

Большие данные

613x-010402D $x=\{1,2,3\}$

осень 2024

Лекция 3: Основные элементы архитектуры Hadoop

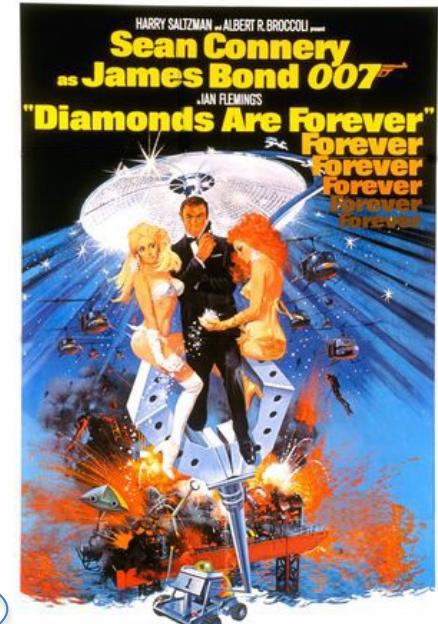
Сергей Борисович Попов
sepo@ssau.ru

Материалы лекций:

https://1drv.ms/f/s!ApFj4iOLPNegvEPfMZL_5jjkXsqfQ

Принципы технологий Big Data

- Разделяй и властвуй
- Обрабатываем там, где храним
- Данные навсегда! («Data Are Forever»)
- In-Memory Processing
- Lazy Evaluation («Ленивые» вычисления)



GFS: The Google file system

19th ACM Symposium on Operating Systems Principles 2003

MapReduce: Simplified Data Processing on Large Clusters

6th Symposium on Operating System Design and Implementation 2004
(OSDI'04)

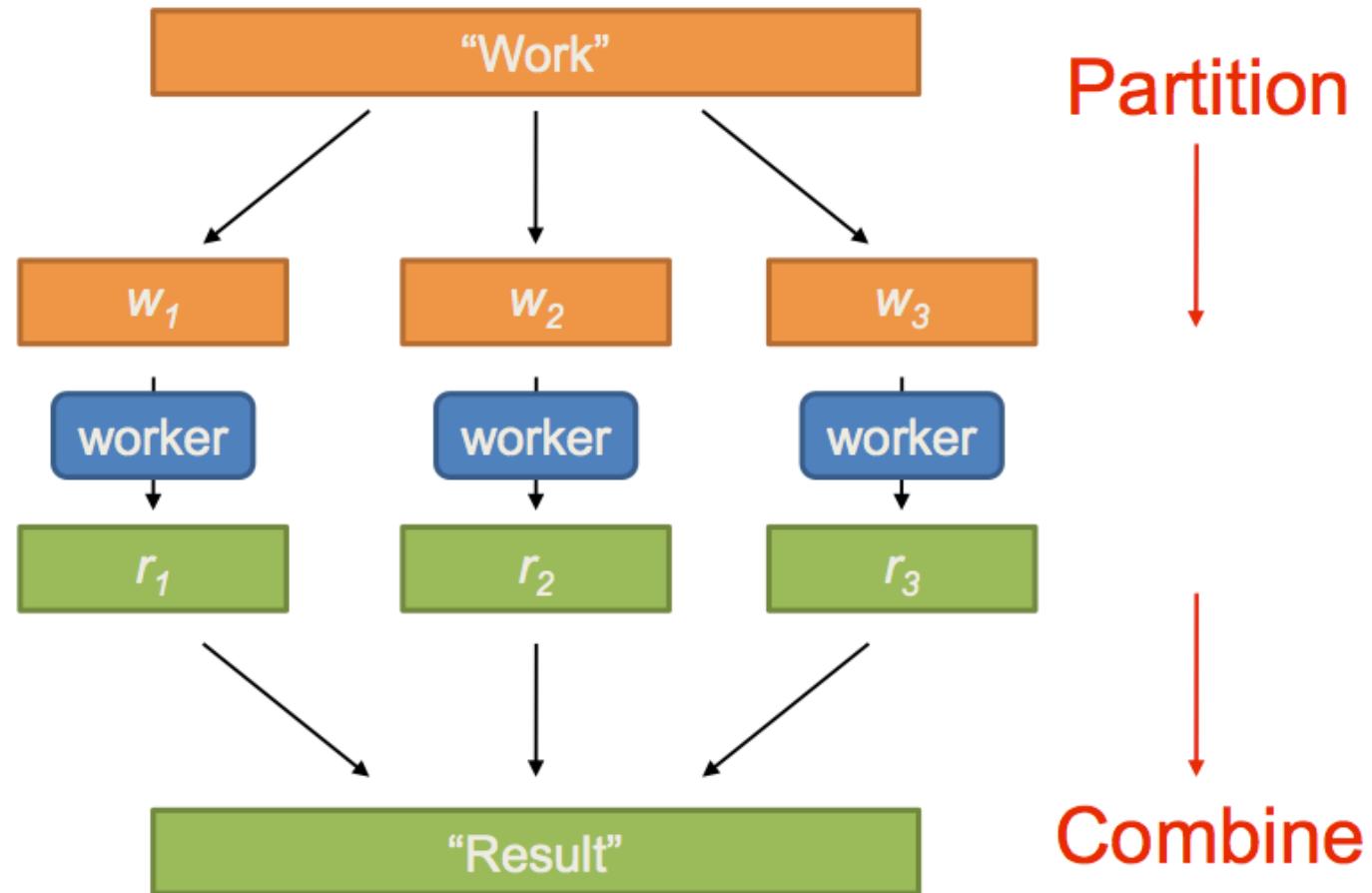
Bigtable: A Distributed Storage System for Structured Data

7th Symposium on Operating Systems Design and Implementation 2006
(OSDI'06)

Программные выступления Google

- [1] Ghemawat, S. The Google file system / Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung // Proceedings of the 19th ACM Symposium on Operating Systems Principles. – Bolton Landing, NY, 2003. – P. 29-43.
- [2] Dean, J. MapReduce: Simplified Data Processing on Large Clusters / Jeffrey Dean, Sanjay Ghemawat // Proceedings of the Sixth Symposium on Operating System Design and Implementation (OSDI'04). – 2004. – P. 137-150.
- [3] Chang, F. Bigtable: A Distributed Storage System for Structured Data / Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber // Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06). – 2006. – P. 205-218.

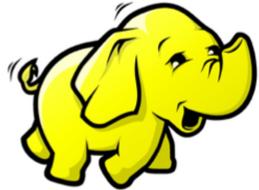
Разделяй и властвуй – Divide and Conquer





“Big Ideas”

- Масштабирование *горизонтальное*, а не *вертикальное*
 - Снятие ограничений SMP и больших shared-memory компьютеров
- Перенос процесса вычислений к данным
 - Ограниченнная пропускная способность коммуникаций
- Последовательная обработка данных
(не использовать произвольный доступ к данным)
 - Значительное время позиционирования головки диска (seeks),
вполне достаточная пропускная способность диска при
последовательном чтении



Системные принципы Hadoop

- Горизонтальное (Scale-Out) масштабирование вместо вертикального (Scale-Up)
- Отправляем код к данным
- Уметь обрабатывать падения и отказы оборудования
- Инкапсуляция сложности работы распределенных и многопоточных приложений

Кластер Hadoop

- “Дешевое” обычное железо (Commodity Hardware)
- Соединенное по сети
- Расположено в одном месте
 - Сервера в стойках в dataцентре

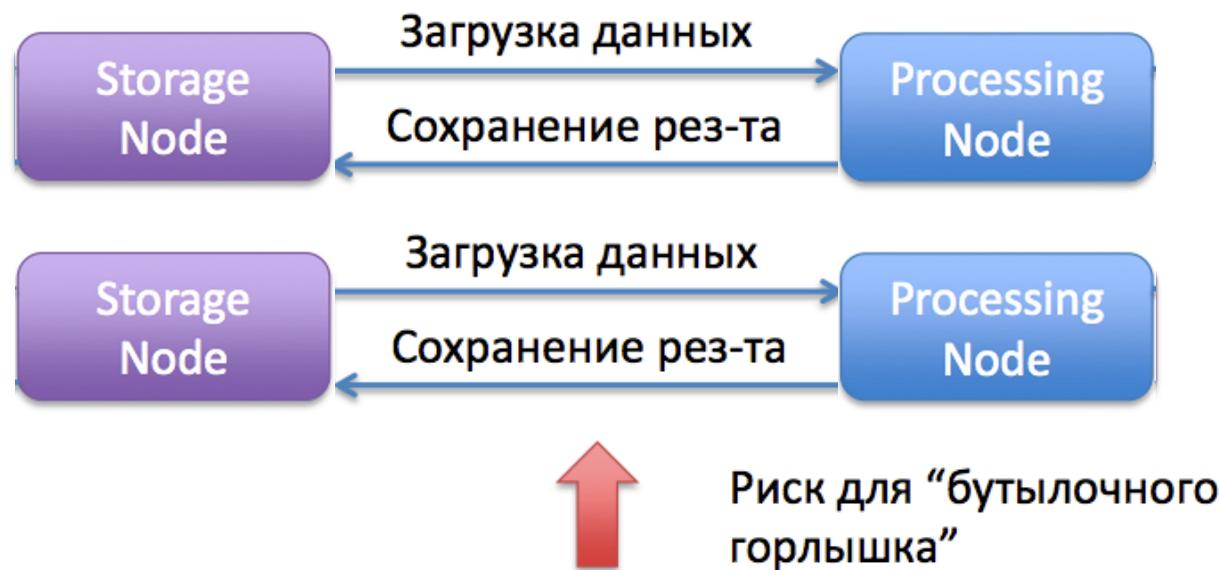


Горизонтальное масштабирование вместо вертикального

- Сложнее и дороже масштабироваться “вверх”
 - Добавить дополнительные ресурсы к существующему железу (CPU, RAM)
 - Закон Мура не успевает за ростом объема данных
 - Если нельзя улучшить железо, то надо покупать более мощное новое
 - Это вертикальное масштабирование
- Горизонтальное масштабирование
 - Добавить больше машин к существующему распределенному окружению
 - Уровень приложения поддерживает добавление/удаление нод
 - Hadoop исповедует такой подход – набор связанных нод
 - Так же очень просто масштабироваться “вниз”

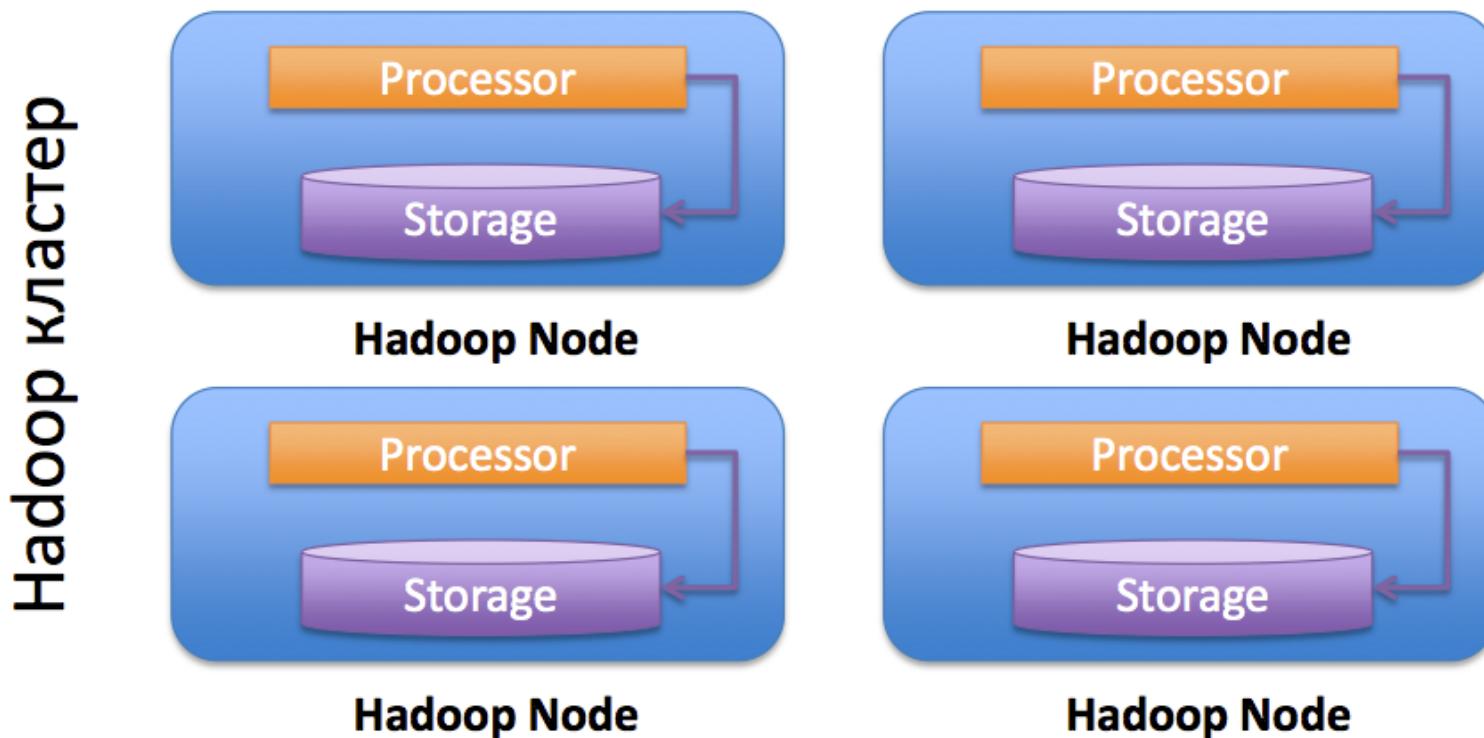
Код к данным

- Традиционная архитектура системы обработки данных
 - Ноды системы разделяются на вычислительные и стораджи, соединяются высокоскоростным линком
 - Многие приложения обработки данных являются CPU-bound, что приводит к проблемам с сетью



Код к данным

- Hadoop сближает вычислительный процессор и данные
 - Код копируется к данным (небольшой расход, Кб)
 - Процессор выполняет код и имеет доступ к локально расположенным данным



Отказы оборудования

- Чем больше количество машин, тем чаще будут отказы железа
 - На больших кластерах (сотни и тысячи машины) отказы будут еженедельно (и даже ежедневно!)
- Hadoop разрабатывался с учетом отказов железа
 - Репликация данных
 - Перезапуск тасков

Инкапсуляция сложности реализации

- Hadoop скрывает многие сложности распределенных и многопоточных систем
 - Небольшое число компонент
 - Предоставляет простой и хорошо определенный интерфейс для взаимодействия между компонентами
- Освобождает разработчика от заботы о проблемах системного уровня
 - Race conditions, ожидание данных
 - Организация передачи данных, распределение данных, доставка кода и т.д.
- Позволяет разработчику фокусироваться на разработке приложения и реализации бизнес-логики

Базовые модули Hadoop



<https://hadoop.apache.org>

- Изначально Hadoop был известен в основном из-за двух ключевых компонент:
 - **HDFS** – Hadoop Distributed File System
 - **MapReduce** – Фреймворк распределённой обработки данных
- Сейчас Apache™ Hadoop® содержит четыре базовых модуля:
 - **Hadoop Common**: The common utilities that support the other Hadoop modules.
 - **Hadoop Distributed File System (HDFS™)**: A distributed file system that provides high-throughput access to application data.
 - **Hadoop YARN**: A framework for job scheduling and cluster resource management.
 - **Hadoop MapReduce**: A YARN-based system for parallel processing of large data sets.

Экосистема Hadoop



Источник: <http://hadoop.apache.org>

- Другие Apache-проекты, связанные с Hadoop:
 - **Ambari™**: web-средства мониторинга/управления Hadoop-кластерами, включая поддержку для Hadoop HDFS, Hadoop MapReduce, Hive, HCatalog, HBase, ZooKeeper, Oozie, Pig and Soop.
 - **Avro™**: система сериализации данных (передача схемы вместе с данными, работа с динамически типизированными объектами).
 - **Cassandra™**: A scalable multi-master database with no single points of failure.
 - **Chukwa™**: A data collection system for managing large distributed systems.
 - **HBase™**: A scalable, distributed database that supports structured data storage for large tables.
 - **Hive™**: A data warehouse infrastructure that provides data summarization and ad hoc querying.
 - **Mahout™**: A Scalable machine learning and data mining library.
 - **Ozone™**: A scalable, redundant, and distributed object store for Hadoop.
 - **Pig™**: A high-level data-flow language and execution framework for parallel computation.
 - **Spark™**: A fast and general compute engine for Hadoop data.
 - **Submarine**: A unified AI platform which allows engineers and data scientists to run Machine Learning and Deep Learning workload in distributed cluster.
 - **Tez™**: A generalized data-flow programming framework, built on Hadoop YARN.
 - **ZooKeeper™**: A high-performance coordination service for distributed applications.

Дистрибутивы Hadoop

- Дистрибутивы Hadoop призваны решить проблему несовместимостей версий
- Вендоры дистрибутивов обеспечивают
 - Интеграционные тесты компонентов Hadoop
 - Инсталляционные пакеты в различных форматах
 - rpm, tarballs и т.д.
 - Могут включать дополнительные скрипты для запуска
 - Некоторые вендоры могут делать backport фич и исправлений багов из Apache
 - Обычно, найденный баги исправляются коммитерами, которые работают на вендоров и отправляются в основной Apache репозиторий

Дистрибутивы Hadoop и платформы Big Data

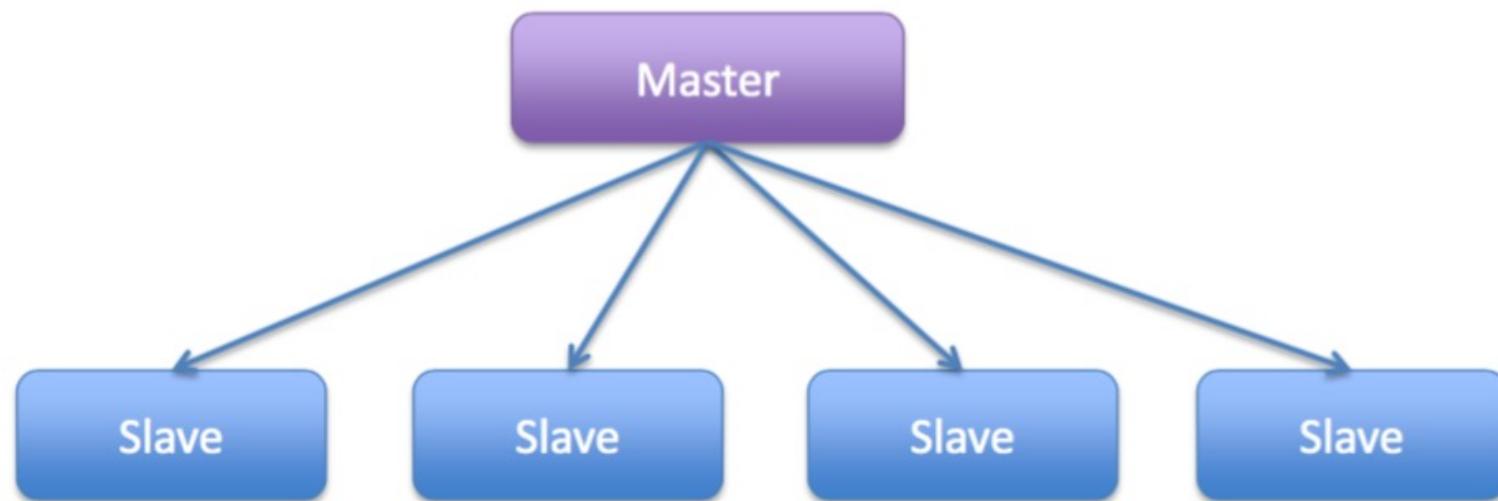
- HPE Ezmeral Data Fabric (бывшая MapR)
- Azure Data Lake Store by Microsoft
- FusionInsight Big Data Platform by Huawei
- Google Cloud Platform
- Amazon EMR
- IBM BigInsights for Apache Hadoop
- Hortonworks Sandbox & Cloudera Director by Cloudera
- Oracle Big Data
- Transwarp Data Hub
- Hadoop as a Service by Idera (Qubole)
- Seabox BigData Platform by Eastern Jin Technology (Seabox Data)



