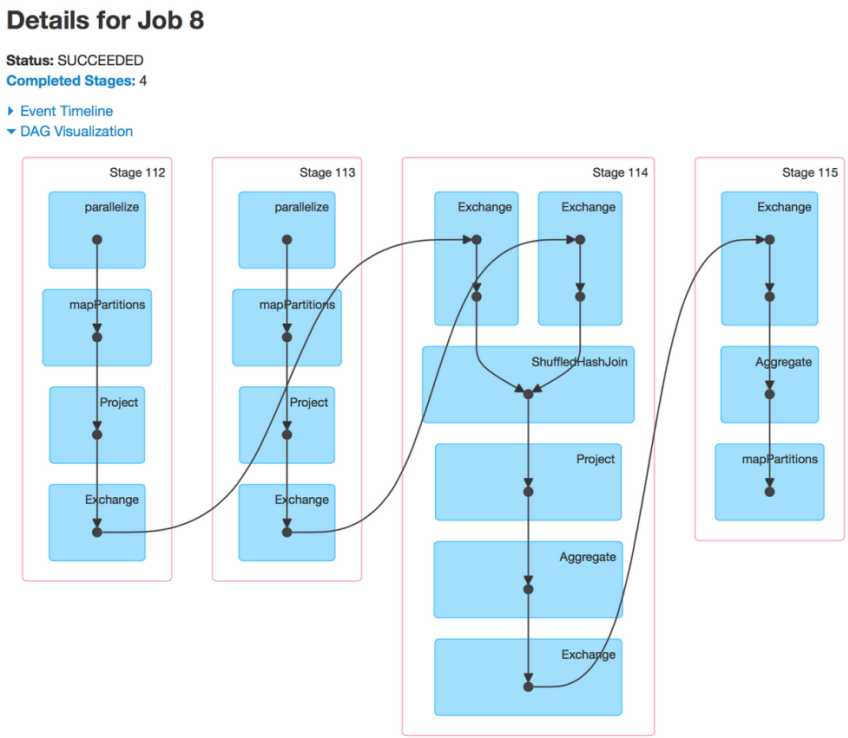


Рисунок 6 – Граф обработки данных, составленный из RDD (показаны синим цветом).

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 . Ниже, в листинге 1, приведен пример построения простого графа обработки в виде линейной цепочки действий, представляющих собой классическое приложение [i]WordCount.

Листинг 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 – 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. В листинге 2 приведен пример материализации приложения [ii]WordCount.

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.

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 . 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.

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, 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).

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. 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).

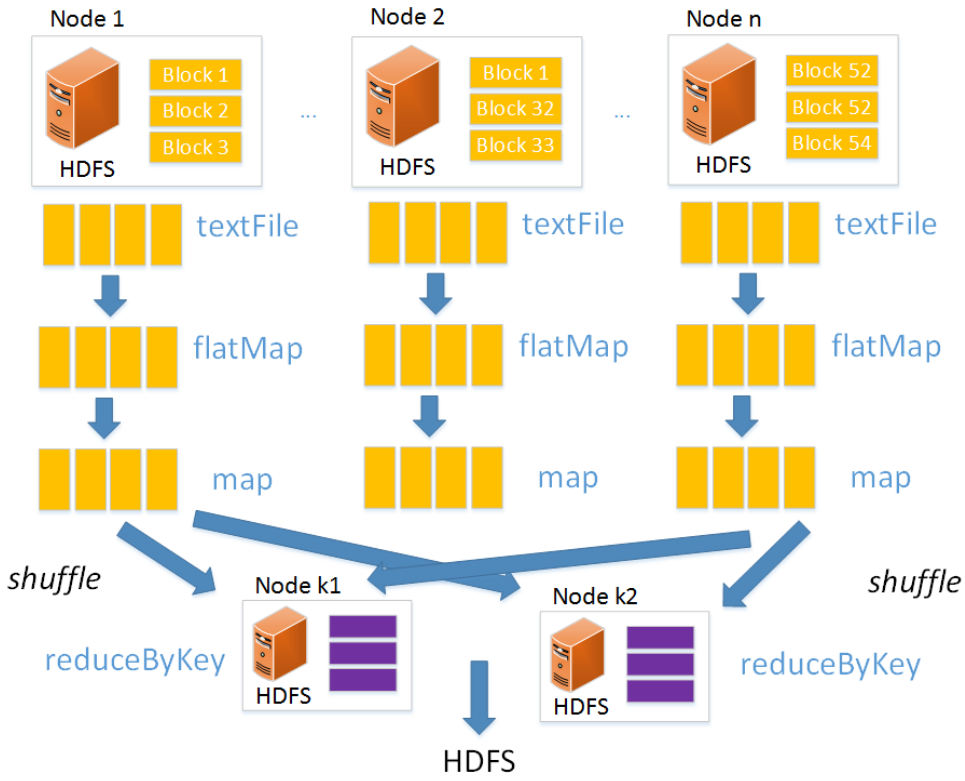
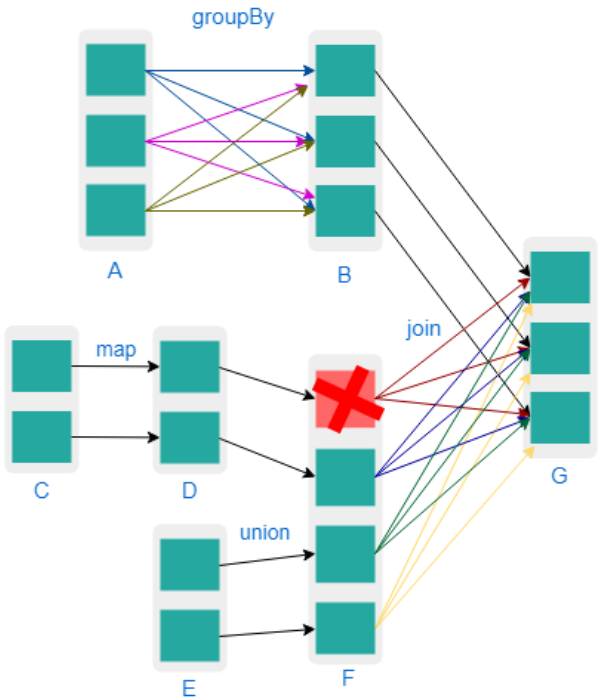


Рисунок 7 – Shuffleоперация в результате трансформации reduceByKey в приложении WordCount

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. 7 показана такая ситуация.

а)



б)

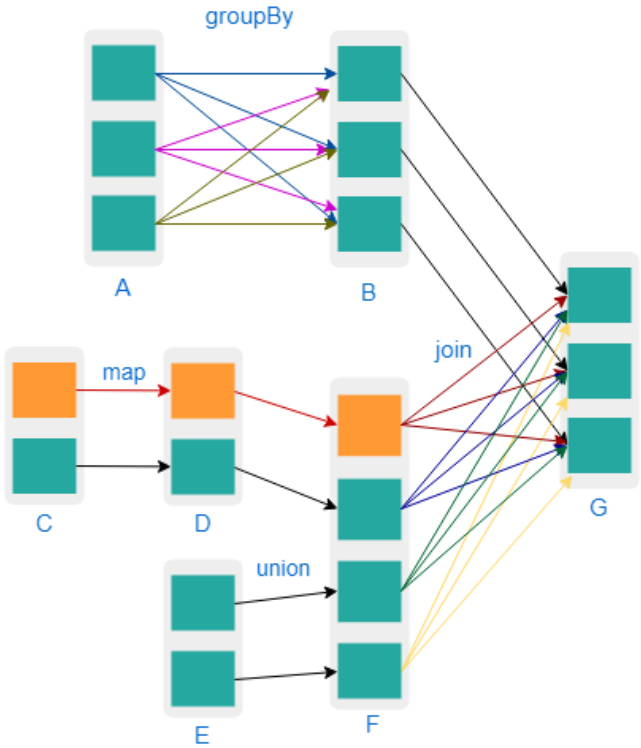


Рисунок 8 – Восстановление утраченных партиций по narrow зависимостям за счет частичного пересчета: (а) – потеря партиции в RDD F; (б) – восстановление партиции в RDD F за счет пересчета родителей данной партиции.

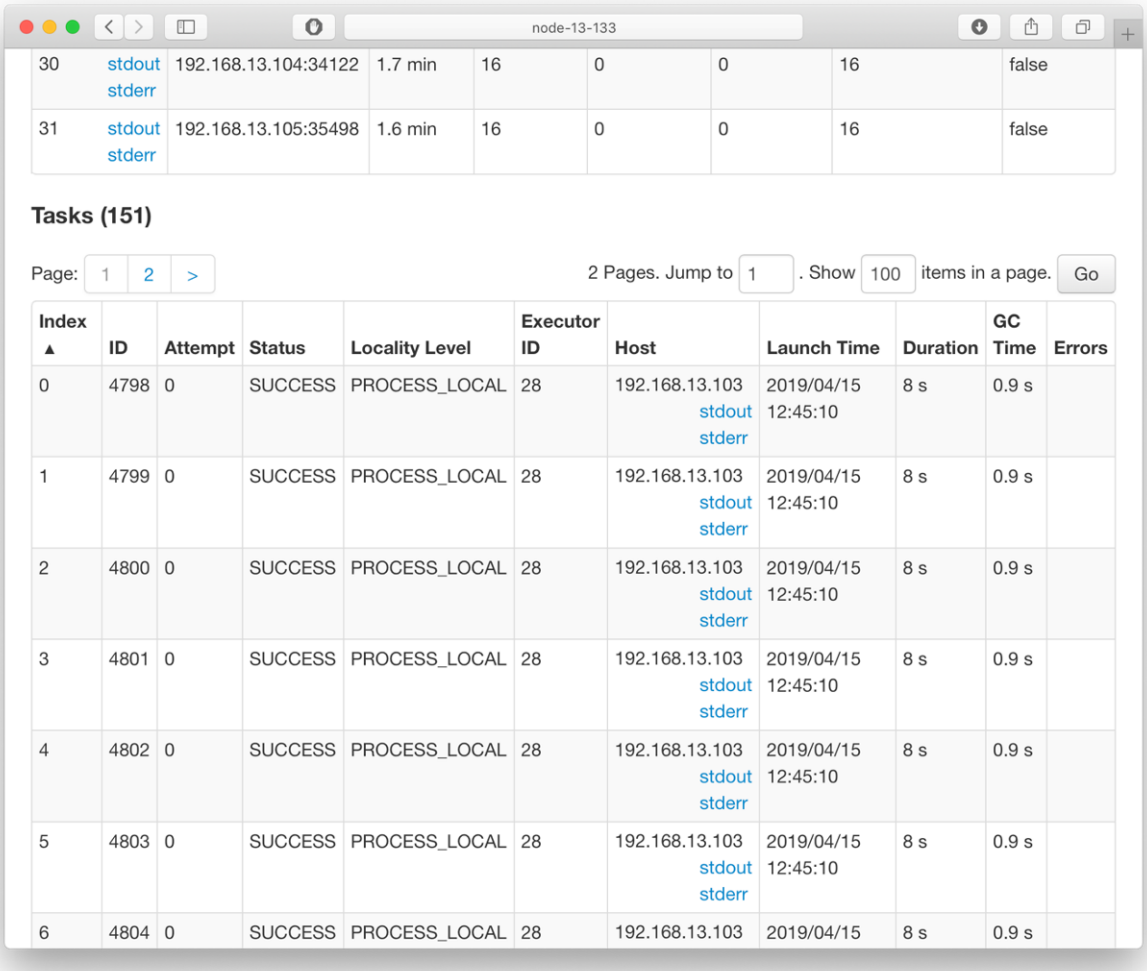


Рисунок 9 – Задачи и их статусы в пользовательском интерфейсе приложения Spark

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), 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. 9 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.

Ветвление и итеративные вычисления

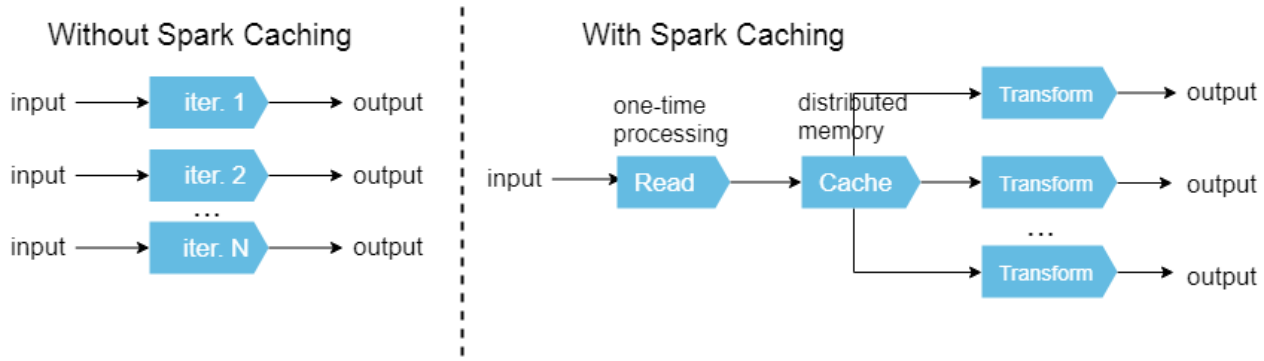


Рисунок 10 – Иллюстрация ветвления вычислений с помощью кэширования.

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. 10. .

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 11 приведена иллюстрация данного процесса.

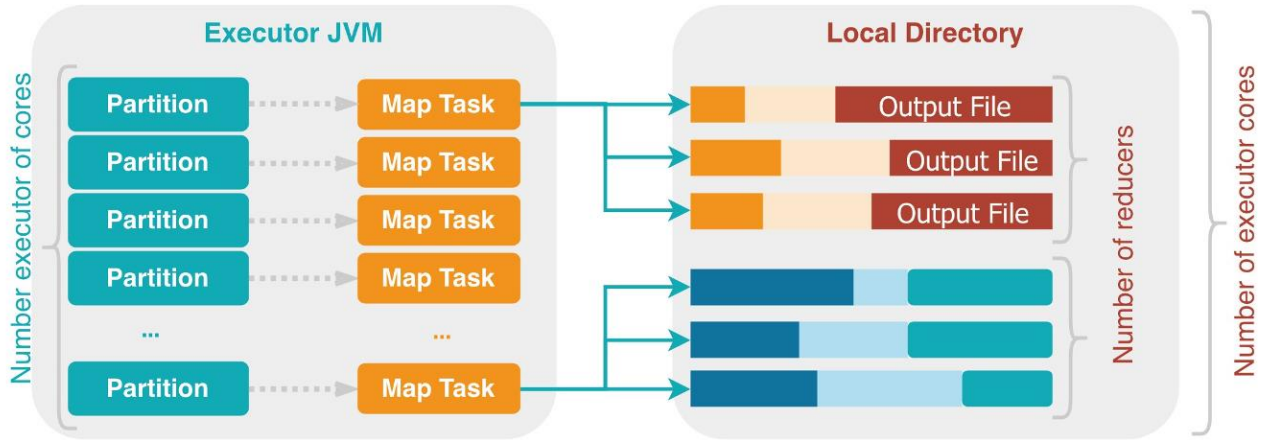


Рисунок 11 – Shuffle – процесс изменяющий размещение данных

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. 12).

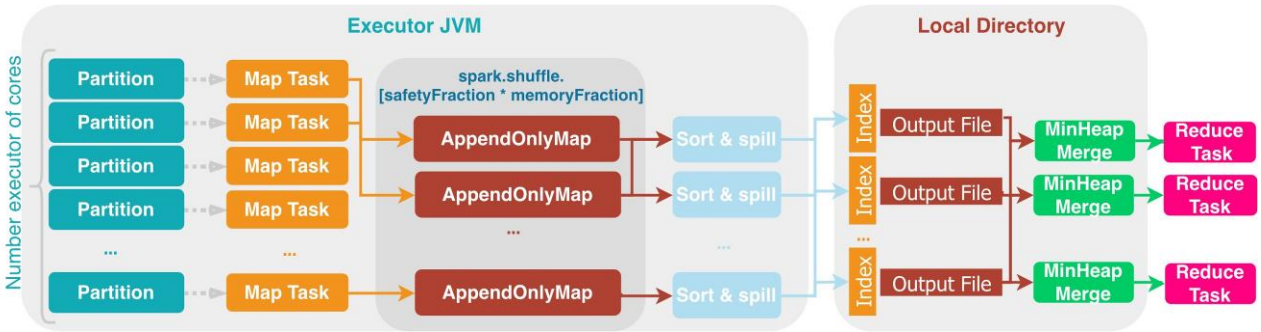


Рисунок 12 – Sort Shuffle – процесс, изменяющий размещение данных

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. 13:

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).

