

分布式计算框架Hadoop

— 典型的分布式系统

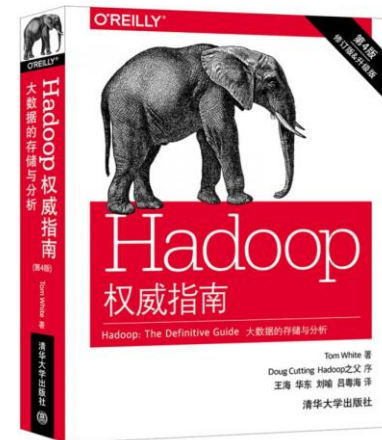
Some slides from:

Nalini Venkatasubramanian @ UIC Irvine

<https://www.ics.uci.edu/~cs237/>

林子雨 厦门大学

<http://dblab.xmu.edu.cn/post/linziyu>



<https://www.oreilly.com/library/view/hadoop-the-definitive/9781491901687/>

Outline

1. Basics
2. MapReduce
3. YARN
4. HDFS
5. ZooKeeper

Hadoop

- Apache top level project
- Open-source implementation of frameworks
- For **reliable, scalable**, distributed computing and data storage
- A **flexible and highly-available** architecture for large scale computation and data processing on a network of **commodity hardware**.



<https://hadoop.apache.org/>

Motivation



1998

The Google File System

2003

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

Bigtable: A Distributed Storage System for Structured Data

2006

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach

Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

MapReduce: Simplified Data Processing on Large Clusters

2004

Jeffrey Dean and Sanjay Ghemawat

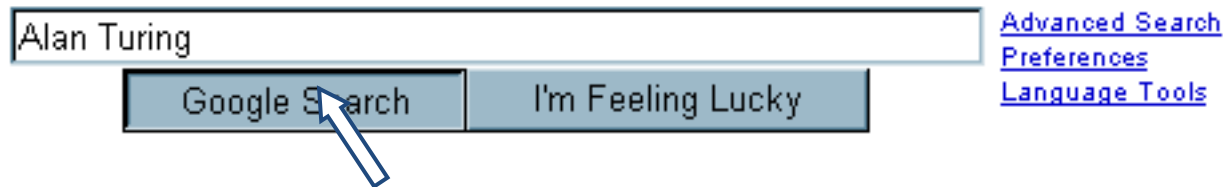
jeff@google.com, sanjay@google.com

Google, Inc.



2013

Motivation



- 200+ processors
- 200+ terabyte database
- 10^{10} total clock cycles
- 0.1 second response time
- 5¢ average advertising revenue

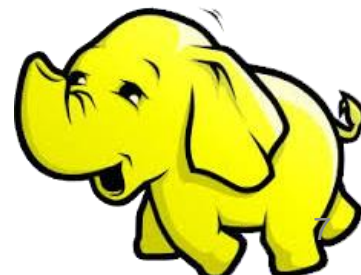
Motivation

- For large scale data processing
- Want to process lots of data (> 1 TB)
- Want to parallelize across hundreds/thousands of CPUs
- ... Want to make this easy

"Google Earth uses **70.5 TB**: 70 TB for the raw imagery and 500 GB for the index data."-2016

Goals of Hadoop

- Abstract and facilitate the storage and processing of large and/or rapidly growing data sets
 - Structured and non-structured data
 - Simple programming models
- High scalability and availability
- Use commodity (cheap!) hardware with little redundancy
- Fault-tolerance
- Move computation rather than data



Hadoop's Developers

2005: Doug Cutting and Michael J. Cafarella developed Hadoop to support distribution for the [Nutch](#) search engine project.

The project was funded by Yahoo!

2006: Yahoo gave the project to Apache Software Foundation.



Doug Cutting

Motivated by Google

2003

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Google*



2004

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat
jeff@google.com, sanjay@google.com
Google, Inc.



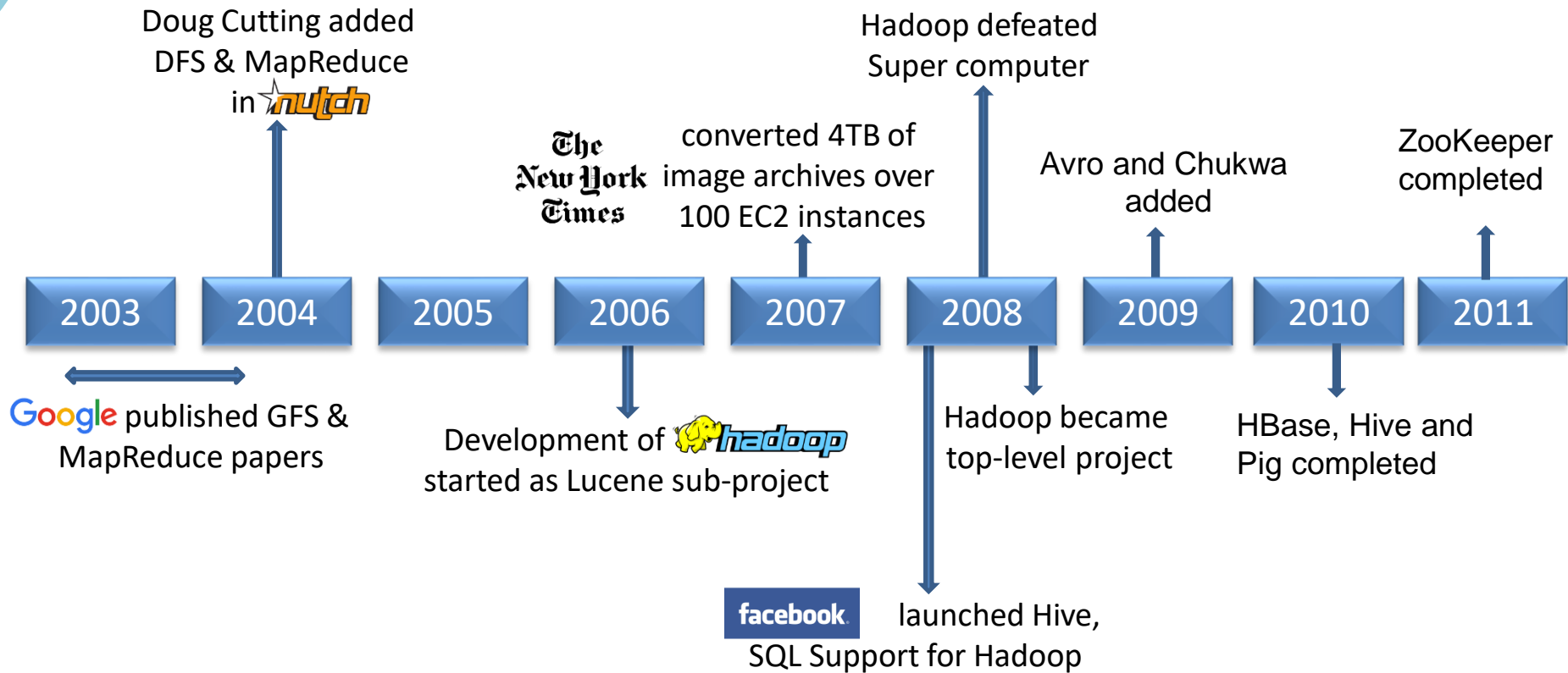
2006

Bigtable: A Distributed Storage System for Structured Data

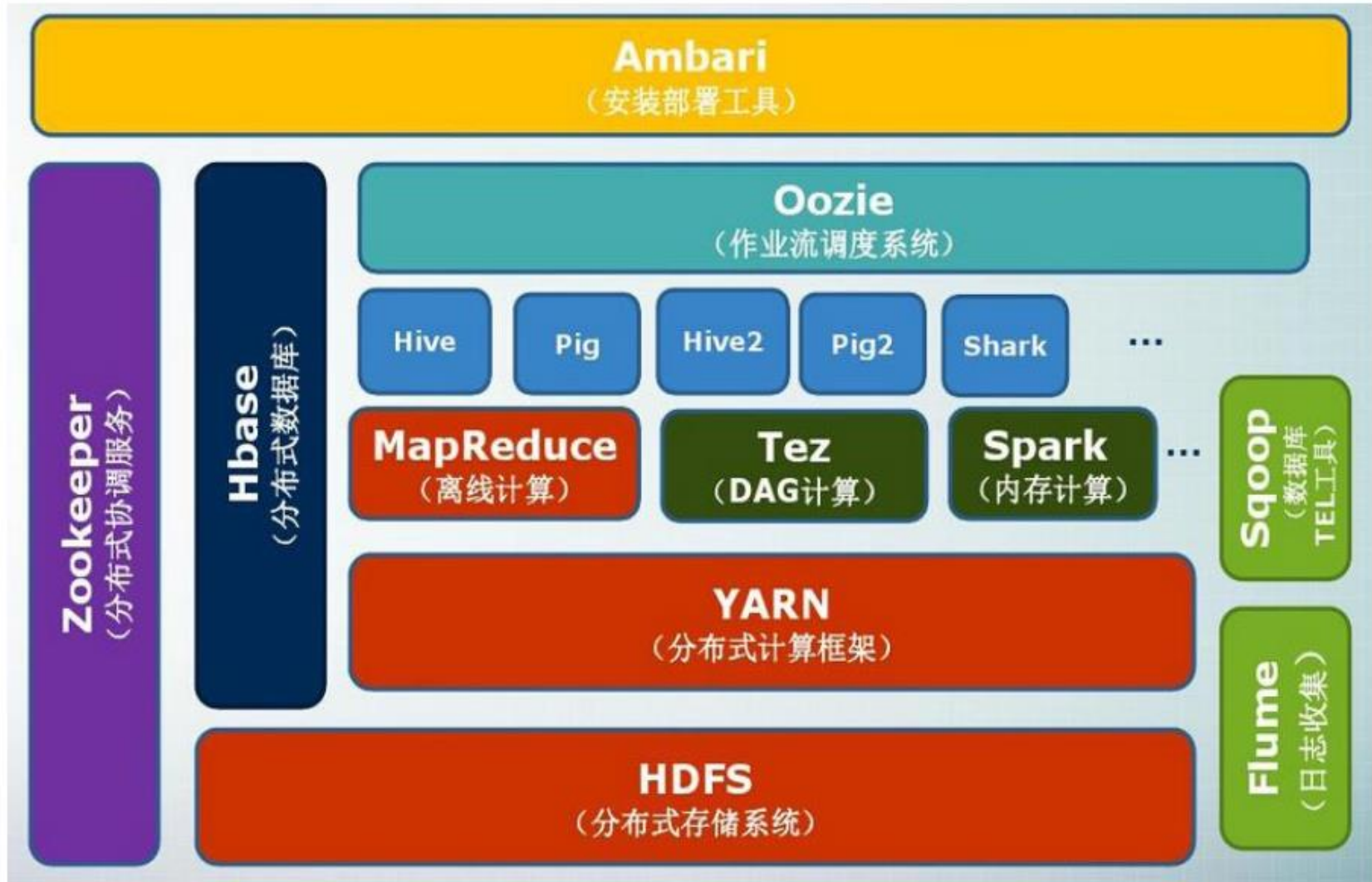
Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com
Google, Inc.



