Запуск

1. Устанавливаем Docker и запускаем консоль (под Linux для запуска следующих комманд нужны права суперпользователя)
2. Скачиваем контейнер с bigtop sandbox

```
docker pull bigtop/sandbox:1.2.1-ubuntu-16.04-
    hdfs_yarn_hive_pig
```

3. Запускаем контейнер. Лучше это сделать из директории, в которой содержатся материалы курса.

```
docker run -d --hostname=quickstart \
    --name=hadoop-sandbox \
    --privileged=true \
    -p 50070:50070 \
    -p 14000:14000 \
    -p 8088:8088 \
    -p 19888:19888 \
    -p 9083:9083 \
    -p 10000:10000 \
    -p 10002:10002  \
    -v `pwd`:/course \
    bigtop/sandbox:1.2.1-ubuntu-16.04-hdfs_yarn_hive_pig
```

4. Ждем некоторое время, когда контейнер запустится, затем в консоли можно выполнить команду

```
docker exec -it hadoop-sandbox bash
```

5. В консоле внутри контейнера

```
apt update -y & apt install -y python3 python3-pip hadoop
    -httpfs
service hadoop-httpfs start
python3 -m pip install mrjob
```

Использование песочницы

Контейнер можно остановить

```
docker stop  hadoop-sandbox
```

и потом опять запустить

```
docker start  hadoop-sandbox
```

Если потребуется пробросить дополнительный порт или раздел, то можно сделать так - создать образ из существующего контейнера, удалить контейнер и запустить новый, указав дополнительные параметры:

```
docker stop hadoop-sandbox
docker commit hadoop-sandbox hadoop-image
docker rm hadoop-sandbox
docker run -d --hostname=quickstart \
    --name=hadoop-sandbox \
    --privileged=true \
    -p 50070:50070 \
    -p 14000:14000 \
    -p 8088:8088 \
    -p 19888:19888 \
    -p 9083:9083 \
    -p 10000:10000 \
    -p 10002:10002  \
    -v `pwd`:/course \
    hadoop-image
```

После запуска контейнера на хост-системе можно проверить доступность сервисов:

- http://localhost:50070 - HDFS WEB UI
- http://localhost:14000 - HttpFS REST API

- http://localhost:8088 - Resource Manager
- http://localhost:19888 - Hadoop History Server

Apache Hadoop

Hadoop [1] - это фреймворк который позволяет распределенно обрабатывать большие объемы данных на кластерах, используя простую программную модель, которая называется MapReduce.

Hadoop начал создаваться как проект с открытым исходным кодом в 2005 году, под влиянием технологии компании Google, которая называлась MapReduce. Первоначально Hadoop предполагалось использовать для индексации документов в поисковой системе **Nutch** [5]. Затем проект перешел под эгиду *Apache Software Foundation*, развивался компанией *Yahoo* и другими компаниями, которые предлагали свои коммерческие дистрибутивы. Пик популярности пришелся на 2010-е годы, затем стал стремительно уменьшаться, что связано с ростом популярности платформ для потоковых вычислений.

На определенном этапе эволюции проект разделился на независимые части: `YARN` (система управление вычислительными ресурсами общего назначения), `HDFS` (распределенная файловая система), непосредственно подсистема для запуска `MapReduce` задач. В настоящее время наблюдается тенденция на переход с `HDFS` на `Apache Ozone` - систему хранения следующего поколения.

HDFS

`Hadoop Distributed File System`[9] - распределенная файловая система, которая развивается в рамках проекта `Hadoop`. С точки зрения пользователя представляет собой классическую иерархическую файловую систему, с файлами, директориями, правами доступа. Но файлы хранятся распределенно, реплицируются, система устойчива к отказам оборудования. Естественно, `HDFS` имеет свои ограничения, которые отличают её от классических локальных файловых систем, а дизайн изначально был рассчитан на использование в рамках обработки данных с помощью MapReduce. Система масштабируема, запускалась на кластере ~ 4000 машин. `HDFS` необязательно использовать вместе с `MapReduce`, система может использоваться независимо, например вместе с `Apache Spark` [7].

Основные особенности

- устойчива к падениям
- масштабируема
- безопасность (права доступа и т.д.)
- написана на Java
- концепция основана на Google GFS
- на узлах кластера используются нативные файловые системы

На кластер устанавливаются сервисы двух видов (кластеры обычно управляются YARN [2] или Kubernetes [10]):

1. `NameNode` - хранение метаинформации
    - дополнительная реплика сервис: `Secondary NameNode`
    - резервный узел `Standby NameNode`
2. `DataNode` - хранение данных, рабочие машины

- чем больше `DataNode`, тем больше информации можно хранить на кластере
- чем больше `DataNode`, тем больше производительность