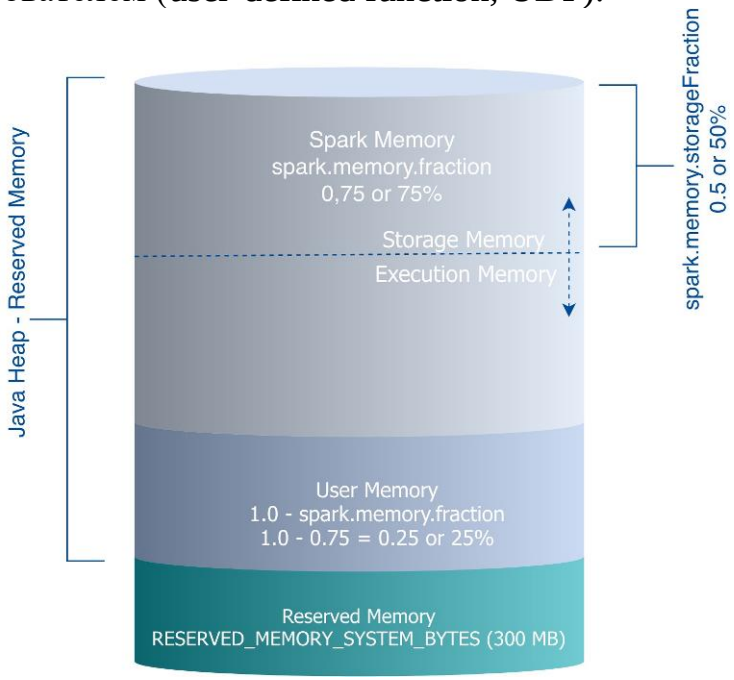


Рисунок 13 –Структура оперативной памяти executor'а приложения Spark

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

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.

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. ~

DataFrame API и Spark SQL

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.
- 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 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 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.

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 , а пример его создания приведен в листинге 3.

Листинг 3.– 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 4.

Листинг 4 – 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.

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. 14 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 5, 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 14 – Схемы датафреймов: (а) followersDf; (б) followingDf

Листинг 5 – 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 6 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 6 – Результат команды explain для скрипта из листинга 5

```
== 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 5 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 7 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 7 – 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 7 представлен в листинге 8.

Листинг 8 – Результат команды explain для скрипта из листинга 7

```
== 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>
```

Использование пользовательских функций (UDF)

Ниже, в листинге 9, 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 9 – 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")
```

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 10.

Листинг 10 – 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
 )
```

```scala
  // 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()
```

. 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.

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.

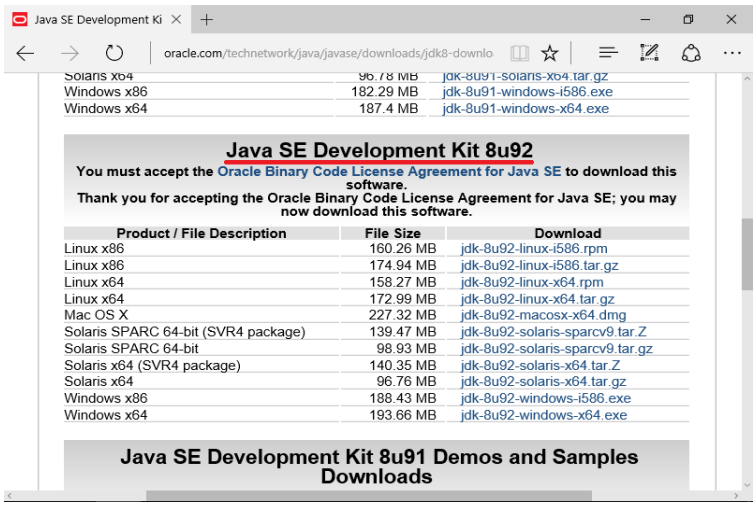


Рисунок 14 – Java SE Development Kit.

Затем проследовать инструкциям инсталлятора и использовать настройки по умолчанию. Необходимо также скачать Apache Maven http://maven.apache.org/download.cgi, что изображено на рис. 15:

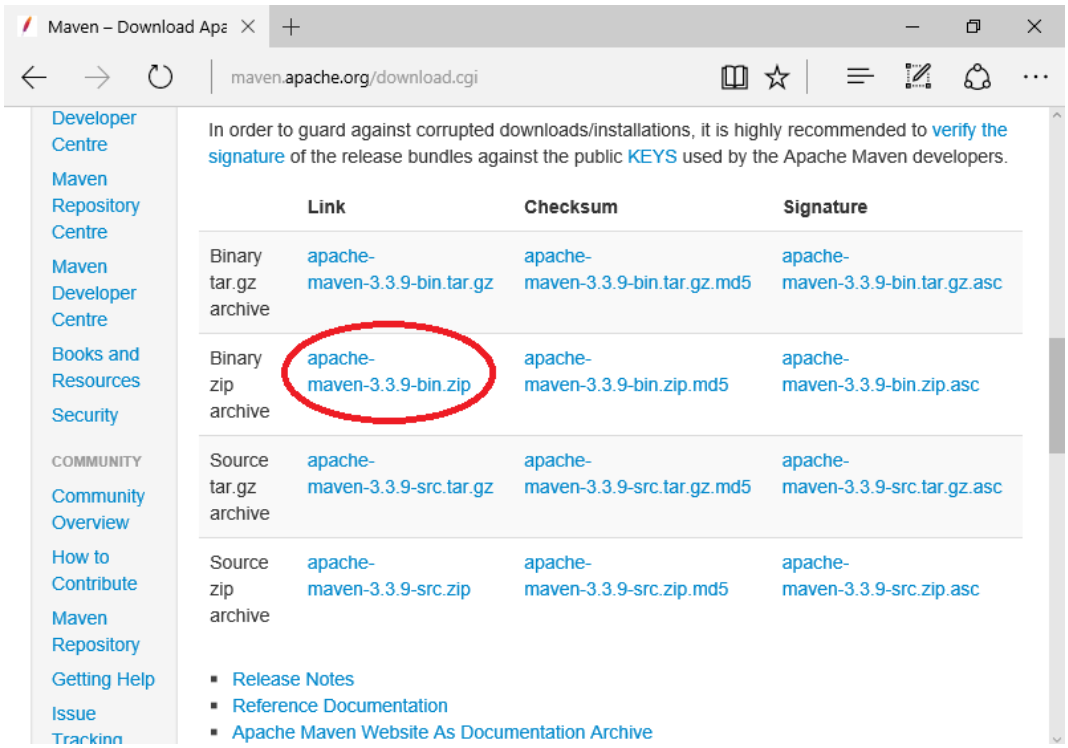


Рисунок 15 – Загрузка Maven с главного сайта.

Извлеките файлы из архива в корень диска C: (см. Рис. 16).

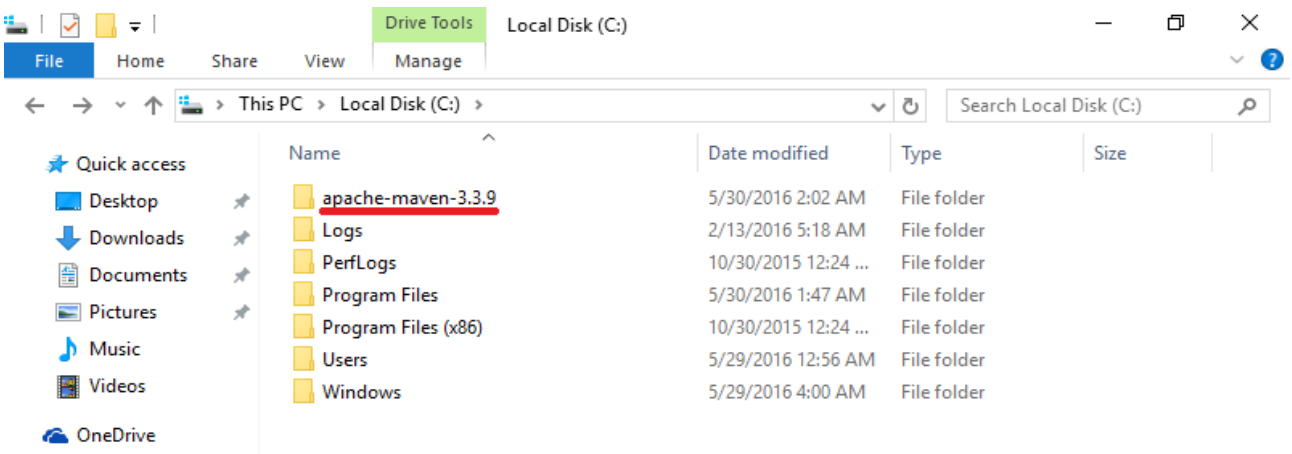


Рисунок 16 – Директория с Apache Maven.

В поиске (нажмите клавишу Windows) найдите и выберите: *Система (Панель управления)/ System (Control Panel)* (рис 17).

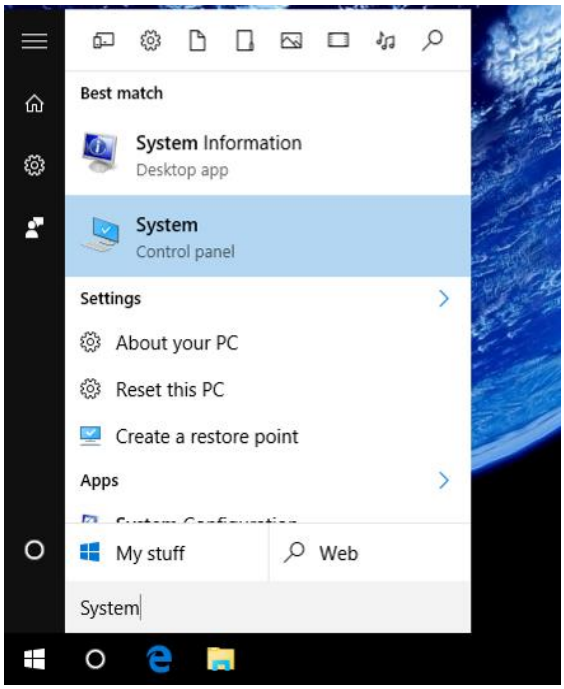


Рисунок 18 – Панель управления.

Нажмите *Расширенные настройки системы (Advanced system settings)*, затем *Переменные среды (Environment Variables)*(см. Рис. 19).

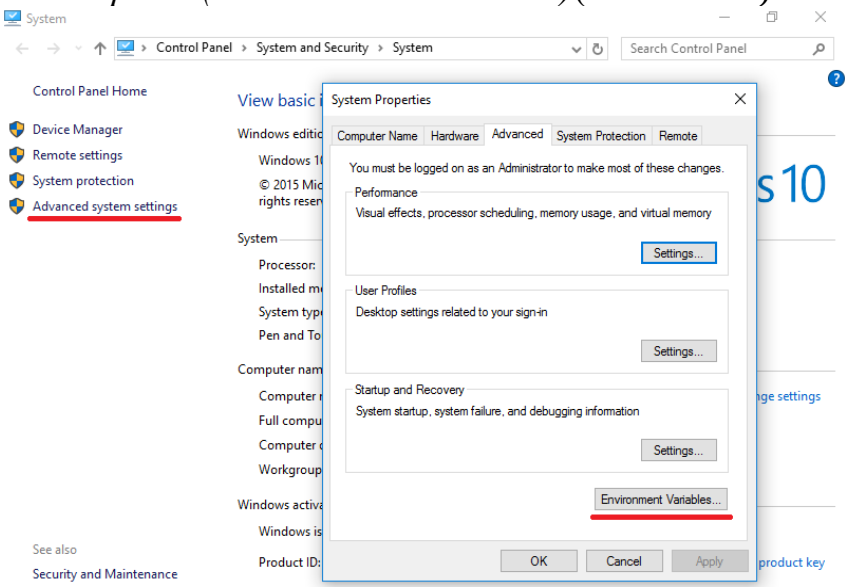


Рисунок 19 – Окно системных свойств.

Добавьте две новые системные переменные: JAVA_HOME и M2_HOME нажав *New* в разделе *System Variables* (см. рис. 20 и рис. 21).

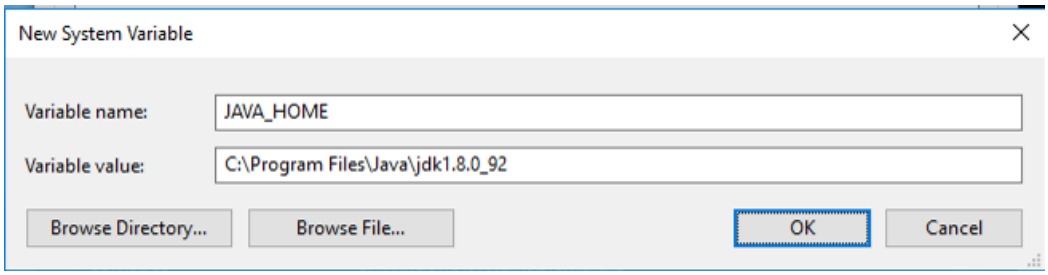


Рисунок 20 – Создание переменной JAVA_HOME.

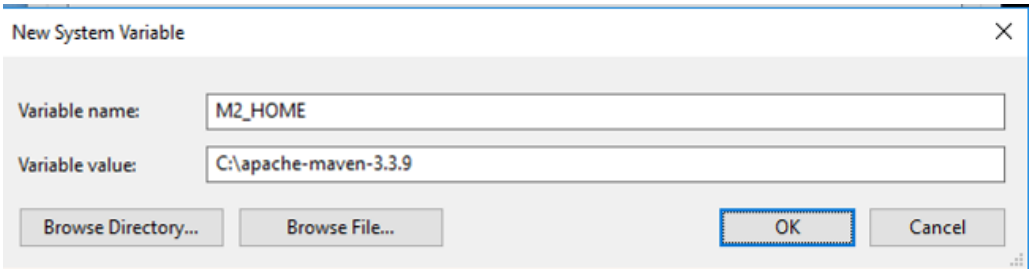


Рисунок21– Создание переменной M2_HOME.

После этого найдите переменную Path (см. Рис. 22) и нажмите Edit.

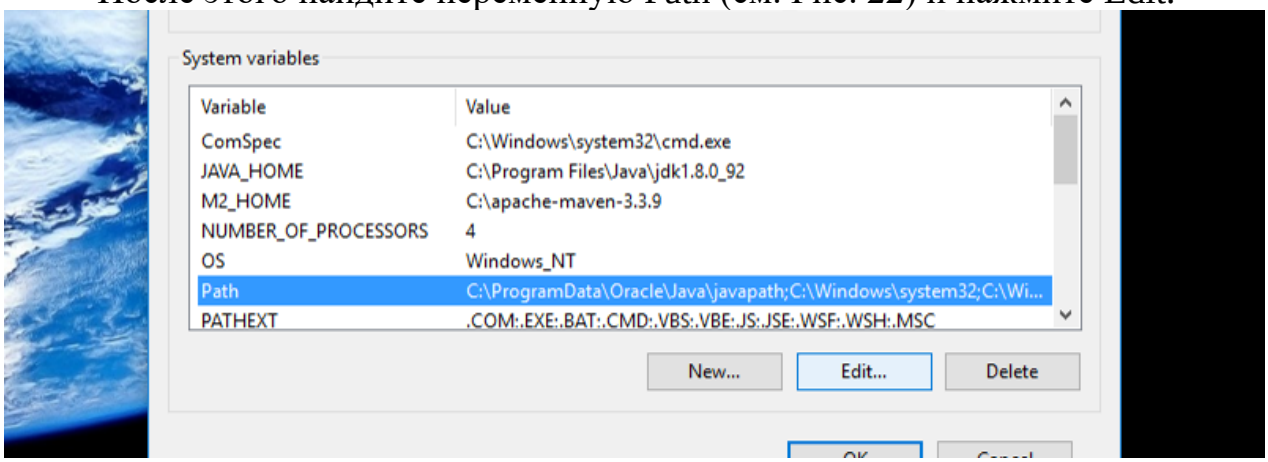


Рисунок 22 – Настройка переменной PATH

Добавьте две следующие строки: %JAVA_HOME%\bin и %M2_HOME%\bin, нажав *New* и заполнив пробелы (см. рис. 23).

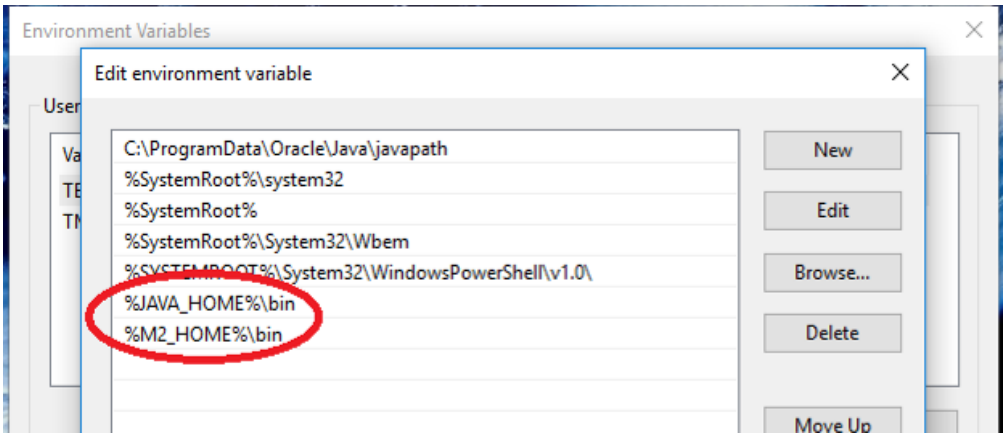


Рисунок 23 – Настройка переменных окружения.

Закройте все окна, нажав OK.

Откройте командную строку, нажав клавишу Windows, и введите cmd (см. Рис. 24). Попробуйте выполнить следующие команды в открывшейся консоли: java и mvn. Команды должны быть доступны.

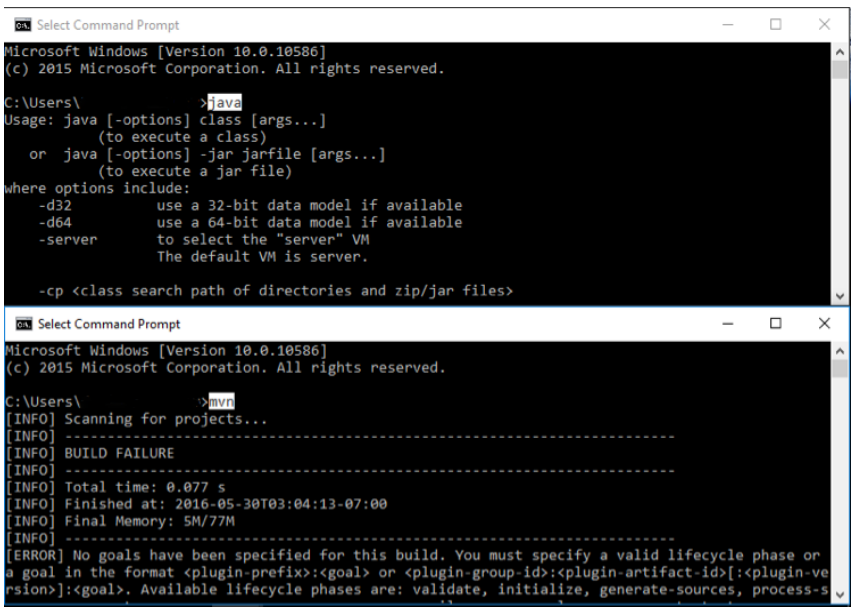


Рисунок 24 – Проверка команд java и mvn в терминале

Скачайте и установите IntelliJ IDEA Community Edition, если она не установлена: http://www.jetbrains.com/idea/download. Используйте настройки по умолчанию (см. Рис. 25).

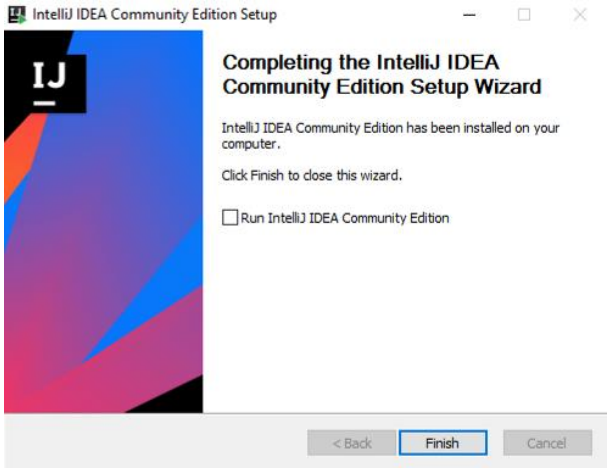


Рисунок 25 – IntelliJ IDEA Окно установки.

Также необходимо установить Scala плагин для IntelliJ IDEA. (рис. 26).

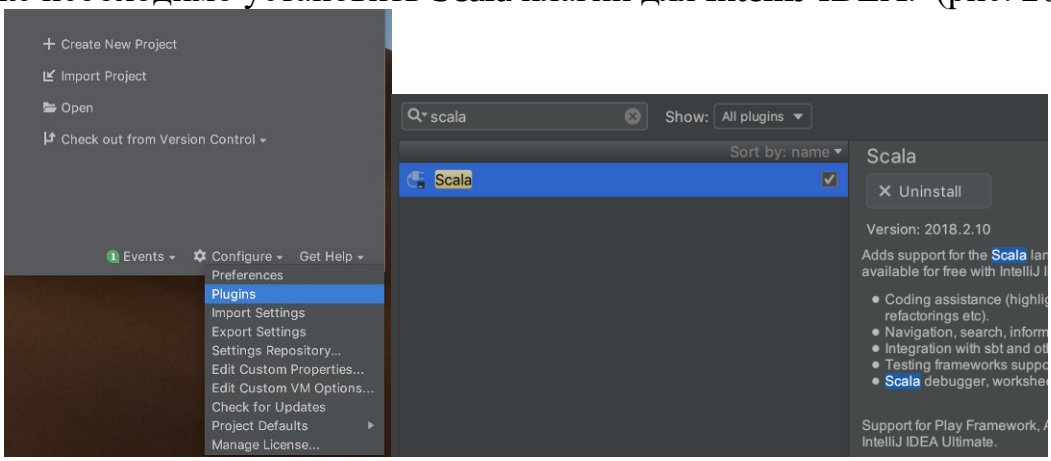


Рисунок 26 – IntelliJ IDEA, установка Scala плагина.

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

Листинг 10 – 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 11 – 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 12 – 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
```