Development of Hadoop



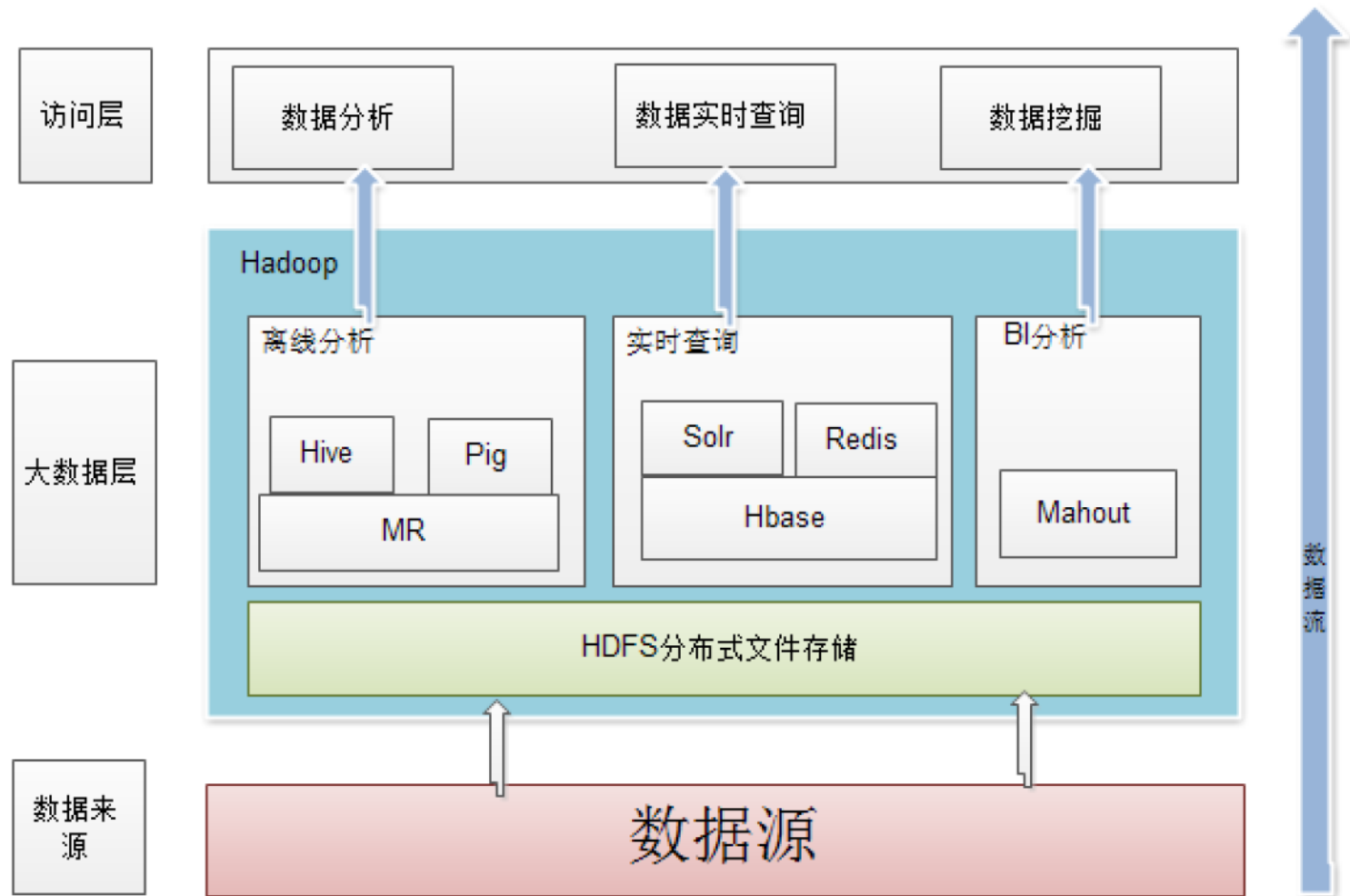
Hadoop Ecosystem



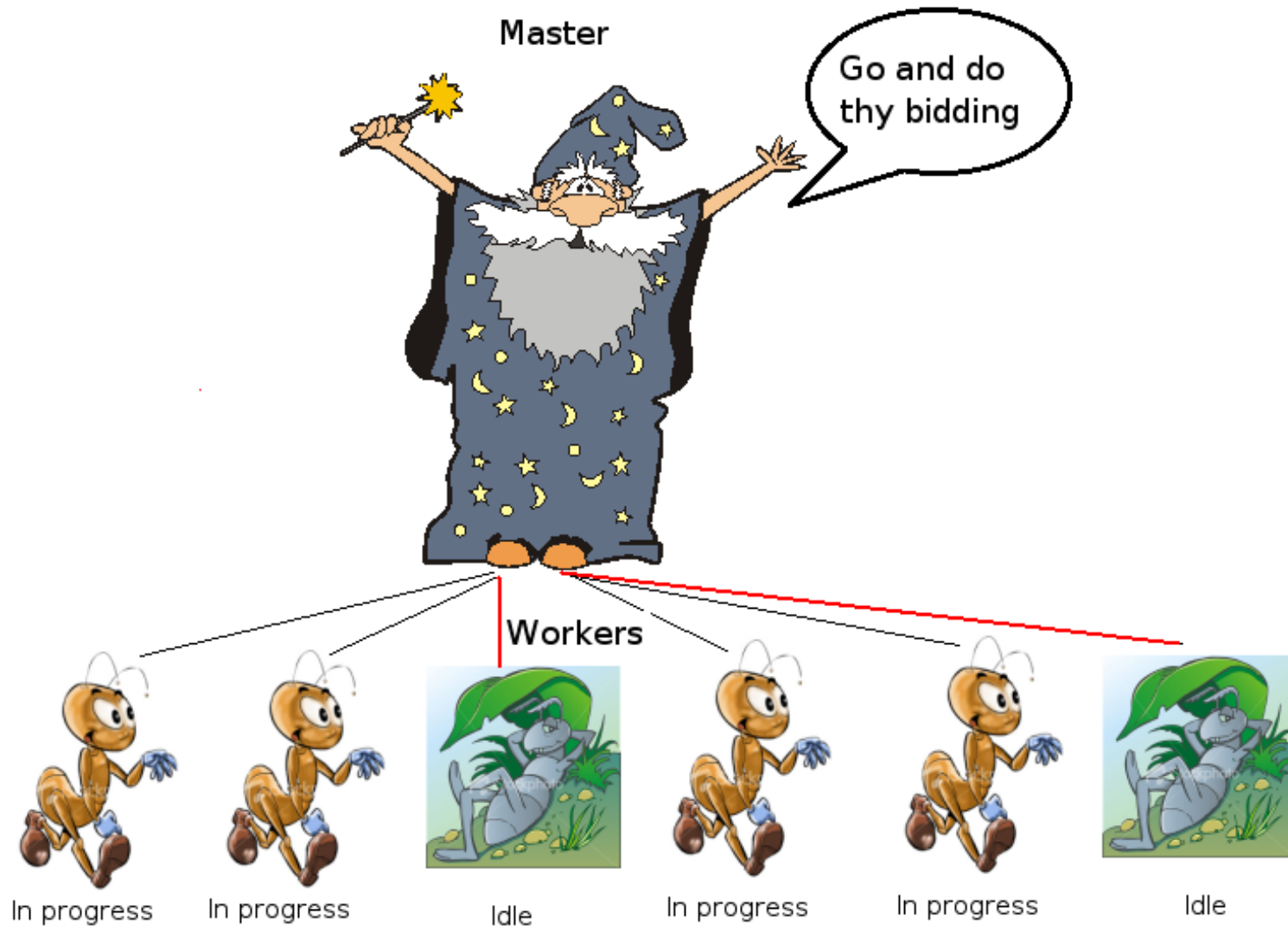
Hadoop Ecosystem

组件	功能
HDFS	分布式文件系统
MapReduce	分布式并行编程模型
YARN	资源管理和调度器
Tez	运行在YARN之上的下一代Hadoop查询处理框架
Hive	Hadoop上的数据仓库
HBase	Hadoop上的非关系型的分布式数据库
Pig	一个基于Hadoop的大规模数据分析平台，提供类似SQL的查询语言Pig Latin
Sqoop	用于在Hadoop与传统数据库之间进行数据传递
Oozie	Hadoop上的工作流管理系统
Zookeeper	提供分布式协调一致性服务
Storm	流计算框架
Flume	一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统
Ambari	Hadoop快速部署工具，支持Apache Hadoop集群的供应、管理和监控
Kafka	一种高吞吐量的分布式发布订阅消息系统，可以处理消费者规模的网站中的所有动作流数据
Spark	类似于Hadoop MapReduce的通用并行框架