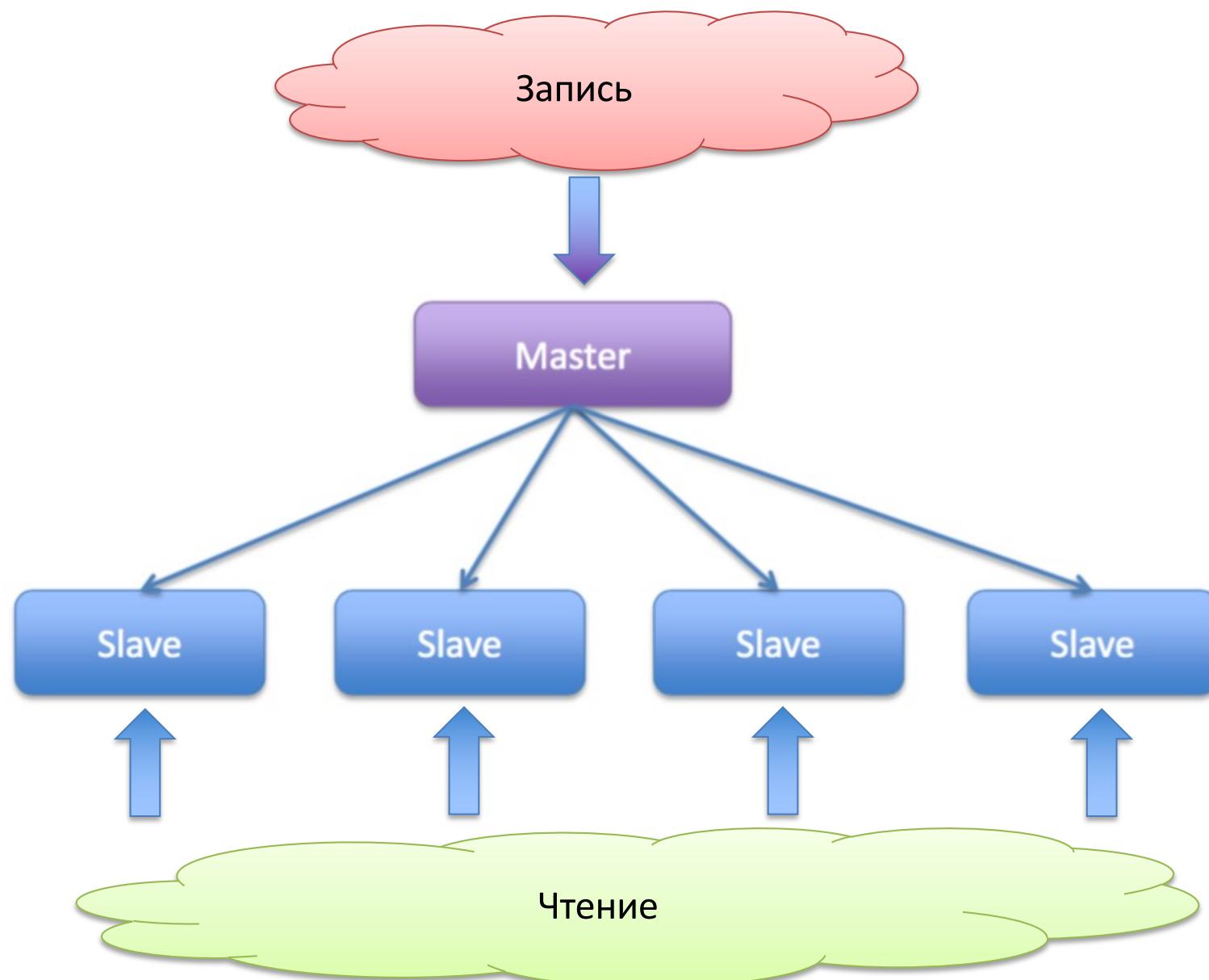
CLOUDERA



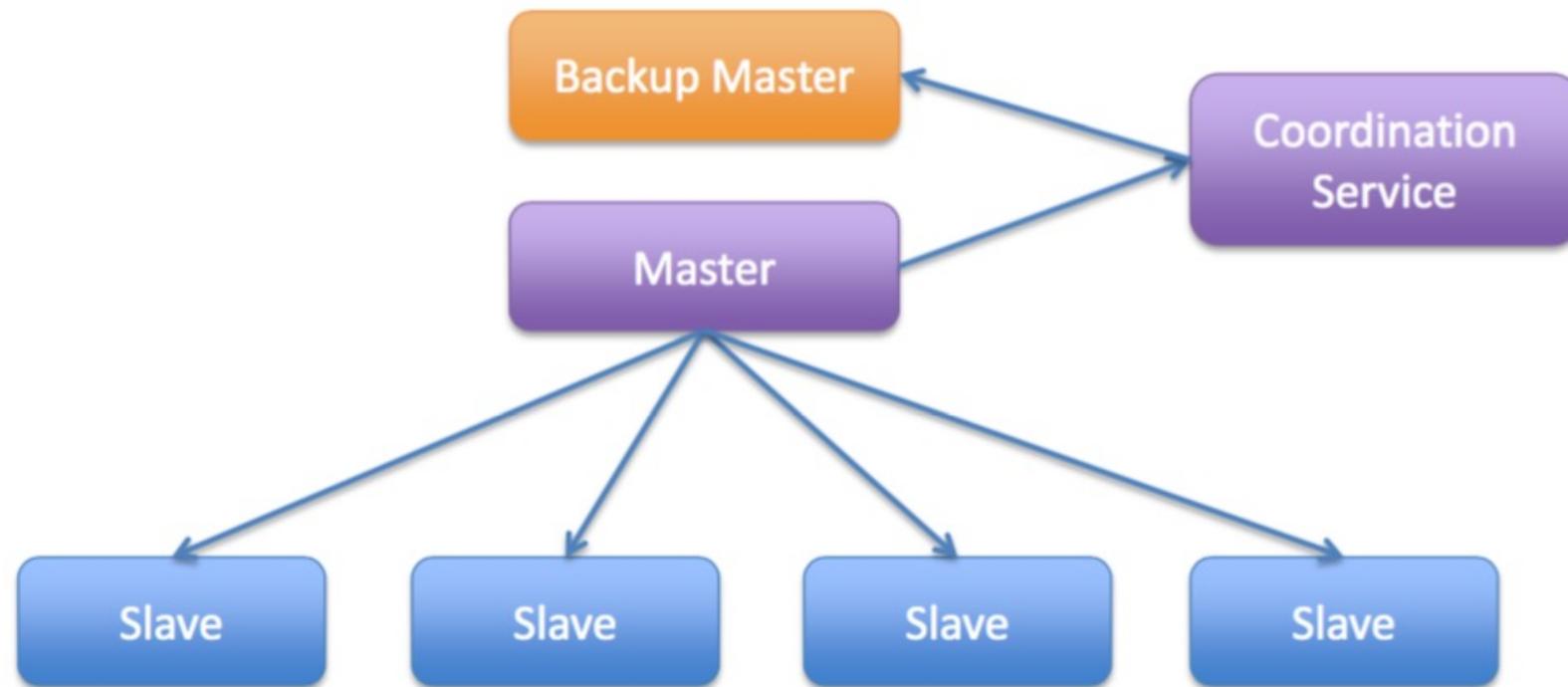
Масштабируемая распределённая система



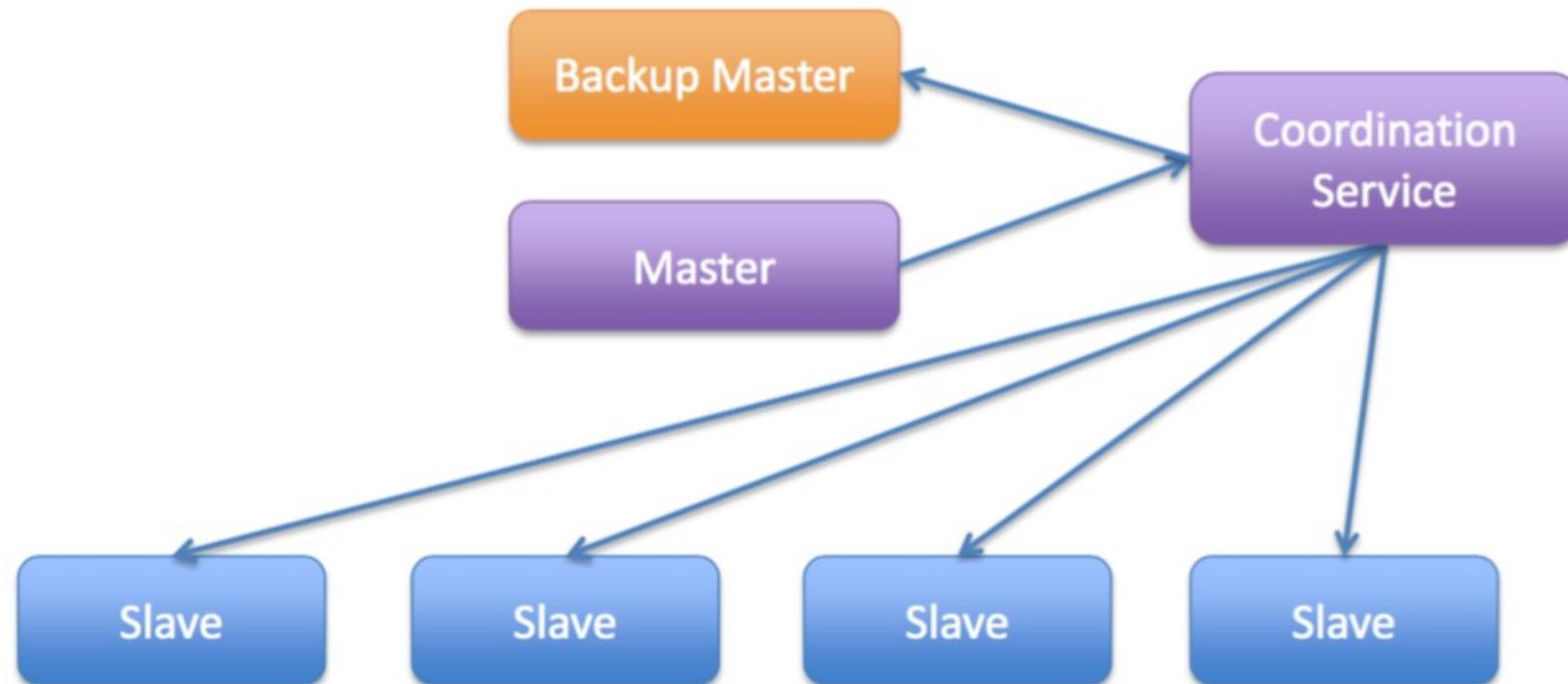
Масштабируемая распределённая система



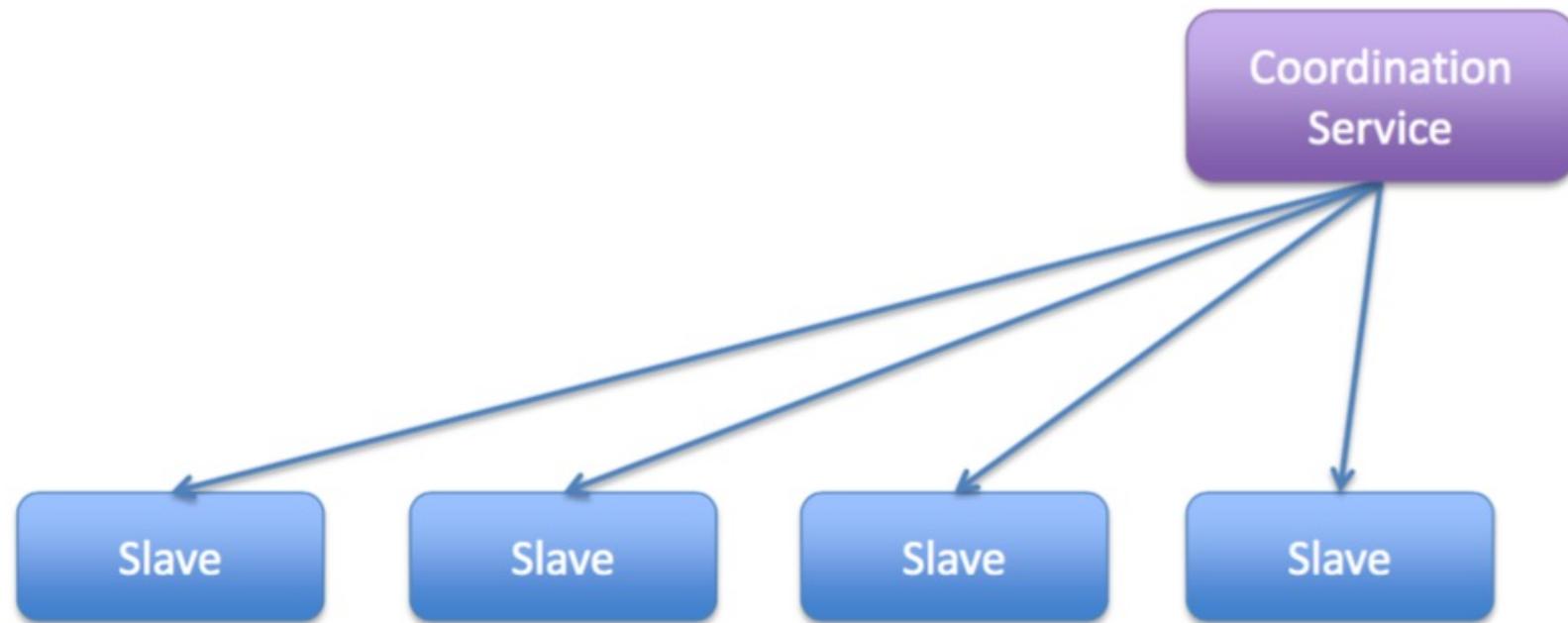
Масштабируемая распределённая система



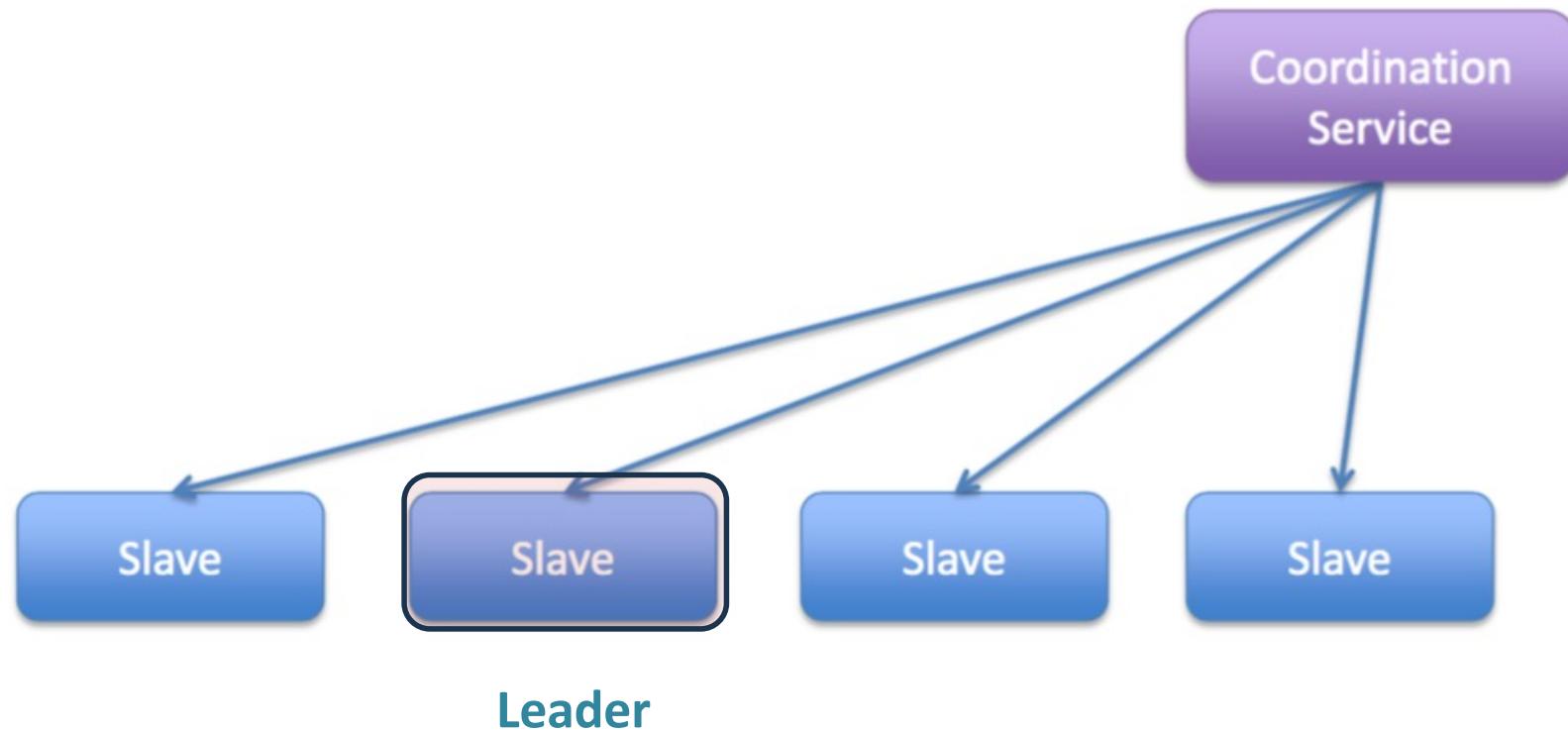
Масштабируемая распределённая система



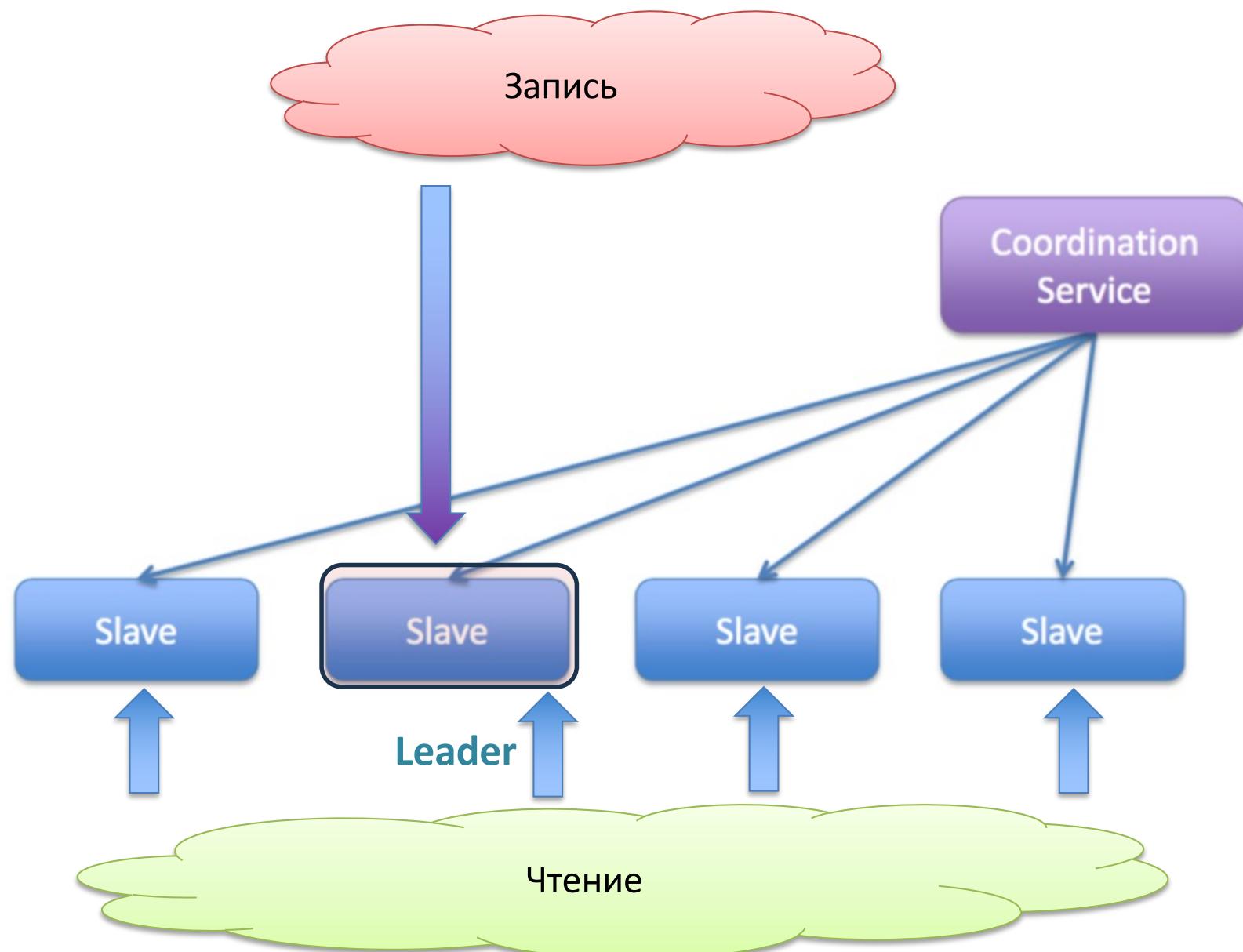
Масштабируемая распределённая система



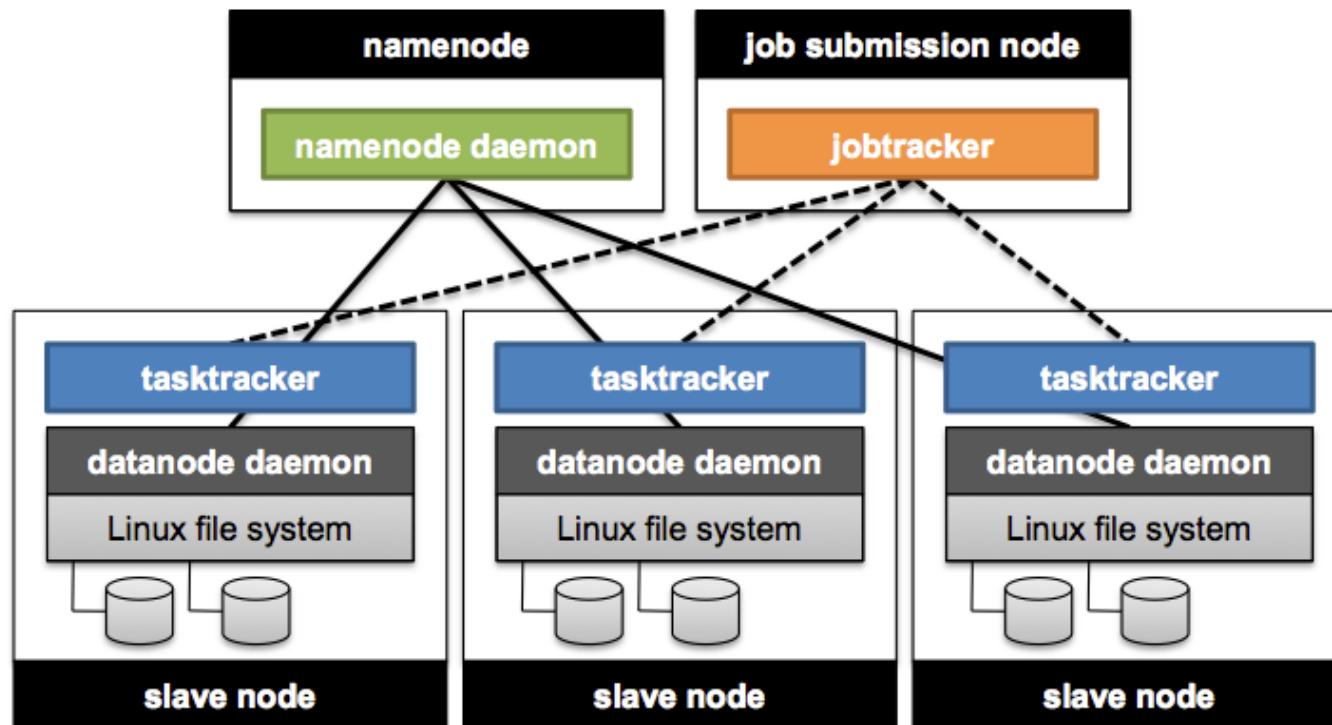
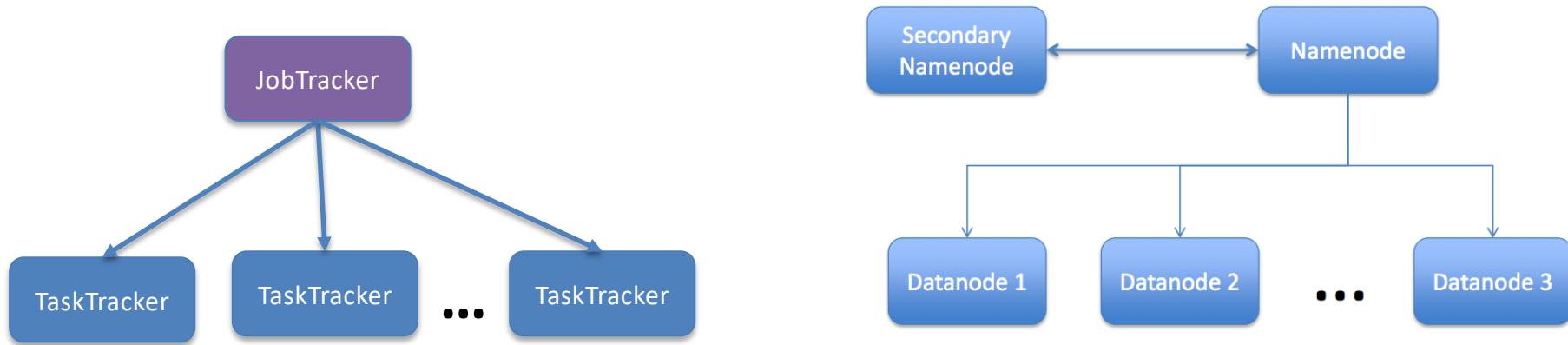
Масштабируемая распределённая система