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 27 – IntelliJ IDEA приветственное окно.

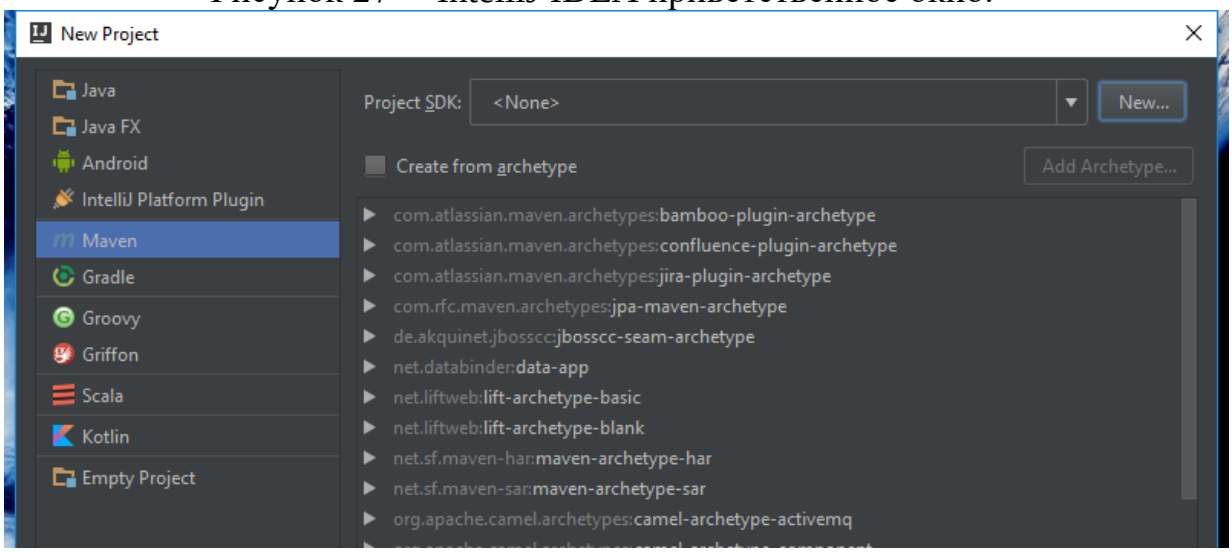


Рисунок 27 – Создание нового проекта.

На левой панели выберите Maven. На правой панели нажмите кнопку *Создать* (см. рис. 27). Укажите домашний каталог JDK (см. Рис. 28). На разных ОС это действие может отличаться.

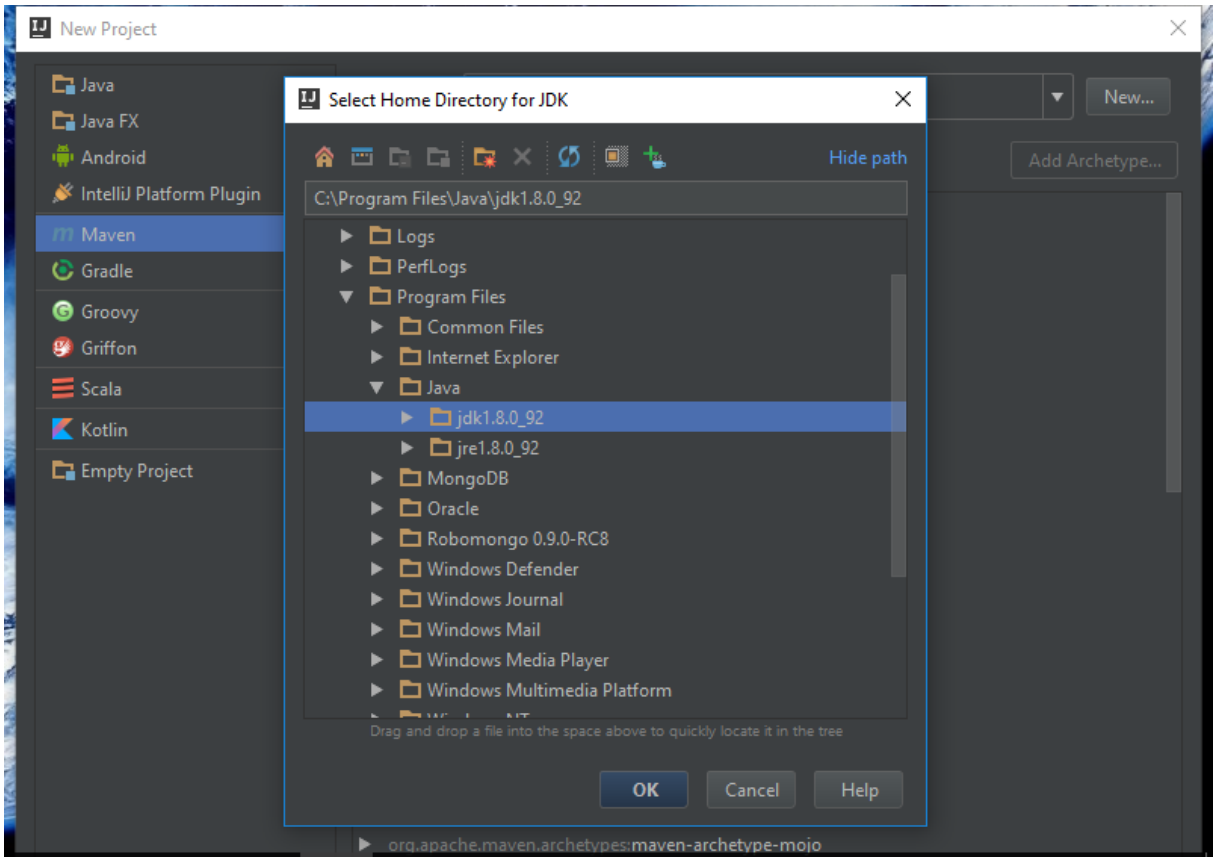


Рисунок 28 – Выбор JDK.

Нажмите *OK*, затем *Далее*. Выполните некоторые дополнительные настройки проекта (см. рис. 29 и рис. 30), если они нужны.

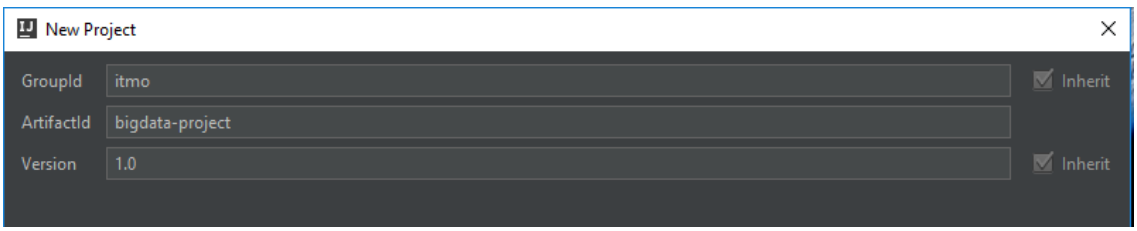


Рисунок 29 – Дополнит ельные настройки проекта.

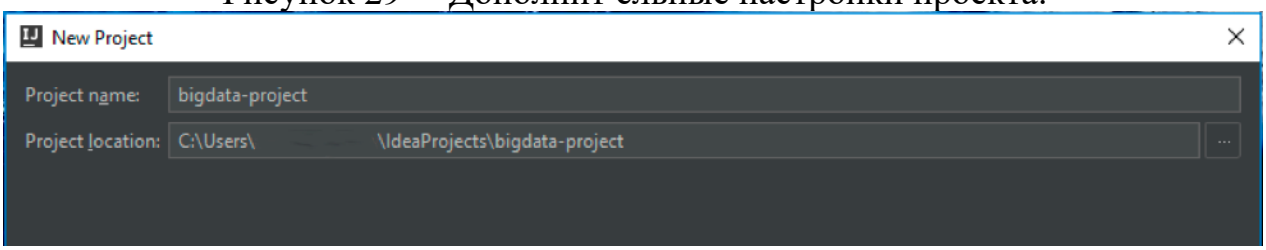


Рисунок 30 – Дополнительные настройки проекта.

Нажмите *Finish*. В открывшемся уведомлении нажмите *Enable Auto-Import* (рис. 31).

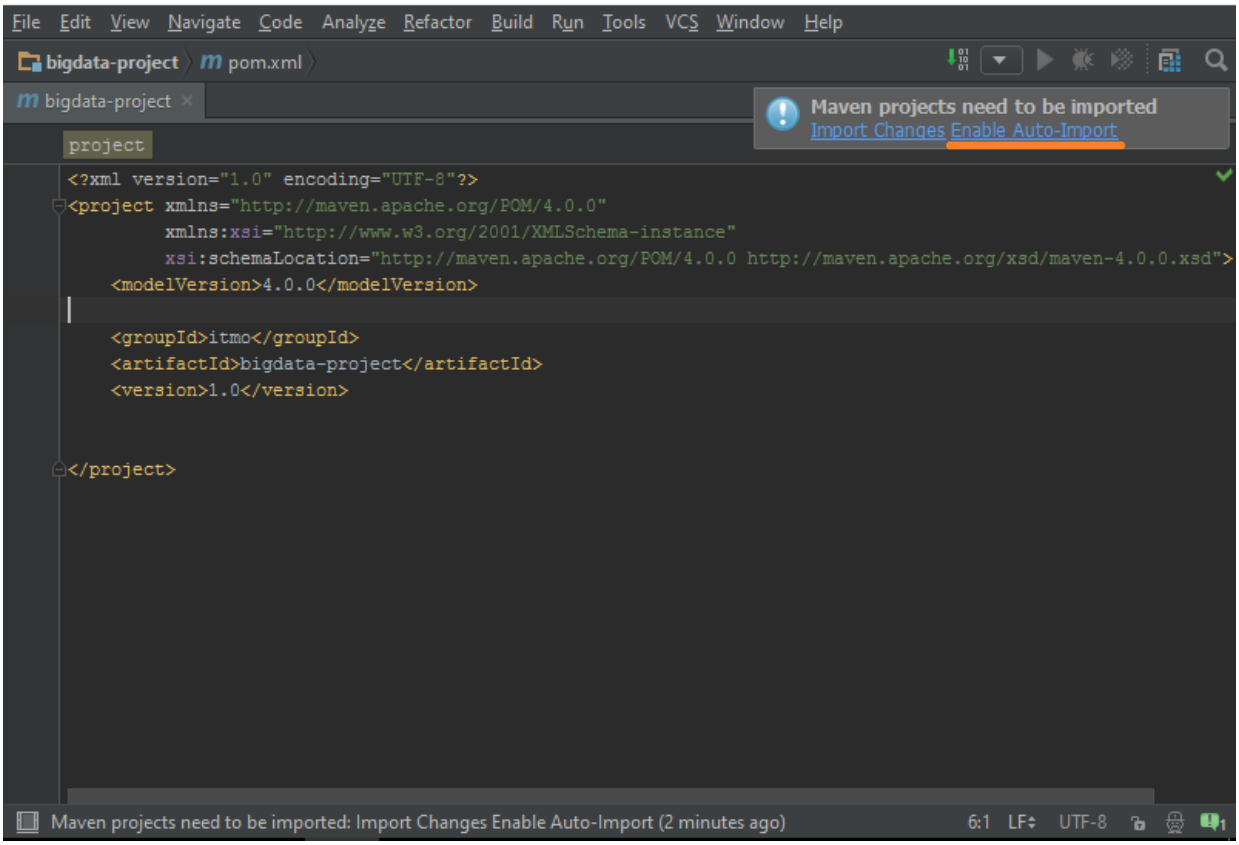


Рисунок 32 – Автоматическое импортирование maven зависимостей.

Создайте папку scala в каталоге src/main (см. рис. 33). Чтобы сделать это, щелкните правой кнопкой мыши на папке main, затем выберите *New* и *Directory*. Напишите имя scala и нажмите кнопку *OK*.