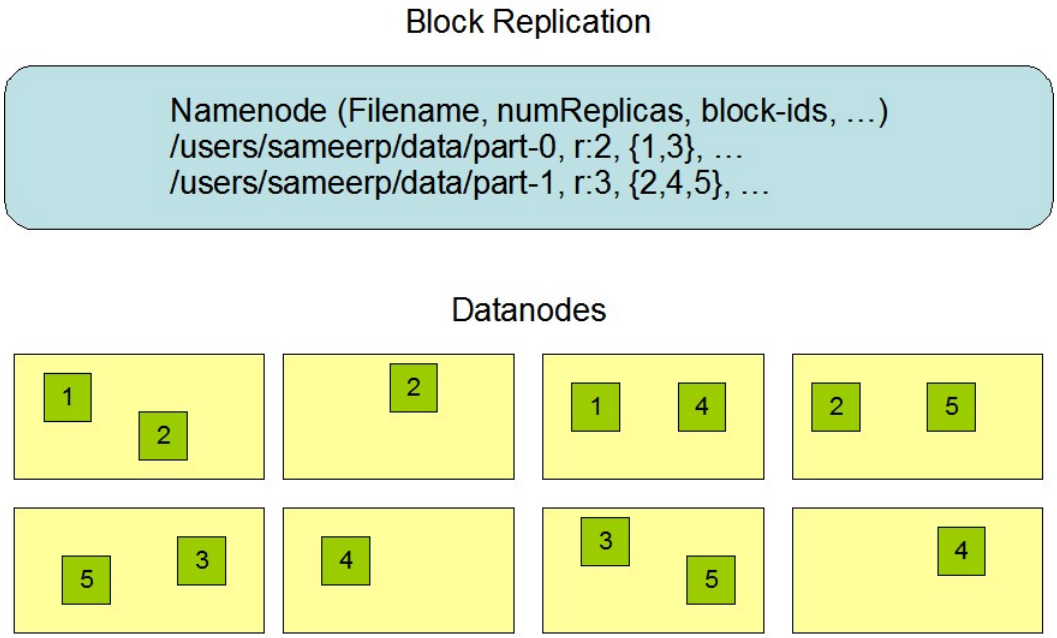

Рис. 2: HDFS Data Nodes, источник [1]

Как говорилось выше, логически HDFS представляет из себя привычную иерархическую файловую систему. При этом можно выделить следующие особенности:

- файлы делятся на блоки (по умолчанию 128Мб)
- каждый кусок хранится на какой-то `DataNode`.
- каждый блок файла копируется по узлам сети (по умолчанию на 3 узла)
- `NameNode` отслеживает каждый состояние узлов
- если узел стал недоступен, то блок автоматически до-реплицируется

- `DataNode` умеют общаться друг с другом
- `HDFS Federation` позволяет использовать множество `NameNode` для горизонтального масштабирования
- есть система контроля доступа и авторизации

В целом процесс получения/записи данных выглядит так:

1. Клиент обращается к `NameNode`
2. Клиент перенаправляется к соответствующему `DataNone` для чтения/записи данных.

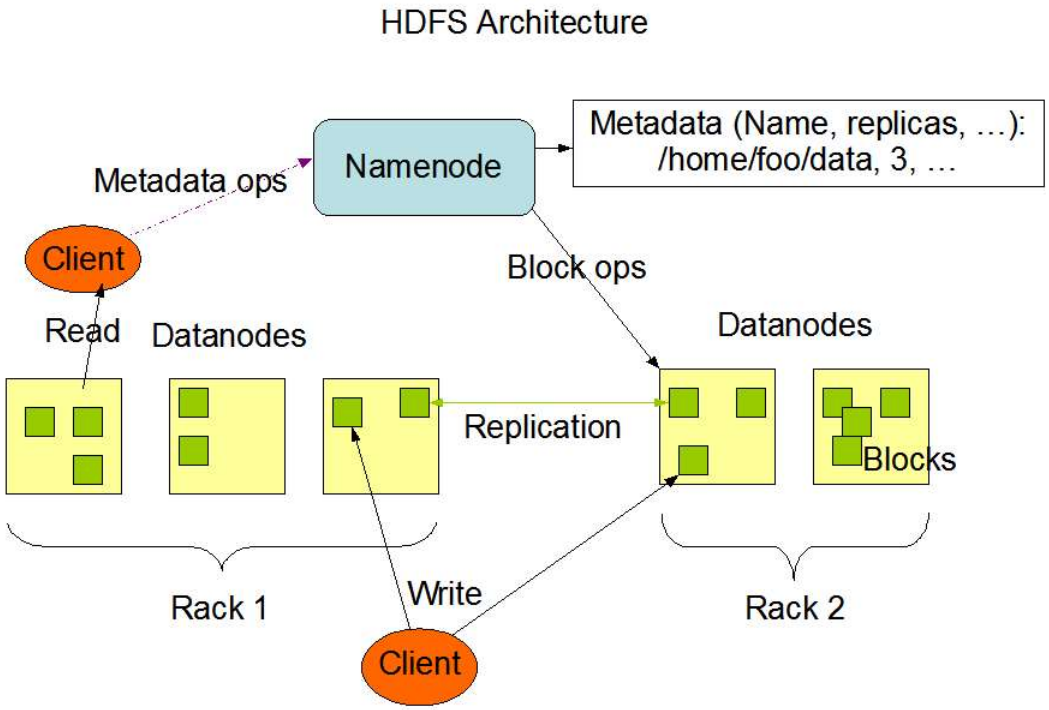

Рис. 3: HDFS Architecture, источник [1]

Данные реплицируются, `NameNode` постоянно общается с `DataNode` и

отслеживает их состояние, перераспределяет данные. `DataNode` могут обмениваться данными напрямую.