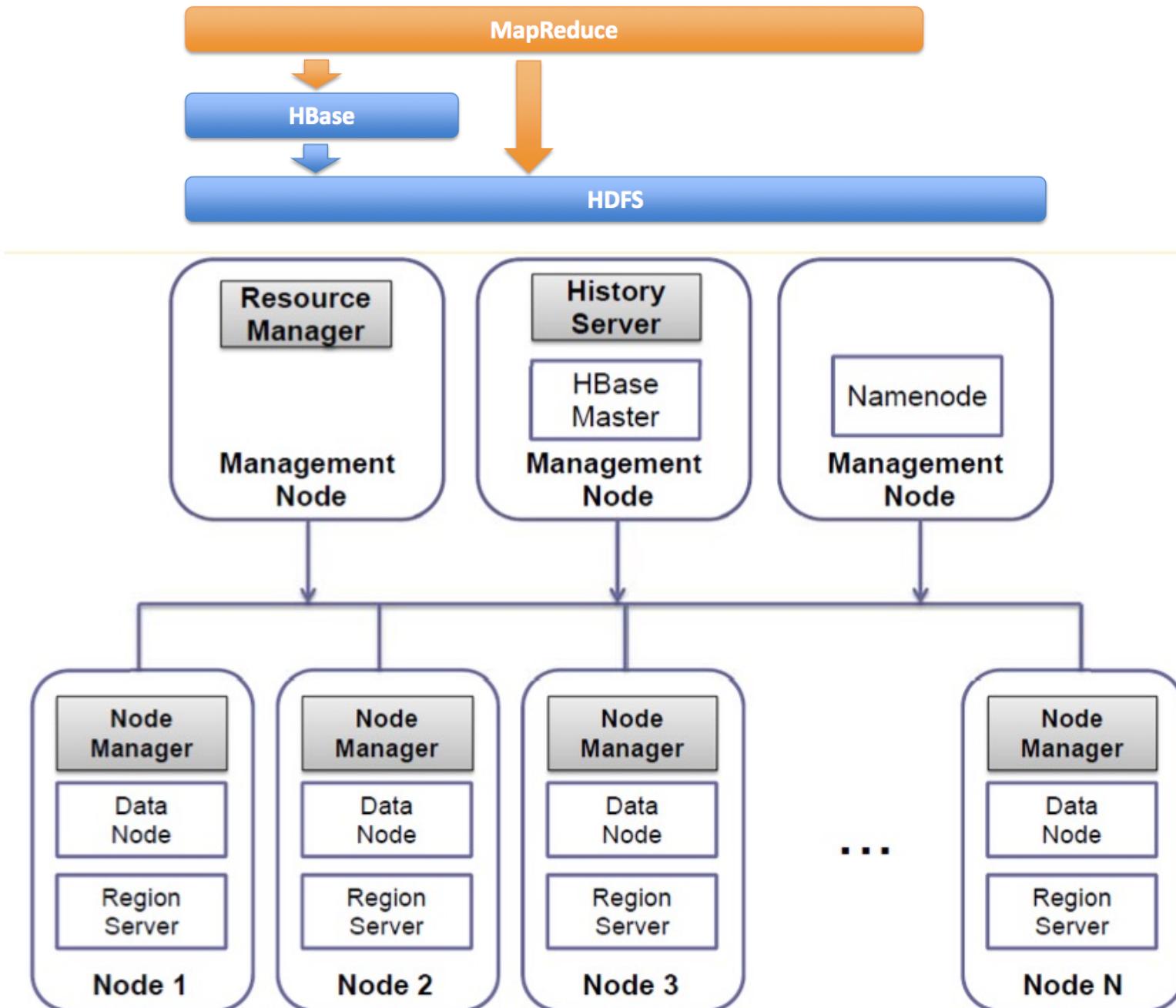
Масштабируемая распределённая система



Сервисы Hadoop 1.0 (MapReduce + HDFS)



Сервисы Hadoop 2.0 (YARN + HDFS + HBase)

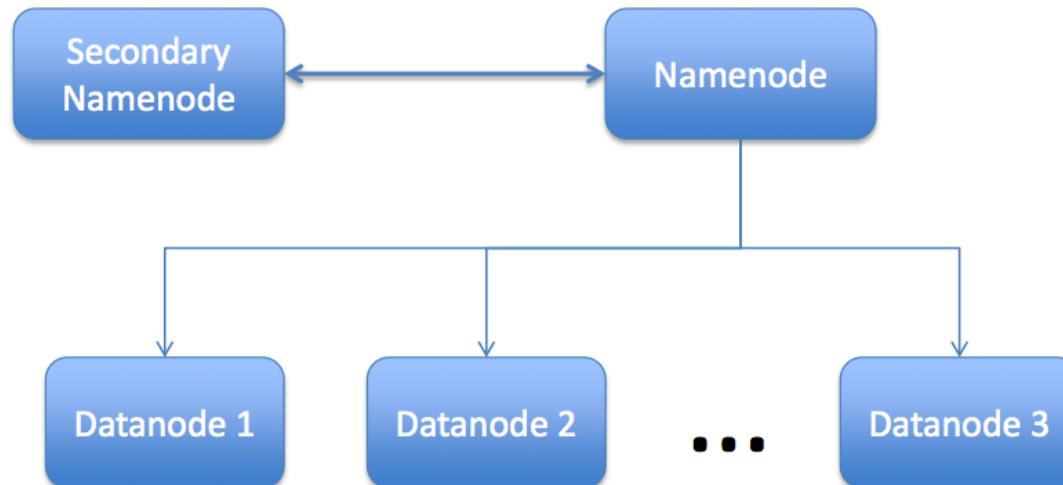


HDFS – Hadoop Distributed File System

- "Один большой диск" для пользователя
- Работает поверх обычных файловых систем (Ext3, Ext4, XFS)
- Основана на *архитектуре* Google's FileSystem
- Отказоустойчивая
- Хорошо подходит для...
 - хранения больших файлов (тера- петабайты, миллионы (но не миллиарды) файлов размером от 100 Мб)
 - последовательной обработки данных (паттерн "write once / read many times", нет операций произвольного доступа, но есть append)
 - использования "обычных" серверов
- Не подходит для...
 - чтения с минимальной задержкой доступа (low-latency reads)
 - работы с большим количеством небольших файлов
 - многопоточной записи данных

Управляющие процессы HDFS (сервисы или демоны)

- **Namenode**
 - отвечает за файловое пространство (Namespace), метаинформацию, расположение блоков файлов
 - запускается на одном (выделенном) узле
- **Datanode**
 - хранит и отдаёт блоки данных
 - отправляет ответы о состоянии на Namenode
 - запускается обычно на всех рабочих узлах кластера
- **Secondary Namenode**
 - периодически обновляет fsimage
 - использует отдельный узел с теми же характеристиками, что и Namenode
 - не используется для обеспечения "высокой доступности" (high-availability)



Файлы и блоки

- Файлы в HDFS состоят из блоков (блок – единица хранения данных)
- Файлы управляются через Namenode, хранятся на Datanode
- Стандартные размеры блоков: 64МВ, 128МВ, 256МВ
- Блоки реплицируются по узлам в процессе записи
 - фактор репликации по умолчанию равен 3
 - обеспечивает отказоустойчивость и оптимизацию доступа к данным
 - один и тот же блок хранится на нескольких Datanode
 - Namenode определяет куда копировать реплики блоков

HDFS-7285: Erasure Coding Support inside HDFS

Без EC: Файл из 6 блоков с 3х-кратной репликацией занимает 18 блоков

Erasure Coding (EC):

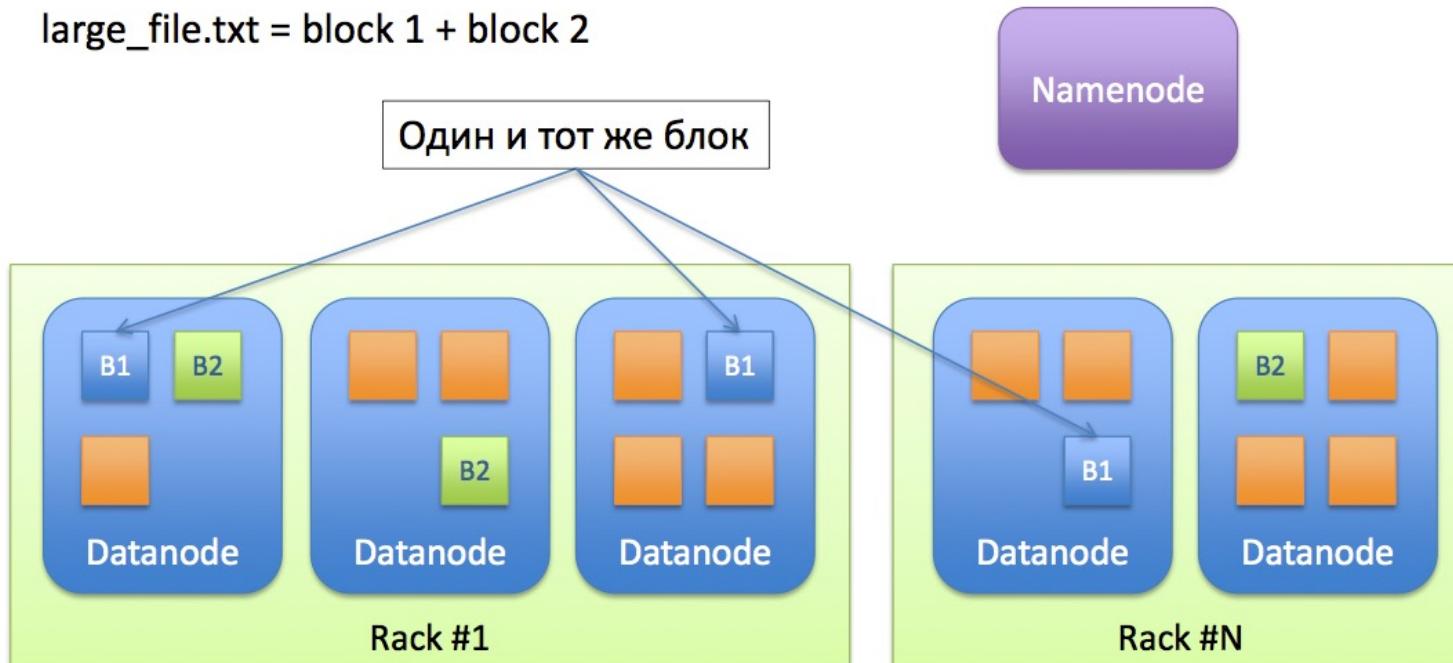
Коэффициент репликации файла EC всегда 1 и не может быть изменен (нет -setrep). EC использует стрипинг, который разделяет последовательные данные (например, файл) на более мелкие единицы (например, бит, байт или блок) и хранит последовательные единицы на разных дисках. Для каждой полосы исходных ячеек данных рассчитывается и хранится определенное количество ячеек четности, процесс которого называется кодированием. Ошибка в любой ячейке полосы восстанавливается путем декодирования на основе выживших данных и ячеек четности.

При развертывании EC файл будет занимать только 9 блоков дискового пространства (6 данных, 3 паритета) .

Стратегия репликации

- Размещение блоков зависит от того в какой стойке стоит сервер (rack aware) – баланс между надёжностью и производительностью
 - Снижение нагрузки на сеть (bandwidth)
 - Улучшение надёжности при размещении реплик в разных стойках
- При репликации на 3 узла:
 - 1-я реплика на произвольный узел (выбор исходя из загруженности)
 - 2-я реплика на другой узел из той же стойки
 - 3-я реплика на узел из другой стойки

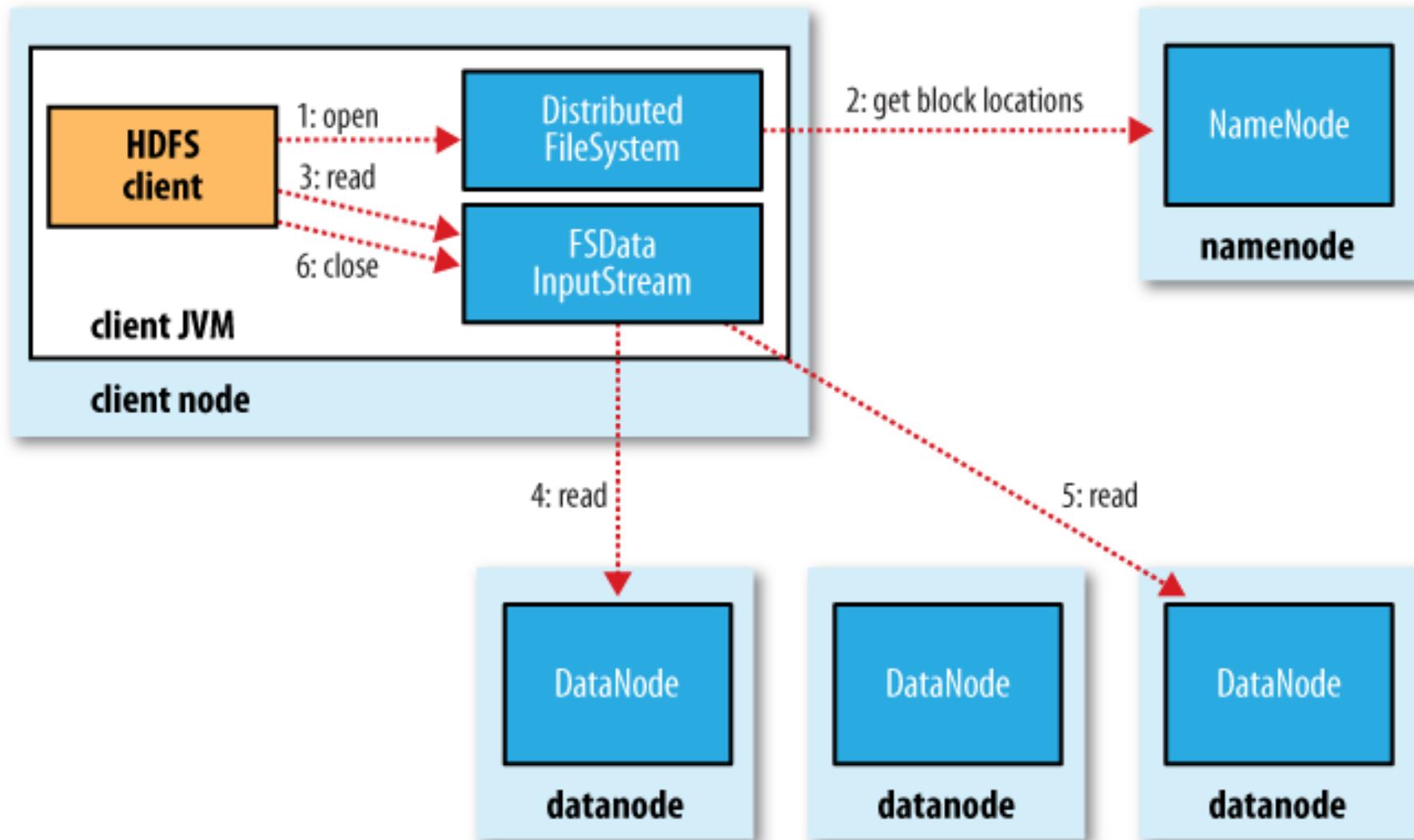
large_file.txt = block 1 + block 2



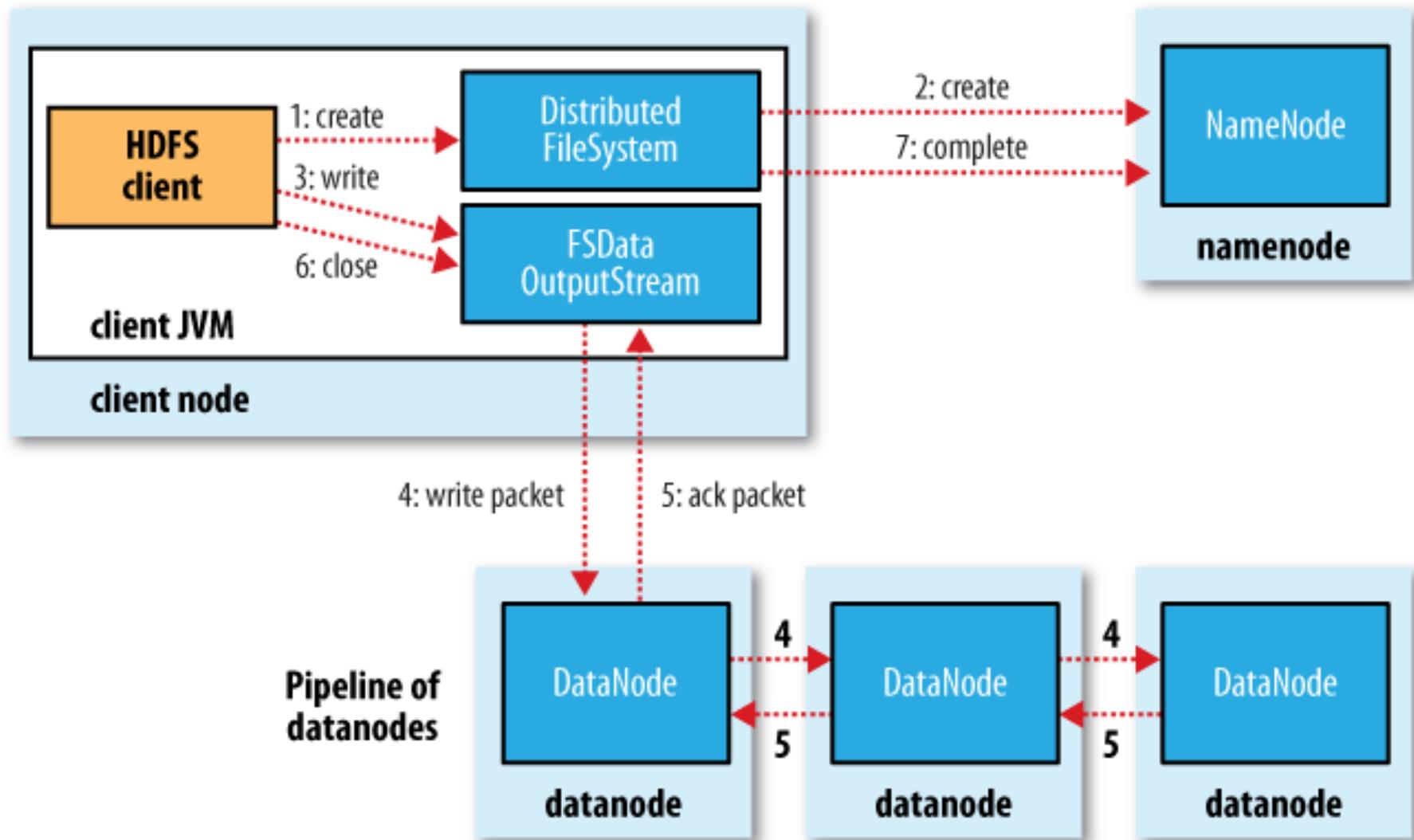
Взаимодействие клиентов с HDFS

- Namenode не выполняет непосредственно операций чтения/записи
- Клиент обращается к Namenode для
 - обновления области имён HDFS (namespace)
 - для получения информации о размещении блоков для чтения/записи
- Клиент взаимодействует напрямую с Datanode для чтения/записи

Чтение данных из HDFS



Запись данных в HDFS



Namenode: использование памяти

- Для быстрого доступа вся мета-информация о блоках хранится в ОЗУ Namenode
 - Чем больше кластер, тем больше ОЗУ требуется
 - Лучше миллионы больших файлов (сотни мегабайт), чем миллиарды маленьких
 - Работает на кластерах из сотен машин
- Hadoop 2+
 - Namenode Federation
 - Каждая Namenode управляет частью блоков
 - Горизонтальное масштабирование Namenode
 - Поддержка кластеров из тысячи машин
 - Детали тут: <http://hadoop.apache.org/docs/r2.0.2-alpha/hadoop-yarn/hadoop-yarn-site/Federation.html>

Namenode: использование памяти

- Изменение размера блока влияет на максимальный размер FS
 - Увеличение размера блока с 64Мб до 128Мб уменьшает число блоков и существенно увеличивает размер места, которое NN может обслуживать
 - Пример:
 - Пусть у нас есть 200Тб = 209,715,200 Мб
 - При размере блока 64Мб это соответствует 3,276,800 блоков
 - $209,715,200\text{мб} / 64\text{мб} = 3,276,800$ блоков
 - При размере блока 128Мб это соответствует 1,638,400 блоков
 - $209,715,200\text{мб} / 128\text{мб} = 1,638,400$ блоков

Fault-tolerance в Namenode

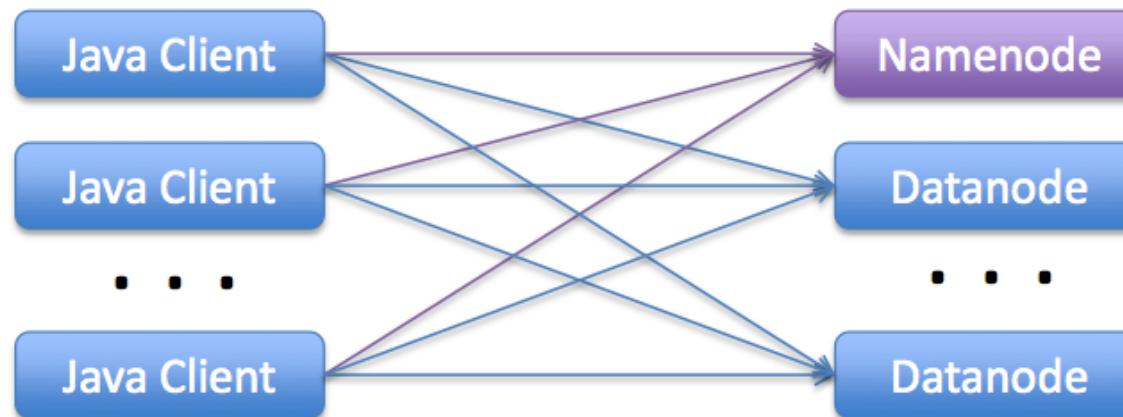
- Процесс демона NN должен быть запущен все время
 - Если демон падает, то HDFS не работает (и обычно весь кластер тоже)
- Namenode – это единственная точка отказа (*single point of failure*)
 - Должна работать на отдельной надежной машине
 - Обычно, это не бывает проблемой
- Hadoop 2+
 - High Availability Namenode
 - Процесс Active Standby всегда запущен и берет на себя управления в случае падения NN
 - Все еще в процессе тестирования
 - Более подробно тут:
 - <http://hadoop.apache.org/docs/r2.0.2-alpha/hadoop-yarn/hadoop-yarn-site/HDFSHighAvailability.html>

Доступ к HDFS

- Способы доступа
 - Direct Access
 - Взаимодействует с HDFS с помощью нативного клиента
 - Java, C++
 - Через Proxy Server
 - Досутп к HDFS через Proxy Server – middle man
 - Серверы REST, Thrift и Avro

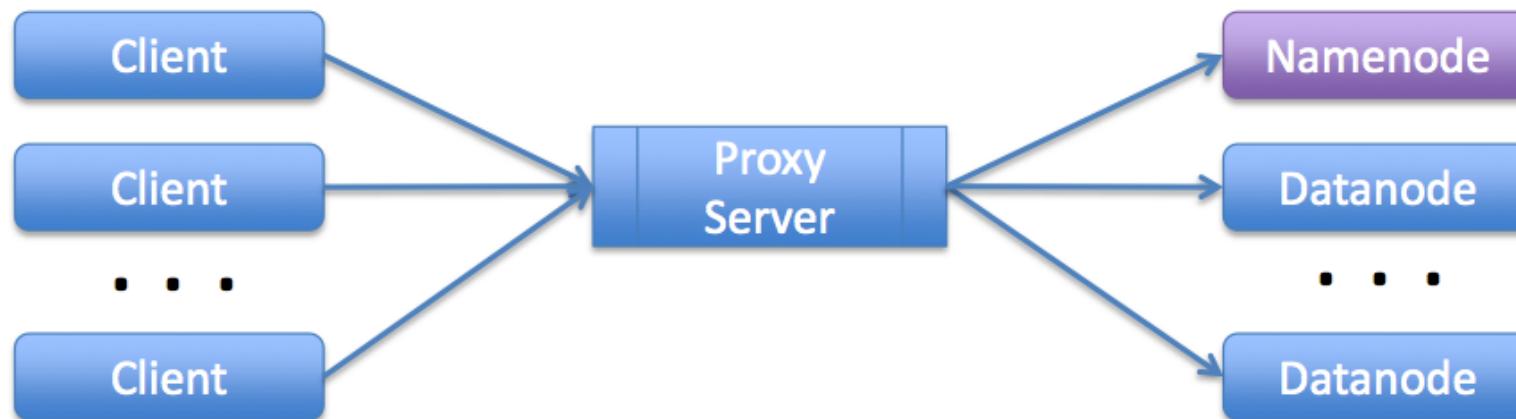
Direct Access

- API для Java и C++
- Клиент запрашивает метаданные (такие, как расположение блоков) от NN
- Клиент напрямую запрашивает данные от DN
- Java API
 - Наиболее часто используется
- Используется для MapReduce



Доступ через Proxy Server

- Клиент работает через внешний Proxy Server
 - Т.о. должна быть независимость от языка
- Существует несколько серверов в поставке с Hadoop
 - Thrift – язык определения интерфейса
 - WebHDFS REST – ответы в формате JSON, XML или Protocol Buffers
 - Avro – механизм сериализации



Команды Shell

- Взаимодействие с FS через стандартный unix-shell
- Использование:
\$hdfs dfs -<command> -<option> <URI>
 - Example *\$hdfs dfs -ls /*
- **URI usage:**
 - HDFS: *\$hdfs dfs -ls hdfs://localhost/to/path/dir*
 - Local: *\$hdfs dfs -ls file:///to/path/file3*
 - Схема и имя хоста NN опционально, по-умолчанию используется параметр из конфигурации
 - В core-site.xml - fs.default.name property