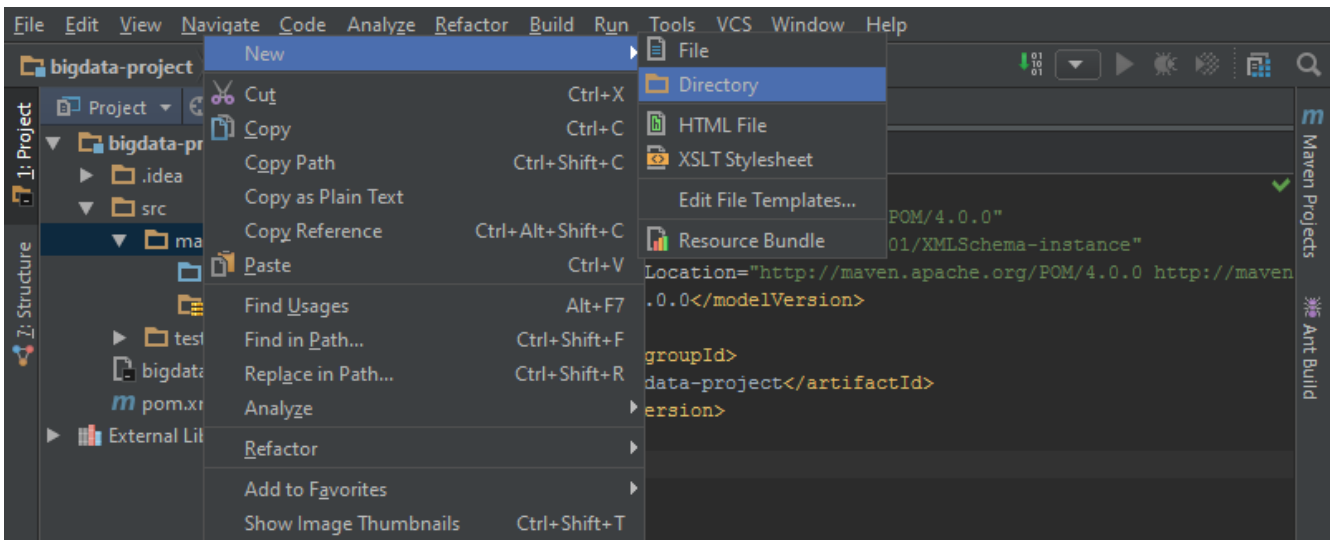


Рисунок 33 – Создание новой папки.

Затем щелкните правой кнопкой мыши папку Scala, выберите *Mark Directory As* и *Sources Root* (см. Рис. 33).

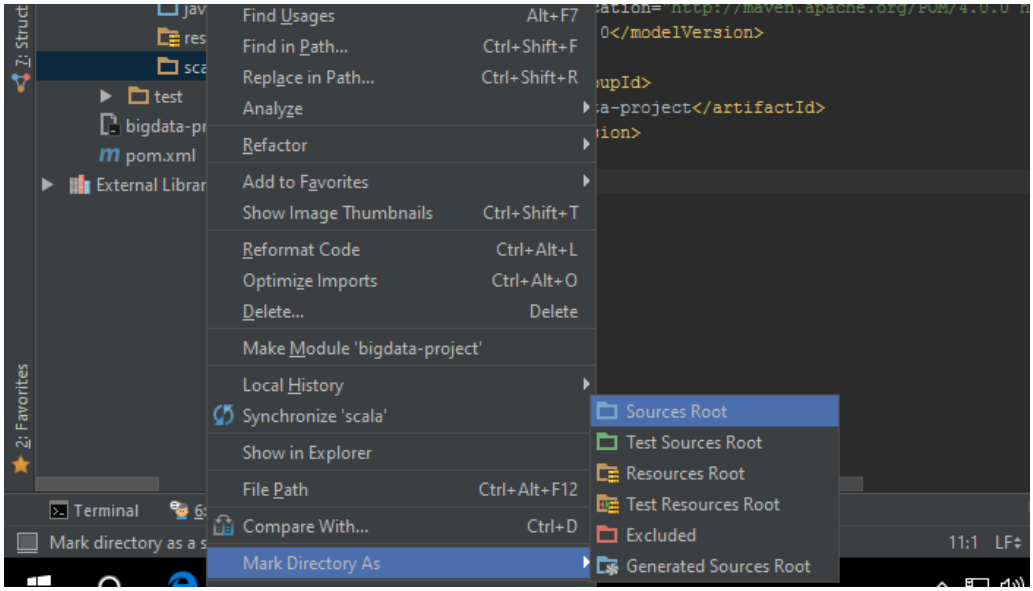


Рисунок 33 – Настройка директории с исходными файлами.

Добавьте следующий код в файл pom.xml (см. листинг. 13)

Листинг 13 – 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. 34), 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.

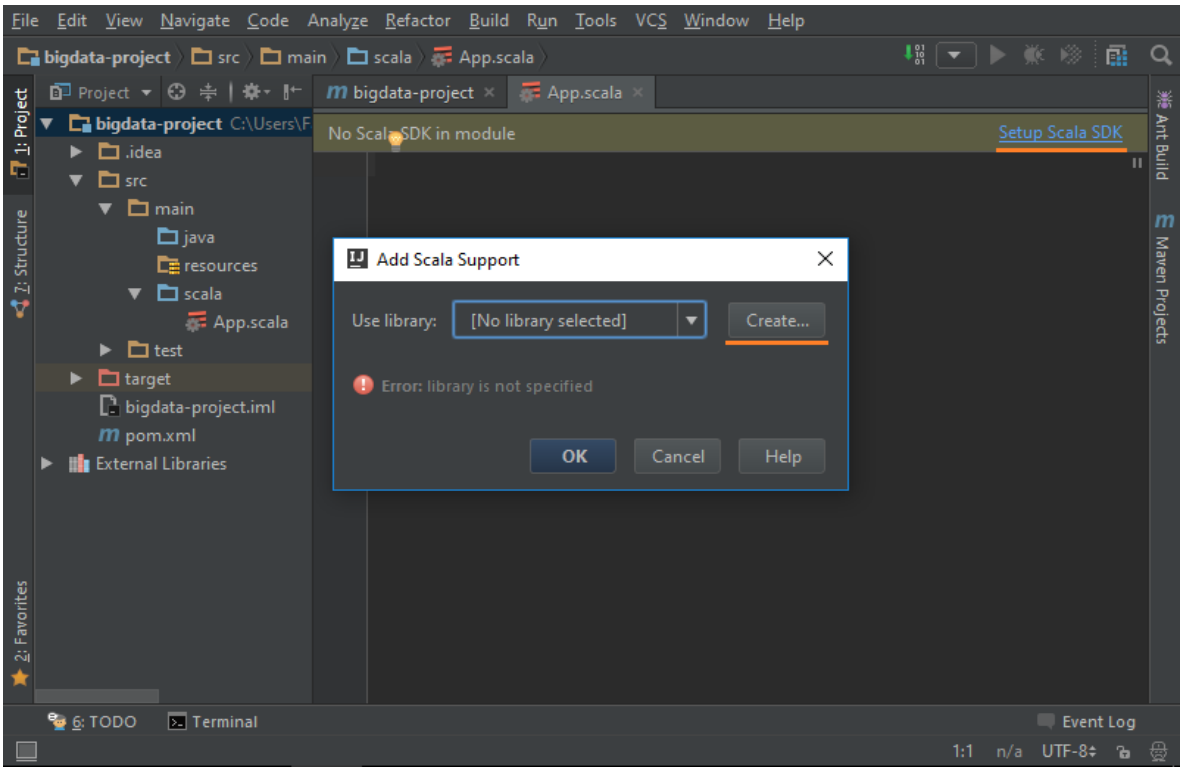


Рисунок 34 – Выбор scala для проекта Intellij Idea.

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

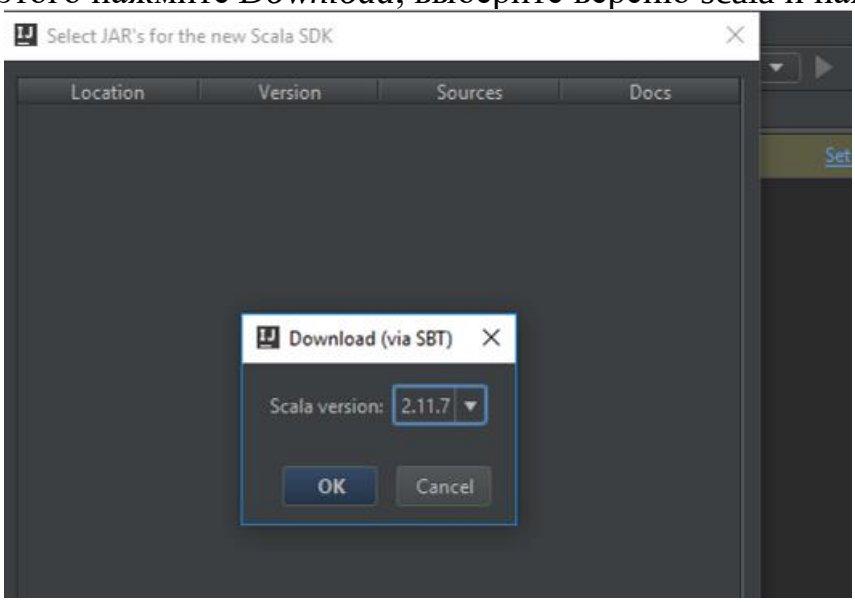


Рисунок 35 – Выбор scala для проекта Intellij Idea.

Будьте терпеливы, так как этот процесс может занять много времени, если у вас слабое интернет-соединение. После загрузки нажмите *OK* (см. рис. 36.

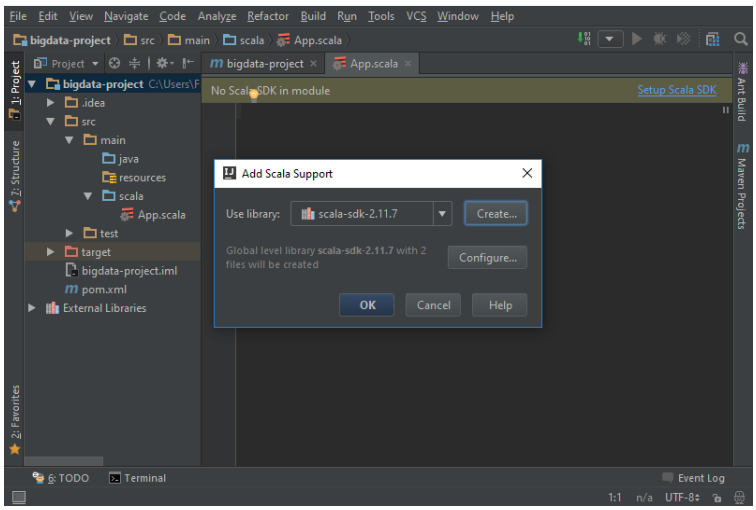


Рисунок 37 – Выбор scala для проекта Intellij Idea.

Запуск Scala проекта в Intellij Idea

Добавьте следующий код в файл app.scala.

Листинг 15 – первая scala программа

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

Наберите *Ctrl + Shift + F10* или нажмите *Run* (см. рис. 38) и выберите App.

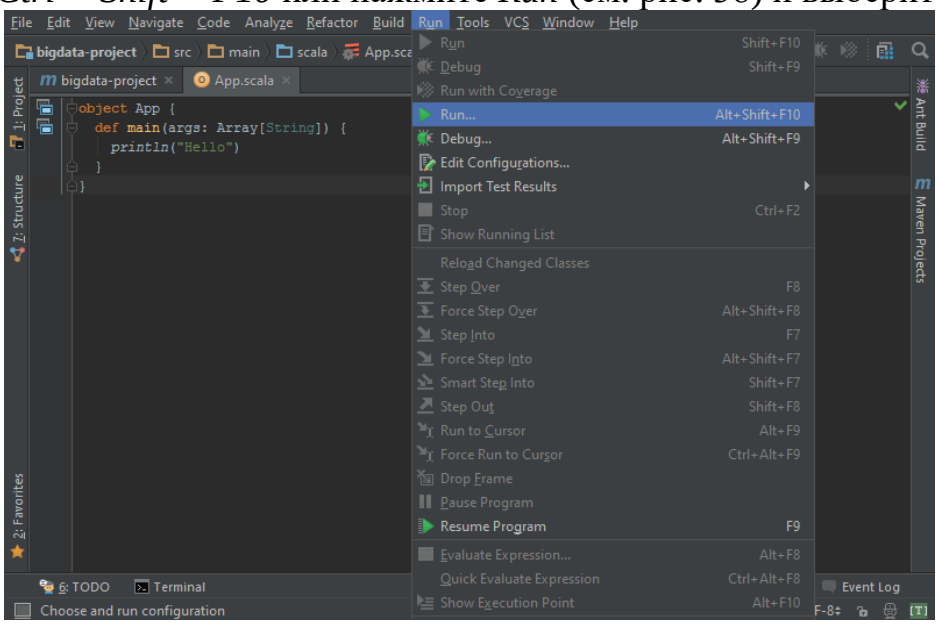


Рисунок 38– Запуск проекта в Intellij Idea.

Вы также можете использовать следующую кнопку (см. рис. 39).

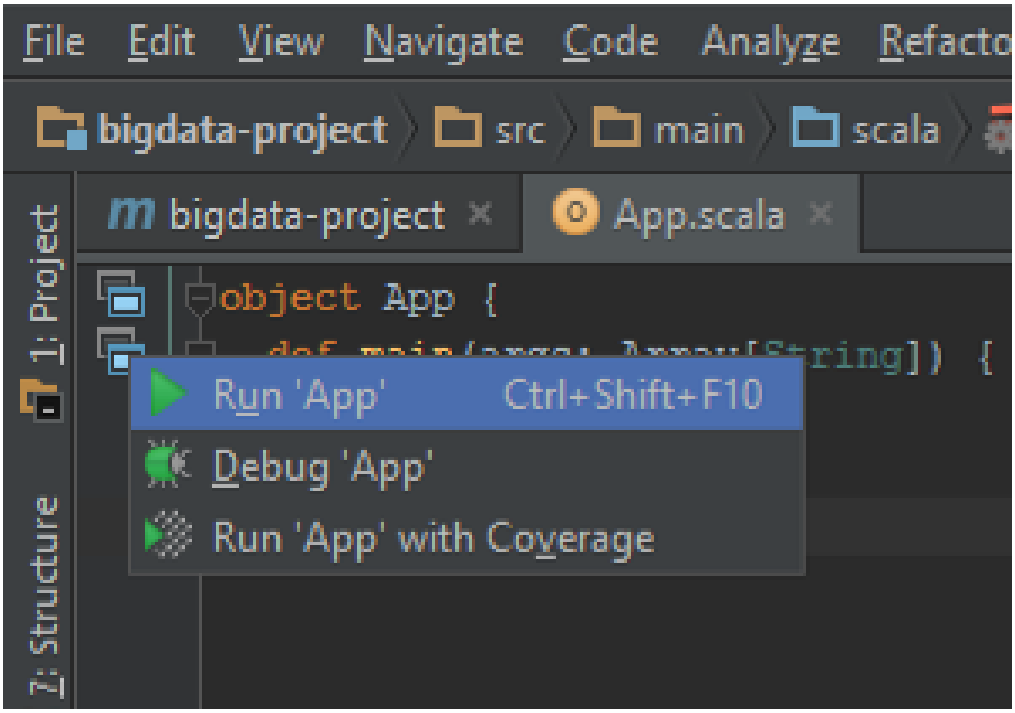


Рисунок 39– Запуск проекта в Intellij Idea.

Итак, код работает (см. рис. 40) и это означает, что настройка выполнена.

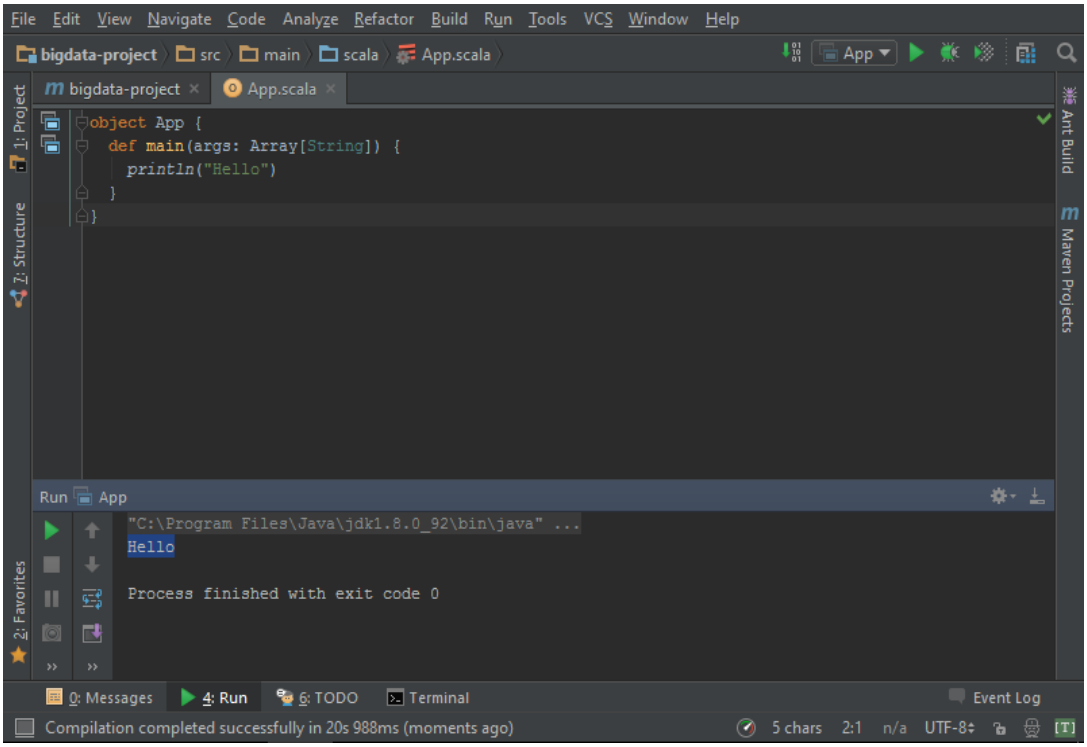


Рисунок 40 – Запуск проекта в Intellij Idea.