Недостатки: - `HDFS` неэффективно работает с большим числом маленьких файлов (размер блока 128Мб) - произвольный доступ к файлам ограничен - файлы неизменяемы (но можно добавить содержимое к файлу)

Командная строка

С файлами `HDFS` можно работать с помощью командной строки, WEB UI или с помощью API для различных языков программирования. Допустим у нас есть некоторый файл **weather.csv**. Проделаем манипуляции с помощью нашей песочницы:

1. Запустим `bash` в контексте песочницы:

```
docker exec -it hadoop-sandbox bash
```

2. Посмотрим на содержимое “домашней” директории:

```
hadoop fs -ls /user/root
```

3. Скопируем файл из локальной файловой системы на `HDFS`

```
hadoop fs -put weather.csv /user/root
```

4. Проверим наличие файла

```
hadoop fs -ls /user/root
```

5. Посмотрим содержимое текстового файла

```
hadoop fs –cat /user/root/weather.csv
```

6. Скопируем файл из HDFS на локальную файловую систему

```
hadoop fs –get /user/root/weather.csv weather2.csv
```

7. Удалим файл на HDFS

```
hadoop fs –rm /user/root/weather.csv
```

Python API

HDFS представляет REST HTTP API, для которого есть клиенты, например, из Python [13]

Запишем данные в файл:

```
import hdfs

# соединение с HDFS
fs = hdfs.InsecureClient("http://localhost:14000",
    user="root")

with fs.write("lines.txt", encoding="utf–8") as w:
    for i in range(200):
        w.write(str(i))
        w.write('\n')
```

Удалим файл:

```
fs.delete("lines.txt")
```

Загрузим файл из локальной файловой системы на HDFS

```
fs.upload("weather.csv", "data/weather.csv")
```

Прочитаем файл

```
with fs.read("weather.csv") as reader:
  content = reader.read()
```

YARN

YARN [2] - Yet Another Resource Negotiator.

Система управления ресурсами, составная часть Apache Hadoop начиная с версии 2.0. Мотивация для создания YARN очевидна: есть большой кластер и нужно запускать на нём какие-то задачи, выделяя каждой задаче необходимые ресурсы (ограничивая и квотируя), наблюдать за состоянием задач и оборудования.

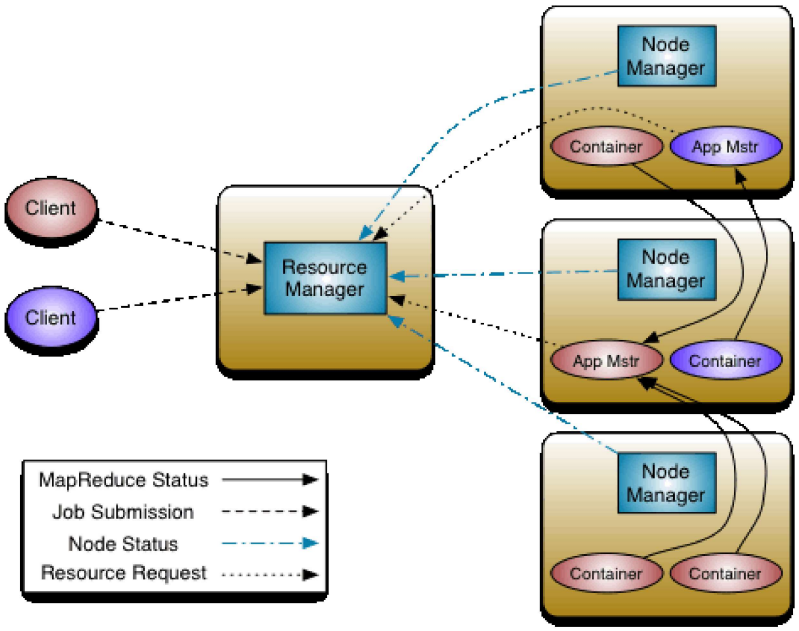

Рис. 4: Архитектура YARN, источник [2]

Кластер YARN состоит из следующих типов машин:

- Resource Manager - управляет ресурсами кластера. Кластер имеет только один основной Resource Manager
- Node Manager - узлы, которые отвечают за запуск задач и приложений.

Основные особенности:

- YARN написан на Java, приложения, которые взаимодействуют с YARN, нативно реализуются на Java
- YARN может взаимодействовать с HDFS, агенты YARN работают на тех же физических серверах, что и DataNode HDFS. Приложение, которое запускает задачу на YARN, может указать, что задача должна выполняться на машине, где хранится определенный блок HDFS
- YARN может использоваться не только для запуска Hadoop, а также для управления любыми распределенными приложениями
- YARN - это универсальная система управления ресурсами. На нем может запускаться, например, Spark[7]

Кластер может иметь несколько тысяч Node Manager. Node Manager запускают задачи и приложения, следят за ресурсами ресурсы, шлют отчеты для Resource Manager.

Схема взаимодействия компонентов:

1. Клиент инициирует запуск приложения и обращается к Resource Manager

2. `Resource Manager` на одном из кластера выделяет необходимые ресурсы для запуска контейнера с приложением. Это приложение называется `Application Master`.
3. `Application Master` запускается и дальше через `Resource Manager` может слать запросы на выделение ресурсов на остальных узлах кластера (с определенным количеством доступной памяти, числа ядер и так далее).