Hadoop URI

scheme://authority/path



hdfs://localhost:8020/user/home

scheme

authority

HDFS path

Команды в shell

- Большинство команд ведет себя схожим образом, что и команды в Unix
 - *cat, rm, ls, du ...*
- Поддержка специфичных для HDFS операций
 - Напр., смена фактора репликации
- Вывод списка команд
 - `$ hdfs dfs –help`
- Показать детальную информацию по команде
 - `$ hdfs dfs -help <command_name>`

Основные команды в shell

- ***cat*** – вывод источника в stdout
 - Весь файл: `$hdfs dfs -cat /dir/file.txt`
 - Полезно вывод перенаправить через pipe в *less*, *head*, *tail* и т.д.
 - Получить первые 100 строк из файла
 - `$hdfs dfs -cat /dir/file.txt | head -n 100`
- ***ls*** – отобразить файловую статистику, для директории отобразить вложенные директории и файлы
 - `$hdfs dfs -ls /dir/`
- ***mkdir*** – создать директорию
 - `$hdfs dfs -mkdir /dir`

Копирование данных в shell

- ***cp*** – скопировать файлы из одного места в другое
 - `$hdfs dfs -cp /dir/file1 /otherDir/file2`
- ***mv*** – перемещение файла из одного места в другое
 - `$hdfs dfs -mv /dir/file1 /dir2`
- ***put*** – копирование файла из локальной FS в HDFS
 - `$hdfs dfs -put localfile /dir/file1`
 - `copyFromLocal`
- ***get*** – копирование файла из HDFS в локальную FS
 - `$hdfs dfs -get /dir/file1 localfile`
 - `copyToLocal`

Удаление и статистика в shell

- ***rm*** – удалить файл (в корзину)
 - `$hdfs dfs -rm /dir/file`
- ***rm -r*** – удалить рекурсивно директорию
 - `$hdfs dfs -rm -r /dir`
- ***du*** – отобразить размер файла или директории в байтах
 - `$hdfs dfs -du /dir/`
- ***du -h*** – отобразить размер файла или директории в удобно-читаемом формате
 - `$hdfs dfs -du -h /dir/
65M /dir`

Остальные команды в shell

- Другие команды
 - *chmod, count, test, tail и т.д.*
- Чтобы узнать больше
 - `$hdfs dfs -help`
 - `$hdfs dfs -help <command>`

Команда fsck

- Проверка неконсистентности файловой системы
- Показывает проблемы
 - Отсутствующие блоки
 - Недореплицированные блоки
- Не устраняет проблем, только информация
 - Namenode попытается автоматически исправить проблемы
- **\$ hdfs fsck <path>**
 - Напр., \$ *hdfs fsck /*

Права в HDFS

- Ограничения на уровне файла/директории
 - Сходство с моделью прав в POSIX
 - Read (r), Write (w) и Execute (x)
 - Разделяется на пользователя, группу и всех остальных
- Права пользователя определяются исходя из прав той ОС, где он запускает клиентское приложение
- Авторизация через Kerberos
 - Hadoop 0.20.20+
 - <http://hadoop.apache.org/common/docs/r0.23.0/hadoop-yarn/hadoop-yarnsite/ClusterSetup.html>

```
[cloudera@localhost ~]$ hadoop fs -ls /user/cloudera/wordcount/output
Found 3 items
-rw-r--r--  3 cloudera cloudera  0 2014-03-15 11:56 /user/cloudera/wordcount/output/_SUCCESS
drwxr-xr-x - cloudera cloudera  0 2014-03-15 11:56 /user/cloudera/wordcount/output/_logs
-rw-r--r--  3 cloudera cloudera 31 2014-03-15 11:56 /user/cloudera/wordcount/output/part-00000
```

Команда **DFSAadmin**

- Команды для администрирования HDFS
 - `$hdfs dfsadmin <command>`
 - Напр.: `$hdfs dfsadmin –report`
- ***report*** – отображает статистику по HDFS
 - Часть из этого также доступна в веб-интерфейсе
- ***safemode*** – переключения между режимом safemode для проведения административных работ
 - Upgrade, backup и т.д.

Балансер HDFS

- Блоки в HDFS могут быть неравномерно распределены по всем Datanode'ам кластера
 - К примеру, при добавлении новых машин они будут какое-то время почти пустыми, т.к. новые данные будут записываться исходя из топологии
- Балансер – это утилита, которая автоматически анализирует расположение блоков в HDFS и старается его сбалансировать
 - `$ hdfs balancer`

HDFS API

File System Java API

- org.apache.hadoop.fs.FileSystem
 - Абстрактный класс, которые представляет абстрактную файловую систему
 - (!) Это именно класс, а не интерфейс
- Реализуется в различных вариантах
 - Напр., локальная или распределенная

Реализации *FileSystem*

- Hadoop предоставляет несколько конкретных реализаций
 - *org.apache.hadoop.fs.LocalFileSystem*
 - Подходит для нативных FS, использующих локальные диски
 - *org.apache.hadoop.hdfs.DistributedFileSystem*
 - Hadoop Distributed File System (HDFS)
 - *org.apache.hadoop.hdfs.HftpFileSystem*
 - Доступ к HDFS в read-only режиме через HTTP
 - *org.apache.hadoop.fs.ftp.FTPFileSystem*
 - Файловая система поверх FTP-сервера
- Различные реализации для разных задач
- Для работы с HDFS обычно используется
 - *org.apache.hadoop.hdfs.DistributedFileSystem*

Пример SimpleLocalLs.java

```
public class SimpleLocalLs {  
    public static void main(String[] args) throws Exception{  
  
        Path path = new Path("/");  
        if ( args.length == 1 ){  
            path = new Path(args[0]);  
        }  
  
        Configuration conf = new Configuration();  
        FileSystem fs = FileSystem.get(conf);  
  
        FileStatus [] files = fs.listStatus(path);  
        for (FileStatus file : files ){  
            System.out.println(file.getPath().getName());  
        }  
    }  
}
```

FileSystem API: Path

- Объект Path в Hadoop представляет файл или директорию
 - *java.io.File* сильно завязан на локальную FS
- Path – это на самом деле URI в FS
 - HDFS: `hdfs://localhost/user/file1`
 - Local: `file:///user/file1`
- Пример:
 - `new Path("/test/file1.txt");`
 - `new Path("hdfs://localhost:9000/test/file1.txt");`

Объект Configuration

- Объект Configuration хранит конфигурацию сервера и клиента
 - Довольно много где используется в Hadoop
 - HDFS, MapReduce, Hbase,..
- Использует простую парадигму key-value
 - Является враппером над java.util.Properties,
 - Который в свою очередь враппер над java.util.Hashtable
- Получения значения параметра
 - *String name = conf.get("fs.default.name");*
 - returns null если свойства не существует
- Получения значения параметра и вернуть значение по-умолчанию, если не существует
 - *String name = conf.get("fs.default.name", "hdfs://localhost:9000");*
- Также есть типизированные варианты
 - getBoolean, getInt, getFloat и т.п.
 - float size = conf.getFloat("file.size");

Объект Configuration

- Обычно инициализируется значениями через конфигурационный файлы из CLASSPATH (напр., conf/coresite.xml и conf/hdfs-site.xml)
 - *Configuration conf = new Configuration();
conf.addResource(
 new Path(HADOOP_HOME + "/conf/coresite.xml"));*
- *conf.addResource()* может принимать как String, так и Path
 - *conf.addResource("hdfs-site.xml")*
 - *conf.addResource(new Path("/my/location/site.xml"))*
- По-умолчанию загружает
 - core-default.xml
 - Расположен в hadoop-common-X.X.X.jar/core-default.xml
 - core-site.xml

Чтение данных из файла

- Создать объект *FileSystem*
- Открыть *InputStream*, указывающий на *Path*
- Скопировать данные по байтам используя *IOUtils*
- Закрыть *InputStream*

Пример ReadFile.java

```
public class ReadFile {  
    public static void main(String[] args) throws IOException {  
        Path file = new Path("/path/to/file.txt");  
        FileSystem fs = FileSystem.get(new Configuration()); // Open FileSystem  
  
        InputStream input = null;  
        try {  
            input = fs.open(file); // Open InputStream  
            IOUtils.copyBytes(input, System.out, 4096); // Copy from Input to Output  
            Stream  
        } finally {  
            IOUtils.closeStream(input); // Close stream  
        }  
    }  
}
```

Запись данных в файл

- Создать объект *FileSystem*
- Открыть *OutputStream*
 - Указывает на *Path* из *FileSystem*
 - Используем *FSDataOutputStream*
 - Автоматически создаются все директории в пути, если не существуют
- Копируем данные по байтам используя *IOUtils*

Пример WriteToFile.java

```
public class WriteToFile {  
    public static void main(String[] args) throws IOException {  
        String text = "Hello world in HDFS!\n";  
        InputStream in = new BufferedInputStream(  
            new ByteArrayInputStream(text.getBytes()));  
  
        Path file = new Path("/path/to/file.txt");  
        Configuration conf = new Configuration();  
        FileSystem fs = FileSystem.get(conf); // Create FileSystem  
        FSDataOutputStream out = fs.create(file); // Open OutputStream  
        IOUtils.copyBytes(in, out, conf); // Copy Data  
    }  
}
```

FileSystem: запись данных

- ***fs.append(path)*** – дописать к существующему файлу
 - Поддержка для HDFS
- Нельзя записать в середину файла
- ***FileSystem.create(Path)*** создает все промежуточные директории для заданного каталога (по умолчанию)
 - Если это не нужно, то надо использовать
 - *public FSDataOutputStream create(Path f, boolean overwrite)*
 - *overwrite = false*

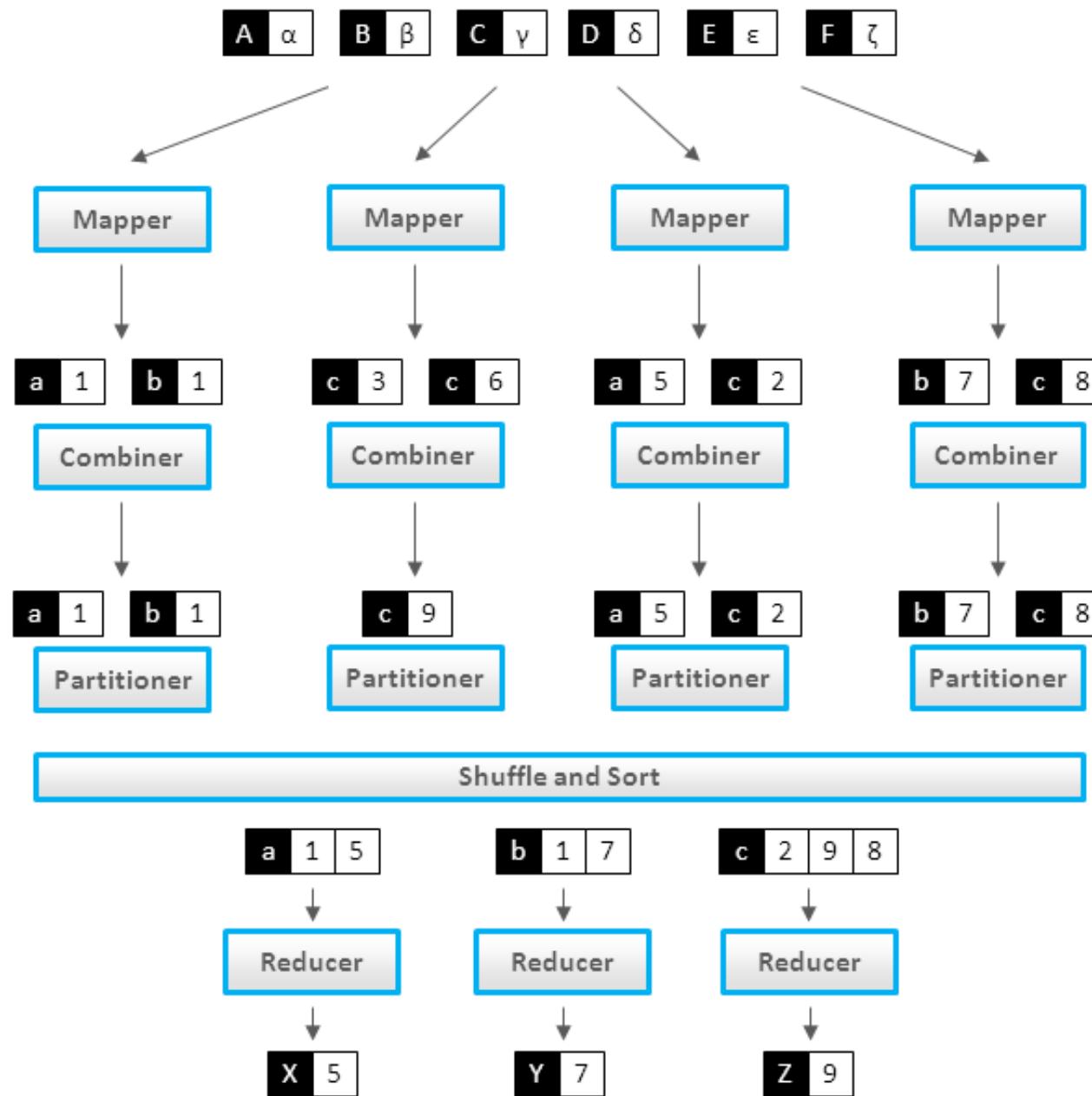
FileSystem: подстановки (globbing)

- *FileSystem* имеет поддержку матчинга имени файла по заданному паттерну используя метод *globStatus()*
 - *FileStatus [] files = fs.globStatus(glob);*
- Используется для выбора списка файлов по шаблону
- Примеры шаблонов
 - ? – любой один символ
 - * - любые 0 и больше символов
 - [abc] – любой символ из набора в скобках
 - [a-z]
 - [^a] – любой символ, кроме указанного
 - {ab,cd} – любая строка из указанных в скобках

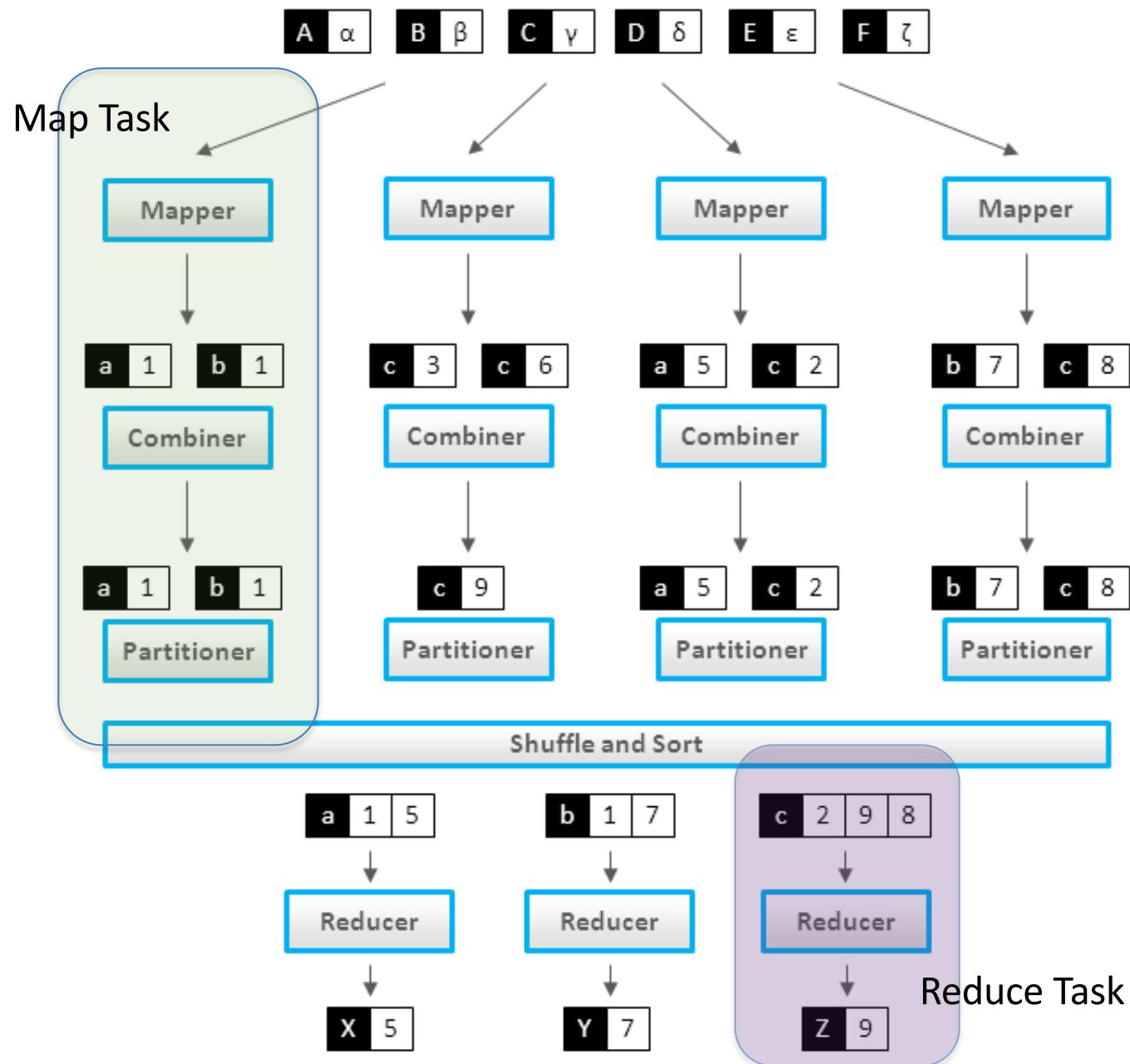
MapReduce

- Программист определяет две основные функции:
map (k_1, v_1) \rightarrow list(k_2, v_2)
reduce (k_2 , list(v_2^*) \rightarrow list(k_3, v_3)
 - Все значения с одинаковым ключом отправляются на один и тот же reducer
- ... и опционально:
partition (k_2 , number of partitions) \rightarrow partition for k_2
 - Часто просто хеш от key, напр., $hash(k_2) \bmod n$
 - Разделяет множество ключей для параллельных операций reduce
combine (k_2, v_2) \rightarrow list(k_2, v_2')
 - Мини-reducers которые выполняются после завершения фазы map
 - Используется в качестве оптимизации для снижения сетевого трафика на reduce

Общая схема MapReduce



Общая схема MapReduce



Hadoop API

- Типы **API**
 - *org.apache.hadoop.mapreduce*
 - Новое API, будем использовать в примерах
 - *org.apache.hadoop.mapred*
 - Старое API, лучше не использовать
- Класс **Job**
 - Представляет сериализованную Hadoop-задачу для запуска на кластере
 - Необходим для указания путей для *input/output*
 - Необходим для указания формата данных для *input/output*
 - Необходим для указания типов классов для *mapper, reducer, combiner* и *partitioner*
 - Необходим для указания типов значений пар *key/value*
 - Необходим для указания количества редьюсеров (но не мапперов!)

Hadoop API

- Класс ***Mapper***
 - *void setup(Mapper.Context context)*
 - Вызывается один раз при запуске таска
 - *void map(K key, V value, Mapper.Context context)*
 - Вызывается для каждой пары key/value из *input split*
 - *void cleanup(Mapper.Context context)*
 - Вызывается один раз при завершении таска
- Класс ***Reducer/Combiner***
 - *void setup(Reducer.Context context)*
 - Вызывается один раз при запуске таска
 - *void reduce(K key, Iterable<V> values, Reducer.Context context)*
 - Вызывается для каждого key
 - *void cleanup(Reducer.Context context)*
 - Вызывается один раз при завершении таска
- Класс ***Partitioner***
 - *int getPartition(K key, V value, int numPartitions)*
 - Возвращает номер партиции (reducer) для конкретного ключа

“Hello World”: Word Count

```
Map(String docid, String text):  
    for each word w in text:  
        Emit(w, 1);
```

```
Reduce(String term, Iterator<Int> values):  
    int sum = 0;  
    for each v in values:  
        sum += v;  
    Emit(term, value);
```

WordCount, Configure Job

- *Job job = Job.getInstance(getConf(), "WordCount");*
 - Инкапсулирует все, что связано с задачей
 - Контролирует процесс выполнения задачи
- Код задачи упаковывается в *jar*-файл
 - Фреймворк *MapReduce* отвечает за дистрибуцию *jar*-файла по кластеру
 - Самый простой способ определить *jar*-файл для класса это вызвать
job.setJarByClass(getClass());

WordCount, Configure Job

- Определить input
 - Может быть либо путь к файлу, директории или шаблон пути
 - Напр.: `/path/to/dir/test_*`
 - Директория преобразуется ко списку файлов внутри
 - Input определяется через класс имплементации ***InputFormat***. В данном случае ***TextInputFormat***
 - Отвечает за сплит входных данных и чтение записей
 - Определяет тип входных данных для пар key/value (***LongWritable*** и ***Text***)
 - Разбивает файл на записи, в данном случае по-строчно.
 - Каждый вызов маппера получает на вход одну строку

```
TextInputFormat.addInputPath(job, new Path(args[0]));
job.setInputFormatClass(TextInputFormat.class);
```

WordCount, Configure Job

- Определить output
 - Определить типы для *output key* и *values* для функций *map* и *reduce*
 - В случае, если типы для *map* и *reduce* отличаются, то
 - *setMapOutputKeyClass()*
 - *setMapOutputValueClass()*

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
```

WordCount, Configure Job

- Определить классы для *Mapper* и *Reducer*
 - Как минимум, надо будет реализовать эти классы

```
job.setMapperClass(WordCountMapper.class);
job.setReducerClass(WordCountReducer.class);
```

- Опционально, надо определить класс для Combiner
 - Часто он будет совпадать с классом *Reducer* (но не всегда!)