Первое Spark приложение

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

Листинг 16 – 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 17 – 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. 41). Возможно, вы увидите ошибку.

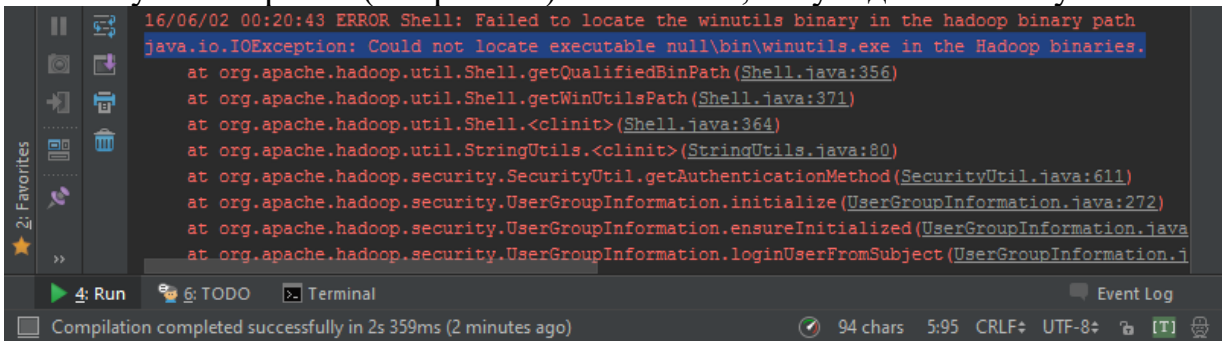


Рисунок 41 – Ошибка в windows.

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 17 – Установка системного свойства HADOOP_HOME.

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

Перезапустите проект (см. рис. 42).

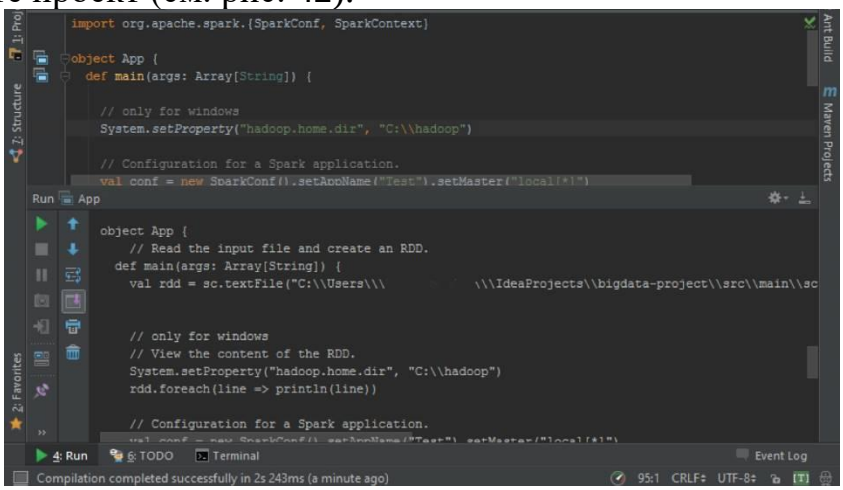


Рисунок 42 – Успешный запуск Spark проекта в IntelliJ Idea.

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

Лабораторные задания

1 Установить и настроить Apache Spark.

2 Получить у преподавателя индивидуальное задание по разработке Spark проекта и выполнить его.

3 Оформить отчёт.

ЛАБОРАТОРНАЯ РАБОТА 7.
Apache Kafka. ПОТОКОВАЯ ОБРАБОТКА ДАННЫХ

Цель работы: Изучение Apache Kafka, получение практических навыков потоковой обработки данных.

Kafka был разработан в компании LinkedIn в 2011 году и с тех пор значительно усовершенствовался. Сегодня Kafka – это целая платформа, обеспечивающая избыточность, достаточную для хранения огромных объемов данных. Здесь предоставляется шина сообщений с колоссальной пропускной способностью, на которой можно в реальном времени обрабатывать абсолютно все проходящие через нее данные.

Apache Kafka – это платформа для потоковой передачи событий.

С технической точки зрения, потоковая передача событий — это практика сбора данных в режиме реального времени из источников событий, таких как базы данных, датчики, мобильные устройства, облачные службы и программные приложения, в виде потоков событий; надежное хранение этих потоков событий для последующего извлечения; манипулирование, обработка и реагирование на потоки событий в режиме реального времени, а также ретроспективно; и маршрутизация потоков событий к различным технологиям назначения по мере необходимости. Таким образом, потоковая передача событий обеспечивает непрерывный поток и интерпретацию данных, так что нужная информация оказывается в нужном месте и в нужное время.

Kafka — это распределенная система, состоящая из **серверов** и **клиентов,** которые обмениваются данными по высокопроизводительному сетевому протоколу TCP . Его можно развернуть на «голом железе», виртуальных машинах и контейнерах как в локальной, так и в облачной среде.

Серверы : Kafka запускается как кластер из одного или нескольких серверов, которые могут охватывать несколько центров обработки данных или облачных регионов. Некоторые из этих серверов образуют уровень хранения, называемый брокерами. На других серверах работает Kafka Connect для непрерывного импорта и экспорта данных в виде потоков событий для интеграции Kafka с вашими существующими системами, такими как реляционные базы данных, а также с другими кластерами Kafka. Чтобы вы могли реализовать критически важные варианты использования, кластер Kafka отличается высокой масштабируемостью и отказоустойчивостью: если какой-либо из его серверов выйдет из строя, другие серверы возьмут на себя его работу, чтобы обеспечить непрерывную работу без потери данных.

Клиенты : они позволяют вам писать распределенные приложения и микросервисы, которые считывают, записывают и обрабатывают потоки событий параллельно, в масштабе и отказоустойчивым образом даже в случае сетевых проблем или сбоев машины. Kafka поставляется с некоторыми такими клиентами, которые дополняются десятками клиентов, предоставляемых сообществом Kafka: клиенты доступны для Java и Scala,

включая библиотеку Kafka Streams более высокого уровня , для Go, Python, C/C++ и многих других программ. языков, а также REST API.

Рассмотрим особенности Kafka.

Распределенный

Распределенной называется такая система, которая в сегментированном виде работает сразу на множестве машин, образующих цельный кластер; поэтому для конечного пользователя они выглядят как единый узел. Распределенность Kafka заключается в том, что хранение, получение и рассылка сообщений у него организовано на разных узлах (так называемых «брокерах»). Важнейшие плюсы такого подхода – высокодоступность и отказоустойчивость.

Горизонтально масштабируемый

Давайте сначала определимся с тем, что такое вертикальная масштабируемость. Допустим, у нас есть традиционный сервер базы данных, и он постепенно перестает справляться с нарастающей нагрузкой. Чтобы справиться с этой проблемой, можно просто нарастить ресурсы (CPU, RAM, SSD) на сервере. Это и есть **вертикальное масштабирование** – на машину навешиваются дополнительные ресурсы. При таком «масштабировании вверх» возникает два серьезных недостатка:

Есть определенные пределы, связанные с возможностями оборудования. Бесконечно наращиваться нельзя.

1. Такая работа обычно связана с простоями, а большие компании не могут позволить себе простоев.

Горизонтальная масштабируемость решает ровно ту же проблему, просто мы подключаем все больше машин. При добавлении новой машины никаких простоев не происходит, при этом, количество машин, которые можно добавить в кластер, ничем не ограничено. Загвоздка в том, что не во всех системах поддерживается горизонтальная масштабируемость, многие системы не рассчитаны на работу с кластерами, а те, что рассчитаны – обычно очень сложны в эксплуатации.

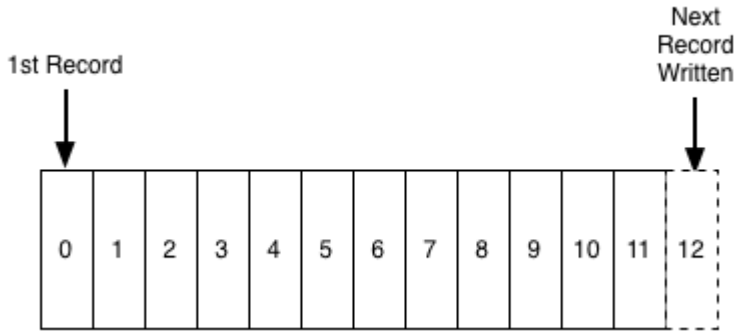
Отказоустойчивость

Для нераспределенных систем характерно наличие так называемой единой точки отказа. Распределенные системы проектируются таким образом, чтобы их конфигурацию можно было корректировать, подстраиваясь под отказы. Кластер Kafka из пяти узлов остается работоспособным, даже если два узла лягут. Необходимо отметить, что для обеспечения отказоустойчивости обязательно приходится частично жертвовать производительностью, поскольку чем лучше ваша система переносит отказы, тем ниже ее производительность.

Журнал коммитов

Журнал коммитов (также именуемый «журнал опережающей записи», «журнал транзакций») – это долговременная упорядоченная структура данных,

причем, данные в такую структуру можно только добавлять. Записи из этого журнала нельзя ни изменять, ни удалять. Информация считывается слева направо; таким образом гарантируется правильный порядок элементов.



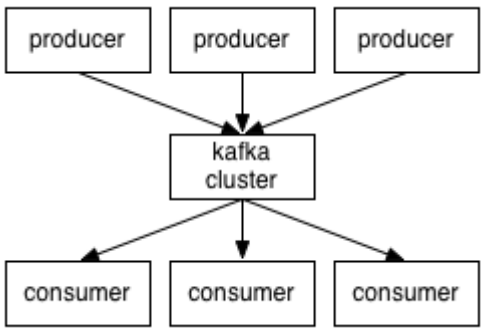
В сущности, Kafka хранит все свои сообщения на диске, а при упорядочивании сообщений в виде вышеописанной структуры можно пользоваться последовательным считыванием с диска.

- Операции считывания и записи выполняются за постоянное время O(1) (если известен ID записи), что, по сравнению с операциями O(log N) на диске в другой структуре, невероятно экономит время, так как каждая операция подвода головок затратна.
- Операции считывания и записи не влияют друг на друга (операция считывания не блокирует операцию записи и наоборот, чего не скажешь об операциях со сбалансированными деревьями).

Два этих момента радикально увеличивают производительность, поскольку она совершенно не зависит от размера данных. Kafka работает одинаково хорошо, будь у вас на сервере 100KB или 100TB данных.

Как все это работает?

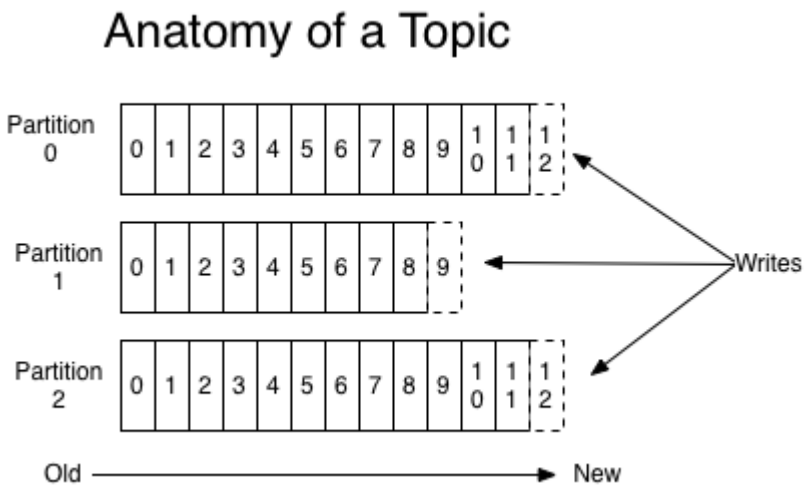
Приложения (**генераторы**) посылают сообщения (**записи**) на узел Kafka (**брокер**), и указанные сообщения обрабатываются другими приложениями, так называемыми **потребителями**. Указанные сообщения сохраняются в **теме**, а потребители подписываются на тему для получения новых сообщений.



Темы могут разрастаться, поэтому крупные темы подразделяются на более мелкие **секции** для улучшения производительности и масштабируемости. (пример: *допустим, вы сохраняли пользовательские запросы на вход в систему;*

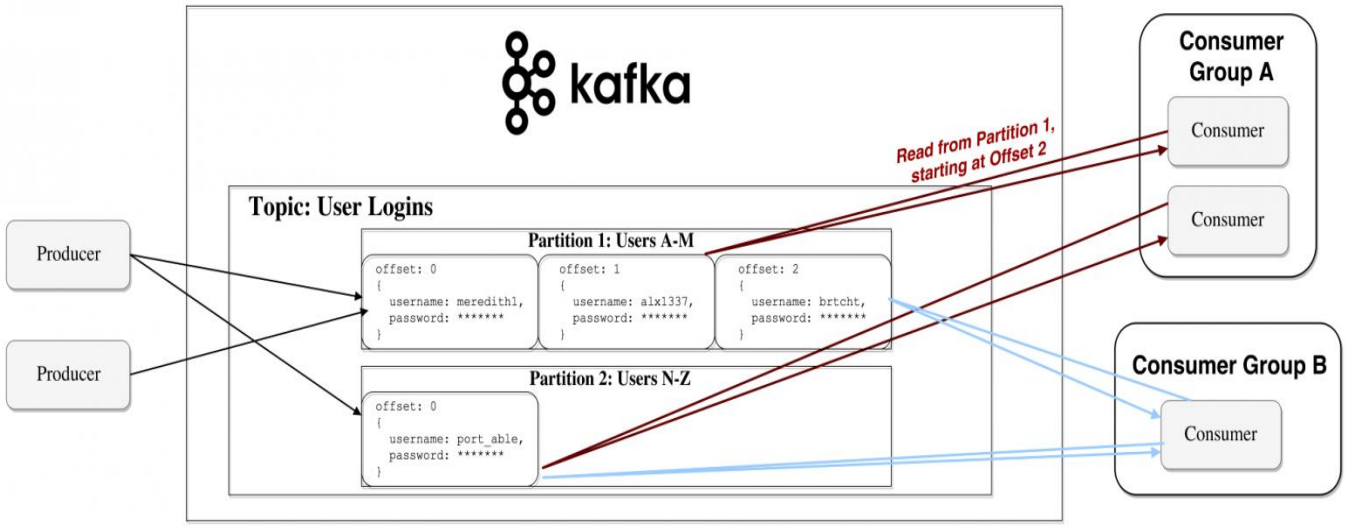
в таком случае можно распределить их по первому символу в имени пользователя)

Kafka гарантирует, что все сообщения в пределах секции будут упорядочены именно в той последовательности, в которой поступили. Конкретное сообщение можно найти по его смещению, которое можно считать обычным индексом в массиве, порядковым номером, который увеличивается на единицу для каждого нового сообщения в данной секции.



В Kafka соблюдается принцип «тупой брокер – умный потребитель». Таким образом, Kafka не отслеживает, какие записи считываются потребителем и после этого удаляются, а просто хранит их в течение заданного периода времени (например, суток), либо до тех пор, пока не будет достигнут некоторый порог. Потребители сами опрашивают Kafka, не появилось ли у него новых сообщений, и указывают, какие записи им нужно прочесть. Таким образом, они могут увеличивать или уменьшать смещение, переходя к нужной записи; при этом события могут переигрываться или повторно обрабатываться.

Следует отметить, что на самом деле речь идет не об одиночных потребителях, а о группах, в каждой из которых – один или более процессов-потребителей. Чтобы не допустить ситуации, когда два процесса могли бы дважды прочесть одно и то же сообщение, каждая секция привязывается лишь к одному процессу-потребителю в пределах группы.