Идея в том, чтобы отделить две логических компоненты в системе распределенной обработки данных: `Resource Manager` - глобальные ресурсы, `Application Master` - ресурсы приложения.

В `YARN` используется концепция контейнера:

- контейнер - это набор ресурсов, который выделяется со стороны `Resource Manager` по запросу
- контейнер выделяет права приложению для использования ограниченного числа ресурсов
- внутри контейнера можно запускать любое приложение
- для запуска контейнеров `Node Manager` используют специфичные для операционной системы средства. На Linux может использоваться Docker.
- ресурсов на всех может не хватить - используется планировщик, запросы ставятся в очередь по приоритету
- логика планировщика может настраиваться
- `Node Manager`, `Resource Manager` имеют Web UI, с помощью которого можно просматривать логи, следить за задачами и т.п.

Если кластер работает в большой компании, но часто возникает проблема совместного использования ресурсов. Для этого применяются планировщики, которые могут кастомизироваться в зависимости от конкретных нужд. Основные планировщики:

FIFO scheduler

- задачи выполняются в порядке очереди
- нет понятия приоритета
- не рекомендовано к использованию

Fair scheduler

- был разработан компанией *Facebook*
- пытается распределить ресурсы таким образом, чтобы все запущенные приложения получили одинаковую долю ресурсов
- есть набор пулов, каждая задача распределяется в определенный пул
- каждому пулу могут принадлежать определенное число ресурсов
- каждый пользователь прикрепляется к одному пулу, он может использовать ресурсы только из этого пула
- позволяет администратору настраивать доступное число ресурсов

Capacity scheduler

- был разработан компанией *Yahoo*
- предназначается для больших кластеров
- позволяет настраивать число ресурсов для каждого пользователя
- используется очередь задач с приоритетами

Упражнение

Запустите песочницу и пройдите по ссылке http://localhost:8088 - это Web UI Resource Manager. На главной странице список приложений, для

каждого приложения можно посмотреть логи, его статус, тип, ресурсы. Исследуйте возможности этого интерфейса. Ответьте на вопрос: достаточно ли этого для администрирования большого кластера YARN?

MapReduce

Парадигма вычислений

Это модель вычислений и тип организации фреймворка вычислений, который был популяризован компанией *Google* в начале 2000-х годов. Название произошло от двух функций, известных в функциональном программировании. На Python в функциональном стиле (неэффективно) их можно реализовать так:

```
def our_map(
    lst: List[T],
    func: Callable[[T], R]
) -> List[R]:

    if len(lst) == 0:
        return []
    return [func(x)] + our_map(lst[1:], func)

def our_reduce(
    lst: List[T],
    x0: R,
    func: Callable[[T, R], R]
) -> R:
    if len(lst) == 0:
        return x0

    return func(
        lst[0],
        our_reduce(lst[1:], x0, func)
    )
```

Смысл функции `map`: задан упорядоченный список, каждый элемент которого независимо преобразуется определенным образом.

Смысл функции `reduce`: свернуть список к одному значению. Заметим, что вычисление функции `map` можно легко распараллелить, разделив входной список на независимые куски и посчитав их отдельно.

Логически систему `MapReduce` можно представить так:

- Входные данные
    1. Список гомогенных пар ключ/значение, произвольного типа
- Параметры алгоритма
    1. Функция `map`, которая преобразует одну пару ключ/значение в набор пар ключ/значение.
    2. Функция `reduce`, которая преобразующая список значений, имеющих одинаковый ключ, в набор пар ключ/значение.
- Результат
    1. Пары ключ/значения, собранные в результате работы функций `reduce`.

Легко видеть, что функции `map` и `reduce` могут быть выполнены параллельно для своих кусков данных. Теперь можно обобщить основную логическую сущность движка `MapReduce`: пользователь предоставляет источник данных (пары ключ/значение) и две функции. Движок сам решает как оптимальным образом выполнить вычисления.

Прививем “игрушечную” имплементацию движка на языке Python:

`run_mapper` принимает на вход данные и функцию. Затем происходит итерирование по каждой паре входных данных, на выходе получается итератор, который генерирует выходные пары:

```
def run_mapper(
    func: Callable[
            [K1, V1],
            Iterable[Tuple[K2, V2]
        ],
    data: List[Tuple[K1, V1]]
) -> Iterable[Tuple[K2, V2]]:

    for k1, v1 in data:
        yield from func(k1, v1)
```

`run_reducer` выполняет второй шаг, когда данные сгруппированы по ключу:

```
def run_reducer(
    func: Callable[
            [K2, List[V2]],
            Iterable[Tuple[K3, V3]
        ],
    data: Iterable[Tuple[K2, List[V2]]]
) -> Iterable[Tuple[K3, V3]]:

    for k2, lst_v2 in data:
        yield from func(k2, lst_v2)
```