```
job.setCombinerClass(WordCountReducer.class);
```

- Запуск задачи
 - Запускает задачу и ждет окончания ее работы
 - true в случае успеха, false в случае ошибки

```
job.waitForCompletion(true)
```

WordCount, Configure Job

```
public class WordCountJob extends Configured implements Tool{
    @Override
    public int run(String[] args) throws Exception {
        Job job = Job.getInstance(getConf(), "WordCount");
        job.setJarByClass(getClass());

        TextInputFormat.addInputPath(job, new Path(args[0]));
        job.setInputFormatClass(TextInputFormat.class);

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);
        job.setCombinerClass(WordCountReducer.class);

        TextOutputFormat.setOutputPath(job, new Path(args[1]));
        job.setOutputFormatClass(TextOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        return job.waitForCompletion(true) ? 0 : 1;
    }
}
```

WordCount, Configure Job

```
public class WordCountJob extends Configured implements Tool{  
    public static void main(String[] args) throws Exception {  
        int exitCode = ToolRunner.run(  
            new WordCountJob(), args);  
        System.exit(exitCode);  
    }  
}
```

WordCount, Mapper

- Класс **Mapper** имеет 4 *generic* параметра
 - Параметры
 - input key
 - input value
 - output key
 - output value
 - Используются типы из hadoop's IO framework
 - org.apache.hadoop.io
- В задаче должен быть реализован метод **map()**
- В **map()** передается
 - пара key/value
 - Объект класса Context
 - Используется для записи в output новой пары key/value
 - Работы со счетчиками
 - Отображения статуса таска

WordCount, Mapper

```
public class WordCountMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

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

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        StringTokenizer tokenizer = new StringTokenizer(value.toString());
        while (tokenizer.hasMoreTokens()) {
            text.set(tokenizer.nextToken());
            context.write(text, one);
        }
    }
}
```

WordCount, Reducer

- Класс ***Reducer*** имеет 4 *generic* параметра
 - Параметры
 - input key
 - input value
 - output key
 - output value
 - Типы параметров из *map output* должны соответствовать *reduce input*
- MapReduce группирует пары *key/value* от мапперов по ключу
 - Для каждого ключа будет набор значений
 - *Reducer input* отсортирован по ключу
 - Сортировка значений для ключа не гарантируется
- Объект типа Context используется с той же целью, что и в ***Mapper***

WordCount, Reducer

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

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values,
                         Context context)
        throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable value : values) {
            sum += value.get();
        }
        context.write(key, new IntWritable(sum));
    }
}
```

Reducer в качестве Combiner

- Объединяет данные на стороне *Mapper* для уменьшения количества передаваемых данных на *Reducer*
 - Годится в том случае, когда тип пары key/value в output у *Reducer* такие же, как и у *Mapper*
- Фреймворк MapReduce не гарантирует вызов *Combiner*
 - Нужно только с точки зрения оптимизации
 - Логика приложения не должна зависеть от вызова вызов *Combiner*

Типы данных в Hadoop

- ***Writable***
 - Определяет протокол де- сериализации. Каждый тип в Hadoop должен быть ***Writable***
- ***WritableComparable***
 - Определяет порядок сортировки. Все ключи должны быть ***WritableComparable*** (но не значения!)
- ***Text, IntWritable, LongWritable*** и т.д.
 - Конкретные реализации для конкретных типов
- ***SequenceFiles***
 - Бинарно-закодированная последовательность пар *key/value*

Комплексные типы данных в Hadoop

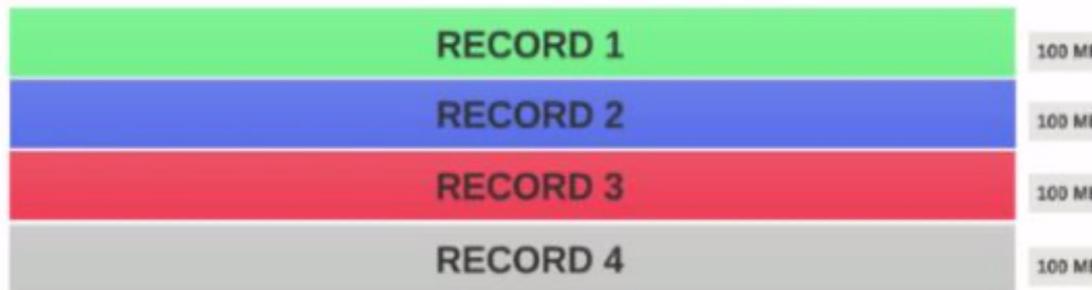
- Как можно реализовать комплексный тип?
- Простой способ:
 - Закодировать в **Text**
 - Напр., (x,10) = “x:10”
 - Для раскодирования использовать специальный метод парсинга
 - Просто, работает, но...
- Сложный (правильный) способ:
 - Определить реализацию своего типа **Writable(Comparable)**
 - Необходимо реализовать методы **readFields, write, (compareTo)**
 - Более производительное решение, но сложнее в реализации

InputSplit

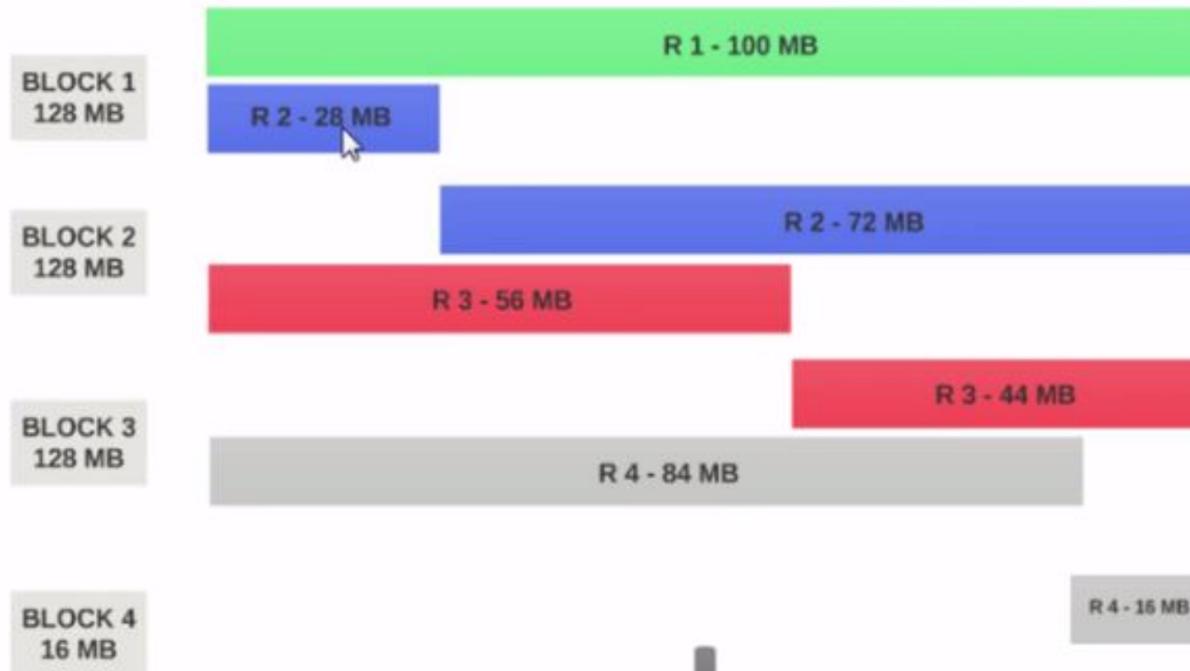
- ***Split*** – это набор логически организованных записей
 - Строки в файле
 - Строки в выборке из БД
- Каждый экземпляр *Mapper* будет обрабатывать один *split*
 - Функция *map(k, v)* обрабатывает только одну пару key/value
 - Функция *map(k, v)* вызывается для каждой записи из *split*
- Различные сплиты реализуются расширением класса ***InputSplit***
 - Hadoop предлагает много различных реализаций ***InputSplit***
 - FileSplit, TableSplit и т.д.

Split vs. Block

Файл **400 МБ**, состоящий из **4 записей**



Если HDFS **Размер блока** настроен как **128 МБ**, то 4 записи не будут равномерно распределены между блоками.



InputFormat

- Определяет формат входных данных
- Создает *input splits*
- Определяет, как читать каждый *split*
 - Разбивает каждый *split* на записи
 - Предоставляет реализацию класса **RecordReader**

```
public abstract class InputFormat<K, V> {  
    public abstract  
        List<InputSplit> getSplits(JobContext context)  
            throws IOException, InterruptedException;  
    public abstract  
        RecordReader<K,V> createRecordReader(InputSplit split,  
            TaskAttemptContext context )  
            throws IOException, InterruptedException;  
}
```

InputFormat

- Hadoop предоставляет много готовых классов-реализаций ***InputFormat***
 - TextInputFormat
 - LongWritable/Text
 - NLineInputFormat
 - *NLineInputFormat.setNumLinesPerSplit(job, 100);*
 - DBInputFormat
 - TableInputFormat (HBASE)
 - ImmutableBytesWritable/Result
 - StreamInputFormat
 - SequenceFileInputFormat
- Выбор нужного формата производится через
 - *job.setInputFormatClass(*InputFormat.class);*

OutputFormat

- Определяет формат выходных данных
- Реализация интерфейса класса
OutputFormat<K,V>
 - Проверяет *output* для задачи
 - Напр. путь в HDFS
 - Создает реализацию ***RecordWriter***
 - Непосредственно пишет данные
 - Создает реализацию ***OutputCommitter***
 - Отменяет *output* в случае ошибки таска или задачи

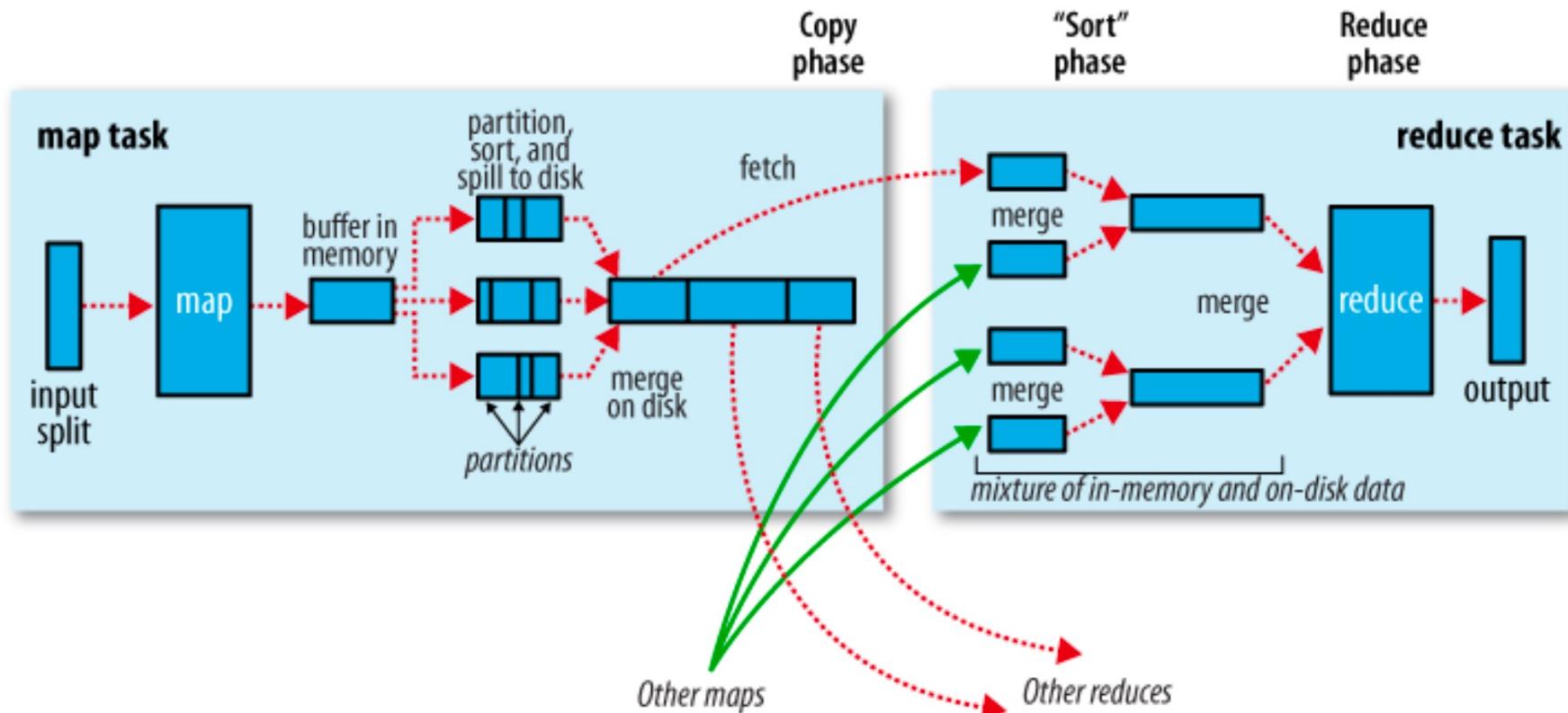
OutputFormat

- Hadoop предоставляет много готовых классов-реализаций ***OutputFormat***
 - TextOutputFormat
 - DBOutputFormat
 - TableOutputFormat (HBASE)
 - MapFileOutputFormat
 - SequenceFileOutputFormat
 - NullOutputFormat
- Выбор нужного формата производится через
 - `job.setOutputFormatClass(*OutputFormat.class);`
 - `job.setOutputKeyClass(*Key.class);`
 - `job.setOutputValueClass(*Value.class);`

Shuffle и Sort в Hadoop

- На стороне *Map*
 - Выходные данные буферизуются в памяти в циклическом буфере
 - Когда размер буфера достигает предела, данные “скидываются” (*spilled*) на диск
 - Затем все такие “сброшенные” части объединяются (*merge*) в один файл, разбитый на части
 - Внутри каждой части данные отсортированы
 - *Combiner* запускается во время процедуры объединения
- На стороне *Reduce*
 - Выходные данные от мапперов копируются на машину, где будет запущен редьюсер
 - Процесс сортировки (*sort*) представляет собой многопроходный процесс объединения (*merge*) данных от мапперов
 - Это происходит в памяти и затем пишется на диск
 - Итоговый результат объединения отправляется непосредственно на редьюсер

Shuffle и Sort в MapReduce



Process	Time ----->
User Program	MapReduce() ... wait ...
Master	Assign tasks to worker machines...
Worker 1	Map 1 Map 3
Worker 2	Map 2
Worker 3	Read 1.1 Read 1.3 Read 1.2 Reduce 1
Worker 4	Read 2.1 Read 2.2 Read 2.3 Reduce 2

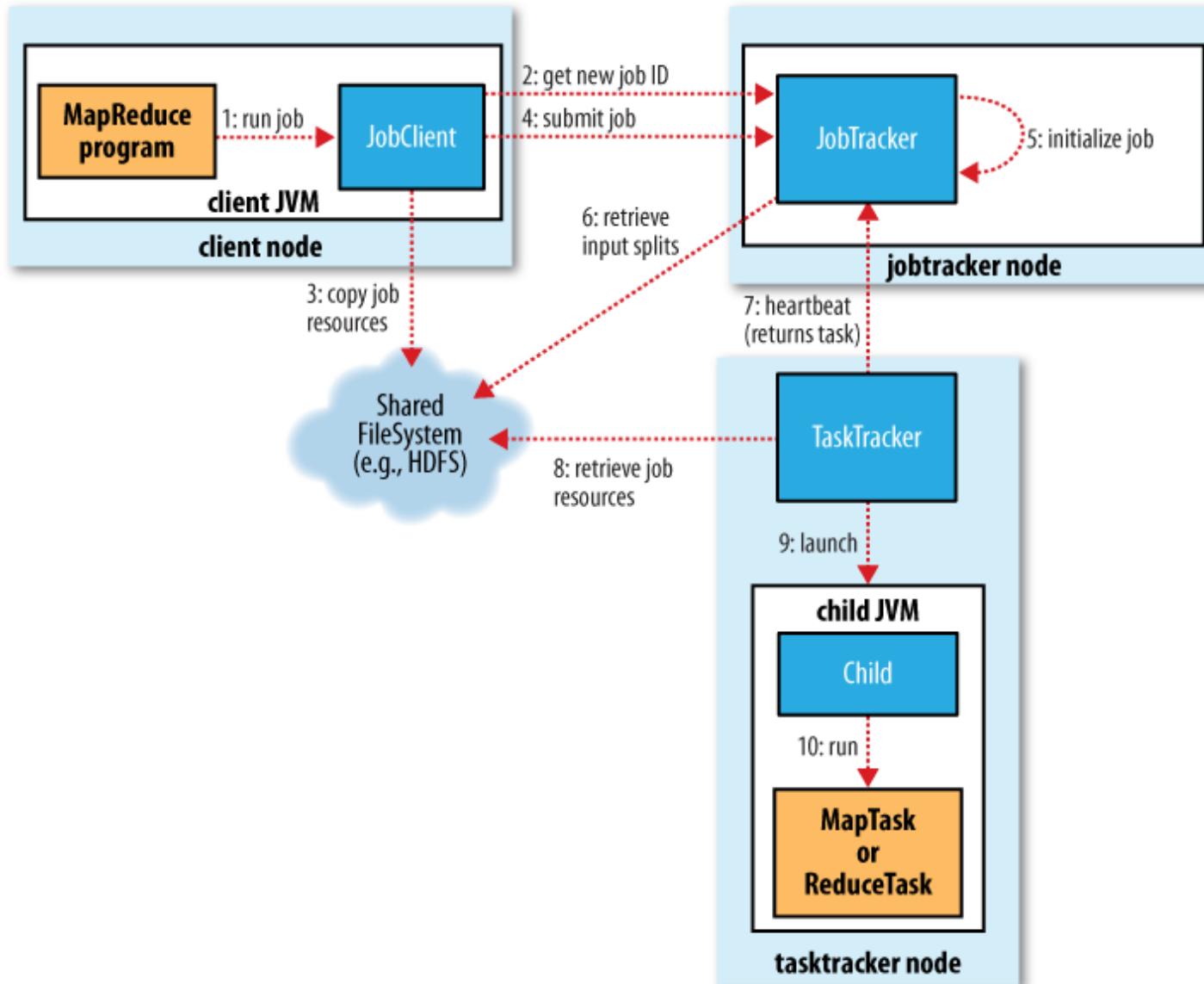
Hadoop MapReduce (ver.1)

- MapReduce работает поверх демонов
 - *JobTracker* и *TaskTracker*
- ***JobTracker***
 - Управляет запуском тасков и определяет, на каком *TaskTracker* таск будет запущен
 - Управляет процессом работы MapReduce задач (*jobs*)
 - Мониторит прогресс выполнения задач
 - Перезапускает зафайленные или медленные таски
- MapReduce имеет систему ресурсов основанную на слотах (*slots*)
 - На каждом *TaskTracker* определяется, сколько будет запущено слотов
 - Таск запускается в одном слоте
 - M мапперов + R редьюсеров = N слотов
 - Для каждого слота определяется кол-во потребляемой ОЗУ
 - Такая модель не слишком удобная с точки зрения утилизации ресурсов

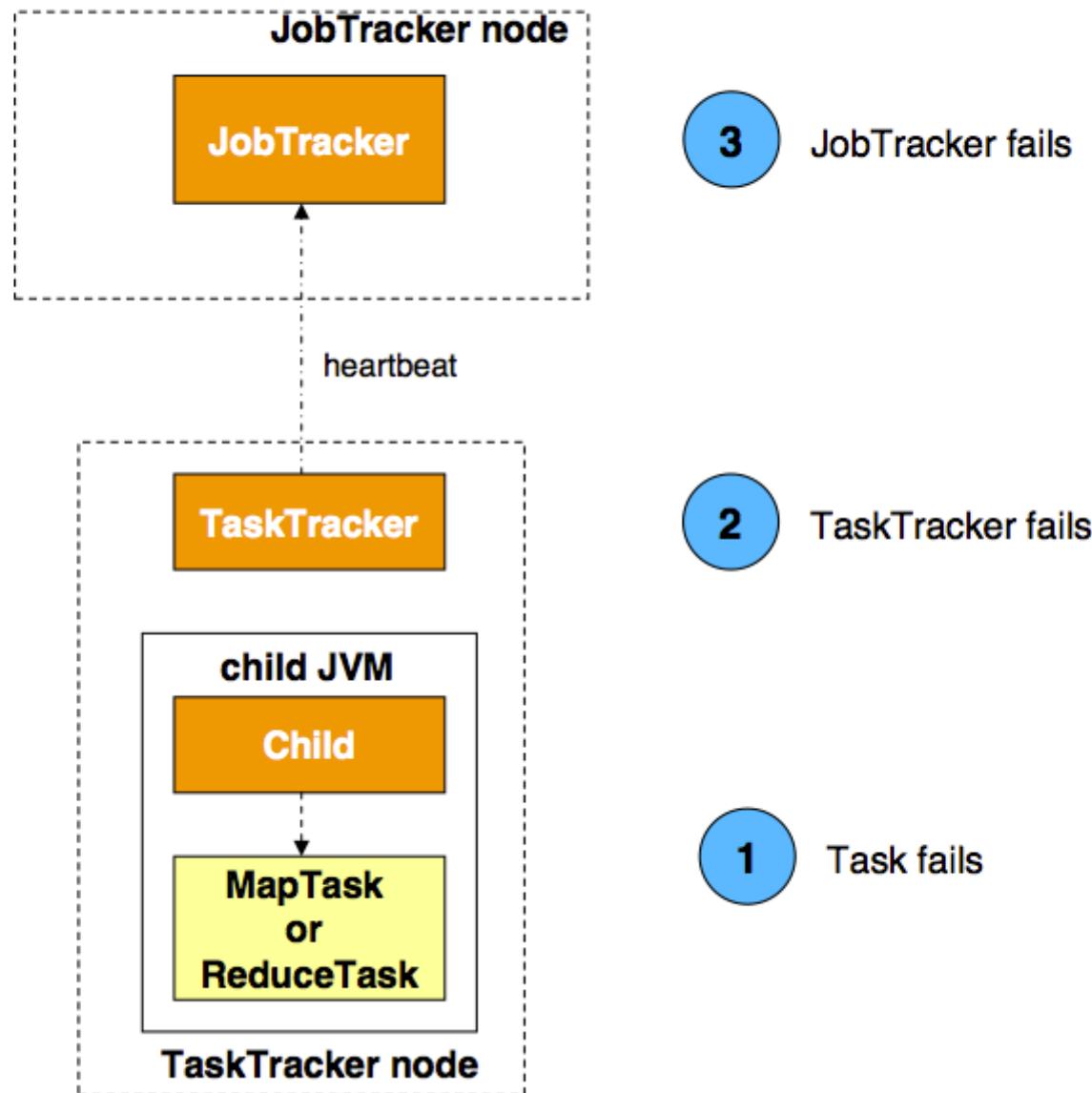
MapReduce “Runtime”

- Управление запуском задач
 - Назначает воркерам таски *map* или *reduce*
- Управление “*data distribution*”
 - Перемещает код к данным
 - Запускает таски (по возможности) локально с данными
- Управление синхронизацией
 - Собирает, сортирует и объединяет промежуточные данные
- Управление обработкой ошибкой и отказов
 - Определяет отказ воркера и перезапускает таск
- Все работает поверх распределенной FS

Запуск задания в классическом фреймворке



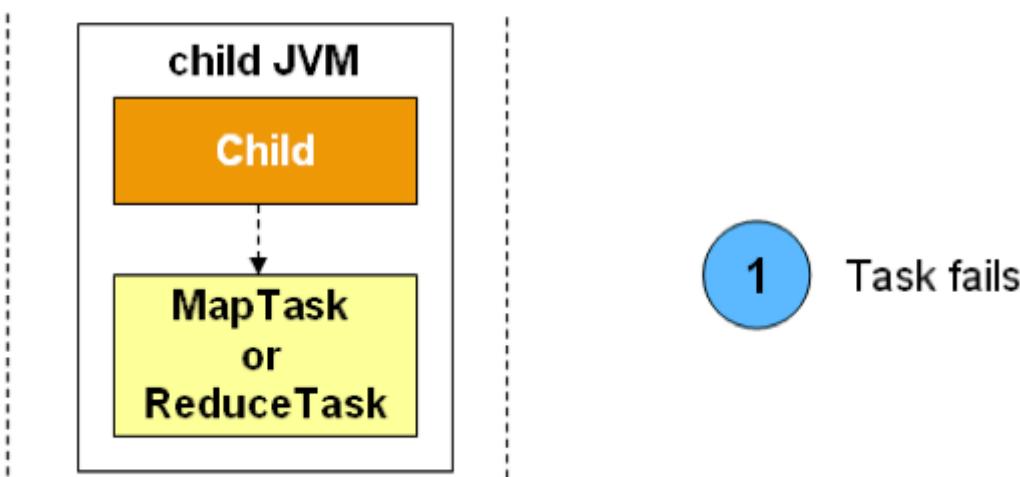
Fault tolerance



Fault tolerance

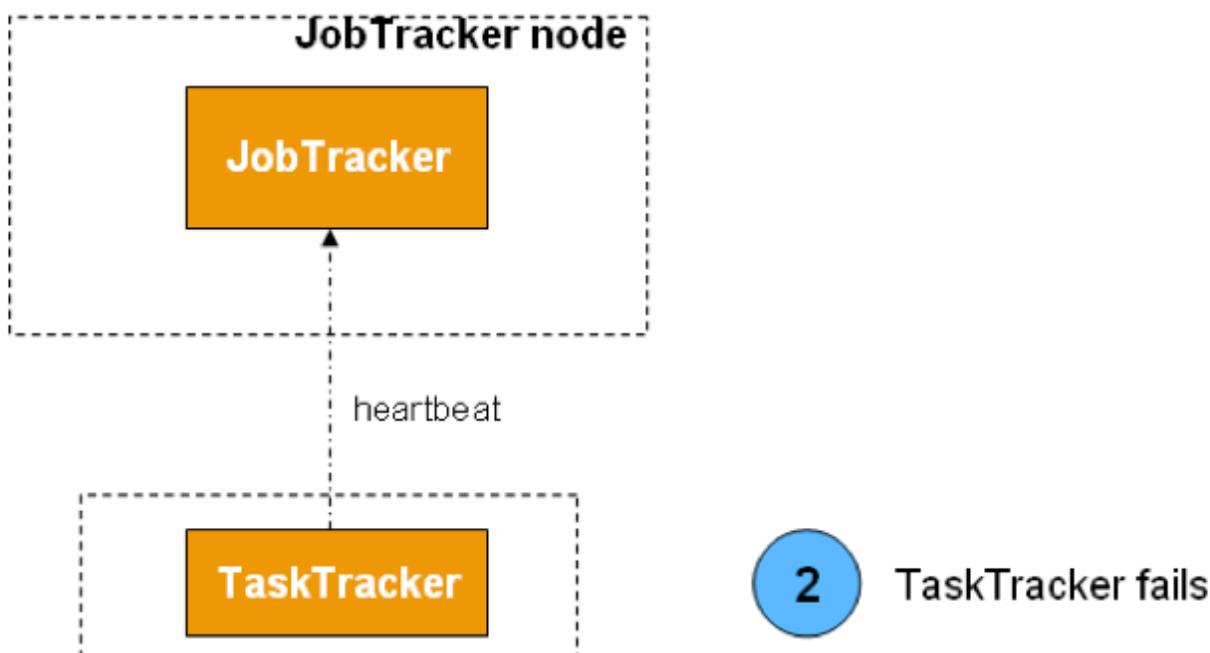
- **Task Failure**

- If a child task fails, the child JVM reports to the TaskTracker before it exits. Attempt is marked failed, freeing up slot for another task.
- If the child task hangs, it is killed. JobTracker reschedules the task on another machine.
- If task continues to fail, job is failed.