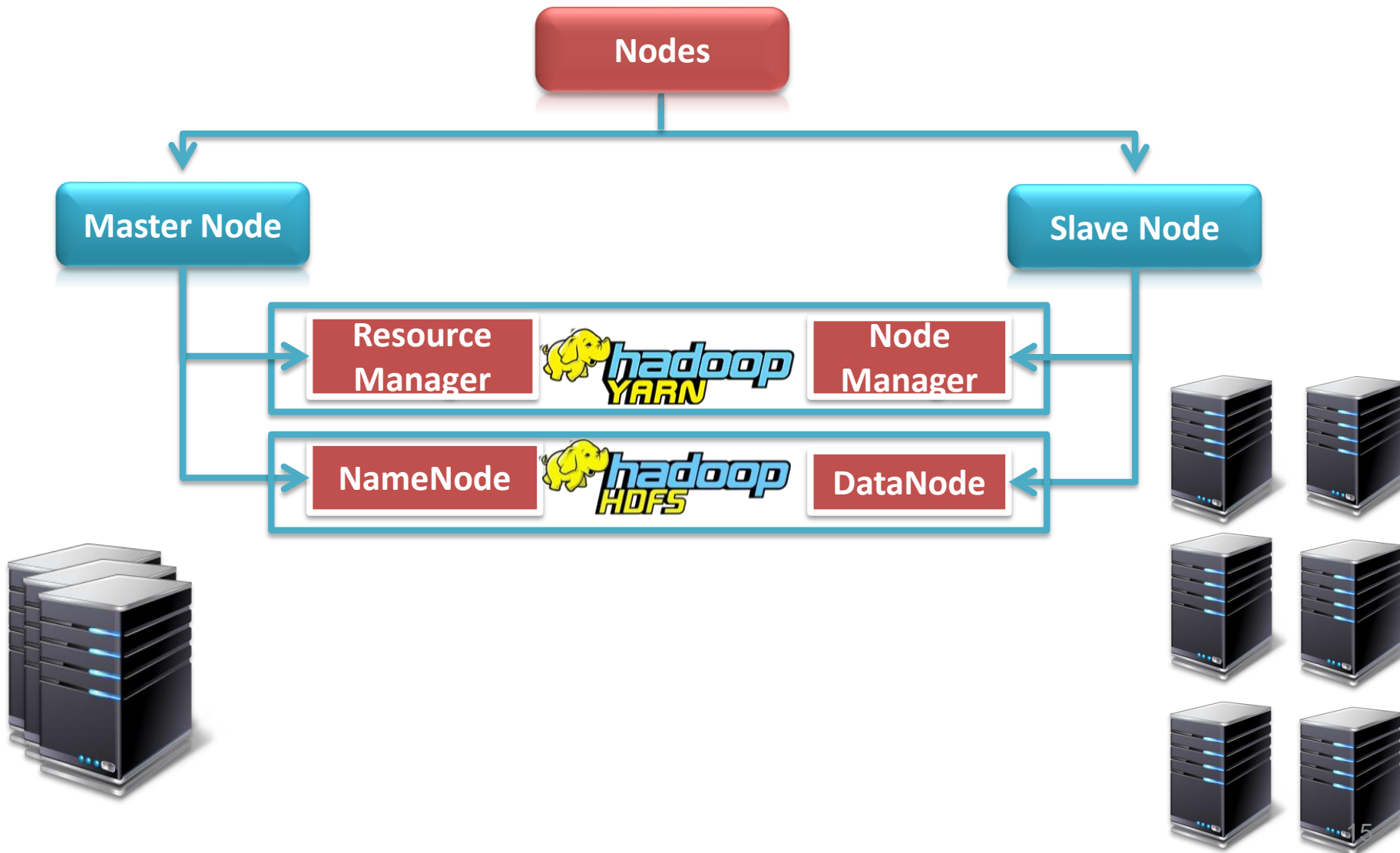
Application of Hadoop



Hadoop Architecture



Hadoop Architecture





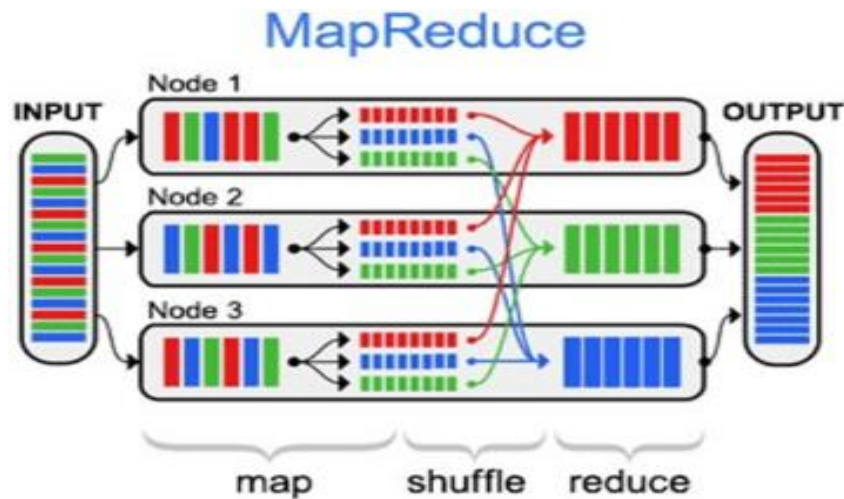
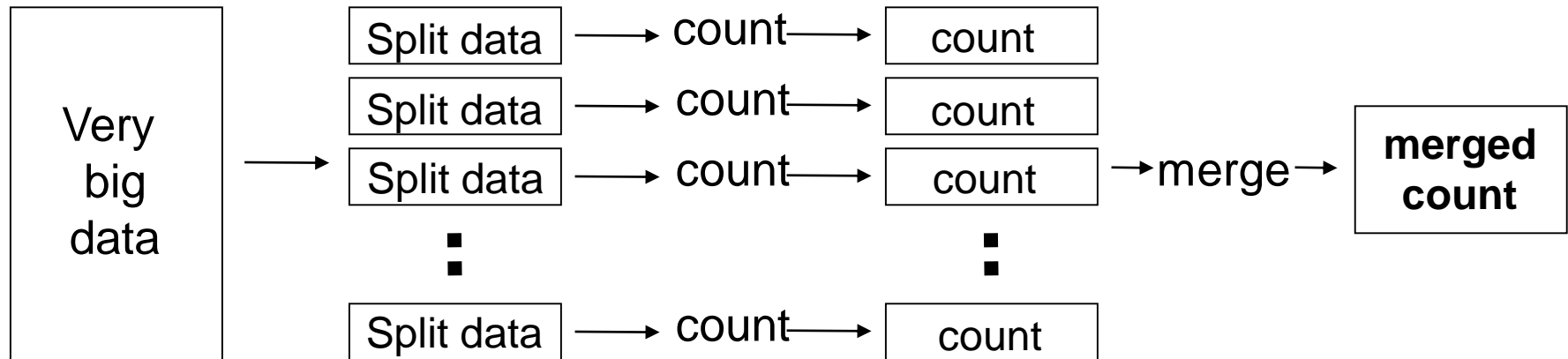
Outline

1. Basics
2. MapReduce
3. YARN
4. HDFS
5. ZooKeeper

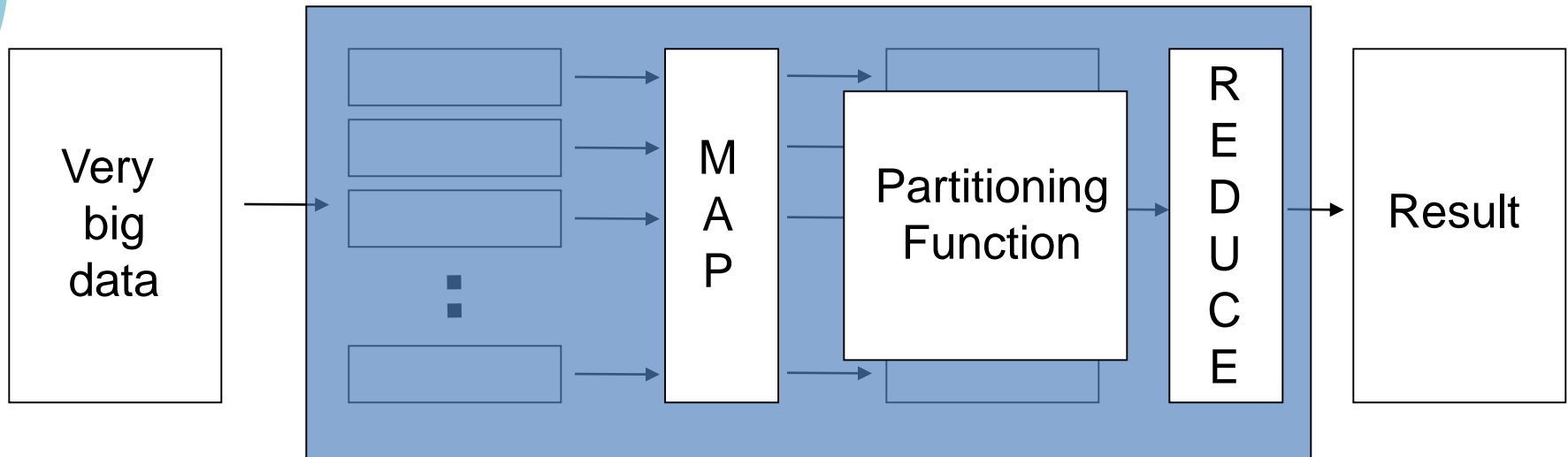
MapReduce

- A programming model (& its associated implementation)
- For processing large data set
- Exploits large set of commodity computers
- Executes process in distributed manner
- Offers high degree of transparencies

Distributed Word Count

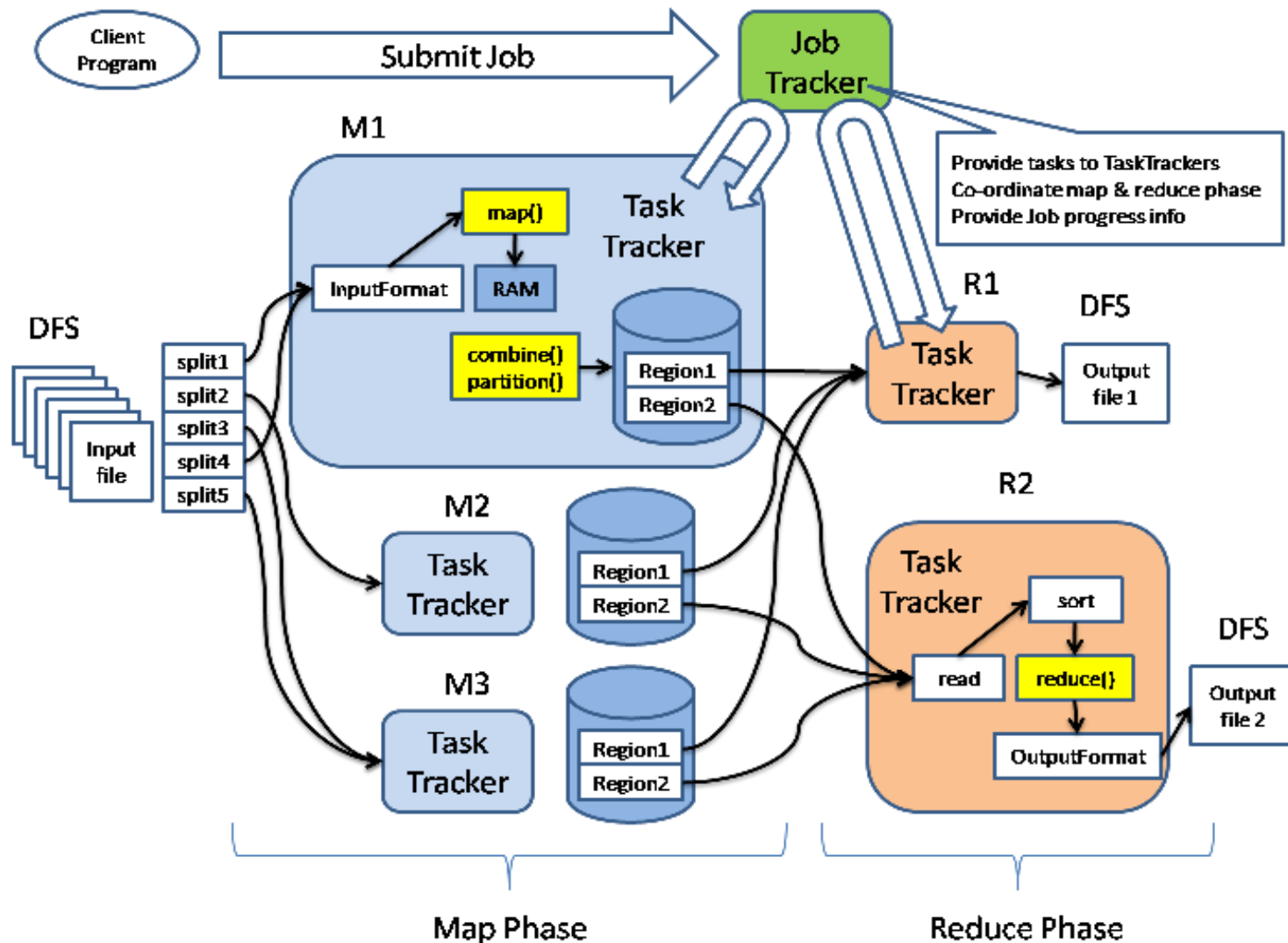


MapReduce Procedure



- Map:
 - Accepts *input* key/value pair
 - Emits *intermediate* key/value pair
- Reduce :
 - Accepts *intermediate* key/value* pair
 - Emits *output* key/value pair

MapReduce Framework



MapReduce Process

(org.apache.hadoop.mapred)

- JobClient
 - Submit job
- JobTracker
 - Manage and schedule job, split job into tasks
- TaskTracker
 - Start and monitor the task execution
- Child
 - The process that really execute the task

Inter Process Communication

IPC/RPC (org.apache.hadoop.ipc)

- Protocol

Client Role

Server Role

JobClient <-----JobSubmissionProtocol-----> JobTracker

TaskTracker <----- InterTrackerProtocol -----> JobTracker

TaskTracker <-----TaskUmbilicalProtocol -----> Child

Mapper

- Records from the data source
 - lines out of files, rows of a database, etc.
 - key*value pairs: e.g., (filename, line)
- map() produces one or more *intermediate* values along with an output key from the input.