совмещаем все в месте получаем:

```
def run_engine(
    map_func: Callable[
                [K1, V1],
                Iterable[Tuple[K2, V2]
              ],
    reduce_func: Callable[
                   [K2, List[V2]],
                   Iterable[Tuple[K3, V3]]
                 ],
    data: List[Tuple[K1, V1]]
) -> Iterable[Tuple[K3, V3]]:

    after_mapper = defaultdict(list)
    for k_2, v_2 in self._mapper(map_func, data):
        after_mapper[k_2].append(v_2)

    to_reducer = sorted(
        after_mapper.items(),
        key=lambda x: x[0]
    )

    after_reducer = sorted(
        self._reducer(reduce_func, to_reducer),
        key=lambda x: x[0]
    )

    return after_reducer
```

Этот игрушечный движок можно применить в классическом примере `word_count`. Допустим у нас имеется три текста, мы хотим посчитать частотность слов. В данном примере на входе пары виду > (text_id: int, text: str)

```
texts = [(1, 'The Scheduler is responsible....'),
         (2, 'YARN supports the notion of ...'),
         (3, 'The timeline readers are separate...')
]
```

Реализуем `map` и `reduce`:

```python
def map_func(
    k: int,
    v: str
) -> Iterable[Tuple[str, int]]:

    # разделим текст на слова
    words = [x for x in v.lower().split() if x.
        isalpha()]
    # для каждого слова сгенерируем пару
    for word in words:
        yield (word, 1)

def reduce_func(
    k: str,
    v: Iterable[int]
) -> Iterable[Tuple[str, int]]:

    yield k, sum(v)
```

Теперь можно запустить:

```python
run_engine(
    map_func,
    reduce_func,
    texts
)
```

В результате получим список:

```
[('a', 2),
 ('allocating', 1),
 ('allows', 1),
 ('and', 2),
 ('applications', 1),
 ('are', 2),
 ('component', 1),
 ('constraints', 2),
 ('daemons', 1)
  ...
]
```

По сути, Hadoop имплементируют тот же самый движок (на основе YARN), но:

1. В качестве входных данных выступает файл или файлы на HDFS.
2. Результат записывается на HDFS.
3. Шаги выполняются параллельно на кластере.
4. Есть возможность задавать различные форматы входных и выходных файлов.
5. Логику работы функций сортировки можно менять.

Библиотека MRjob

Изначально алгоритмы для Hadoop писались на языке Java, в настоящий момент она почти не используется из-за сложностей и избыточности, хотя формально это самый оптимальный способ. Вместо этого применяются специализированные системы (Apache Hive [3]) или используется механизм Hadoop Streaming, который позволяет имплементировать функции map и reduce на произвольном языке. Существует удобная библиотека mrjob [11] для языка Python, с помощью которой можно запускать Hadoop-задачи без особых усилий. Отметим, что mrjob может запускать задачи не только на Hadoop, но и на Apache Spark [7].

Приведем пример реализации классического алгоритма word—count с помощью mrjob. В основном нужно наследоваться от класса MRJob и переопределить некоторые методы.

```
# файл job.py

from mrjob.job import MRJob
from mrjob.protocol import TextProtocol
import re

WORD_RE = re.compile(r"[\w']+")


class MRWordFreqCount(MRJob):
    OUTPUT_PROTOCOL = TextProtocol

    # тут имплементируется функция map
    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            word = word.lower()
            yield (word, 1)

    def combiner(self, word, counts):
        yield word, sum(counts)

    # тут имплементируется функция reduce
    def reducer(self, word, counts):
        yield word, str(sum(counts))

if __name__ == '__main__':
    MRWordFreqCount.run()
```

Допустим у нас есть некоторый текстовый файл `texts.txt`. Мы можем запустить нашу программу локально, без всякого кластера Hadoop:

```
python3 job.py < texts.txt
```

Для того, чтобы запустить в песочнице на кластере Hadoop: 1. Запустите bash в контексте контейнера

```
docker exec –it hadoop–sandbox bash
```

2. Скопируйте файл на HDFS

```
hadoop fs –put texts.txt /user/root/texts.txt
```

3. Запустите `job.py` с нужным

```
python3 job.py –r hadoop  hdfs:///user/root/weather.csv –
    o hdfs:///user/root/result
```

4. Посмотрите на результат

```
hadoop fs –cat "hdfs:///user/root/result/*"
```

Hadoop Streaming

На самом деле `MRJob` использует механизм, который называется `Hadoop Streaming`. С помощью его можно писать MapReduce-задачи на любом языке. Приведем пример с использованием Python:

Напишем скрипт, который реализует функцию `map`:

```
# файл mapper.py
import re
import sys

WORD_RE = re.compile(r"[\w']+")

for line in sys.stdin:
    line = line.strip()
    words = WORD_RE.findall(line):

    for word in words:
        print("{}\t{}".format(word, 1))
```

Теперь скрипт, который реализует функцию `reduce`:

```python
# файл reducer.py
import sys

cur_word = None
cur_count = 0
word = None

for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)

    count = int(count)

    if cur_word == word:
        cur_count += count
    else:
        if len(current_word) >= 1:
            print("{}\t{}".format(
                cur_word, cur_count
            )
        cur_count = count
        cur_word = word

if current_word == word:
    print("{}\t{}".format(cur_word, cur_count)
```

Для того, чтобы запустить это в песочнице, нужно выполнить команду:

```
yarn jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar\
 -input texts.txt\
 -output stat\
 -file mapper.py\
 -file reducer.py\
 -mapper "python mapper.py"\
 -reducer "python reducer.py"
```

Заметим, что таким образом можно использовать любой скриптовый язык, но производительность и удобство в таком случае оставляет же-

лать лучшего.