Долговременное хранение на диске

Как упоминалось выше, Kafka на самом деле хранит свои записи на диске и ничего не держит в оперативной памяти. Да, возможен вопрос, есть ли в этом хоть капля смысла. Но в Kafka действует множество оптимизаций, благодаря которым такое становится осуществимым:

1. В Kafka есть протокол, объединяющий сообщения в группы. Поэтому при сетевых запросах сообщения складываются в группы, что позволяет снизить сетевые издержки, а сервер, в свою очередь, сохраняет партию сообщений за один присест, после чего потребители могут сразу выбирать большие линейные последовательности таких сообщений.
2. Линейные операции считывания и записи на диск происходят быстро. Известна проблема: современные диски работают сравнительно медленно из-за необходимости подвода головок, однако, при крупных линейных операциях такая проблема исчезает.
3. Указанные линейные операции сильно оптимизируются операционной системой путем **опережающего чтения** (заблаговременно выбираются крупные группы блоков) и **запаздывающей записи** (небольшие логические операции записи объединяются в крупные физические операции записи).
4. Современные ОС кэшируют диск в свободной оперативной памяти. Такая техника называется **страничный кэш**.
5. Поскольку Kafka сохраняет сообщения в стандартизированном двоичном формате, который не изменяется на протяжении всей цепочки (генератор->брокер->потребитель), здесь уместна оптимизация нулевого копирования. В таком случае ОС копирует данные из страничного кэша прямо на сокет, практически обходя стороной приложение-брокер, относящееся к Kafka.

Благодаря всем этим оптимизациям Kafka доставляет сообщения практически так же быстро, как и сама сеть.

Распределение и репликация данных

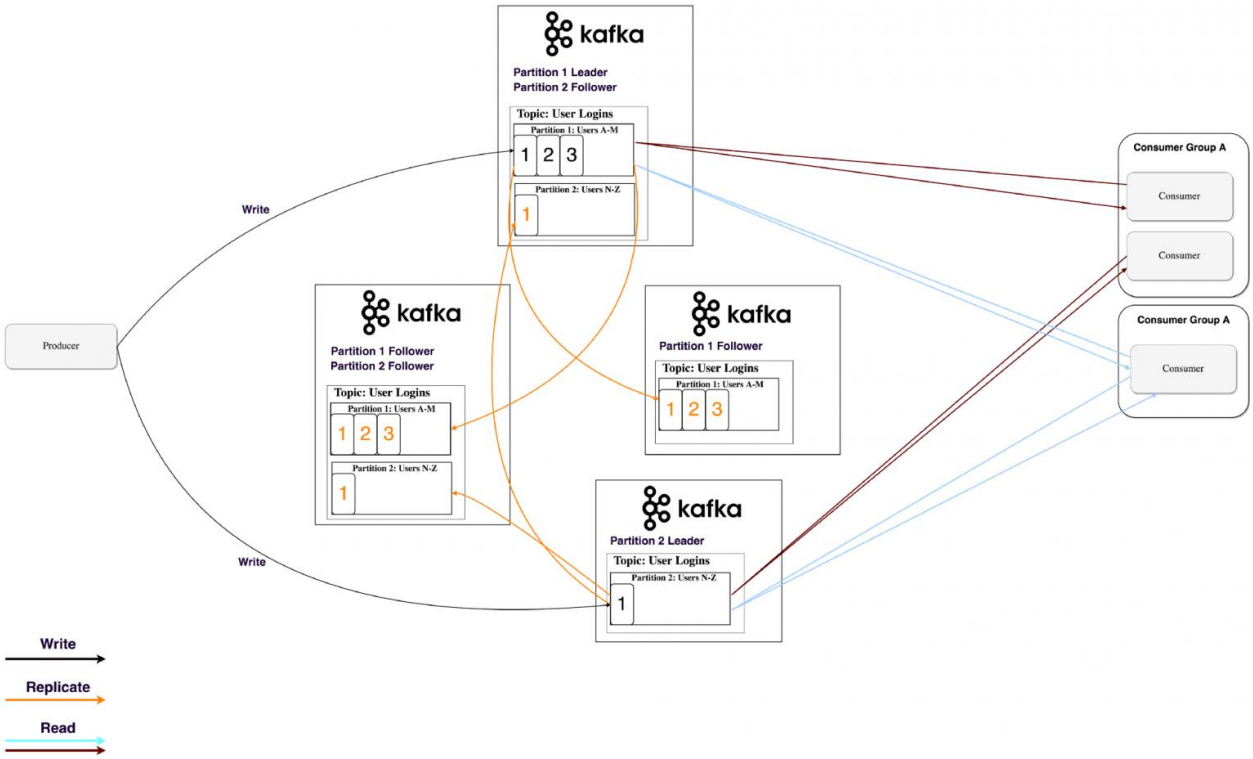
Теперь давайте обсудим, как в Kafka достигается отказоустойчивость, и как он распределяет данные между узлами.

Репликация данных

Данные с сегмента реплицируются на множестве брокеров, чтобы данные сохранились, если один из брокеров откажет.

В любом случае, один из брокеров всегда “владеет” секцией: этот брокер — именно тот, на котором приложения выполняют операции считывания и записи в секцию. Этот брокер называется «**ведущим секции**». Он реплицирует получаемые данные на ***N*** других брокеров, так называемых **ведомыми**. На ведомых также хранятся данные, и любой из них может быть выбран в качестве ведущего, если актуальный ведущий откажет.
Так можно сконфигурировать гарантии, обеспечивающие, что любое

сообщение, которое будет успешно опубликовано, не потеряется. Когда есть возможность изменить коэффициент репликации, можно частично пожертвовать производительностью ради повышенной защиты и долговечности данных (в зависимости от того, насколько они критичны).



Таким образом, если один из ведущих когда-нибудь откажет, его место может занять ведомый.

Однако чтобы генератор/потребитель мог записывать/считывать информацию в данной секции, приложению нужно знать, какой из брокеров здесь ведущий. Эту информацию нужно где-то взять.

Для хранения таких метаданных в Kafka используется сервис под названием **Zookeeper**.

Что такое Zookeeper?

Zookeeper – это распределенное хранилище ключей и значений. Оно сильно оптимизировано для считывания, но записи в нем происходят медленнее. Чаще всего Zookeeper применяется для хранения метаданных и обработки механизмов кластеризации (пульс, распределенные операции обновления/конфигурации, т.д.).

Таким образом, клиенты этого сервиса (брокеры Kafka) могут на него подписываться – и будут получать информацию о любых изменениях, которые могут произойти. Именно так брокеры узнают, когда ведущий в секции меняется. Zookeeper исключительно отказоустойчив (как и должно быть), поскольку Kafka сильно от него зависит.

Он используется для хранения всевозможных метаданных, в частности:

- Смещение групп потребителей в рамках секции (хотя, современные клиенты хранят смещения в отдельной теме Kafka)
- ACL (списки контроля доступа)—используются для ограничения доступа /авторизации
- Квоты генераторов и потребителей — максимальные предельные количества сообщений в секунду
- Ведущие секций и уровень их работоспособности

Как генератор/потребитель определяет ведущего брокера данной секции?

Ранее Генератор и Потребители непосредственно подключались к Zookeeper и узнавали у него эту (а также другую) информацию. Теперь Kafka уходит от такой связки и, начиная, соответственно, с версий 0.8 и 0.9 клиенты, во-первых, выбирают метаданные непосредственно у брокеров Kafka, а брокеры обращаются к Zookeeper.

Поток метаданных

Потоки

Потоковый процессор в Kafka отвечает за всю следующую работу: принимает непрерывные потоки данных от входных тем, каким-то образом обрабатывает этот ввод и подает поток данных на выходные темы (либо на внешние сервисы, базы данных, в корзину, да куда угодно…) Простую обработку можно выполнять непосредственно на API генераторов/потребителей, однако, более сложные преобразования – например, объединение потоков, в Kafka выполняется при помощи интегрированной библиотеки Streams API.

Этот API предназначен для использования в рамках вашей собственной базы кода, на брокере он не работает. Функционально он подобен API потребителя, облегчает горизонтальное масштабирование обработки потоков и распределение его между несколькими приложениями (подобными группам потребителей).

Обработка без сохранения состояния

Обработка без сохранения состояния — это поток детерминированной обработки, не зависящий ни от каких внешних факторов. В качестве примера рассмотрим вот такое простое преобразование данных: прикрепляем информацию к строке

"Hello" -> "Hello, World!"

Потоково-табличный дуализм

Важно понимать, что потоки и таблицы – это, в сущности, одно и то же. Поток можно интерпретировать как таблицу, а таблицу – как поток.

Поток как таблица

Если обратить внимание, как выполняется синхронная репликация базы данных, то очевидно, что речь идет о **потоковой репликации**, где любые изменения в таблицах отправляются на сервер копий (реплику). Поток Kafka можно интерпретировать точно так же – как поток обновлений для данных, которые агрегируются и дают конечный результат, фигурирующий в таблице. Такие потоки сохраняются в локальной RocksDB (по умолчанию) и называются **KTable**.

Таблица как поток

Таблицу можно считать мгновенным снимком, отражающим последнее значение для каждого ключа в потоке. Аналогично, из потоковых записей можно составить таблицу, а из обновлений таблицы — сформировать поток с логом изменений.

Обработка с сохранением состояния

Некоторые простые операции, например, map() или filter(), выполняются без сохранения состояния, и нам не приходится хранить каких-либо данных, касающихся их обработки. Однако, на практике большинство операций выполняется с сохранением состояния (напр. count()), поэтому вам, естественно, придется хранить состояние, сложившееся на настоящий момент.

Проблема с поддержанием состояния в потоковых процессорах заключается в том, что эти процессоры иногда отказывают! Где же хранить это состояние, чтобы обеспечить отказоустойчивость?

Упрощенный подход – просто хранить все состояние в удаленной базе данных и подключаться к этому хранилищу по сети. Проблема в том, что тогда теряется локальность данных, а сами данные многократно перегоняются по сети – оба фактора существенно тормозят ваше приложение. Более тонкая, но важная проблема заключается в том, что активность вашего задания потоковой обработки будет жестко зависеть от удаленной базы данных – то есть, это задание будет несамодостаточным (вся ваша обработка может рухнуть, если другая команда внесет в базу данных какие-то изменения).

Итак, какой же подход лучше?

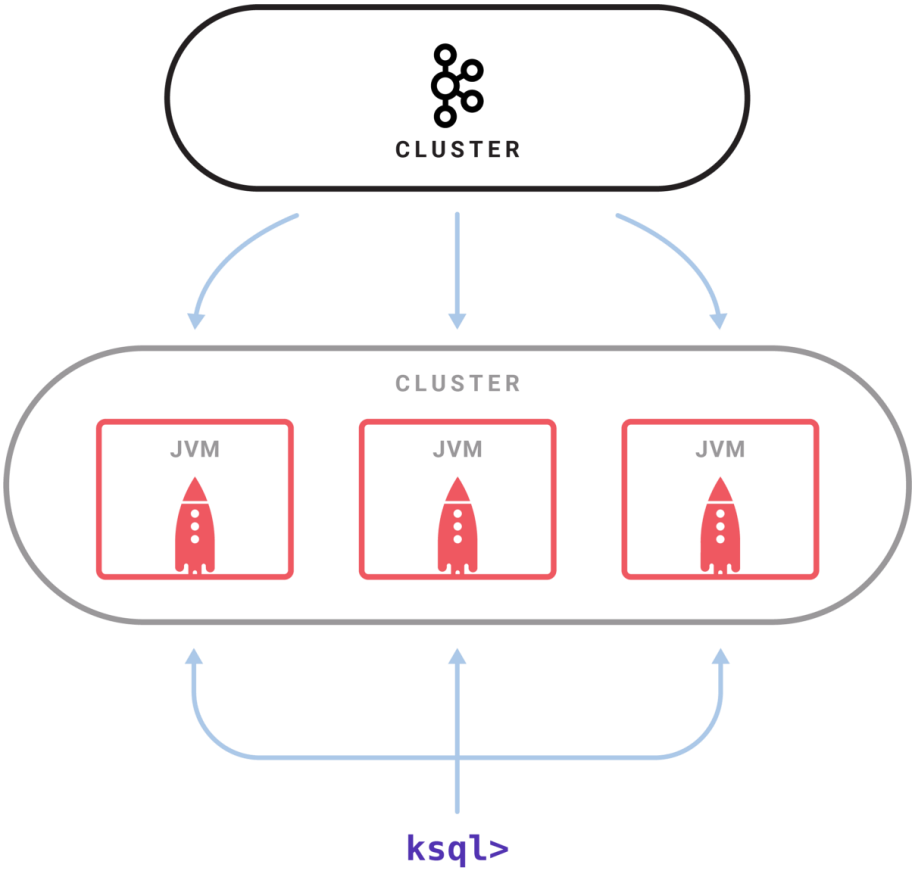
Вновь вспомним о дуализме таблиц и потоков. Именно благодаря этому свойству потоки можно преобразовывать в таблицы, расположенные именно там, где происходит обработка. Также при этом получаем механизм, обеспечивающий отказоустойчивость – мы храним потоки на брокере Kafka.

Потоковый процессор может сохранять свое состояние в локальной таблице (например, в RocksDB), которую будет обновлять входной поток (возможно, после каких-либо произвольных преобразований). Если этот процесс сорвется, то мы сможем восстановить соответствующие данные, повторно воспроизведя поток.

Можно добиться даже того, чтобы удаленная база данных генерировала поток и, фактически, широковещательно передавала лог изменений, на основании которого вы будете перестраивать таблицу на локальной машине.

KSQL

Как правило, код для обработки потоков приходится писать на одном из языков для JVM, поскольку именно с ней работает единственный официальный клиент Kafka Streams API.



KSQL позволяет писать простые потоковые задания на знакомом языке, напоминающем SQL.

Описание Apache Kafka

С подробной документацией по Apache Kafka можно ознакомиться по ссылке https://kafka.apache.org/documentation/

Лабораторные задания

1 Ознакомиться с документацией по Apache Kafka по ссылке https://kafka.apache.org/documentation/

2 Установить и настроить Apache Kafka

3 Получить у преподавателя индивидуальное задание по обработке данных с использованием Apache Kafka.

4 Оформить отчёт.

ЛАБОРАТОРНАЯ РАБОТА № 8.
ИЗУЧЕНИЕ ВОЗМОЖНОСТЕЙ И РАБОТА С ДОКУМЕНТО-ОРИЕНТИРОВАННОЙ БД MongoDB

Цель работы Изучение особенностей и возможностей MongoDB, получение практических навыков работы с документо-ориентированной БД

Знакомство с MongoDB

В последнее время для задач оперативной обработки плохоструктурированных данных все большую популярность приобретают NoSQL (от англ.: «*Not Only SQL*» – «*Не только SQL*») БД. Они используются не только как элемент хранилища данных, но все чаще как само хранилище.

NoSQL – это ряд технологий, подходов, проектов направленных на реализацию моделей баз данных, имеющих существенные отличия от СУБД, работающих с языком SQL. Чаще всего данные в NoSQL решении представляются в виде хеш-таблиц, деревьев, документов и других структур.

Одной из самых популярных NoSQL СУБД в настоящее время является MongoDB. MongoDB – это документно-ориентированная база данных с открытым исходным кодом.

Основными особенностями MongoDB являются:

– документно-ориентированное хранилище;

– полная поддержка индексов;

– репликация данных;

– высокая доступность данных;

– способность к горизонтальному масштабированию;

– авто-шардинг;

– поддержка запросов;

– поддержка Map/Reduce;

– поддержка GridFS.

Более подробное описание БД и её особенностей представлено на официальном сайте http://www.mongodb.org/

Так как MongoDB относится к NoSQL базам данных, и является документно-ориентированной, то каждая запись в ней является документом без жестко заданной схемы. Каждый документ может содержать вложенные документы.

MongoDB обладает хорошей скоростью работы с данными (чтения/записи), хорошей масштабируемостью. Благодаря отличным реализациям репликации и шардинга, базу данных mongo легко реплицировать на кластер компьютеров или настроить шардинг (возможность разнести данные по нескольким серверам). Кроме того MongoDB обладает системой распределенных вычислений с высокой степенью отказоустойчивости.

Репликация – это тиражирование изменений данных с главного сервера БД на одном или нескольких зависимых серверах.

Шардинг – разделение данных на уровне ресурсов, разбиение данных по какому-либо признаку. Концепция шардинга заключается в логическом разделении данных по различным ресурсам.

Для управления документами используется нотация JSON, для их хранения – BSON .

MongoDB не поддерживает модель транзакционной целостности ACID. Это означает, что в MongoDB отсутствует понятие «транзакция». Например, данные, изменяемые одним клиентом, одновременно могут читаться другим. Атомарность присутствует, но только на уровне целого документа.

Установка MongoDB.

Для начала работы с MongoDB её необходимо установить. К методическим указаниям прилагаются архивы с MongoDB, для различных операционных систем. Установка и работа с MongoDB в методических указаниях рассматривается на примере операционной системы Windows7.

Для других операционных систем семейства Windows действия по установке/настройке и работе с MongoDB аналогичны.

В подпапке «Windows» папки «MongoInstall» необходимо выбрать архив, подходящий к версии вашей ОС: «mongodb-win32-i386-2.4.3.zip» для 32-разрядной ОС и «mongodb-win32-x86_64-2.4.3.zip» для 64-разрядной ОС. Предпочтительной является версия для 64-х битной ОС, т.к. 32-х битная версия имеет ограничения функциональности. Максимальный размер БД в 32-разрядной версии приложения составляет 2 Гб.