Mapper

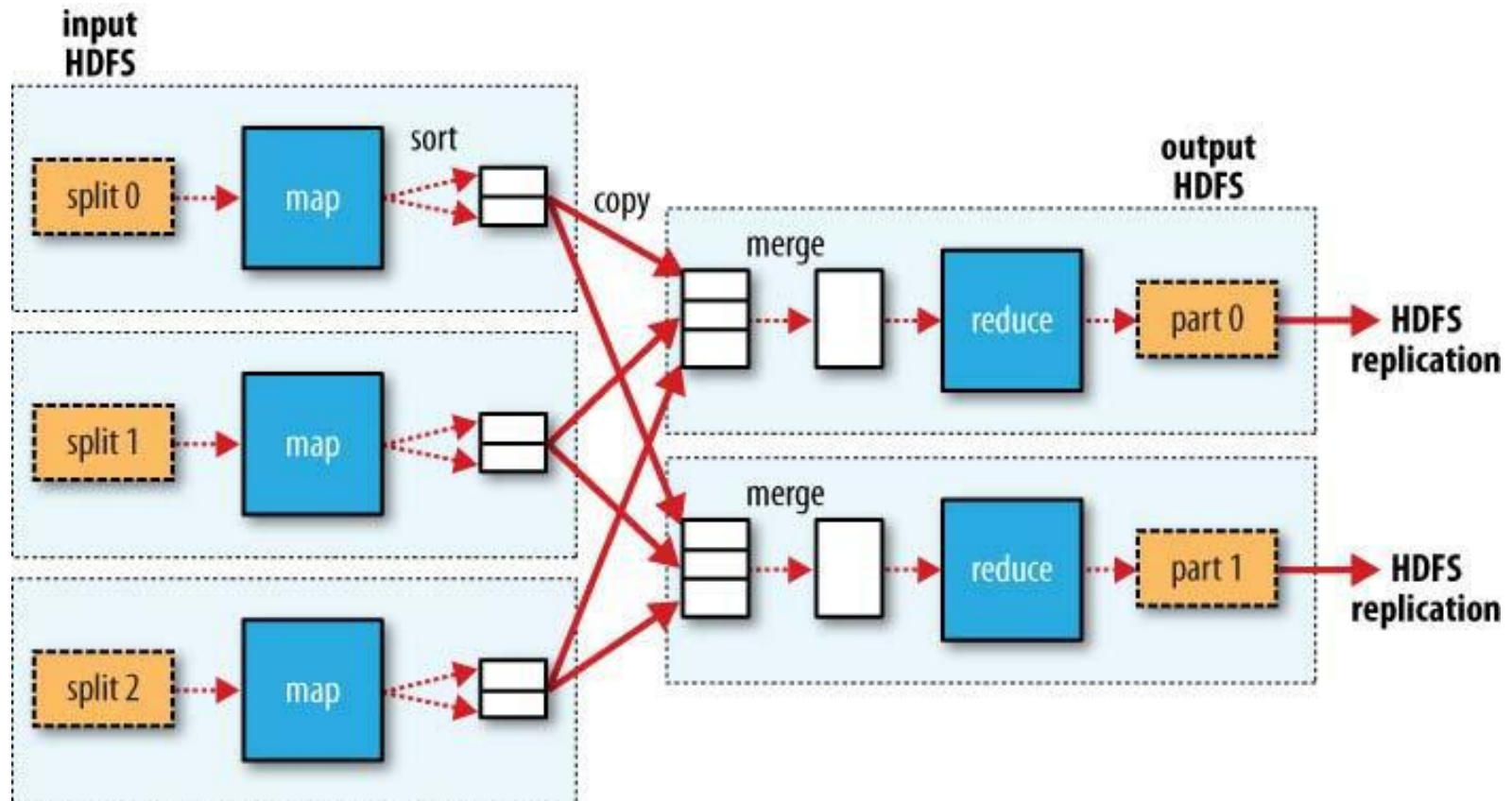
- 每一个Mapper类的实例生成了一个Java进程
 - 在某一个InputSplit上执行
- 有两个额外的参数OutputCollector以及Reporter
 - 前者用来收集中间结果
 - 后者用来获得环境参数以及设置当前执行的状态。

Reducer

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- `reduce()` combines those intermediate values into one or more *final values* for that same output key

(in practice, usually only one final value per key)

MapReduce Data Flow



Partition&Shuffle

- 在Map工作完成之后，每一个 Map函数会将结果传到对应的Reducer所在的节点
- 用户可以提供一个Partitioner类，用来决定一个给定的<key,value>对传输的具体位置

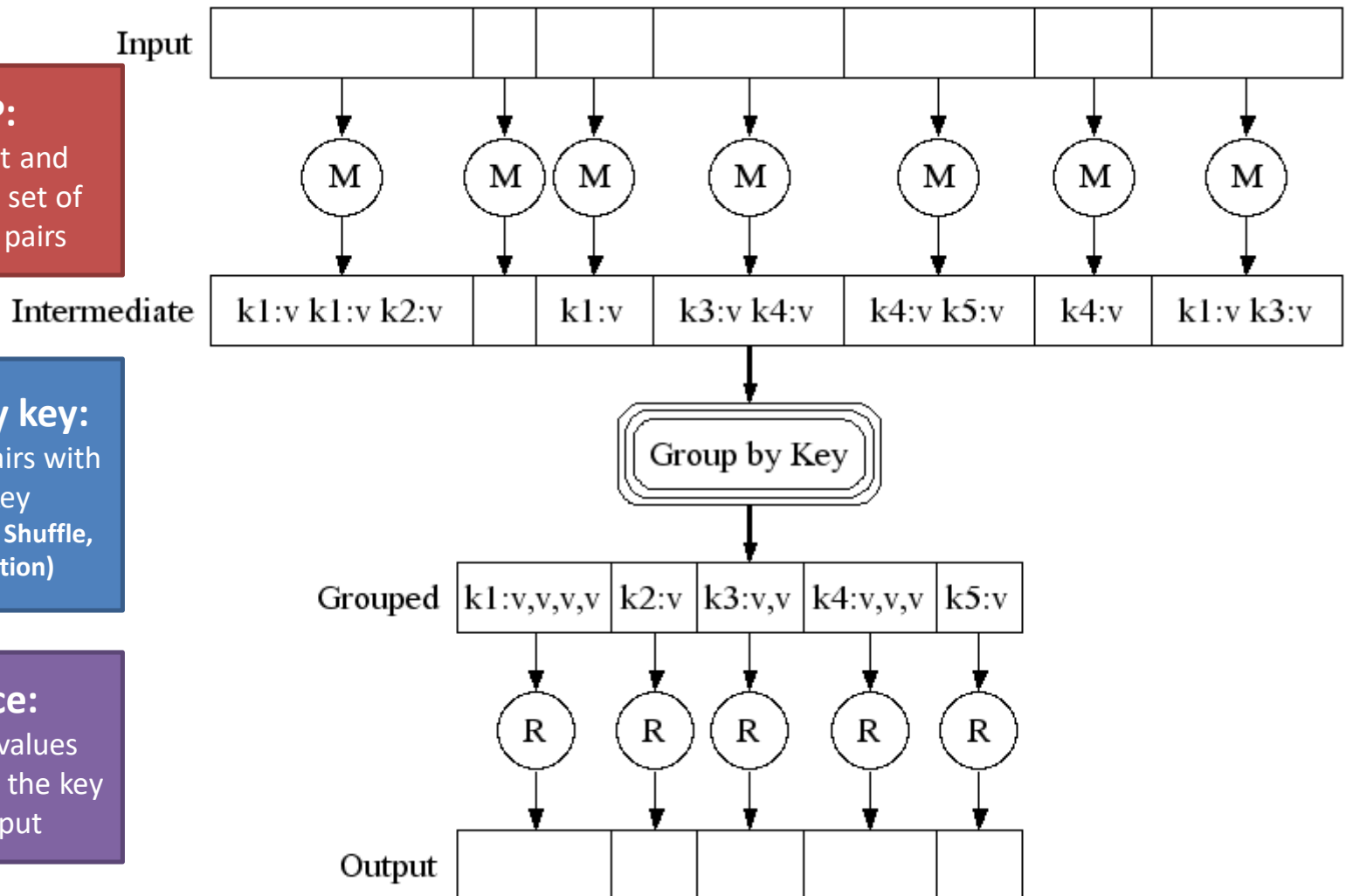
Sort

- 传输到每一个节点上的所有的Reduce函数接收到的`<key,value>`对会被Hadoop自动排序
 - 即Map生成的结果传送到一个节点时，会被自动排序
- **Default** : $\text{hash}(\text{key}) \bmod R$
- **Guarantee**:
 - Relatively well-balanced partitions
 - Ordering guarantee within partition

Partitioning Function

MAP:

Read input and produces a set of key-value pairs



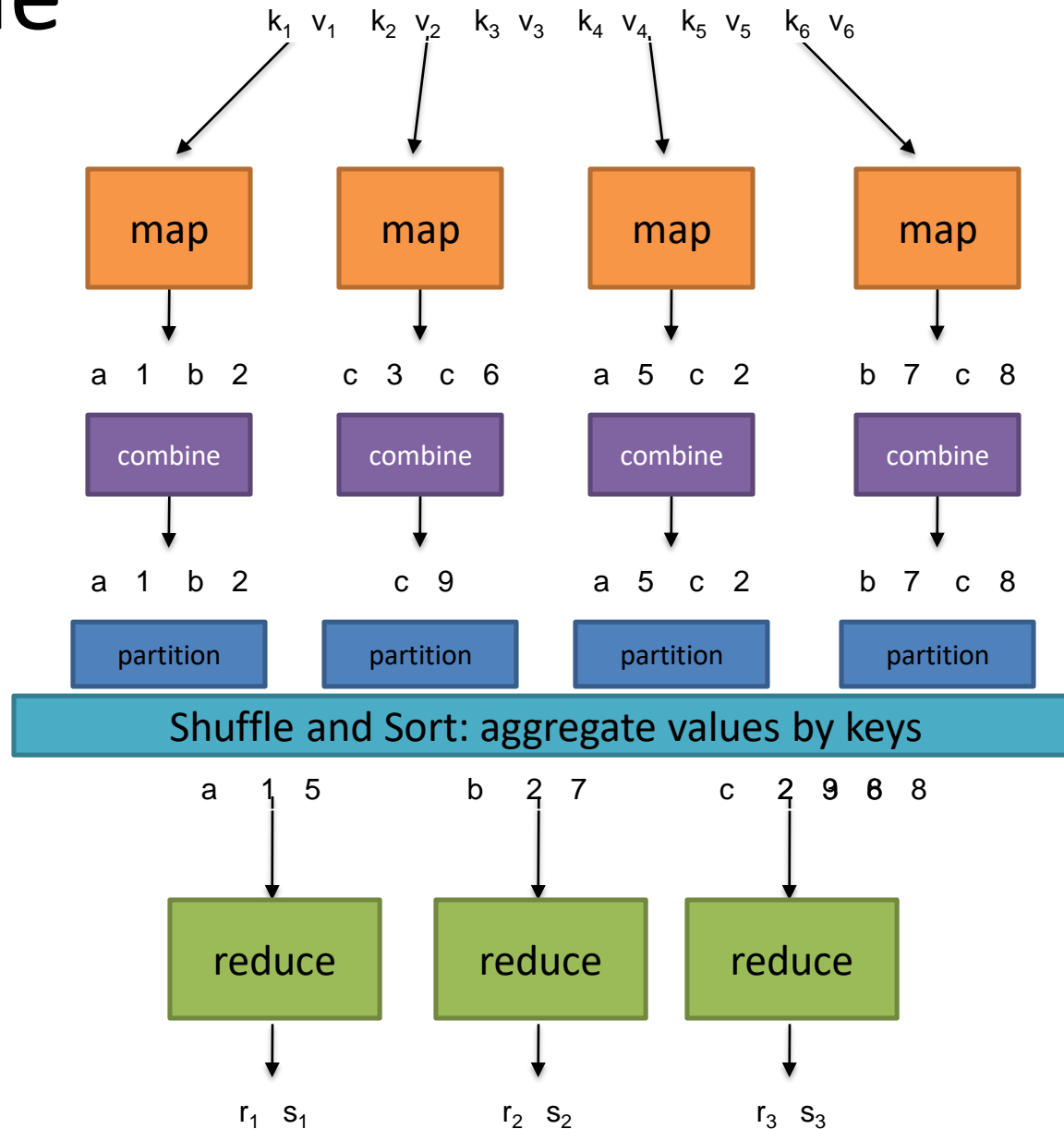
Group by key:

Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

Reduce:

Collect all values belonging to the key and output

Example



MapReduce

```
Class MapReduce{
    Class Mapper ...{
        Map code;
    }
    Class Reduer ...{
        Reduce code;
    }
    Main(){
        JobConf Conf=new JobConf("MR.Class");
        Other code;
    }
}
```

MapReduce Transparencies

Plus Google Distributed File System :

- Parallelization
- Fault-tolerance
- Locality optimization
- Load balancing



Example Word Count: Map

```
public static class MapClass extends MapReduceBase
implements Mapper {
    private final static IntWritable one= new IntWritable(1);
    private Text word = new Text();

    public void map(WritableComparable key, Writable value,
OutputCollector output, Reporter reporter)
throws IOException {
        String line = ((Text)value).toString();
        StringTokenizer itr = new StringTokenizer(line);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);
        }
    }
}
```



Example Word Count: Reduce

```
public static class Reduce extends MapReduceBase
implements Reducer {
    public void reduce(WritableComparable key, Iterator
values, OutputCollector output, Reporter reporter)
throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += ((IntWritable) values.next()).get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```



Example Word Count: Main

```
public static void main(String[] args) throws IOException
{
    //checking goes here
    JobConf conf = new JobConf();

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    conf.setMapperClass(MapClass.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    conf.setInputPath(new Path(args[0]));
    conf.setOutputPath(new Path(args[1]));