Основные концепции Hadoop MapReduce

Итак, перейдем к более низкому уровню. Apache Hadoop реализует парадигму MapReduce, позволяя параллельно обрабатывать огромное количество данных на кластере, по возможности минимизируя сетевые операции и скрывая особенности имплементации от программиста. При этом:

- достигается автоматическая распараллеливание и распределение кода/данных по кластеру
- предлагаются утилиты мониторинга выполнения задачи
- разработчик просто должен реализовать две функции: `map` и `reduce`
- `Data Locality` - данные по возможности обрабатываются на том узле кластера, где они хранятся
- входные и выходные данные хранятся на `HDFS`
- выполнение задачи устойчиво к отказам оборудования

Стадии выполнения Job

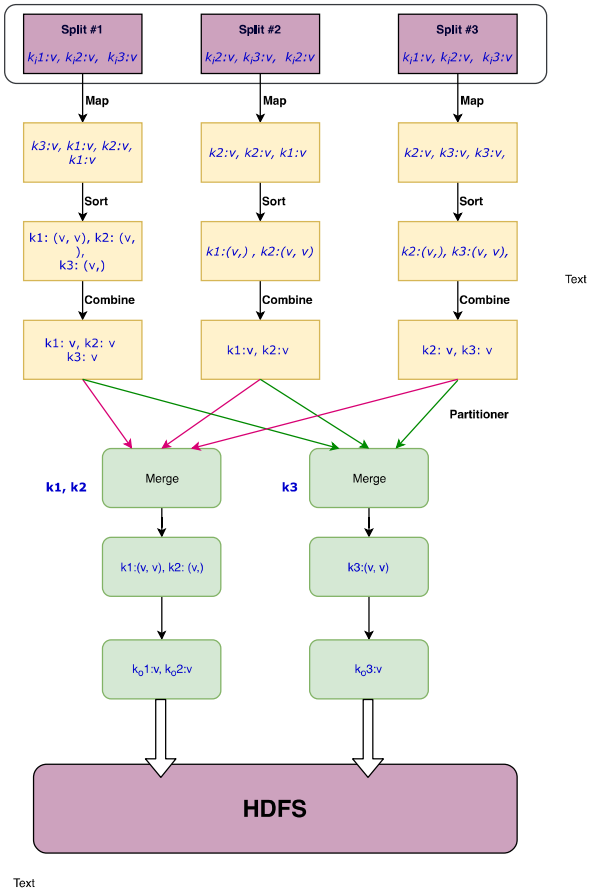

Рис. 5: Схема MapReduce

Дадим основным определения:

Job - процесс выполнения конкретной задачи на конкретном наборе данных, с конкретными функциями `map` и `reduce`.

Task - запуск какой-то подзадачи (`Mapper` - выполняет функцию `map` или `Reducer` - выполняет функцию `reduce`) на каком-то определенном куске данных

Task Attempt - попытка запуска **Task**. Hadoop может запускать одну задачу на выполнение несколько раз. Это происходит если предыдущая попытка не увенчалась успехом или медленно работает (speculative execution).

InputFormat - то, как должны интерпретироваться данные во входном файле на `HDFS`, перед тем, как они попадут в Mapper, которому нужны пары ключ-значения. По умолчанию используется `TextInputFormat`, который интерпретирует входной файл как текстовый файл. Это актуально при написании программы на Java. Помимо всего прочего, `InputFormat` определяет, как входные данные делятся на независимые куски

OutputFormat - то, как пары ключ-значения после работы Reducer'а должны записываться на `HDFS` (на самом деле не обязательно на `HDFS`). По умолчанию используется `TextOutputFormat`, который записывает пары ключ-значения в текстовом виде.

Combiner - опциональная функция, похожая на `Reducer`. Чаще всего применяется для оптимизации вычислений, когда функция `reduce` коммутативна ($a \circ b = b \circ a$) и ассоциативна ($(a \circ b) \circ c = a \circ (b \circ c)$). Идея в том, что мы можем применить операцию `reduce` на части данных на тех же узлах, на которых запускались `Mapper`ы, когда они ещё не сгруппированы. Если это возможно, то это ускорит вычисления.

Partitioner - опциональная функция, которая делит пространство

ключей. Данная функция выполняется на каждом ключе после выполнения `Mapper`'а, от результата зависит в какой `Reducer` (из количество известно) попадут данные, ассоциированные с ключом. По умолчанию используется хэш-функция. Иногда нужно задать пользовательский `Partitioner`, это может быть связано с особенностями алгоритмов или для повышения производительности, чтобы перераспределить нагрузку на узлы, которые выполняют функции `reduce`.

Secondary Sort - MapReduce сортирует записи по ключам перед тем, как они попадут в `Reducer`'ы. Но список значений для конкретного ключа не отсортирован, иногда их нужно получить в отсортированном виде. Этот процесс опционален и называется вторичной сортировкой, его можно имплементировать самостоятельно, используя композитный ключ.