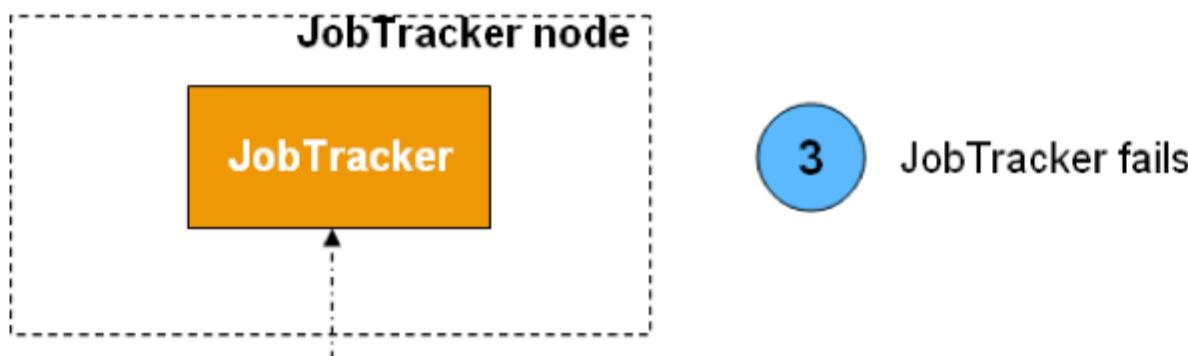
Fault tolerance

- **TaskTracker Failure**
 - JobTracker receives no heartbeat
 - Removes TaskTracker from pool of TaskTrackers to schedule tasks on.



Fault tolerance

- **JobTracker Failure**
 - Single point of failure. Job fails



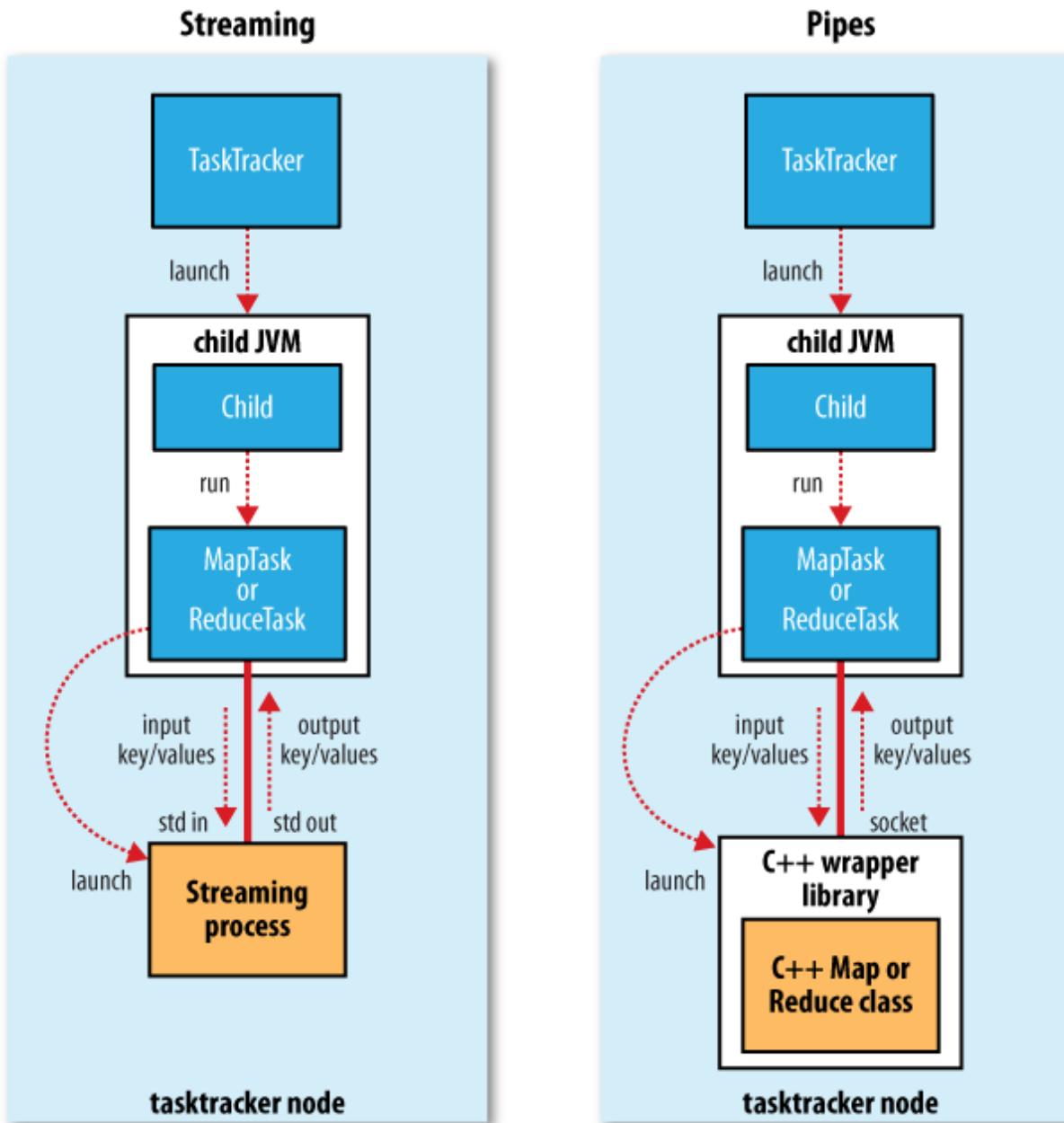
Hadoop Streaming

- Используется стандартный механизм ввода/вывода в Unix для взаимодействия программы и Hadoop
- Позволяет разрабатывать MR задачи почти на любом языке программирования, который умеет работать со стандартным вводом/выводом
- Обычно используется:
 - Для обработки текста
 - При отсутствии опыта программирования на Java
 - Быстрого написания прототипа

Streaming в MapReduce

- На вход функции *map()* данные подаются через стандартный ввод
- В *map()* обрабатываются они построчно
- Функция *map()* пишет пары *key/value*, разделяемые через символ табуляции, в стандартный вывод
- На вход функции *reduce()* данные подаются через стандартный ввод, отсортированный по ключам
- Функция *reduce()* пишет пары *key/value* в стандартный вывод

Использование задач Streaming и Pipes



WordCount на Python

Map: countMap.py

```
#!/usr/bin/python
import sys

for line in sys.stdin:
    for token in line.strip().split(" "):
        if token: print token + '\t1'
```

WordCount на Python

Reduce: countReduce.py

```
#!/usr/bin/python
import sys

(lastKey, sum)=(None, 0)

for line in sys.stdin:
    (key, value) = line.strip().split("\t")
    if lastKey and lastKey != key:
        print lastKey + '\t' + str(sum)
        (lastKey, sum) = (key, int(value))
    else:
        (lastKey, sum) = (key, sum + int(value))
if lastKey:
    print lastKey + '\t' + str(sum)
```

Запуск и отладка

Тест в консоли перед запуском

```
$ cat test.txt | countMap.py | sort | countReduce.py
```

Запуск задачи через Streaming Framework

```
yarn jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*jar \
-D mapred.job.name="WordCount Job via Streaming" \
-files countMap.py, countReduce.py \
-input text.txt \
-output /tmp/wordCount/ \
-mapper countMap.py \
-combiner countReduce.py \
-reducer countReduce.py
```

Python vs. Java

```
#!/usr/bin/python
import sys

for line in sys.stdin:
    for token in line.strip().split(" "):
        if token: print token + '\t'

#!/usr/bin/python
import sys

(lastKey, sum)=(None, 0)

for line in sys.stdin:
    (key, value) = line.strip().split("\t")
    if lastKey and lastKey != key:
        print lastKey + '\t' + str(sum)
        (lastKey, sum) = (key, int(value))
    else:
        (lastKey, sum) = (key, sum + int(value))
if lastKey:
    print lastKey + '\t' + str(sum)
```

```
public class WordCountJob extends Configured implements Tool{
    static public class WordCountMapper
        extends Mapper<LongWritable, Text, Text, IntWritable> {

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

        @Override
        protected void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
            StringTokenizer tokenizer = new StringTokenizer(value.toString());
            while (tokenizer.hasMoreTokens()) {
                text.set(tokenizer.nextToken());
                context.write(text, one);
            }
        }
    }

    static public class WordCountReducer
        extends Reducer<Text, IntWritable, Text, IntWritable> {

        @Override
        protected void reduce(Text key, Iterable<IntWritable> values,
                             Context context)
            throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable value : values) {
                sum += value.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }

    @Override
    public int run(String[] args) throws Exception {
        Job job = Job.getInstance(getConf(), "WordCount");
        job.setJarByClass(getClass());

        TextInputFormat.addInputPath(job, new Path(args[0]));
        job.setInputFormatClass(TextInputFormat.class);

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);
        job.setCombinerClass(WordCountReducer.class);

        TextOutputFormat.setOutputPath(job, new Path(args[1]));
        job.setOutputFormatClass(TextOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        return job.waitForCompletion(true) ? 0 : 1;
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(
            new WordCountJob(), args);
        System.exit(exitCode);
    }
}
```

Отладка в Streaming

- Можно обновлять счетчики и статус таска
 - Для этого надо отправить строку в *standard error* в формате “*streaming reporter*”

reporter:counter:<counter_group>,<counter>,<increment_by>

```
#!/usr/bin/python
import sys

for line in sys.stdin:
    for token in line.strip().split(" "):
        if token:
            sys.stderr.write("reporter:counter:Tokens,Total,1\n")
            print token[0] + '\t1'
```

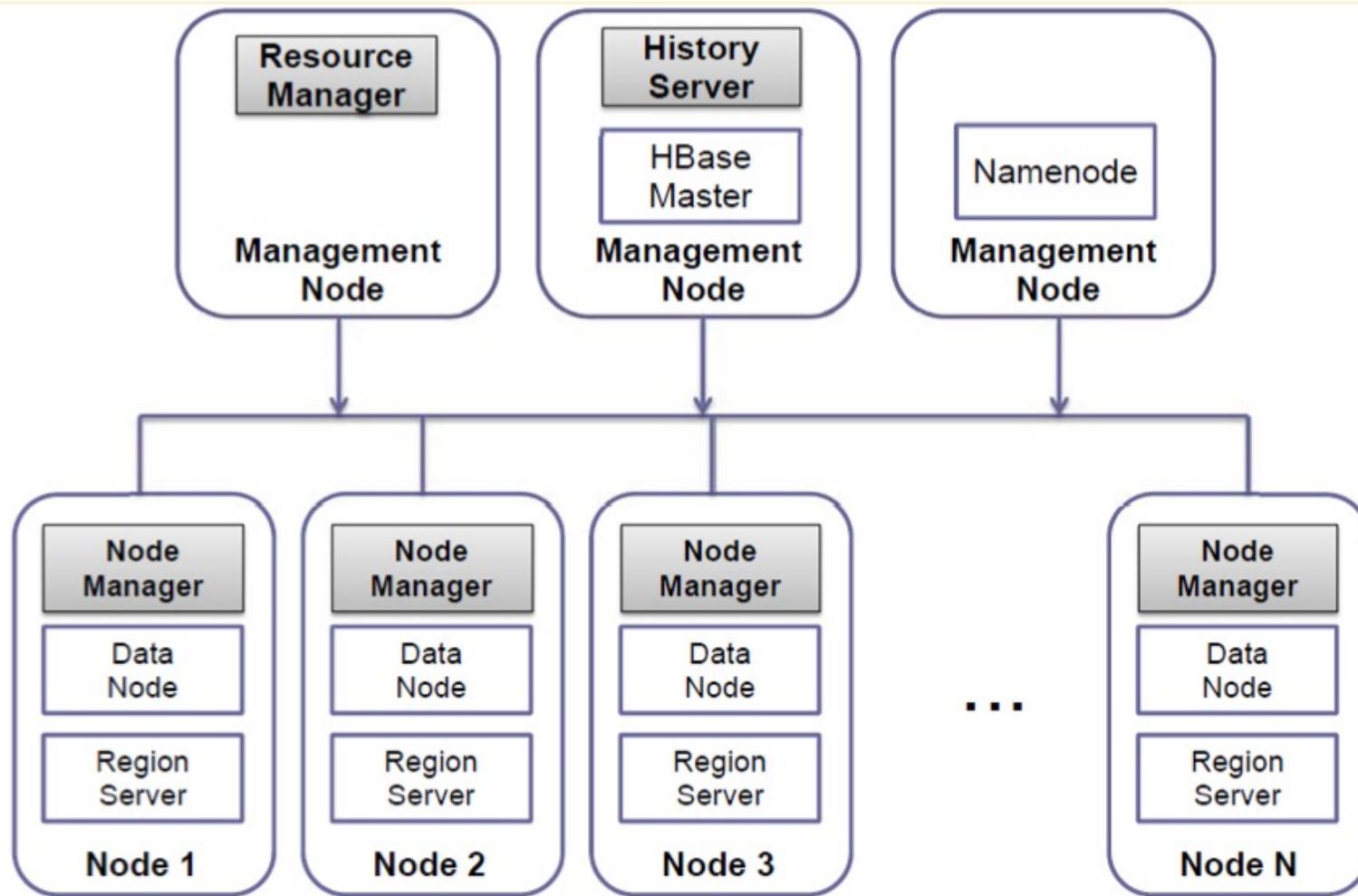
YARN

- **Yet Another Resource Negotiator (YARN)**
- **Отвечает за**
 - Cluster Resource Management
 - Scheduling
- Различные приложения могут быть запущены на **YARN**
 - MapReduce как один из вариантов
 - <http://wiki.apache.org/hadoop/PoweredByYarn>
- Также упоминается/называется как **MapReduce2.0, NextGenMapReduce**
 - Это не совсем правильные названия, т.к. YARN не привязан к MapReduce

YARN vs. Old MapReduce

- Перед YARN в Hadoop были демоны JobTracker и TaskTracker
 - JobTracker отвечает за управление ресурсами и мониторинг/управление тасками
 - Обработка failed tasks
 - Task Bookkeeping
- Подход, основанный на JobTracker, имеет свои минусы
 - Scalability Bottleneck – 5,000+ nodes, 40 000 tasks
 - Гибкость управления *cluster resource sharing and allocation*
 - Slot based approach
 - Напр., 12 слотов на машину независимо, насколько от «размера» таска
 - Единственная парадигма MapReduce
- В 2010 году в **Yahoo!** Начали разработку нового поколения MapReduce - **YARN**

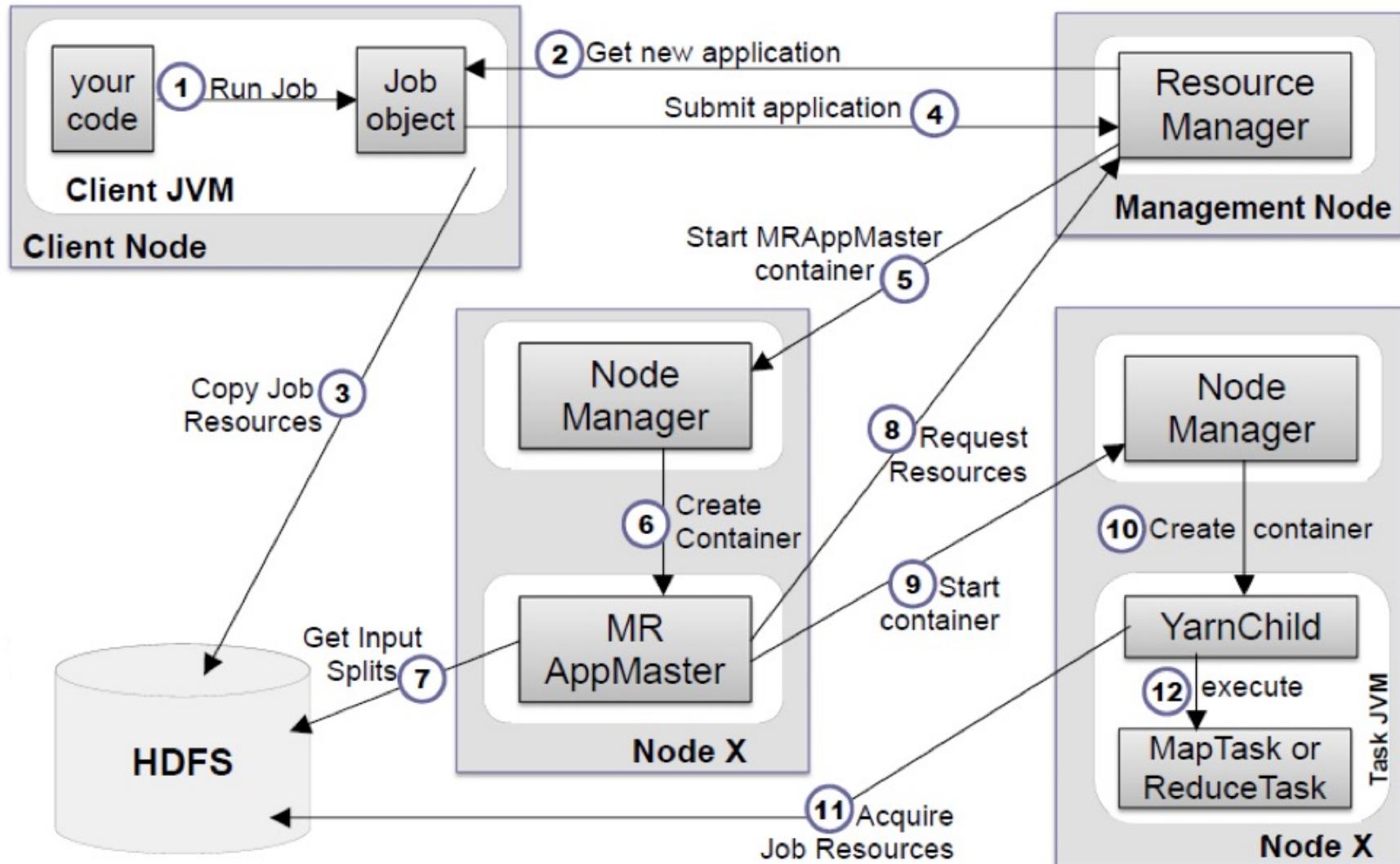
Пример схемы демонов YARN



Компоненты MapReduce на YARN

- **Client**
 - Отправляет и запускает MR-задачу
- **Resource Manager**
 - Контролирует использование ресурсов по всему Hadoop-кластеру
- **Node Manager**
 - Запускается на каждой ноде кластера
 - Создает **execution container**
 - Мониторит его использование
- **MapReduce Application Master**
 - Координирует и управляет MR-задачи
 - Общается с **Resource Manager** про шедулинг тасков
 - Таски запускаются через **NodeManager**
- **HDFS**
 - Общее место хранилища ресурсов и данных задач между компонентами YARN

Выполнение MR-задачи на YARN



Выполнение MR-задачи на YARN

- Клиент отправляет MR-задачу используя объект класса *Job*
 - Клиент запускается в своей собственной JVM
- Код задачи взаимодействует с **Resource Manager** для получения мета-данных приложения (в частности, **application id**)
- Код задачи перемещает все данные, необходимые для задачи, в HDFS для того, чтобы они были доступны в последствии
- Код задачи отправляет приложение на **Resource Manager**
- **Resource Manager** выбирает **Node Manager** с доступными ресурсами и запрашивает контейнер для **MRAppMaster**
- **Node Manager** выделяет контейнер для **MRAppMaster**
 - **MRAppMaster** будет заниматься выполнением и координацией MR-задачи
- **MRAppMaster** берет необходимые данные для задачи из HDFS

Выполнение MR-задачи на YARN

- **MRAppMaster** получает информацию от **Resource Manager** о доступных ресурсах
 - **Resource Manager** выберет **Node Manager** с наибольшим кол-вом ресурсов
- **MRAppMaster** говорит выбранному **Node Manager** запустить таски Map и Reduce таски
- **NodeManager** создает контейнеры **YarnChild**, которые координируют и выполняют таски
- **YarnChild** получает данные для задачи из HDFS которые нужны для выполнения тасков
- **YarnChild** выполняет таски Map и Reduce

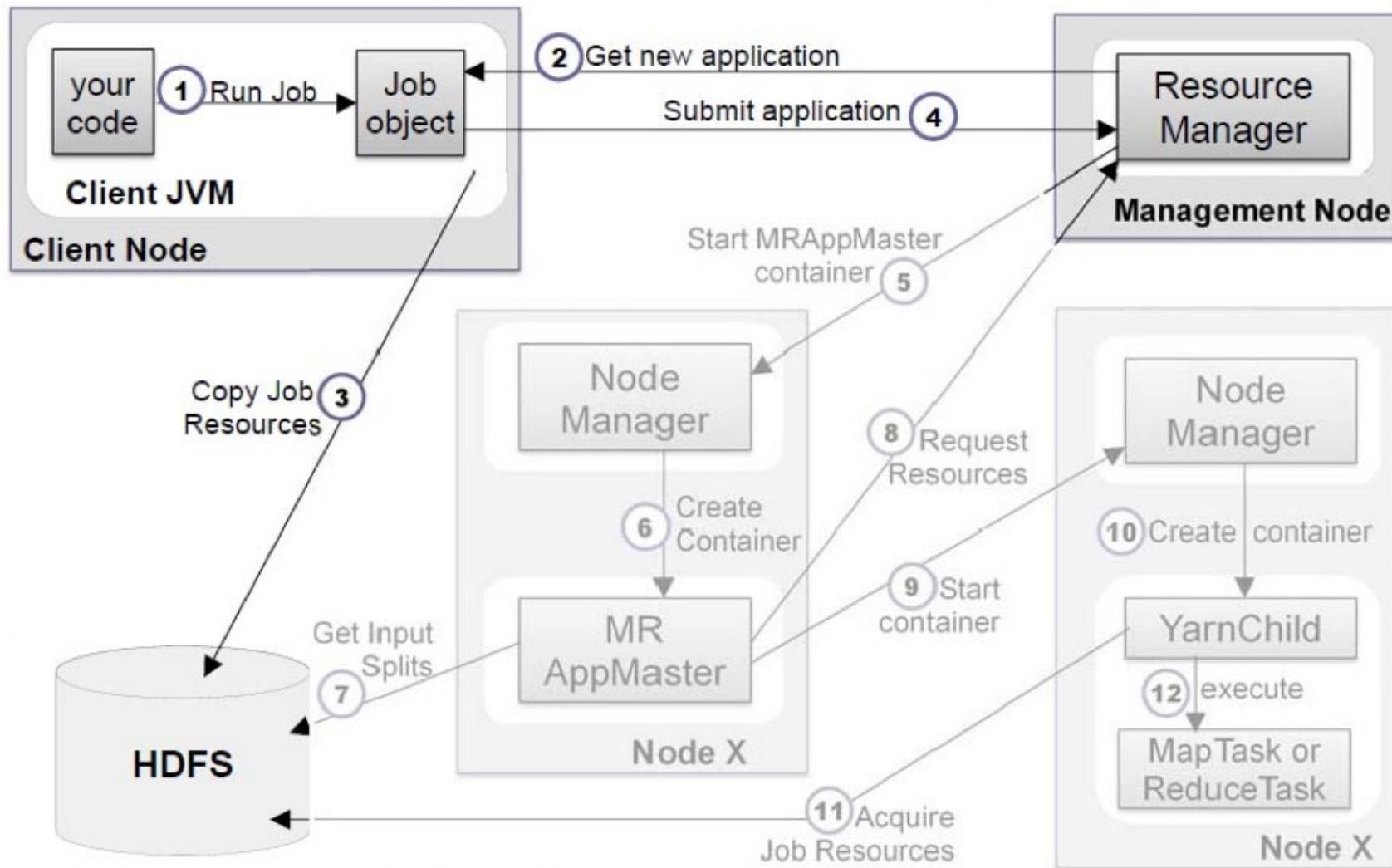
Запуск MapReduce-задачи

- Используется класс для конфигурации задачи
 - *org.apache.hadoop.mapreduce.Job*
- Запуск задачи
 - *job.waitForCompletion(true)*
- Активация протокола YARN
 - *mapreduce.framework.name= yarn*