Далее необходимо распаковать архив в папку, в которой будет проходить работа с mongo. В методических указаниях корневой папкой для mongo является «D:\mongoDB». Распакованный архив содержит папку «bin», содержащую исполняемые файлы mongo. Содержание папки «bin» представлено на рисунке 44.

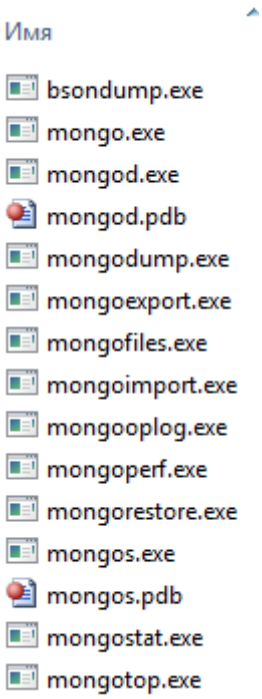


Рисунок 44– Приложения, входящие в состав MongoDB

Все приложения в составе mongo можно условно разделить на следующие группы: приложения ядра базы данных, инструменты для дампа бинарных файлов MongoDB, инструменты импорта и экспорта данных,

инструменты для диагностики и приложения распределенной файловой системы.

Основными приложениями являются приложения «mongod.exe», «mongos.exe» и «mongo.exe».

Приложение «mongod.exe» является главным процессом MongoDB. Оно работает с данными: обрабатывает запросы, управляет форматами данных и т.д. По сути, «mongod.exe» является сервером базы данных.

Приложение «mongos.exe» предназначено для шардинга данных. Оно представляет собой сервис маршрутизации для конфигурирования шардов, обрабатывает запросы от уровня приложений и определяет местоположение данных в кластере.

Приложение «mongo.exe» является консольным клиентом для «mongod.exe», и представляет собой интерфейс тестирования запросов для работы с базой данных и её администрирования.

Обратите внимание, что если ваш ПК работает под управление ОС Windows 7, вам необходимо установить обновление (http://support.microsoft.com/kb/2731284) для разрешения проблемы с маппингом памяти в консольных приложениях.[7]

Данное обновление автоматически устанавливается через центр обновлений Windows, если он у вас включен.

Обновление также прилагается к методическим указаниям: «3rd parties\Fix405791_x32_zip.exe» для 32х систем, и «3rd parties\Fix405791_x64_zip.exe» для 64х систем.

После извлечения файлов mongo, необходимо определить место хранения базы данных. По умолчанию MongoDB ищет файлы БД в папке «C:\data\db». Вы можете создать папку для хранения базы данных в любом месте. У вас должна получиться иерархия, подобная представленной на рисунке 45.

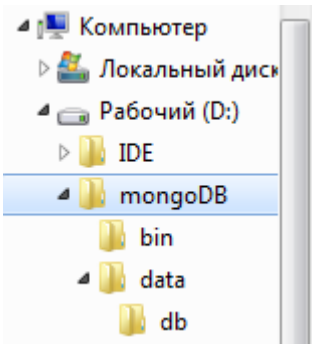


Рисунок 45 – Иерархия папок в MongoDB

MongoDB можно установить в любую папку на компьютере. Для установки достаточно скопировать папки MongoDB в требуемое место.

Mongo может работать в двух режимах:

– в качестве обычного приложения Windows;

– в качестве Windows-сервиса.

В случае использования mongo как сервиса, он автоматически запускается при старте системы.

Запуск MongoDB как Windows приложения

Для запуска MongoDB как Windows приложения необходимо запустить приложение «mongod.exe» с помощью командной строки. Запуск MongoDB для папки «D:\mongoDB\bin» можно произвести следующим образом:

```
D:\mongoDB\bin\mongod
```

или

```
D:
cd \mongoDB\bin
mongod
```

Если у вас папка с базой данных располагается в месте, отличном от стандартного размещения базы данных mongo, то вам необходимо произвести запуск mongo с параметром «--dbpath». Синтаксис команды следующий:

```
--dbpath <путь к папке>
```

Следующий пример показывает, как запустить «mongod.exe» с произвольным путем к папке с БД:

```
mongod --dbpath=D:\mongodb\db
D:\mongoDB\bin\mongod --dbpath=D:\mongodb\db
```

или

```
mongod --dbpath= ../mongodb/db
D:\mongoDB\bin\mongod --dbpath=../mongodb/db
```

В случае успешного запуска консоль перейдет в режим ожидания подключения как показано на рисунке 46.

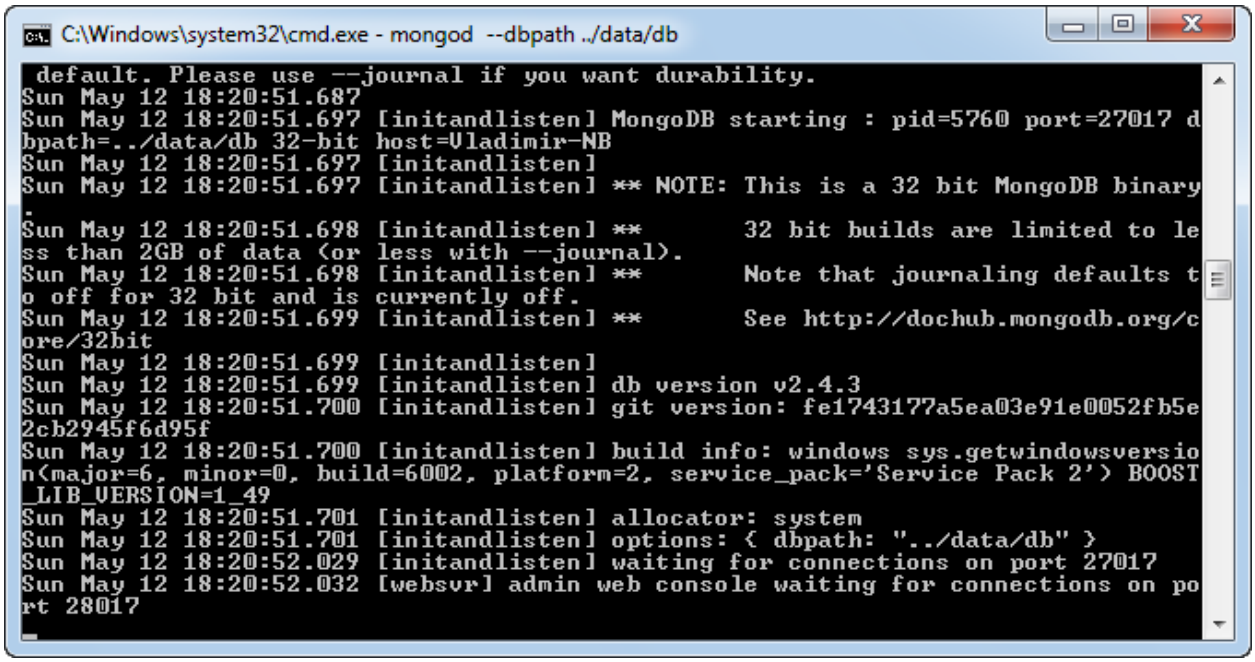


Рисунок 46 – Экранная форма консоли ожидающей подключения клиента

Если после попытки запуска приглашение на ввод команды повторилось, то произошла ошибка запуска. Одной из наиболее частых проблем является изначально неправильно указанный путь к папке с БД. На рисунке 47 показан пример ошибки с неправильно указанным путем к базе данных.

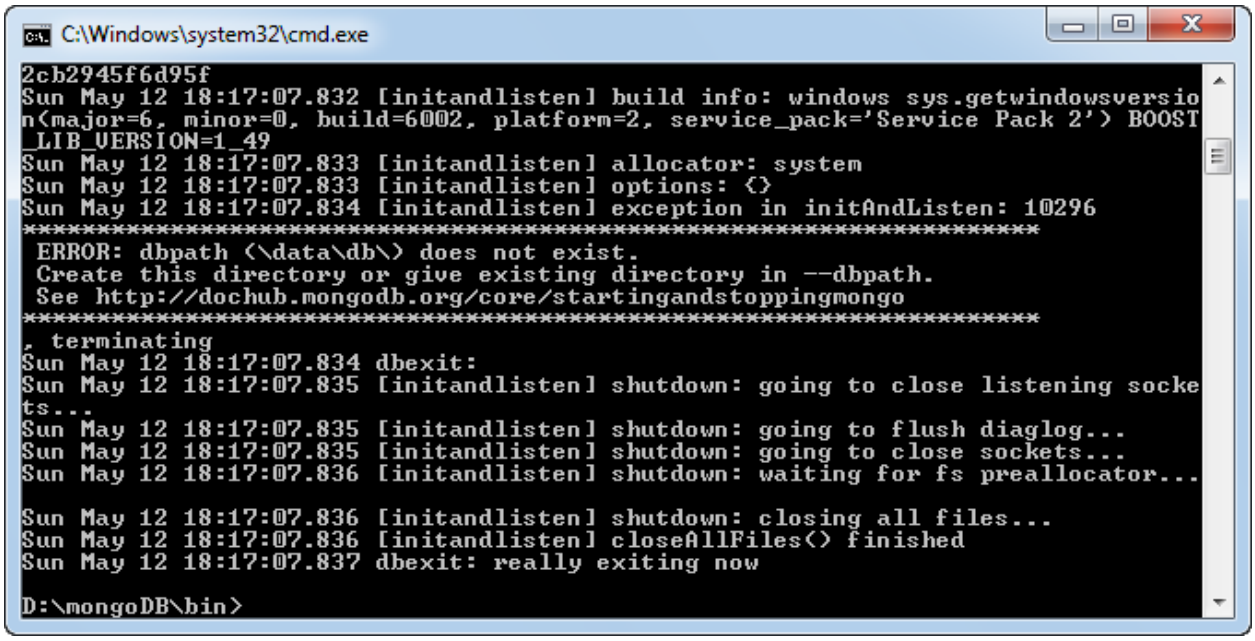


Рисунок 47 – Экранная форма консоли с сообщением об ошибке

Для решения проблемы необходимо указать явно путь к БД. Синтаксис команды «--dbpath» и пример её использования приведены выше.

После успешного запуска окно приложения «mongod.exe» можно свернуть.

Запуск MongoDB как сервис Windows

Обратите внимание, что для установки и запуска MongoDB в качестве сервиса Windows вам потребуются права администратора.

Для использования MongoDB в качестве сервиса необходимо произвести конфигурирование системы: задать пути для логирования и файлов конфигурации.

Первым делом необходимо создать папку для хранения отчетов приложения. Пусть она называется «Log» и находится на одном уровне с папками «bin» и «data».

Следующим шагом является создание файла конфигурирования. Для его создания вам следует создать текстовый файл «mongod.cfg» в папке «D:\MongoDB», и поместить в него строку с путем к файлу отчета: «logpath=<путь к файлу с отчетом>», например:

```
logpath=D:\mongodb\logs\mongo.log
```

Если у вас путь к базе данных отличается от стандартного, то необходимо также прописать и путь к БД. Путь задается следующей строкой: «dbpath=<путь к БД>», например:

```
dbpath=D:\mongodb\data\db
```

Содержание файла конфигурации показано на рисунке 48.

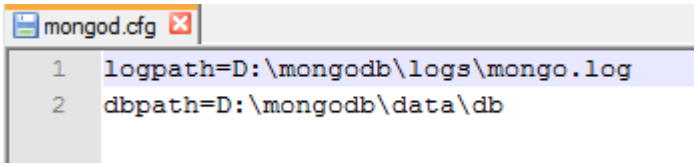


Рисунок 48 – Содержание файла конфигурации MongoDB.

После создания файла конфигурации необходимо произвести установку MongoDB.

Для установки запустите командную строку с правами администратора. Для этого введите «cmd» в строку поиска в меню «Пуск», активируйте контекстное меню приложения «cmd.exe» и выберите пункт меню «Запуск от имени администратора». Запуск командной строки от имени администратора представлен на рисунке 49.

Далее следует запустить «mongod.exe», передав ему в качестве параметра путь к файлу конфигурации и команду на установку. Формат команды: «<путь к mongo> – –config <путь к файлу конфигурации> –install», например:

```
mongod --config D:\mongodb\mongod.cfg –install
D:\mongodb\bin\mongod --config D:\mongodb\mongod.cfg –install
```

При удачной установке вы получите следующее сообщение:

```
Service 'MongoDB' (Mongo DB) installed with command line
'D:\mongodb\bin\mongod.exe --config D:\mongodb\mongod.cfg --service'
```

Далее следует запустить сервис командой «net start»:

```
net start MongoDB
```

Остановка службы осуществляется командой:

```
net stop MongoDB
```

Удаление службы:

```
mongod –remove
```

На этом установка MongoDB как Windows-сервиса закончена.

Также возможно установить mongo как Windows-сервис без создания файла конфигурации, для этого необходимо произвести установку следующей командой:

D:\mongodb\bin\mongod.exe --dbpath=D:\mongodb --logpath=D:\mongodb\log.txt –install

Где:

–-dbpath=<ваш путь к базе данных>,

--logpath=<ваш путь к папке с отчетами>.

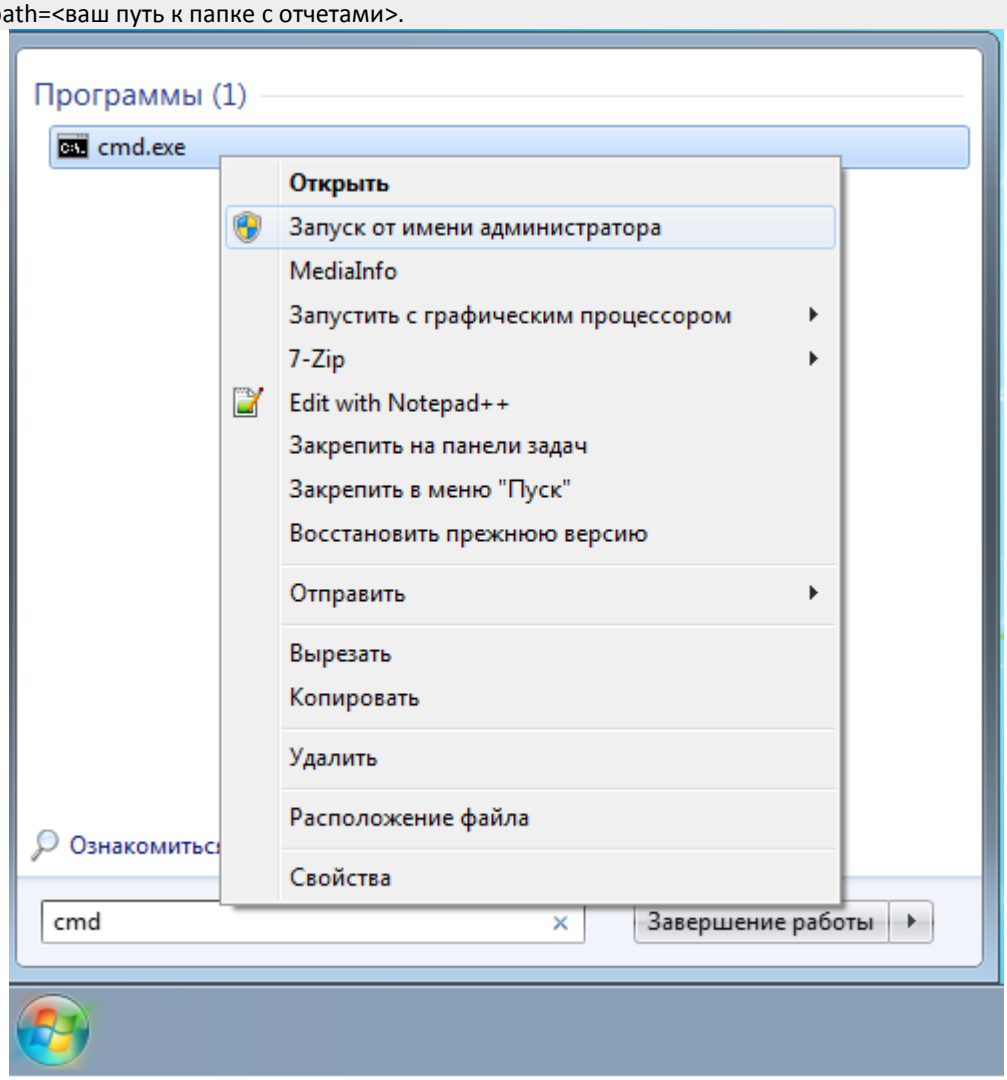


Рисунок 49 – Запуск командной строки от имени администратора

Подключение к MongoDB

Поле запуска сервера необходимо запустить клиент. Для запуска клиента вам необходимо запустить приложение «mongo.exe» через командную строку.

При запуске клиент автоматически подключится к серверу.

Как показано на рисунке 50, после подключения к серверу в окне клиента появится приглашение на ввод команды.

По умолчанию MongoDB создает базу данных с именем «test».