Жизненные цикл `Job` выглядит примерно так, при этом опускаются технические детали, связанные с взаимодействием с `YARN`. Число параллельных процессов, выполняющих `map` и `reduce` определяются пользователем или автоматически до выполнения задания.

1. Hadoop анализирует выходные данные, делит их на части - `splits`.
2. Hadoop определяет на каких машинах будет выполняться маппперы (`Mapper`) и переносит на них код и данные.
3. Hadoop выполняет мапперы (`Mapper`) на каждом сплите. В идеальном случае это должно происходить одновременно.
4. После выполнения `Mapper` происходит группировка и сортировка данных по ключу.
5. Опционально, если он задан пользователем, выполняется `Combiner` (подробности ниже).
6. Происходит процесс, который называется `shuffling`. В этом мо-

мент данные перераспределяются по узлам, которые участвуют в выполнении `Job`. Все данные с одинаковым ключом должны попасть на одну машину. В этот момент происходят тяжелые сетевые операции.
7. После копирования данных они группируются по ключу и запускается `Reducer`.
8. Результат записывается на `HDFS`, при этом, по возможности, сетевые операции сведены к минимуму. Используется та `DataNode`, на которой выполняется процесс `Reducer`'а. В дальнейшем `HDFS` автоматически достигает необходимой степени репликации.

Счетчики

Счетчики (Counters) предоставляют возможность агрегировать какие-то численные сообщения в `Task` и получать конечные значения, когда `Job` завершится. Их значения могут быть получены с помощью API или даже Web UI. Каждый счетчик определяется группой (если необходима группировка) и уникальным именем. Важно заметить, что во время выполнения задачи (`Job`), значения счетчиков могут принимать некорректные значения из-за возможного спекулятивного выполнения.

В библиотеке `mrjob` реализован простой механизм для обращения к счетчикам:

```
class MRWordFreqCount(MRJob):
    OUTPUT_PROTOCOL = TextProtocol

    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            word = word.lower()
            self.increment_counter("letters_stat", "
                first_letter_" + word[0], 1)
            yield (word.lower(), 1)

    def reducer(self, word, counts):
        yield word, str(sum(counts))
```

в этом классическом примере, помимо статистики слов, с помощью счетчиков мы считаем количество встреченных слов в зависимости от их первой буквы. Создастся максимум 26 счетчиков в группе «letter_stat», счетчик с количеством слов на букву «z» будет называться «first_letter_z»

Распределенный кэш

Очень часто алгоритмам в функциях `map` и `reduce` необходимы какие-то данные. Если речь идет о NLP, то это могут быть словари или обученные модели. `Distributed Cache` предоставляет способ для передачи какие-либо данных на машины, которые выполняют задачи (`Task`). Передача данных осуществляется прозрачным для пользователя способом перед запуском задач.

Написание задач на Java

Использование Java - не самый удобный способ, тем не менее это позволяет добиться максимальной производительности и получить

доступ ко все возможностям API. Разберем классический пример WordCount на Java.

В целом исполняемый файл выглядит так

```
1  public class WordCountExample extends Configured
       implements Tool {
2
3      public static void main(final String[] args)
           throws Exception {
4          LoggerUtils.initLogger();
5          ToolRunner.run(new Configuration(), new
               WordCountExample(), args);
6      }
7
8      @Override
9      public int run(final String[] args) throws
           Exception {
10         // установка параметров и запуск Job
11     }
12
13     public static class TextMapper extends Mapper<
           LongWritable, Text, Text, IntWritable> {
14
15         @Override
16         protected void map(final LongWritable key,
               final Text value, final Context context)
                {
17             // код функции map
18         }
19     }
20
21     public static class TextReducer extends Reducer<
           Text, IntWritable, Text, IntWritable> {
22         @Override
23         protected void reduce(final Text key, final
               Iterable<IntWritable> values, final
               Context context) {
24             // код функции reduce
25         }
26     }
27 }
```

В функции, которая запускает `Job` через API мы указываем типы входных и выходных ключей (это необходимо делать, так как API строго типизировано, а при работе с Hadoop Streaming мы используем строки). Дальше задаются классы, имплементирующие `Reducer`, `Mapper` и `Combiner`, формат входных и выходных данных, директорию с входными данными и для записи результатов.

```
1  @Override
2  public int run(final String[] args) throws Exception
       {
3      final Job job = Job.getInstance(getConf());
4      job.setJobName("wordcount");
5      job.setJarByClass(WordCountExample.class);
6
7      job.setOutputKeyClass(Text.class);
8      job.setOutputValueClass(IntWritable.class);
9
10     job.setCombinerClass(TextReducer.class);
11
12     // указываем Mapper и Reducer
13     job.setMapperClass(TextMapper.class);
14     job.setReducerClass(TextReducer.class);
15
16     // удаляем директорию куда
17     // будет писаться результат
18     FileSystem.get(getConf()).delete(new Path(args
           [1]), true);
19     job.setInputFormatClass(TextInputFormat.class);
20     job.setOutputFormatClass(TextOutputFormat.class);
21
22     FileInputFormat.setInputPaths(job, new Path(args
           [0]));
23     FileOutputFormat.setOutputPath(job, new Path(args
           [1]));
24
25     job.waitForCompletion(true);
26
27     return 0;
28 }
```