    JobClient.runJob(conf);
}
```

Example

- Page 1: the weather is good
- Page 2: today is good
- Page 3: good weather is good.

Map output

- Worker 1:
 - (the 1), (weather 1), (is 1), (good 1).
- Worker 2:
 - (today 1), (is 1), (good 1).
- Worker 3:
 - (good 1), (weather 1), (is 1), (good 1).

Reduce Input

- Worker 1:
 - (the 1)
- Worker 2:
 - (is 1), (is 1), (is 1)
- Worker 3:
 - (weather 1), (weather 1)
- Worker 4:
 - (today 1)
- Worker 5:
 - (good 1), (good 1), (good 1), (good 1)

Reduce Output

- Worker 1:
 - (the 1)
- Worker 2:
 - (is 3)
- Worker 3:
 - (weather 2)
- Worker 4:
 - (today 1)
- Worker 5:
 - (good 4)

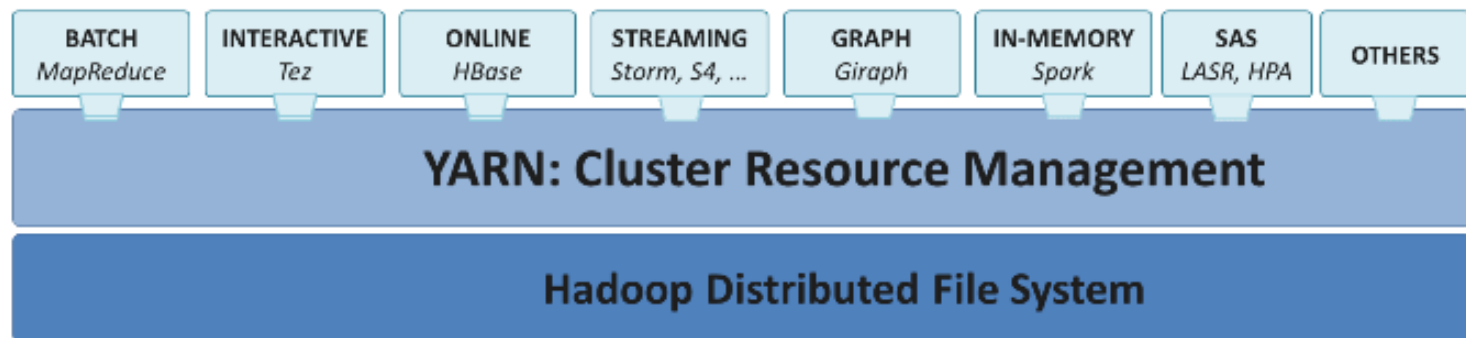


Outline

1. Basics
2. MapReduce
3. YARN
4. HDFS
5. ZooKeeper

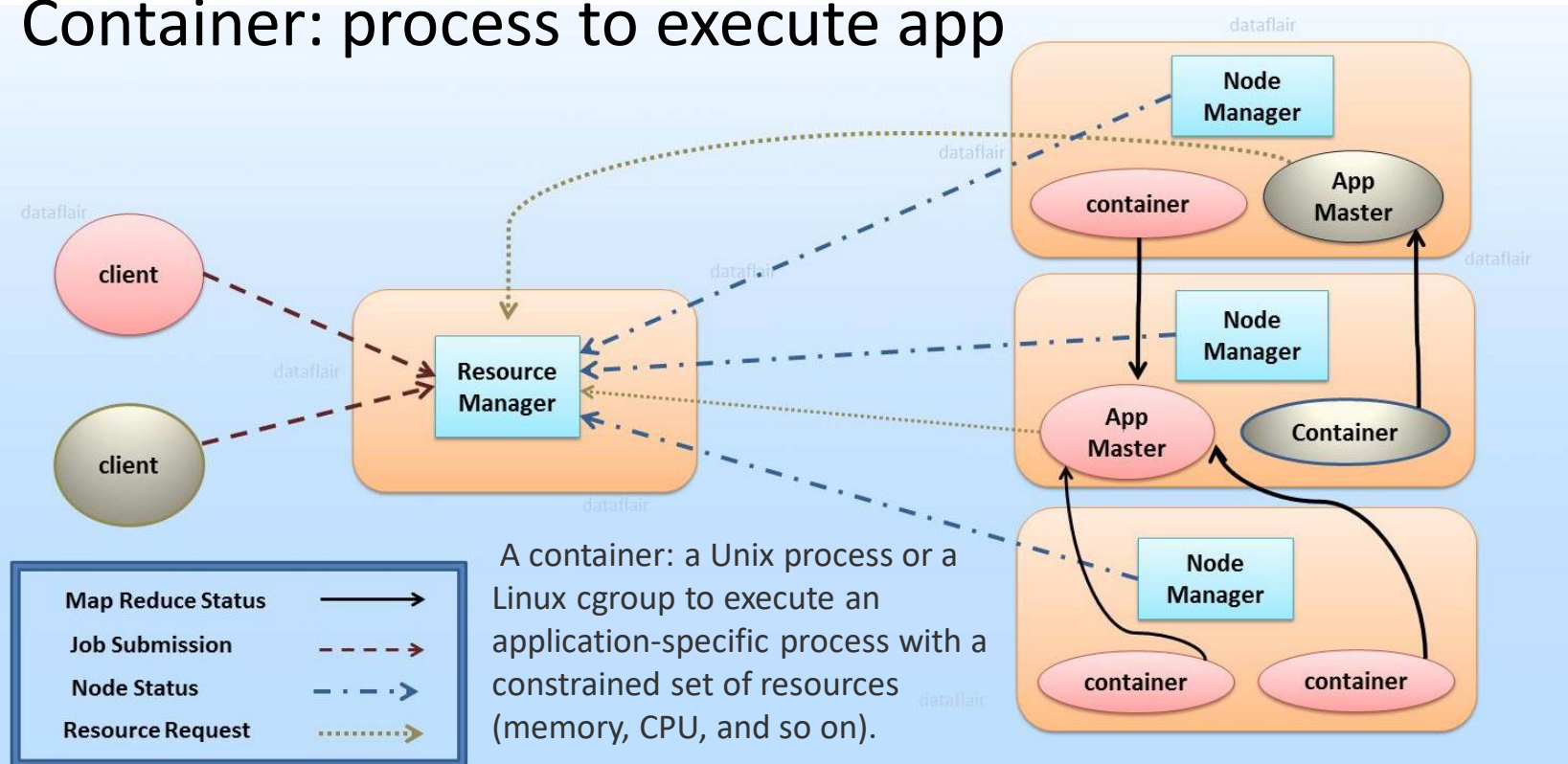
YARN

- **Yet Another Resource Negotiator**
- The resource management layer of Hadoop
- The job scheduler of Hadoop
- Introduced in Hadoop 2.x



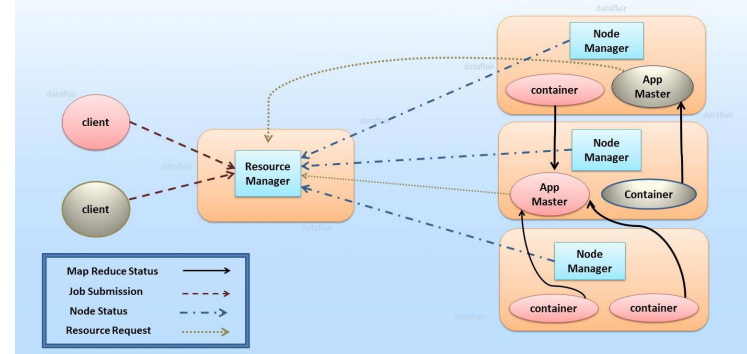
YARN Architecture

- Resource Manager: a master daemon
- Node Manager: slave daemon, one per slave node
- Application Master: one per application
- Container: process to execute app

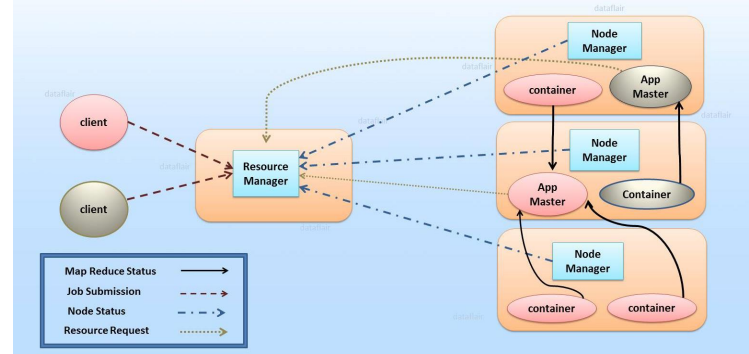


Resource Manager

- Two main components
 - responsible for allocating the resources to the running application
 - pure scheduler
 - no monitoring no tracking for the application
 - doesn't guarantees about restarting failed tasks either due to application failure or hardware failures
- Applications Manager
 - manages running Application Masters in the cluster
 - responsible for starting application masters
 - responsible for monitoring and restarting them upon failures



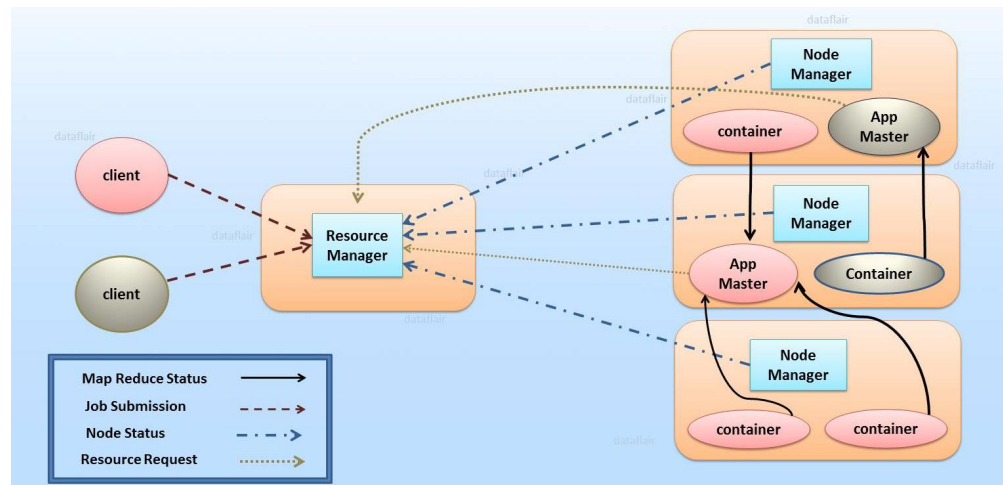
Node Manager



- The slave daemon of YARN
- Responsible for monitoring resource usage and containers and reporting the info. to RM
- Manage the user process on that node
- Track the health of the machine node
- Allows plugging long-running auxiliary services
 - these are application-specific services
 - specified as part of the configurations and loaded during startup
 - A shuffle is a typical auxiliary service by the NMs for MapReduce applications

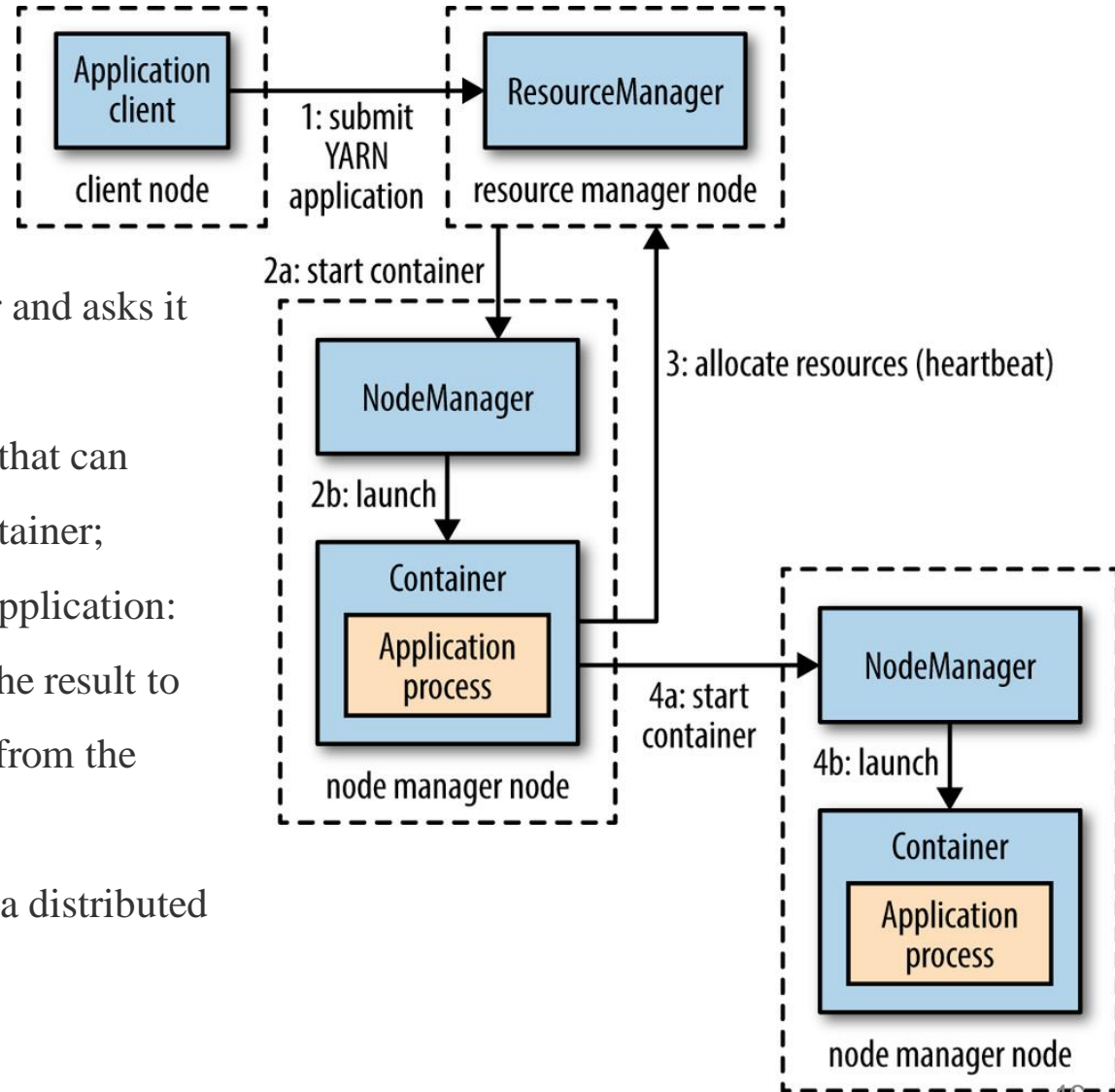
Application Master

- One application master runs per application
- Negotiate resources from the resource manager
- Manage the application life cycle
- Major operations:
 - Acquires containers from the RM's Scheduler, then
 - Contacts the corresponding NMs to start the application's individual tasks



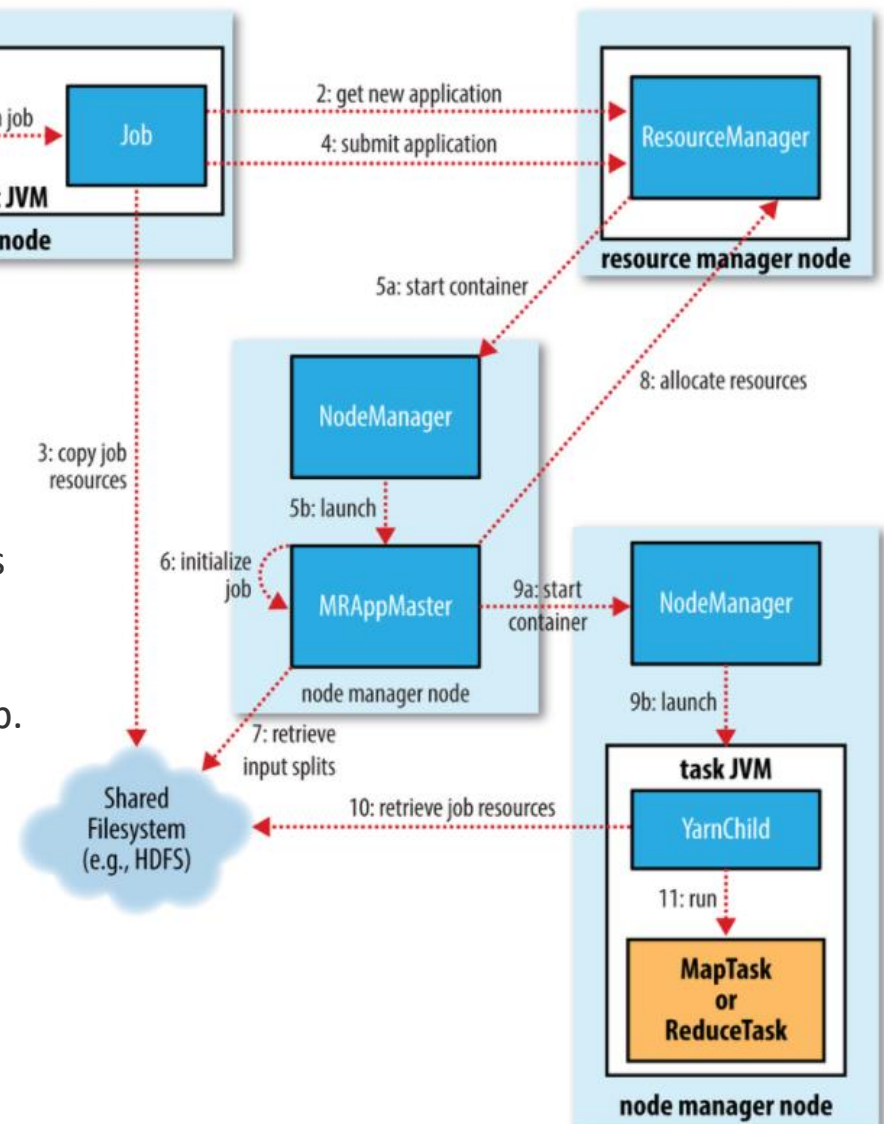
YARN Application Run

1. A client contacts the resource manager and asks it to run an *application master* process;
- 2ab. The RM then finds a node manager that can launch the application master in a container;
3. AM does operations specified by the application: simply run a computation and return the result to the client, or request more containers from the resource managers;
- 4ab. AM uses containers obtained to run a distributed computation (e.g., a MapReduce job).



MapReduce Job Run