Запуск MapReduce-задачи, шаги

- Запрашивается **Application Id** от **Resource Manager**
- **Job Client** проверяет спецификации **output** задачи
- Рассчитывается **Input Splits**
- **Job Resources** копируются в HDFS
 - Jar files, configurations, input splits
- Запуск задачи

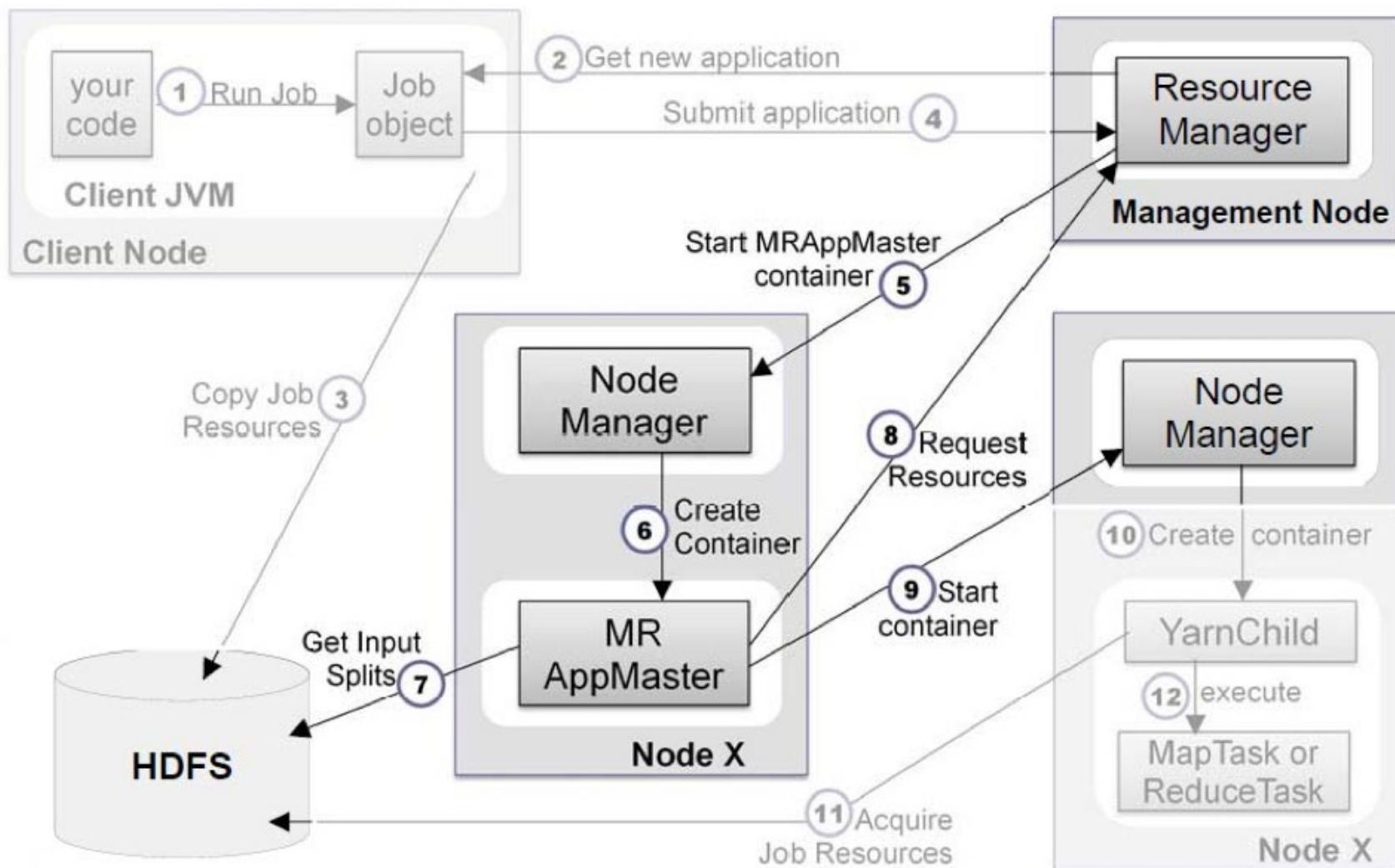
Запуск MapReduce-задачи, шаги



Инициализация задачи, шаги

- **Resource Manager** получает запрос на запуск нового приложения
- **Resource Manager** делегирует это своему внутреннему компоненту – **Scheduler**
 - Существует несколько типов шедулеров
- **Scheduler** запрашивает контейнер для процесса **Application Master**
 - Application Master для MapReduce это MRAppMaster
- **MRAppMaster** инициализирует свои внутренние объекты и выполняет задачу, общаясь с **Resource Manager**

Инициализация MRAppMaster, шаги



Инициализация MRAppMaster, шаги

- Создает внутренние объекты для мониторинга процесса
- Запрашивает **Input Splits**
 - Были созданы клиентов и лежат в HDFS
- Создает таски
 - Мар-таск на сплит
 - Кол-во Reduce-тасков указывается в *mapreduce.job.reduces*
- Решает, как запускать таски
 - В случае небольшой задачи все таски будут запущены в рамках одной *MRAppMaster JVM*
 - Такие задачи называются “*uberized*” или “*uber*”
 - Выполняет таски на *Node Manager*

MRAppMaster и Uber Job

- Если задача слишком маленькая, то **MRAppMaster** будет запускать таски **map** и **reduce** внутри одной JVM
 - Идея в том, что затраты на осуществление распределенности таких тасков будет выше, чем запуск их паралельно
- Задача будет являться **uber** если встречаются все условия:
 - Меньше 10 мапперов
 - *mapreduce.job.ubertask.maxmaps*
 - Один редьюсер
 - *mapreduce.job.ubertask.maxreduces*
 - Размер входных данных меньше размера 1 HDFS-блока
 - *mapreduce.job.ubertask.maxbytes*
- Запуск **uber**-задач можно отключить
 - *mapreduce.job.ubertask.enable=false*

Task Assignment

- Только для не-Uber задач
- **MRAppMaster** определяет контейнер для тасков map и reduce от **Resource Manager**. Запрос включает:
 - Информацию о *data locality* (hosts & racks), которая рассчитывается в *InputFormat* и сохраняется в *InputSplits*
 - Требования по памяти для тасков
- **Scheduler** в **Resource Manager** использует предоставленную информацию для того, чтобы решить, где размещать таски
 - В идеале таск размещается на той же ноде, где и лежат данные для обработки.
 - План Б – внутри той же стойки (rack)

Fine-Grained Memory Model

- В YARN администраторы и разработчики имеют большой контроль над управлением памятью
 - NodeManager
 - Обычно один инстанс на машину
 - Task Containers, которые запускают задачи
 - Множество задач может быть запущено на одном NodeManager
 - Scheduler
 - JVM Heap
 - Память для кода разработчиков
 - Virtual Memory
 - Предотвращает монополизацию машины задачом

Memory Model: Node Manager

- **Node Manager** создает контейнеры, которые запускают задачи map и reduce
 - Сумма всех контейнеров не может превышать заданный лимит
 - Node Manager не будет создавать контейнер если недостаточно доступной памяти
- Лимиты на размер выделенной памяти определяются для каждого *Node Manager*
 - *yarn.nodemanager.resource.memory-mb* в *yarn-default.xml*
 - По-умолчанию 8,192МВ
 - Задается один раз при запуске

Memory Model: Task Memory

- Контролирует лимит физической памяти для каждой задачи
 - Лимит физической памяти, которую таски могут аллоцировать
 - Весь размер используемой физической памяти должен укладываться в конфигурируемое значение
 - *Container Memory Usage = JVM Heap Size + JVM Perm Gen + Native Libraries + Memory used by spawned processes*
 - Таск убивается, если он превышает размер разрешенной физической памяти
- Определяется следующими параметрами:
 - *mapreduce.map.memory.mb* для map tasks
 - *mapreduce.reduce.memory.mb* для reduce tasks
 - Default memory = 1024

Memory Model: JVM Heap

- Помним
 - *Container Memory Usage = JVM Heap Size + JVM Perm Gen + Native Libraries + Memory used by spawned processes*
- **JVM Heap size** может быть определен через:
 - mapreduce.reduce.java.opts
 - mapreduce.map.java.opts

Пример: **mapreduce.map.java.opts=-Xmx2G**

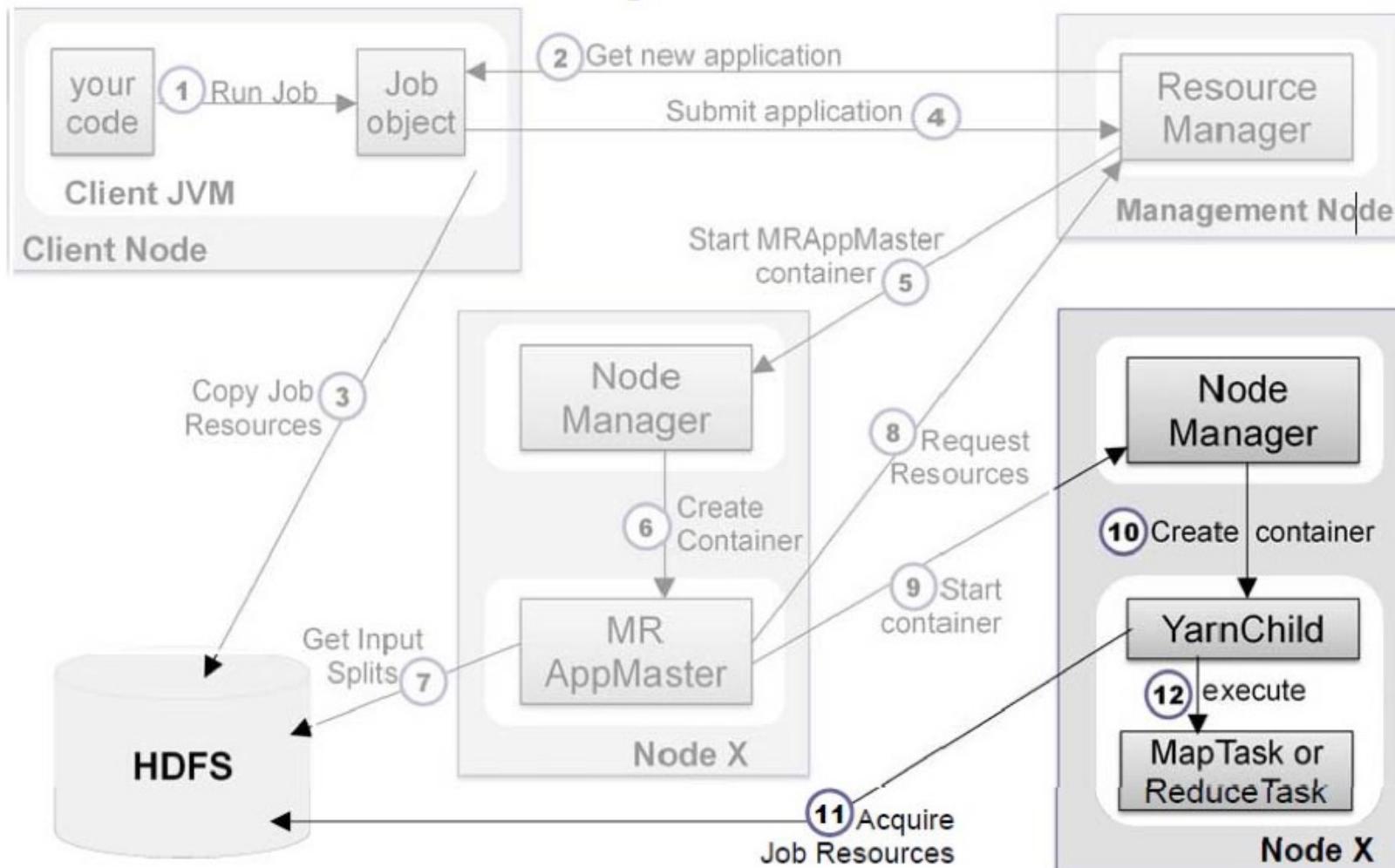
Memory Model: Virtual Memory

- Из документации к *top*
 - "Virtual Memory includes all code, data and shared libraries plus pages that have been swapped out"
- Аллокации **Virtual Memory** ограничиваются **Node Manager**
 - Таск убивается, если превышает размер доступной *Virtual Memory*
- Задается как множитель к лимиту физической памяти
 - По-умолчанию, это 2.1 от размера физической памяти контейнера
 - Напр., если размер контейнера задан в 1Гб физической памяти, тогда лимит виртуальной памяти будет 2.1Гб
 - *yarn.nodemanager.vmem-rmem-ratio* в *yarn-site.xml*
 - Задается один раз при запуске

Memory Model: Пример

- Пусть мы хотим задать heap size для map-таска 512Мб и reduce-таска 1ГБ
- Client's Job Configuration
 - Heap Size:
 - mapreduce.map.java.opts=-Xmx512
 - mapreduce.reduce.java.opts=-Xmx1G
 - Container Limit, требуется extra 512MB сверх Heap space
 - mapreduce.map.memory.mb=1024
 - mapreduce.reduce.memory.mb=1536
- YARN NodeManager Configuration – yarn-site.xml
 - 10 Gigs на NodeManager => 10 mappers или 6 reducers (или некоторая комбинация)
 - yarn.nodemanager.resource.memory-mb=10240
 - Установить свойство Scheduler для выделения 512MB increments
 - yarn.scheduler.capacity.minimum-allocation-mb=512
 - Virtual Memory limit = 2.1 заданной физической памяти
 - 2150.4MB для Map tasks
 - 3225.6MB для Reduce tasks

MapReduce Task Execution Components



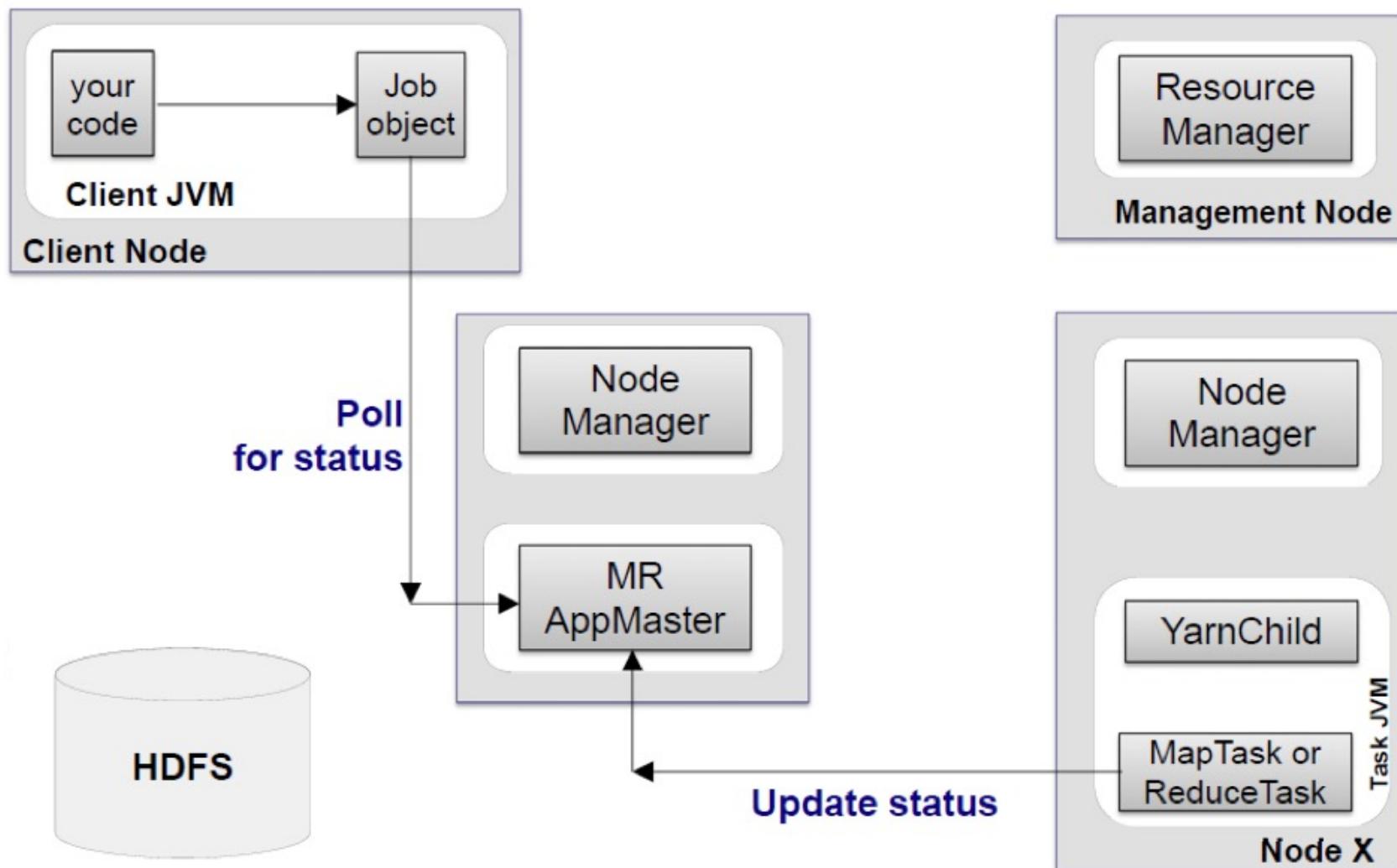
Task Execution

- **MRAppMaster** делает запрос к **Node Manager** на запуск контейнера
 - Containers и Node Manager(ы) уже были выбраны на предыдущем шаге
- Для каждого таска Node Manager стартует контейнер
 - java-процесс **YarnChild** как основной класс
- **YarnChild** запускается в отдельно выделенной JVM
- **YarnChild** копирует локально файлы ресурсов
 - Configuration, jars и .тд.
- **YarnChild** выполняет таски map или reduce

Status Updates

- Таск отправляет статус на **MRAppMaster**
 - Polling каждый 3 сек
- **MRAppMaster** аккумулирует и агрегирует информацию для определения текущего статуса задачи
 - Определяет, что задача выполнилась
- Клиент (**Job object**) запрашивает (poll) **MRAppMaster** обновленный статус
 - По-умолчанию, каждую секунду
 - *mapreduce.client.progressmonitor.pollinterval*
- **Resource Manager Web UI** отображает все запущенные приложения YARN, где каждый ссылается на **Web UI of Application Master**
 - В нашем случае MRAppMaster Web UI

MapReduce Status Updates



Resource Manager Web UI



Logged in as: dr.who

All Applications

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
3	0	0	3	0	0 B	0 B	0 B	0	0	1	0	0

Show 20 entries Search:

ID	User	Name	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1374346532317_0003	shumin	QuasiMonteCarlo	default	Sat, 20 Jul 2013 19:05:02 GMT	Sat, 20 Jul 2013 19:18:54 GMT	FAILED	FAILED		History
application_1374346532317_0002	shumin	QuasiMonteCarlo	default	Sat, 20 Jul 2013 18:57:42 GMT	Sat, 20 Jul 2013 19:00:31 GMT	FINISHED	SUCCEEDED		History
application_1374346532317_0001	shumin	QuasiMonteCarlo	default	Sat, 20 Jul 2013 18:56:03 GMT	Sat, 20 Jul 2013 18:56:29 GMT	FINISHED	SUCCEEDED		History

Showing 1 to 3 of 3 entries First Previous 1 Next Last

Failures

- Могут случиться в:
 - Tasks
 - Application Master (MRAppMaster)
 - Node Manager
 - Resource Manager

Task Failures

- Наиболее часто встречается и проще исправить
- **Task exceptions** и **JVM crashes** передаются на **MRAppMaster**
 - *Attempt* (не таск!) помечается как “failed”
- “Подвисшие” таск будут найдены и killed
 - *Attempt* помечается как “failed”
 - Контролируется через *mapreduce.task.timeout*
- Таск будет считаться failed после 4x попыток (*attempts*)
 - *mapreduce.map.maxattempts*
 - *mapreduce.reduce.maxattempts*

Application Master Failures

- Приложение, выполняемое на **Application Master**, может быть перезапущено
 - По-умолчанию, оно не будет перезапущено и завершиться после одного failure
 - *yarn.resourcemanager.am.max-retries*
- **Resource Manager** получает **heartbeats** от **Application Master (MRAppMaster)** и может его перезапустить в случае ошибки
- Перезапущенный **Application Master** может восстановить последнее состояние тасков
 - Завершенные таски не будут запущены заново
 - *yarn.app.mapreduce.am.job.recovery.enable=true*

Node Manager Failure

- **Failed Node Manager** перестанет отправлять heartbeat-сообщения на **Resource Manager**
- **Resource Manager** добавит в **black list** тот **Node Manager**, который не отправлял отчет более 10 мин
 - *yarn.resourcemanager.nm.liveness-monitor.expiryinterval-ms*
- Таски на **failed Node Manager** восстанавливаются и отправляются на рабочий **Node Manager**

Node Manager Blacklisting

- **MRAppMaster** может добавить в black list **Node Manager**, если на этой ноде большое число failure
- **MRAppMaster** попробует перезапустить таски на нормальных нодах
- Black list не аффектит другие задачи
- По-умолчанию должно произойти 3 failre на ноде чтобы попасть в blacklist
 - *mapreduce.job.maxtaskfailures.per.tracker*

Resource Manager Failures

- Наиболее серьезная проблема, может привести к *downtime*
 - Задачи и таски не смогут запускаться
- **Resource Manager** был спроектирован для автоматического восстановления в таких случаях
 - На данный момент это в процессе разработки
 - Сохраняет состояния в постоянное хранилище
 - Ожидается реализация через ZooKeeper

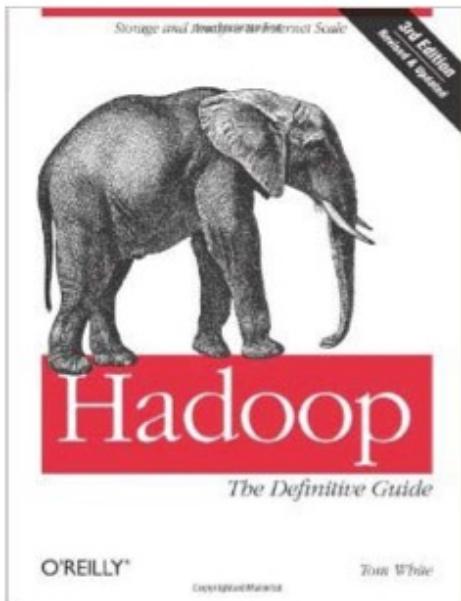
Job Scheduling

- По-умолчанию, используется **FIFO**
 - Поддержка нескольких приоритетов:
 - `VERY_LOW`, `LOW`, `NORMAL`, `HIGH` и `VERY_HIGH`
 - Два пути для задания приоритета
 - `mapreduce.job.priority property`
 - `job.setPriority(JobPriority.HIGH)`
- Два основных типа шедулера
 - CapacityScheduler
 - Общий, многоцелевой кластер
 - Гарантируется емкость ресурсов (*capacity*)
 - FairScheduler
 - Гарантирует честное распределение ресурсов между приложениями

Powered by YARN

- **Apache Giraph**
 - Итеративная система обработки графов
 - Facebook обрабатывает триллион ребер за ~4 мин
- **Spark**
 - Платформа для быстрой аналитики данных
- **Apache HAMA**
 - Фреймворк для массивных научных вычислений над матрицами, графами и сетями
- **Open MPI**

КНИГИ

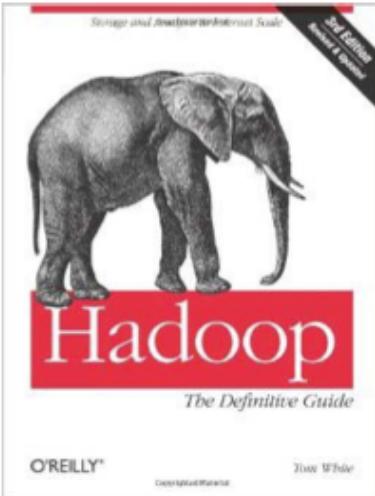


Hadoop: The Definitive Guide
Tom White (Author)
O'Reilly Media; 3rd Edition

Chapter 3: The Hadoop
Distributed Filesystem

Chapter 4: Hadoop I/O

КНИГИ



Hadoop: The Definitive Guide

Tom White (Author)

O'Reilly Media; 3rd Edition

[Chapter 2: MapReduce](#)

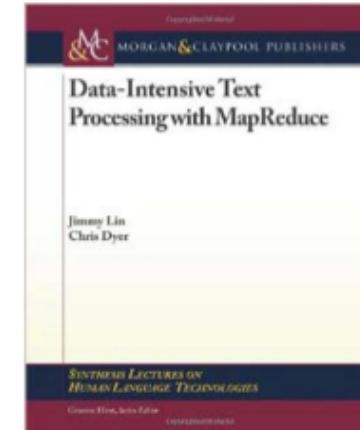
[Chapter 5: Developing a MapReduce Application](#)

[Chapter 7: MapReduce Types and Formats](#)

Data-Intensive Text Processing with MapReduce

Jimmy Lin and Chris Dyer (Authors) (April, 2010)

[Chapter2: MapReduce Basics](#)



Парадигма Map Reduce

Алгоритмы и паттерны

WordCount

- **Описание проблемы**
 - Есть коллекция документов
 - Каждый документ – это набор термов (слов)
 - Необходимо подсчитать кол-во вхождений каждого терма во всех документах
- **Дополнительно**
 - Функция может быть произвольной
 - Напр., файл лога содержит время ответа.
Необходимо подсчитать среднее время.