Для проверки успешности подключения, и работоспособности MongoDB, можно запросить имя базы данных, к которой подключен клиент, для этого необходимо ввести в консоль следующую команду:

```
db.getName()
```

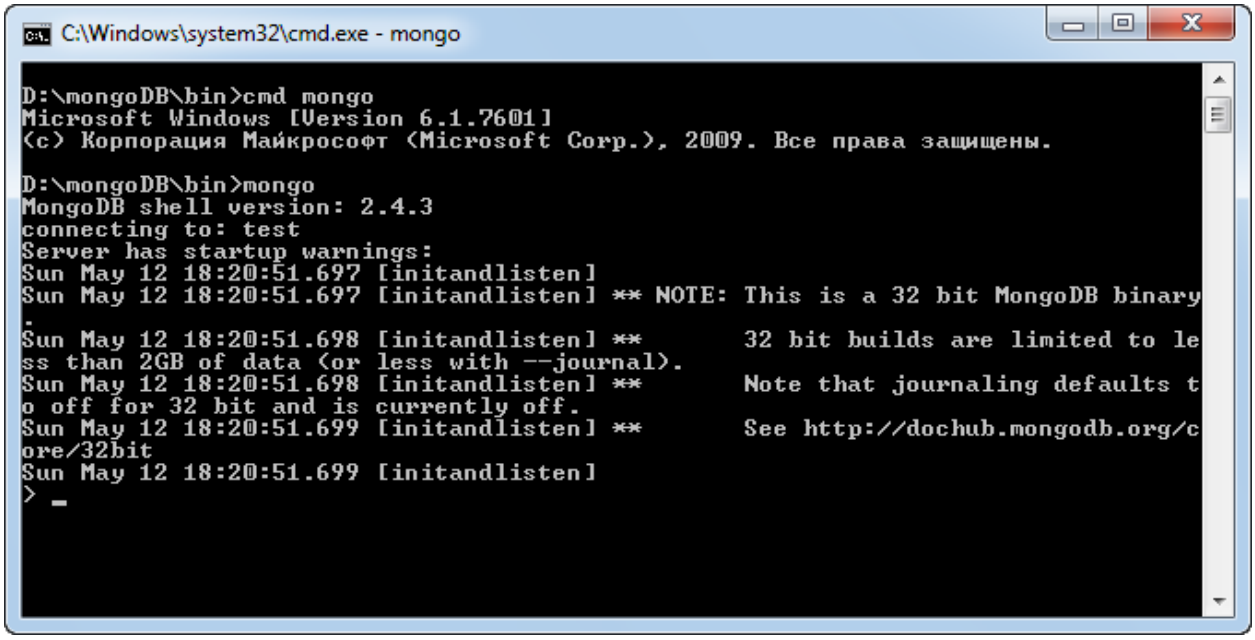


Рисунок 50 – Экранная форма окна приложения «mongo.exe»

Ответом на команду является имя базы данных, в данном случае «test».

Для добавления документа в базу данных необходимо ввести в консоль:

```
db.test.save( {name: "test parameter"})
```

Синтаксис команды следующий: «<объект бд>.<коллекция>.<функция>(<список параметров>)», где:

<объект бд> – Объекта базы данных («db»), в случае если команда не является глобальной;

<коллекция> – Имя коллекции;

<функция> – Имя функции;

<список параметров> – Список параметров функции.

В случае ошибки в команде клиент выдаст соответствующее предупреждение. В случае успешности сообщений выведено не будет.

Для получения всех документов в коллекции необходимо ввести в консоль следующую команду:

```
db.test.find()
```

Оболочка MongoDB Shell

Если следовать инстркуциям выше, на ваш компьютер будет установлен работоспособный экземпляр MongoDB. Убедитесь, что процесс mongod работает, а затем запустите оболочку MongoDB:

./mongo

Если все нормально, то на экране появится приглашение, как на рис. 50.

Если при запуске не указана база данных, то по умолчанию оболочка выбирает базу test. Но чтобы во всех последующих упражнениях оставаться в одном и том же пространстве имен, переключимся на базу данных MyDB.

Создадим. Будем работать с JavaScript-оболочкой, документы представляются в формате JSON (JavaScript Object Notation). Простейший документ, описывающий одного пользователя, мог бы выглядеть так:

{username: "Jones"}

Этот документ содержит одну пару ключ-значение для хранения имени пользователя Jones. Чтобы сохранить этот документ, нужно указать коллекцию. Коллекция users вполне подойдет:

> db.users.insert({username: "Jones"})

После ввода этого предложения может наблюдаться небольшая задержка. В это время на диске создается база данных tutorial и коллекция users. Задержка обусловлена созданием начальных файлов для того и другого.

Если вставка завершилась успешно, то ваш первый документ сохранен. Чтобы убедиться в этом, можно выполнить простой запрос:

> db.users.find()

Ответ будет выглядеть примерно так:

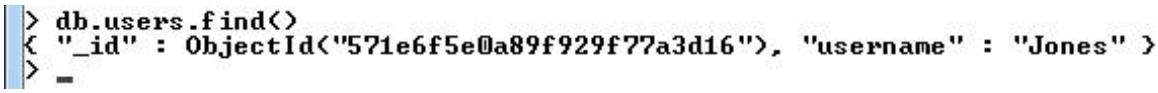


Рисунок 51 – Выполнение простого запроса

Обратите внимание, что в документ добавлено поле _id. Можете считать его значение первичным ключом документа. В любом документе MongoDB обязательно присутствует поле _id, и если в момент создания его значение не задано, то MongoDB генерирует специальный идентификатор объекта. На вашей консоли отобразится не тот же идентификатор, что в примере выше, но он гарантированно будет уникален среди всех значений _id в данной коллекции - единственное непререкаемое требование к этому полю.

Добавим нового пользователя:

> db.users.save({username: "Michel"})

Имея в коллекции более одного документа, мы можем попробовать более сложные запросы. Как и раньше, можно запросить все документы в коллекции:

> db.users.count()

2

Но можно также передать методу find простой селектор запроса. Селектором запроса называется документ, с которым сравниваются все

документы в коллекции. Чтобы найти все документы, в которых поле username равно jones, нужно задать такой селектор:

```
> db.users.find()
{ _id : ObjectId("4bf9bec50e32f82523389314"), username : "Jones" }
{ _id : ObjectId("4bf9bec90e32f82523389315"), username : "Michel" }
```

Селектор запроса {username: "jones"} возвращает все документы, в которых имя пользователя содержит строку jones - документобразец буквально сравнивается со всеми хранящимися в коллекции документами:

```
> db.users.find({username: "jones"})
{ _id : ObjectId("4bf9bec90e32f82523389315"), username : "Jones" }
```

Для обновления нужно задать по меньшей мере два аргумента. Первый определяет, какие документы обновлять, второй - как следует модифицировать отобранные документы. Существует два способа модификации; в этом разделе мы рассмотрим направленную модификацию (targeted modification) - одну из наиболее интересных и уникальных особенностей MongoDB.

Предположим, что пользователь smith решил указать свою страну проживания. Сделать это можно следующим образом:

```
> db.users.update({username: "Jones"}, {$set: {country: "Russia"}})
```

Здесь мы просим MongoDB найти документ, в котором поле username равно smith, и записать в свойство country значение Canada. Если теперь запросить этот документ, то мы увидим, что он обновился:

```
> db.users.find({username: "Jones"})
{"_id" : ObjectId("4bf9ec440e32f82523389316"),
"country" : "Russia", username : "Jones"}
```

Разовьем этот пример. Данные представляются в виде документов, которые, как мы видели в главе 1, могут содержать сложные структуры данных. Предположим, что, помимо профиля, пользователь желает хранить списки своих любимых вещей. Представление такого документа могло бы выглядеть так:

> db.users.update({username: "smith"}, {$unset: {country: 1}})

Ключ favorites указывает на объект, содержащий два других

ключа, указывающих на списки любимых городов и фильмов. Можете ли вы придумать, как с помощью того, что вам уже известно, модифицировать исходный документ о пользователе smith так, чтобы он принял такой вид? На ум должен сразу прийти оператор $set. Отметим, что в этом случае мы практически полностью переписываем документ, и $set ничего не имеет против: (Рисунок 52):

```
{
    username : "Jones",
    favorites : {
        cities : ["Moscow", "Stavropol"],
        movies : ["Crazy Cops", "The War", "12"]
    }
}
```

Рисунок 52 – Complex Document

Точка между favorites и movies означает, что нужно найти ключ favorites, который указывает на вложенный объект с ключом movies, а затем сравнить значение этого вложенного ключа с указанным в запросе. Этот запрос вернет оба документа.:

```
> db.users.update({username : "Jones"},{"$set": {favorites : {movies : ["12", "T
he War", "Crazy! Crazy!"]}}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

Рисунок 53 – Update complex document

Если не задавать никаких параметров, то операция удаления remove удалит из коллекции все документы. Так, чтобы избавиться от коллекции f oo, нужно выполнить такую команду::

> db.users.remove()

You often need to remove only a certain subset of a collection’s documents, and for that, you can pass a query selector to the remove() method. If you want to remove all users whose favorite city is Stavropol, the expression is pretty straightforward:

> db.users.remove({"favorites.cities": "Stavropol"})

Отметим, что операция remove () не уничтожает саму коллекцию, а лишь удаляет из нее документы. Можете считать ее аналогом команд SQL DELETE И TRUNCATE. Чтобы уничтожить коллекцию вместе со всеми построенными над ней индексами, используйте метод drop ():

> db.users.drop()

Создание, чтение, обновление и удаление - основные операции в любой базе данных. Если вы читали внимательно, то теперь можете самостоятельно поэкспериментировать с операциями CRUD в MongoDB. В следующем разделе мы продолжим изучение операций выборки, обновления и удаления, познакомившись с вторичными индексами.

Создание индексов и применение их в запросах

Индексы обычно создаются для повышения скорости выполнения запросов. К счастью, оболочка MongoDB позволяет создавать индексы безо всякого труда. Если ранее вам не доводилось работать с индексами базы данных, то из этого раздела вам станет ясно, зачем они нужны. А если у вас уже есть опыт работы с индексами, то вы узнаете, . как просто они создаются в MongoDB и как можно профилировать выполнение запросов с помощью метода explain().

Индексировать коллекцию имеет смысл, только когда в ней много документов. Поэтому добавим 200 000 простых документов в коллекцию numbers. Поскольку оболочка MongoDB одновременно является интерпретатором JavaScript, то сделать это несложно: (Рисунок 7.8).

```
> for (i=0; i<200000; i++){db.numbers.save({num:i})}
WriteResult({ "nInserted" : 1 })
> db.numbers.count()
200000
>
```

Рисунок 53– Создание коллекции документов

200 000 документов - это немало, так что не удивляйтесь, если на выполнение команды уйдет несколько секунд. По завершении вы можете с помощью парочки запросов убедиться, что все документы на месте::

```
> db.numbers.find()
{ "_id" : ObjectId("571e773c0a89f929f77a3d17"), "num" : 0 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d18"), "num" : 1 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d19"), "num" : 2 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d1a"), "num" : 3 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d1b"), "num" : 4 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d1c"), "num" : 5 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d1d"), "num" : 6 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d1e"), "num" : 7 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d1f"), "num" : 8 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d20"), "num" : 9 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d21"), "num" : 10 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d22"), "num" : 11 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d23"), "num" : 12 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d24"), "num" : 13 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d25"), "num" : 14 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d26"), "num" : 15 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d27"), "num" : 16 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d28"), "num" : 17 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d29"), "num" : 18 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d2a"), "num" : 19 }
Type "it" for more
>
```

Рисунок 54 – Выполнение выборки

Команда count () показывает, что вставлено 200 000 документов. Второй запрос выводит первые 20 результатов. Чтобы показать следующую порцию, выполните команду it:

```
Type "it" for more
> it
{ "_id" : ObjectId("571e773c0a89f929f77a3d2b"), "num" : 20 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d2c"), "num" : 21 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d2d"), "num" : 22 }
{ "_id" : ObjectId("571e773c0a89f929f77a3d2e"), "num" : 23 }
```

Рисунок 55 – Запрос подмножества

Команда it просит оболочку вернуть следующий результирующий набор. Имея набор документов приличного размера, попробуем выполнить несколько запросов. Вас уже не удивит, что для поиска документа по его атрибуту num достаточно такого простого запроса:

```
> db.numbers.find({num : 500})
{ "_id" : ObjectId("571e773c0a89f929f77a3f0b"), "num" : 500 }
```

Более интересны запросы по диапазону, для которых предназначены специальные операторы $gt и $lt. Мы уже встречались с ними в главе 1 (gt означает greater than [больше], a it - less than [меньше]). Вот как запросить документы, для которых значение num больше 199 995:

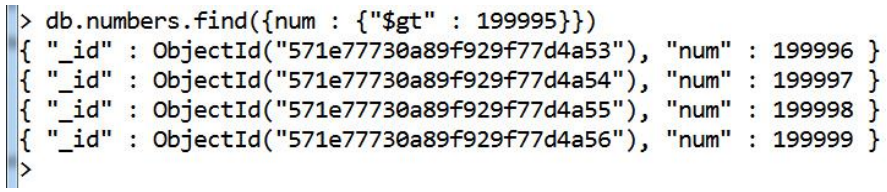


Рисунок 56 – Использование оператора $gt

Как видите, с помощью простого JSON-документа можно сформулировать сложный запрос по диапазону, как на языке SQL. $gt и $lt - всего два из многочисленных ключевых слов, используемых в языке запросов MongoDB; в последующих главах мы встретим много других примеров.

Разумеется, от таких запросов мало толку, если они выполняются неэффективно. В следующем разделе мы впервые задумаемся об эффективности и начнем изучать имеющиеся в MongoDB средства индексирования.

Если вы долго работали с реляционными базами данных, то, наверное, знакомы с командой SQL explain. Она описывает путь выполнения запроса и позволяет выявить медленные операции, показывая, какие индексы были использованы. В MongoDB тоже имеется вариант explain с аналогичной функциональностью. Чтобы понять, как эта команда работает, применим ее к одному из предыдущих запросов. Выполните такую команду::

> db.numbers.find({num: {"$gt": 199995 }}).explain()

Результат должен выглядеть примерно так, как показано в листинге ниже.

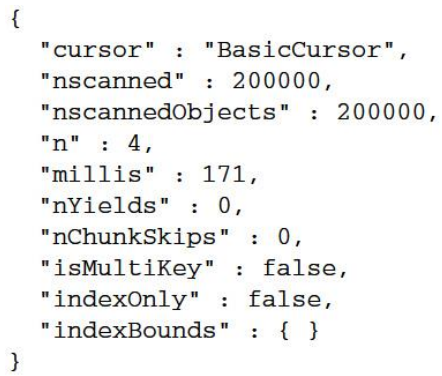


Рисунок 57 – Результат команды explain()

Изучение распечатки, выданной explain () показывает, что для

возврата всего четырех результатов (n) серверу пришлось просканировать всю коллекцию из 200 000 (nscanned) документов. Тип курсора BasicCursor подтверждает, что для формирования результирующего набора индексы не использовались. Столь большая разница между количеством просмотренных и возвращенных документов свидетельствует о неэффективности выполнения запроса. В реальном приложении, когда и коллекция больше, и сами документы объемнее, время выполнения запроса окажется существенно больше 171 миллисекунды, как в данном примере.

Этой коллекции явно недостает индекса. Построить индекс по ключу num можно с помощью метода ensure index (). Введите такую команду:

> db.numbers.ensureIndex({num: 1})

Как и для любой другой операции MongoDB, например выборки или обновления, методу ensurelndex () передается документ, определяющий, по каким ключам следует индексировать. В данном случае документ {num: 1} говорит, что над коллекцией numbers нужно построить индекс по ключу num в порядке возрастания. Убедиться в том, что индекс действительно построен, позволит метод get Indexes ():

```
> db.numbers.getIndexes()
[
  {
    "name" : "_id_",
    "ns" : "tutorial.numbers",
    "key" : {
      "_id" : 1
    }
  },
  {
    "_id" : ObjectId("4bfc646b2f95a56b5581efd3"),
    "ns" : "tutorial.numbers",
    "key" : {
      "num" : 1
    },
    "name" : "num_1"
  }
]
```

Рисунок 58 – Результат команды getIndexes()

Теперь над этой коллекцией построено два индекса. Первый - по стандартному ключу _id - автоматически строится для любой коллекции;

второй - по ключу num - мы только что создали сами. Если сейчас выполнить тот же запрос с помощью метода explain (), то будет заметна ощутимая разница во времени выполнения, что отражено в листинге ниже.

```
> db.numbers.find({num: {"$gt": 199995 }}).explain()
{
  "cursor" : "BtreeCursor num_1",
  "indexBounds" : [
    [
      {
        "num" : 199995
      },
      {
        "num" : 1.7976931348623157e+308
      }
    ]
  ],
  "nscanned" : 5,
  "nscannedObjects" : 4,
  "n" : 4,
  "millis" : 0
}
```

Рисунок 59 – Результат команды explain()

Теперь, когда используется индекс по ключу num, запрос просматривает только пять документов. В результате общее время выполнения сократилось со 171 мс до менее чем 1 мс.

Лабораторные задания

1 Установить и настроить MongoDB.

2 Получить у преподавателя индивидуальное задание по обработке данных в MongoDB

3 Оформить отчёт.