- The client submits the MapReduce job.
- RM coordinates the allocation of compute resources on the cluster.
- NMs launch and monitor the compute containers on machines in the cluster.
- The AM coordinates the tasks running the MR job.
- The AM and the MR tasks run in containers scheduled by the RM and managed by NMs.
- DFS is used for sharing job files among entities.



Resource Requests

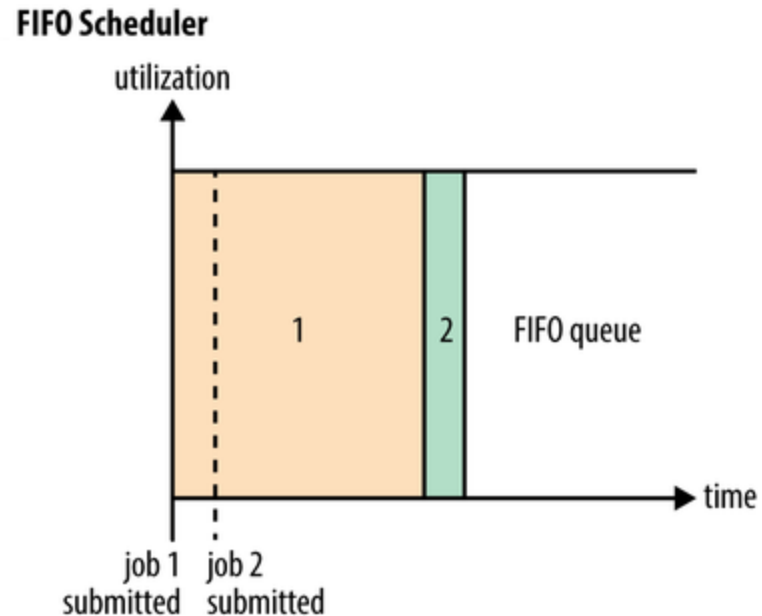
- YARN has a flexible model for making resource requests.
- A request for a set of containers
 - Required resources for each container
- Allow locality constraint
 - to request a container on a specific node or rack, or anywhere on the cluster (off-rack).
 - If not met: either no allocation made or, constraint loosened
 - critical in reduce comm. cost, e.g., MR task run at DataNode
- Allow dynamic request during application execution
 - resource requests at any time while app is running
 - MR jobs: mapper requested up front, reducer requested later; if tasks fail, additional containers will be requested

Scheduling in YARN

- Three schedulers are available in YARN
- FIFO Scheduler
- Capacity Scheduler
- Fair Scheduler

Pro: simple

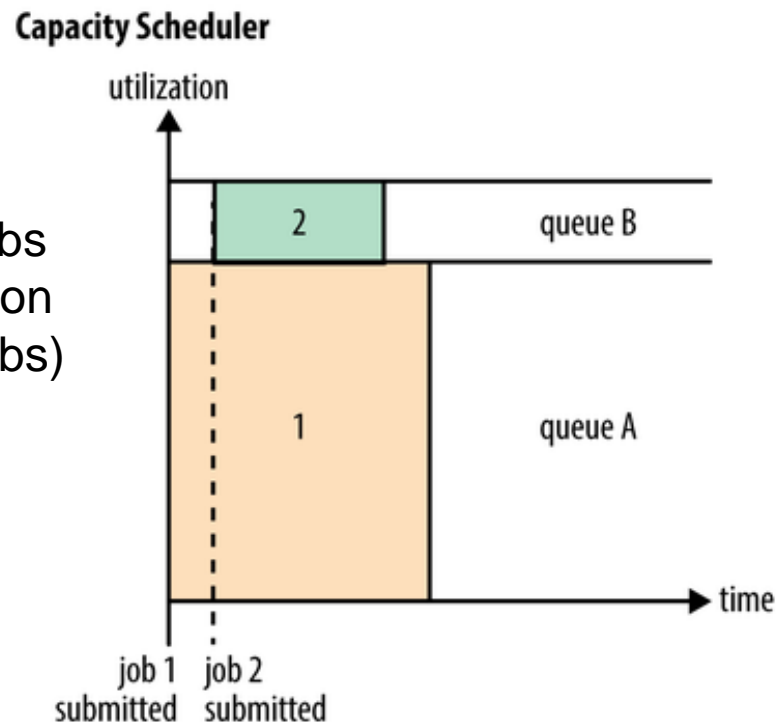
Con: small jobs suffer



Capacity Scheduler

- Short job first policy
- A separate dedicated queue for small job to start as soon as submitted

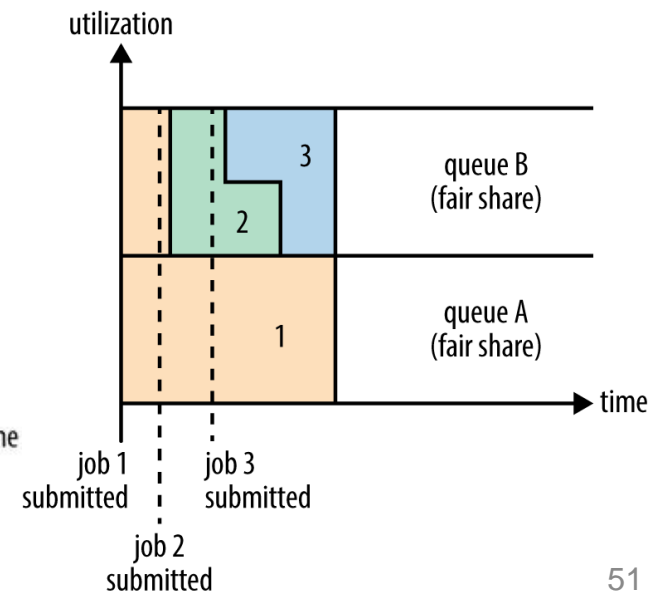
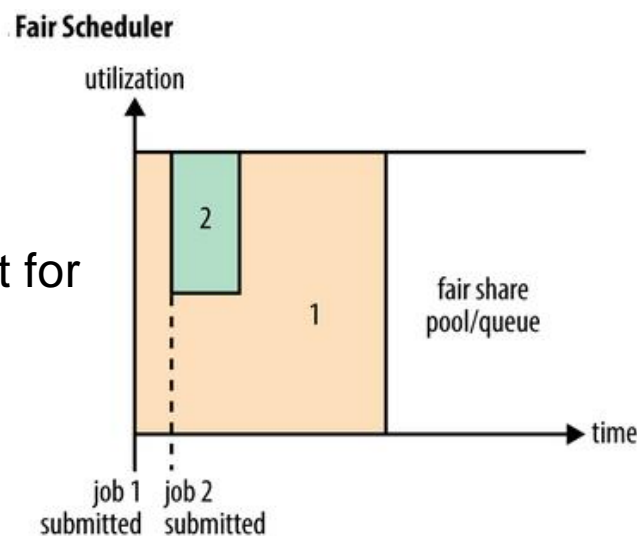
Pro: reduce waiting time of shot jobs
Con: cost of overall cluster utilization
(resources reserved for small jobs)



Fair Scheduler

- Dynamic resource split from large/long jobs
 - No need to reserve capacity
- An example:
 - When small job starts, it is allocated half of resources
- Can be queue based
 - Intra, inter

Pro: resource efficiency
Con: small jobs need to wait for resource release



Delay Scheduling

- Upon locality not met
 - Loosen it, or wait for resource becoming available
- Delay scheduling
 - waiting a short time (no more than a few seconds) can dramatically increase the chances of meeting locality
 - supported by Capacity scheduler and Fair scheduler
- NM periodically sends heartbeat request to RM
 - by default, one per second
 - carry info. about the running containers and available resources (each heartbeat is a “scheduling opportunity”)
- Scheduler waits for a given number of opportunities before loosening the locality

Dominant Resource Fairness (DRF)

- **Dominant resource** of a user: the resource that user has the **biggest share** of.
 - Total resources: $\langle 8\text{CPU}; 5\text{GB} \rangle$
 - User 1 allocation: $\langle 2\text{CPU}; 1\text{GB} \rangle$, $2/8 = 25\%\text{CPU}$ and $1/5 = 20\%\text{RAM}$
 - Dominant resource of User 1 is **CPU** ($25\% > 20\%$)
- **Dominant share** of a user: the **fraction** of the **dominant resource** she is allocated.
 - User 1 dominant share is **25%**.

Dominant Resource Fairness (DRF)

- Apply **max-min fairness** to **dominant shares**: give every user an equal share of her dominant resource.