WordCount, baseline

class Mapper

method Map (docid id, doc d)

for all term t in doc d do

Emit(term t, count 1)

class Reducer

method Reduce (term t, counts [c1, c2,...])

sum = 0

for all count c in [c1, c2,...] do

sum = sum + c

Emit(term t, count sum)

WordCount, “In-mapper combining”, v.1

- Минусы baseline
 - Много лишних счетчиков от *Mapper*
 - Агрегируем их для каждого документа

```
class Mapper
    method Map (docid id, doc d)
        H = new AssociativeArray
        for all term t in doc d do
            H{t} = H{t} + 1
        for all term t in H do
            Emit(term t, count H{t})
```

WordCount, Combiner

- Для всех документов *Mapper* используем *Combiner*

```
class Mapper
    method Map (docid id, doc d)
        for all term t in doc d do
            Emit(term t, count 1)

class Combiner
    method Combine (term t, [c1, c2,...])
        sum = 0
        for all count c in [c1, c2,...] do
            sum = sum + c
        Emit(term t, count sum)

class Reducer
    method Reduce (term t, counts [c1, c2,...])
        sum = 0
        for all count c in [c1, c2,...] do
            sum = sum + c
        Emit(term t, count sum)
```

WordCount, “In-mapper combining”, v.2

```
class Mapper
    method Initialize
         $H = \text{new } \textbf{AssociativeArray}$ 

    method Map (docid id, doc d)
        for all term t in doc d do
             $H\{t\} = H\{t\} + 1$ 

    method Close
        for all term t in H do
            Emit(term t, count H\{t\})
```

WordCount, “In-mapper combining”, v.2

- “In-mapper combining”
 - “Заворачиваем” функционал комбайнера в *mapper* путем сохранения состояния между вызовами функции *map()*
- Плюсы
 - Скорость
 - Почему это быстрее, чем стандартный *Combiner*?
- Минусы
 - Требуется “ручное” управление памятью
 - Потенциальная возможность для багов связанных с сортировкой порядка элементов

Среднее значение, v.1

```
class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)

class Reducer
    method Reduce(string t, integers [r1, r2, ...])
        sum = 0
        cnt = 0
        for all integers r in [r1, r2, ...] do
            sum = sum + r
            cnt = cnt + 1
        avg = sum / cnt
        Emit(string t, integer avg)
```

Можно ли использовать Reducer в качестве Combiner?

Среднее значение, v.2

```
class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)

class Combiner
    method Combine(string t, integers [r1, r2, ...])
        sum = 0
        cnt = 0
        for all integers r in [r1, r2, ...] do
            sum = sum + r
            cnt = cnt + 1
        Emit(string t, pair(sum, cnt))

class Reducer
    method Reduce(string t, pairs[(s1,c1),(s2,c2) ...])
        sum = 0
        cnt = 0
        for all pairs p in [(s1,c1),(s2,c2) ...] do
            sum = sum + p.s
            cnt = cnt + p.c
        avg = sum / cnt
        Emit(string t, integer avg)
```

Почему это не работает?

Среднее значение, v.3

```
class Mapper
    method Map(string t, integer r)
        Emit(string t, pair (r,1))

class Combiner
    method Combine(string t pairs[(s1,c1),(s2,c2) ...]])
        sum = 0
        cnt = 0
        for all pairs p in [(s1,c1),(s2,c2) ...]) do
            sum = sum + p.s
            cnt = cnt + p.c
        Emit(string t, pair(sum, cnt))

class Reducer
    method Reduce(string t, pairs[(s1,c1),(s2,c2) ...])
        sum = 0
        cnt = 0
        for all pairs p in [(s1,c1),(s2,c2) ...]) do
            sum = sum + p.s
            cnt = cnt + p.c
        avg = sum / cnt
        Emit(string t, pair (avg, cnt))
```

А так будет работать?

Среднее значение, v.4

```
class Mapper
    method Initialize
        S = new AssociativeArray
        C = new AssociativeArray

    method Map (string t, integer r)
        S{t} = S{t} + r
        C{t} = C{t} + 1

    method Close
        for all term t in S do
            Emit(term t, pair(S{t}, C{t}))
```

Distinct Values (Unique Items Counting)

- Описание проблемы
 - Есть множество записей
 - Каждая запись содержит поле F и производное число признаков категорий G = {G1, G2, ...}.
- Задача
 - Подсчитать общее число уникальных значений поля F для каждого подмножества записей для каждого значения в каждой категории

Record 1: F=1, G={a, b}

Record 2: F=2, G={a, d, e}

Record 3: F=1, G={b}

Record 4: F=3, G={a, b}

Result:

a -> 3 // F=1, F=2, F=3

b -> 2 // F=1, F=3

d -> 1 // F=2

e -> 1 // F=2

Distinct Values, v.1

- Решение в две фазы (две задачи MapReduce)
- Первая фаза
 - *Mapper* пишет все уникальные пары $[G, F]$
 - *Reducer* подсчитывает общее кол-во вхождений такой пары
 - Основная цель этой фазы – гарантировать уникальность значений F
- Вторая фаза
 - Пары $[G, F]$ группируются по G и затем считается общее кол-во элементов в каждой группе

Distinct Values, v.1

```
// phase 1
class Mapper
    method Map(null, record [value f, categories [g1, g2,...]])
        for all category g in [g1, g2,...]
            Emit(record [g, f], count 1)

class Reducer
    method Reduce(record [g, f], counts [n1, n2, ...])
        Emit(record [g, f], null )
```

```
// phase 2
class Mapper
    method Map(record [f, g], null)
        Emit(value g, count 1)

class Reducer
    method Reduce(value g, counts [n1, n2,...])
        Emit(value g, sum( [n1, n2,...] ))
```

Distinct Values, v.2

- Требуется только одна фаза MapReduce
 - *Mapper*
 - Пишет значение и категории
 - *Reducer*
 - Исключает дубликаты из списка категорий для каждого значения
 - Увеличивает счетчик для каждой категории
 - В конце *Reducer* пишет общее кол-во для каждой категории
- Первая фаза
 - Данный подход не очень хорошо масштабируется
 - Подходит для небольшого числа категорий
 - Напр. парсинг действий пользователей из web-логов
 - *Combiners* позволяют уменьшить кол-во дубликатов перед фазой *Reduce*

Distinct Values, v.2

class Mapper

method Map(null, record [value f, categories [g1, g2,...]])
for all category g in [g1, g2,...]
Emit(value f, category g)

class Reducer

method Initialize

H = new AssociativeArray : category -> count

method Reduce(value f, categories [g1, g2,...])
[g1', g2',..] = ExcludeDuplicates([g1, g2,...])
for all category g in [g1', g2',...]
H{g} = H{g} + 1

method Close

for all category g in H do
Emit(category g, count H{g})

Cross-Correlation

- Описание проблемы
 - Есть множество кортежей объектов
 - Для каждой возможной пары объектов посчитать число кортежей, где они встречаются вместе
 - Если число объектов N , то N^2 объектов будет обработано
- Применение
 - Анализ текстов
 - Кортежи – предложения, объекты – слова
 - Маркетинг
 - Покупатели, кто покупает одни товары, обычно покупают и другие товары
- Если N^2 небольшое и можно построить матрицу в памяти, то реализация довольно проста

Cross-Correlation: Pairs

- Каждый *Mapper* принимает на вход кортеж
 - Генерит все пары соседних объектов
 - Для всех пар выполняет *emit* (*a, b*) → *count*
- Reducer суммирует все *count* для всех пар
 - *Combiners?*

```
class Mapper
    method Map(null, items [i1, i2,...] )
        for all item i in [i1, i2,...]
            for all item j in [i1, i2,...]
                Emit(pair [i j], count 1)
```

```
class Reducer
    method Reduce(pair [i j], counts [c1, c2,...])
        s = sum([c1, c2,...])
        Emit(pair[i j], count s)
```

Cross-Correlation: Pairs

- Плюсы
 - Нет затрат по памяти
 - Простая реализация
- Минусы
 - Множество пар надо отсортировать и распределить по редьюсерам (sort & shuffle)
 - *Combiner* вряд ли поможет (почему?)

Cross-Correlation: Stripes

- Основная идея:
 - Группировать пары вместе в ассоциативный массив
 - Каждый *Mapper* принимает на вход последовательность
 - Генерит все пары рядом расположенных объектов
 - Для каждого объекта выполняет
emit a → { b: count_b, c: count_c, d: count_d ... }
 - *Reducer'ы* выполняют поэлементное суммирование ассоциативных массивов
 - +
$$\begin{array}{r} a \rightarrow \{ b: 1, d: 5, e: 3 \} \\ + a \rightarrow \{ b: 1, c: 2, d: 2, f: 2 \} \\ \hline a \rightarrow \{ b: 2, c: 2, d: 7, e: 3, f: 2 \} \end{array}$$

Cross-Correlation: Stripes

class Mapper

method Map(null, items [i1, i2,...])
for all item i in [i1, i2,...]
 H = new AssociativeArray : item -> counter
 for all item j in [i1, i2,...]
 H{j} = H{j} + 1
 Emit(item i, stripe H)

class Reducer

method Reduce(item i, stripes [H1, H2,...])
 H = new AssociativeArray : item -> counter
 H = merge-sum([H1, H2,...])
 for all item j in H.keys()
 Emit(pair [i j], H{j})

Cross-Correlation: Stripes

- Плюсы
 - Намного меньше операций сортировки и shuffle
 - Возможно, более эффективное использование *Combiner*
- Минусы
 - Более сложная реализация
 - Более “тяжелые” объекты для передаче данных
 - Ограничения на размеры используемой памяти для ассоциативных массивов
- Pairs vs Stripes
 - Обычно, подход со *stripes* быстрее, чем с *pairs*

Реляционные паттерны MapReduce

Selection

```
class Mapper  
    method Map(rowkey key, value t)  
        if t satisfies the predicate  
            Emit(value t, null)
```

Projection

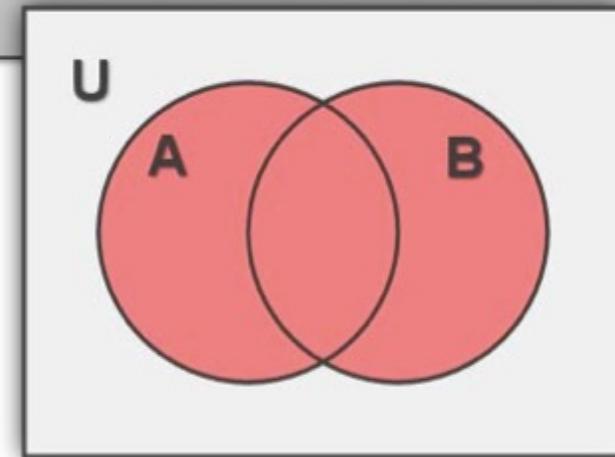
```
class Mapper
    method Map(rowkey key, value t)
        value g = project(t) // выбрать необходимые поля в g
        Emit(tuple g, null)

    // используем Reducer для устранения дубликатов
    class Reducer
        method Reduce(value t, array n) // n - массив из nulls
            Emit(value t, null)
```

Union

```
// на вход подаются элементы из двух множеств A и B
class Mapper
    method Map(rowkey key, value t)
        Emit(value t, null)

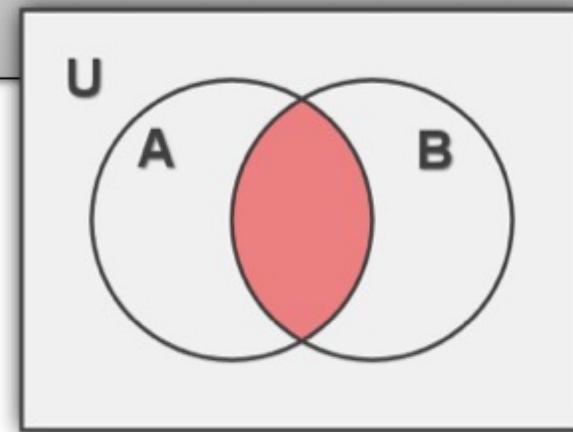
class Reducer
    method Reduce(value t, array n) // n - массив из nulls
        Emit(value t, null)
```



Intersection

```
// на вход подаются элементы из двух множеств A и B
class Mapper
    method Map(rowkey key, value t)
        Emit(value t, null)

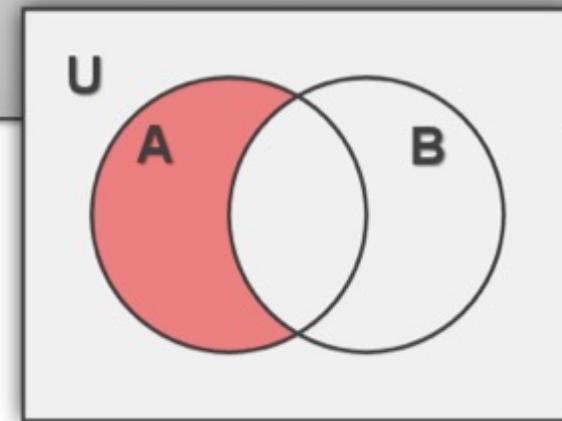
class Reducer
    method Reduce(value t, array n) // n - массив из nulls
        if n.size() = 2
            Emit(value t, null)
```



Difference

```
// на вход подаются элементы из двух множеств A и B
class Mapper
    method Map(rowkey key, value t)
        Emit(value t, string t.SetName) // t.SetName либо 'A' либо 'B'

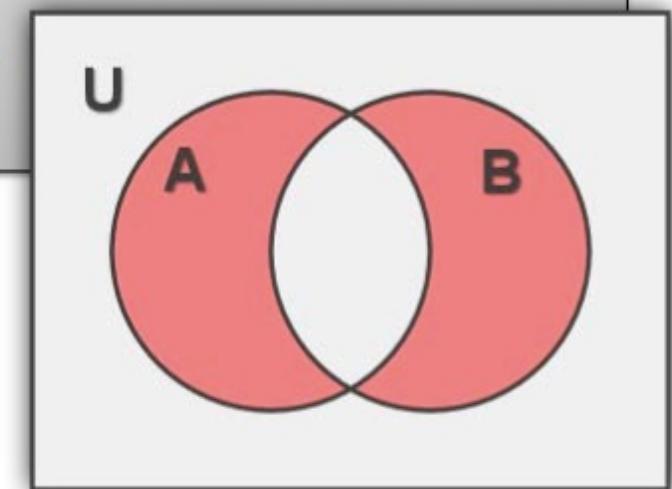
class Reducer
    // массив n может быть ['A'], ['B'], ['A' 'B'] или ['A', 'B']
    method Reduce(value t, array n)
        if n.size() = 1 and n[1] = 'A'
            Emit(value t, null)
```



Symmetric Difference

```
// на вход подаются элементы из двух множеств A и B
class Mapper
    method Map(rowkey key, value t)
        Emit(value t, string t.SetName) // t.SetName либо 'A' либо 'B'

class Reducer
    // массив n может быть ['A'], ['B'], ['A' 'B'] или ['A', 'B']
    method Reduce(value t, array n)
        if n.size() = 1 and (n[1] = 'A' or n[1] = 'B')
            Emit(value t, null)
```



GroupBy и Aggregation

```
class Mapper
    method Map(null, tuple [value GroupBy, value AggregateBy, value ...])
        Emit(value GroupBy, value AggregateBy)

class Reducer
    method Reduce(value GroupBy, [v1, v2,...])
        // aggregate() : sum(), max(),...
        Emit(value GroupBy, aggregate( [v1, v2,...] ) )
```

Группировка и агрегирование может выполняться за один MapReduce этап следующим образом. Mapper извлекает из каждого кортежа значения GroupBy и AggregateBy и выходную пару. Редуктор принимает значения, которые должны агрегироваться, уже сгруппированными и вычисляет функцию агрегации. Типичные функции агрегации, такие как сумма или максимум могут вычисляться в потоковом режиме, следовательно, одновременная обработка всех значений не требуется. Тем не менее, в некоторых случаях может потребоваться две фазы MapReduce.

Repartition Join

- *Reduce Join, Sort-Merge Join*
- Описание задачи
 - Объединить два множества A и B по ключу k
- Решение
 - *Mapper* проходит по всем значениям каждого множества
 - Выбирает ключ k и маркирует каждое значение тегом, определяющим множество, откуда пришло значение
 - *Reducer* получает все значения, объединенные по одному ключу и размещает их по двум корзинам, соответствующим каждому множеству
 - После этого проходит по обеим корзинам и генерит значения из двух множеств с общим ключом

Repartition Join

```
class Mapper
    method Map(null, tuple [join_key k, value v1, value v2,...])
        Emit(join_key k, tagged_tuple [set_name tag, values [v1, v2, ...]] )

class Reducer
    method Reduce(join_key k, tagged_tuples [t1, t2,...])
        H = new AssociativeArray : set_name -> values
        for all tagged_tuple t in [t1, t2,...] // separate values into 2 arrays
            H{t.tag}.add(t.values)
        for all values a in H{'A'} // produce a cross-join of the two arrays
            for all values b in H{'B'}
                Emit(null, [k a b] )
```

Repartition Join

- Минусы
 - *Mapper* отправляет в output все данные, даже для тех ключей, которые есть только в одном множестве
 - *Reducer* должен хранить все значения для одного ключа в памяти
 - *Нужно самостоятельно управлять памятью в случае, если данные в нее не помещаются*

Replicated Join

- *Map Join, Hash Join*
- Часто требуется объединять два множества разных размеров – маленькое и большое
 - Напр. список пользователей с логом их активности
- Для этого можно использовать хеш-таблицу, куда загружать все элементы маленького множества, сгруппированных по ключу k
- Затем, иди по элементам большого множества в *Mapper* и выполнять lookup-запрос к этой хеш-таблице

Replicated Join

```
class Mapper
    method Initialize
        H = new AssociativeArray : join_key -> tuple from A
        A = load()
        for all [ join_key k, tuple [a1, a2,...] ] in A
            H{k} = H{k}.append( [a1, a2,...] )

    method Map(join_key k, tuple B)
        for all tuple a in H{k}
            Emit(null, tuple [k a B] )
```

TF-IDF на MapReduce

TF-IDF

- Term Frequency – Inverse Document Frequency
 - Используется при работе с текстом
 - В Information Retrieval

TF-IDF

- **TF** (*term frequency* — частота слова) — отношение числа вхождения некоторого слова к общему количеству слов документа.
 - Таким образом, оценивается важность слова в пределах отдельного документа.

$$tf(t, d) = \frac{n_i}{\sum_k n_k}$$

где n_i есть число вхождений слова в документ, а в знаменателе — общее число слов в данном документе.

- **IDF** (*inverse document frequency* — обратная частота документа) — инверсия частоты, с которой некоторое слово встречается в документах коллекции.

$$idf(t, D) = \log \frac{|D|}{|(d_i \supset t_i)|}$$

Где:

- $|D|$ — количество документов в корпусе;
- $|(d_i \supset t_i)|$ — кол-во документов, в которых встречается t_i (когда $n_i \neq 0$).

$$tfidf(t, d, D) = tf(t, d) \times idf(t, D)$$

TF-IDF

Что нужно будет вычислить

- Сколько раз слово T встречается в данном документе (tn)
- Сколько слов в документе (sn)
- Сколько документов, в котором встречается данное слово T (n)
- Общее число документов (N)

TF-IDF

- **Job 1:** Частота слова в документе
- *Mapper*
 - Input: $(docname, contents)$
 - Для каждого слова в документе надо сгенерить пару $(word, docname)$
 - Output: $((word, docname), 1)$
- *Reducer*
 - Суммирует число слов в документе
 - Outputs: $((word, docname), tf)$
- *Combiner* такой же как и *Reducer*

Здесь где-то надо подсчитать число всех слов в документе sn для вычисления $tf = tn / sn$

TF-IDF

- **Job 2:** Кол-во документов для слова
- *Mapper*
 - Input: $((word, docname), tf)$
 - Output: $(word, (docname, tf, 1))$
- *Reducer*
 - Суммирует единицы чтобы посчитать n
 - Output: $((word, docname), (tf, n))$

TF-IDF

- **Job 3:** Расчет TF-IDF
- *Mapper*
 - Input: $((word, docname), (tf, n))$
 - Подразумевается, что N известно (его легко подсчитать)
 - Output: $((word, docname), (TF*IDF))$
- *Reducer*
 - Не требуется

TF-IDF, масштабируемость

- Несколько MapReduce задач позволяют реализовать сложные алгоритмы и улучшить масштабируемость
 - Думая в стиле MapReduce часто означает разделение комплексных задач на более мелкие
- Стоит следить за тем, сколько используется ОЗУ, при работе с большим объемом данных
 - Каждый раз, когда необходимо хранить данные в памяти, это может стать потенциальной проблемой масштабируемости

Ресурсы

Data-Intensive Text Processing with MapReduce

Jimmy Lin and Chris Dyer (Authors) (April, 2010)

Chapter3: MapReduce Algorithm Design



<http://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/>