`Mapper` выглядит следующим образом. При токенизации используется токенайзер из `Lucene` [4].

```
@Override
protected void map(final LongWritable key, final Text
     value, final Context context)  {
    try {
        // создаем поток токенов из строки
        final TokenStream ts = classicAnalyzer.
            tokenStream("", value.toString());
        ts.reset();

        // проходим по всем "словам"
        while(ts.incrementToken()) {
            final CharSequence word = ts.getAttribute
                (CharTermAttribute.class);
            final String str = word.toString();

            textBuffer.set(str);
            // выдаем пару ключ/значение
            context.write(textBuffer, new IntWritable
                (1));
        }
        ts.close();
    } catch (Exception ex) {
        LOG.error("", ex);
    }

}
```

В `Reducer` просто суммируем результат.

```
1  @Override
2  protected void reduce(final Text key, final Iterable<
       IntWritable> values, final Context context)
       throws IOException, InterruptedException {
3      int s = 0;
4      for (IntWritable v : values) {
5          s += v.get();
6      }
7
8      if (s > 2) {
9          context.write(key, new IntWritable(s));
10     }
11 }
```

Как видно, код выглядит переусложнено, необходимо думать об эффективной сериализации и дисериализации данных.

Замечания

1. Нужно иметь веские причины, чтобы начинать новый проект с использование Apache Hadoop. В последние годы наблюдается тенденция на переход к облачным решениям и/или потоковой обработке данных, что позволяет производить анализ в реальном времени. Apache Hadoop на фоне более современных средств выглядит громоздко.

2. Написание задач для Apache Hadoop на Java в настоящее время не выглядит разумным. Естественным образом для Hadoop появились доменно-специфичные языки, например [3] или Pig [6], которые позволяют писать задачи не зная Java и/или Python.

3. Для старта `Job` Apache Hadoop иногда нужно минимум несколько минут, время тратится на подготовку данных, выделение ре-

сурсов и так далее. Это не позволяет эффективно имплементировать итерационные алгоритмы.

4. Не все алгоритмы можно реализовать в парадигме MapReduce. Это касается, в частности, многих алгоритмов на графах.

5. Очень часто задачи MapReduce комбинируют в некоторую последовательность, которые должны выполняться друг за другом. В `mrjob` для этого есть удобный API.

6. Hadoop написан на Java, иногда требуется тонкая настройка сборщика мусора.

Упражнения

Задание 1

Загрузите файл со статьями статей из Википедии: https://yadi.sk/d/ObKNNcaFWEsK-w. Каждая строка в файле имеет следующий формат:

> URL статьи <tab> название статьи <tab> текст

Распакуйте и скопируйте файл на `HDFS` с помощью команды:

```
hadoop fs –put wiki.txt /user/root/wiki.txt
```

Сохраните файл `jop.py` следующего содержания:

```
from mrjob.job import MRJob
from mrjob.protocol import TextProtocol
import re

WORD_RE = re.compile(r"\w+")

class MRWordFreqCount(MRJob):
    OUTPUT_PROTOCOL = TextProtocol
    `
    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield word.lower(), 1

    def combiner(self, word, counts):
        yield word, sum(counts)

    def reducer(self, word, counts):
        yield word, str(sum(counts))


if __name__ == '__main__':
    MRWordFreqCount.run()
```

Запустите задачу с использованием локального движка `MRJob`

```
time python3 job.py wiki.txt —o result_local
```

затем с помощью `hadoop`

```
time python3 job.py —r hadoop hdfs:///user/root/wiki.txt
    \
    —o hdfs:///user/root/result_hadoop
```

затем скопируйте 50 первых строчек из файла `wiki.txt`:

```
head —n 50 wiki.txt > wiki_trunc.txt
hadoop fs —put wiki_trunc.txt /user/root/wiki_trunc.txt
```

и повторите вычисления на новом файле. Какие выводы можно сделать, сравнивая время работы этих 4 тестов.

Задание 2

Используйте данные из предыдущего задания.

Напишите с помощью `mrjob` или на `Hadoop` на `Java` следующие программы (возможно понадобится использовать pipeline'ы [12]

1. Напишите программу, которая находит самое длинное слово.
2. Напишите программу, которая находит среднюю длину слов.
3. Напишите программу, которая находит самое частоупотребляемое слово, состоящее из латинских букв.
4. Все слова, которые более чем в половине случаев начинаются с большой буквы и встречаются больше 10 раз.
5. Напишите программу, которая с помощью статистики определяет устойчивые сокращения вида «пр.», «др.», …
6. Напишите программу, которая с помощью статистики определяет устойчивые сокращения вида «т.п.», «н.э.», …
7. Напишите программу, которая с помощью статистики находит имена, употребляющиеся в статьях.

Из подзаданий 5-7 достаточно сделать любые два.

Задание 3

Есть большой набор данных с текстом, нужно подсчитать статистику взаимовстречаемости слов в одном предложении. На вход подаются тексты, на выходе - пары вида ((apple, eat), 42). Это означает, что слово «apple» встретилось со словом «eat» в 42 предложениях. Подумайте, как можно оптимально реализовать подобный подсчет с

помощью Hadoop. Какие могут быть подводные камни и варианты реализации.