- **Equalize** the **dominant share** of the users.
 - Total resources: $\langle 9\text{CPU}; 18\text{GB} \rangle$
 - User 1 wants $\langle 1\text{CPU}; 4\text{GB} \rangle$; Dominant resource: RAM $1/9 < 4/18$
 - User 2 wants $\langle 3\text{CPU}; 1\text{GB} \rangle$; Dominant resource: CPU $3/9 > 1/18$

► $\max(x, y)$

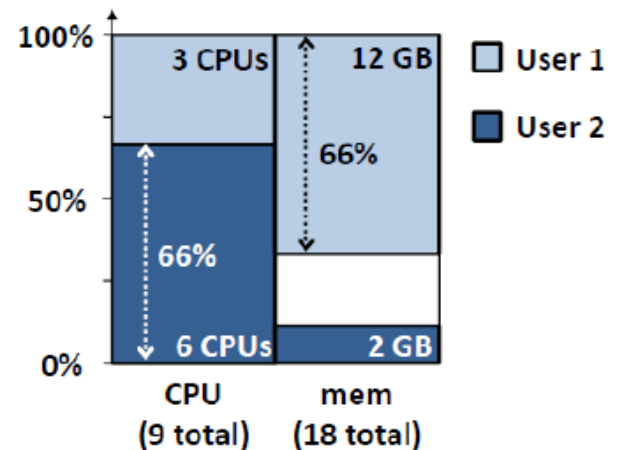
$$x + 3y \leq 9$$

$$4x + y \leq 18$$

$$\frac{4x}{18} = \frac{3y}{9}$$

User 1: $x = 3$: $\langle 33\% \text{CPU}, 66\% \text{GB} \rangle$

User 2: $y = 2$: $\langle 66\% \text{CPU}, 16\% \text{GB} \rangle$

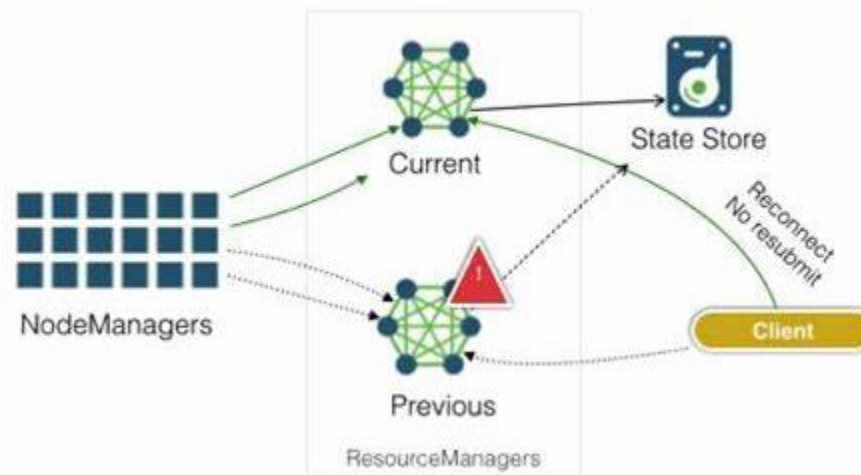


Resource Manager High Availability

- RM is SPOF before to Hadoop v2.4
- Backup pair: Active-Standby
- Transition-to-active
 - By the admin or the failover-controller based automatic failover
- Automatic failover
 - The master has an option to embed the Zookeeper based ActiveStandbyElector to decide which RM should be Active

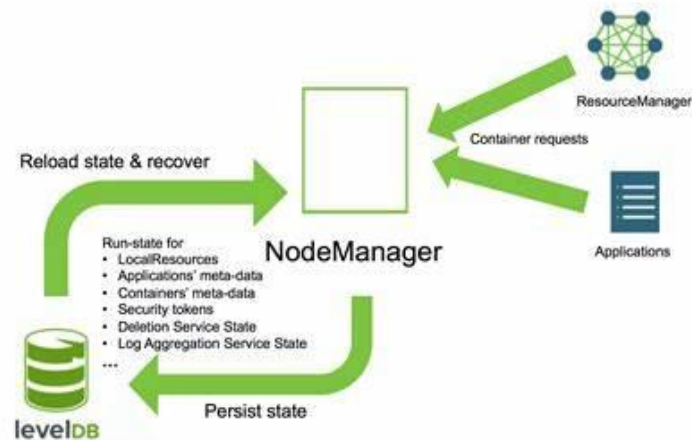
Resource Manager Restart

- Enhance RM with a pluggable state-store to persist application/attempt state and other credentials information .
- RM will reload this information from state-store upon restart and re-kick the previously running applications.
- Users are not required to re-submit the applications.



Node Manager Restart

- The NM stores any necessary state to a local state-store as it processes container-management requests.
 - Upon restarting, NM recovers by
 - first loading state for various subsystems and
 - then letting those subsystems perform recovery using the loaded state.
- Work-preserving NodeManager Restart



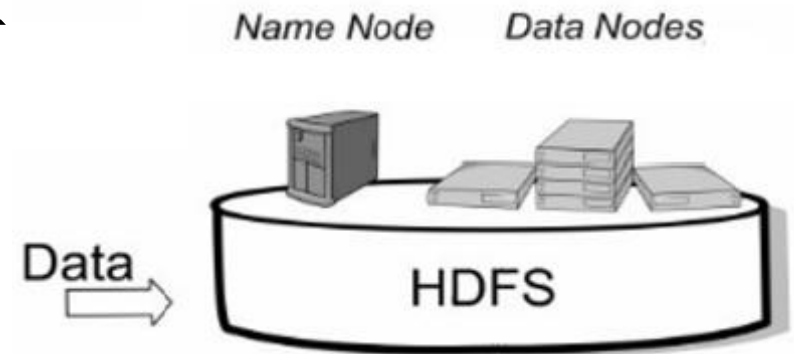


Outline

1. Basics
2. MapReduce
3. YARN
4. HDFS
5. ZooKeeper

HDFS

- Google GFS的开源实现
- 容量大： terabytes or petabytes
 - 将数据保存到大量的节点当中
 - 支持很大单个文件
- 高可靠性、快速访问、高可扩展
 - 按块存储、并行读取、效率高
 - 大量的数据复制
- HDFS是针对MapReduce设计
 - 尽可能本地局部性进行访问



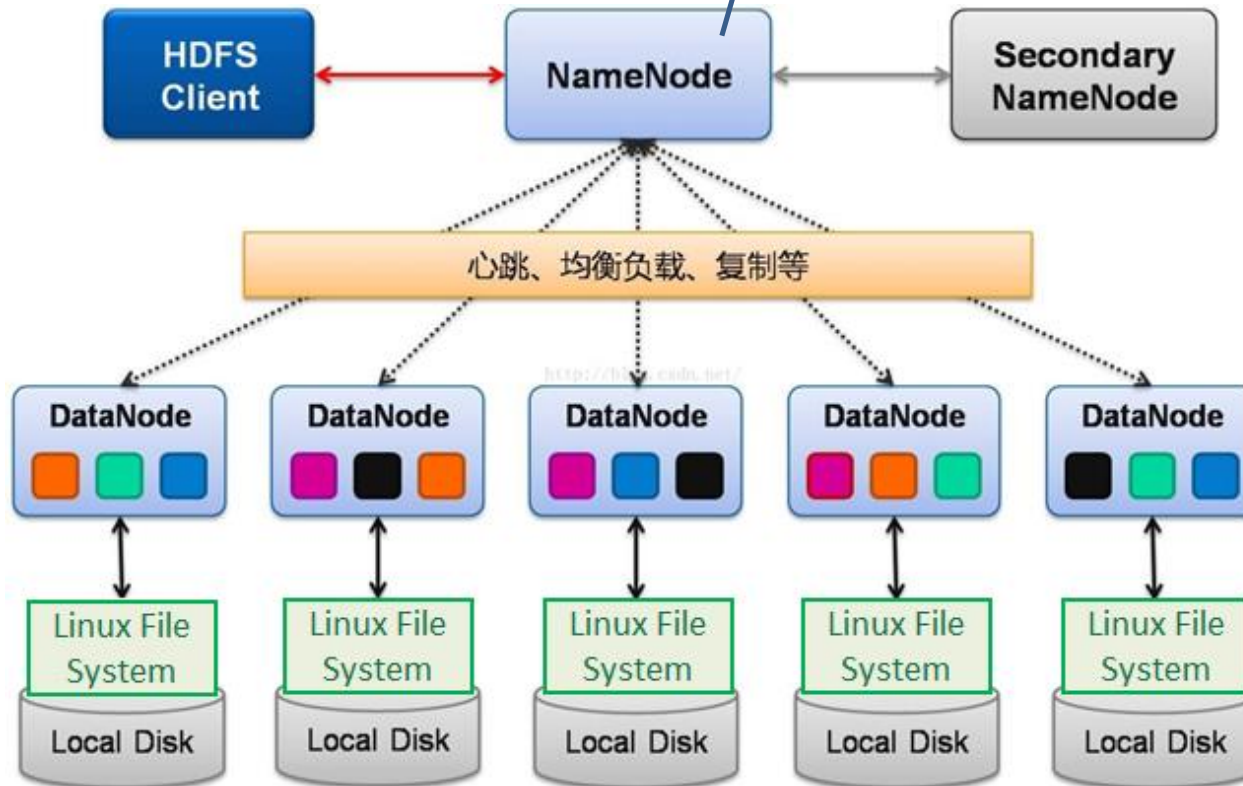
HDFS适用场景

- 大量地从文件中顺序读
 - HDFS对顺序读进行了优化
 - 随机的访问负载较高
- 数据支持一次写入，多次读取
 - 不支持数据更新（但可以直接进行文件替换）
- 数据不进行本地缓存
 - 文件很大，且顺序读

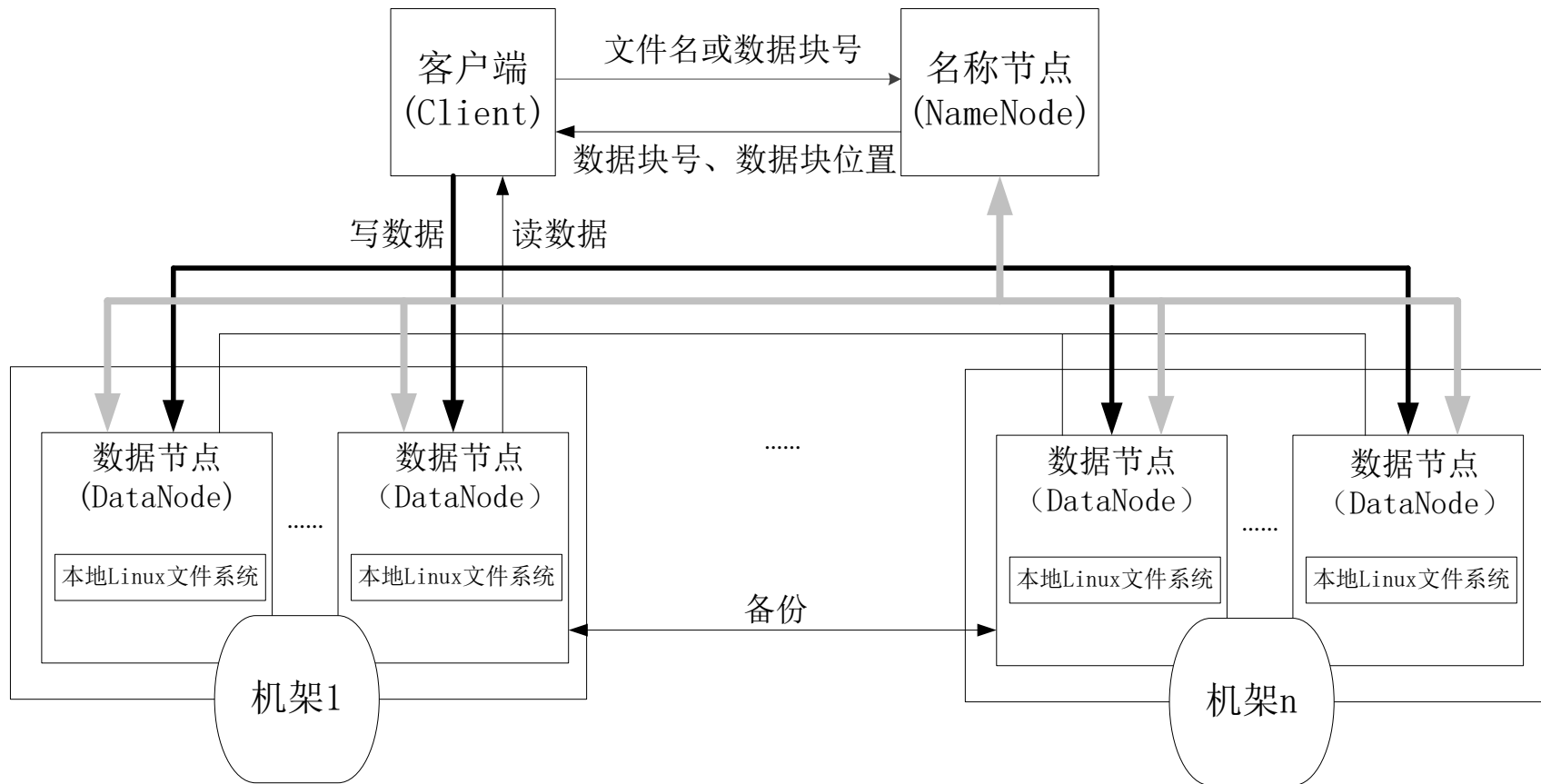
HDFS系统架构

- Master-Slave架构
- 节点的本地OS具体管理本地资源
- HDFS处于中间件层次，整合集群存储

支持联邦模式：
多个NameNode各自负责命名空间的一部分，如“/user”



HDFS系统架构



HDFS的基本设计

- 基于块的文件存储
- 使用传统的分级文件命名体系
 - 命名空间包含目录、文件和块
- 数据块有复制放置
 - 副本的默认数目是3
 - Erasure coding in Hadoop 3.0
- 默认的块的大小是128MB
 - 减少元数据的量
 - 有利于顺序读写
 - （在磁盘上数据顺序存放）

HDFS节点通信

- 所有HDFS通信协议都是基于TCP/IP协议
- Client-NameNode:
 - 通过一个可配置的端口向NameNode主动发起TCP连接;
 - 使用客户端协议与NameNode进行交互。
- NameNode-DataNode:
 - 使用数据节点协议进行交互
- Client-DataNode:
 - 通过RPC通信
- 名称节点不会主动发起RPC，而是响应来自客户端和数据节点的RPC请求

HDFS数据

NameNode

- 存储元数据
- 元数据保存在内存中
- 保存文件, block, datanode 之间的映射关系

DataNode

- 存储文件内容
- 文件内容保存在磁盘
- 维护了block id到datanode本地文件的映射关系

Name Node: Stores Meta Data

Meta Data:
/data/pristine/catalina.log.> 1, 2, 4
/data/pristine/myfile. >3,5

Data Node 1

1

2

4

5

Data Node 2

5

2

3

Data Node 3

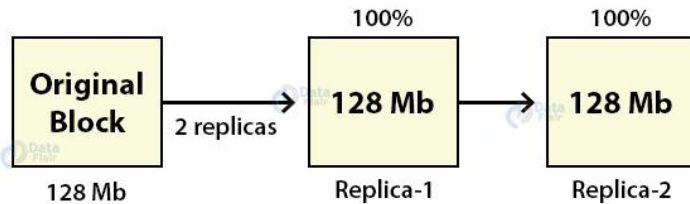
4

1

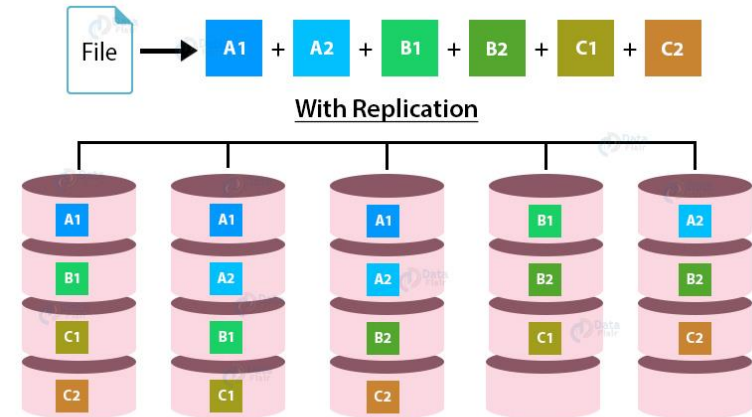
3

HDFS数据

Block Replication



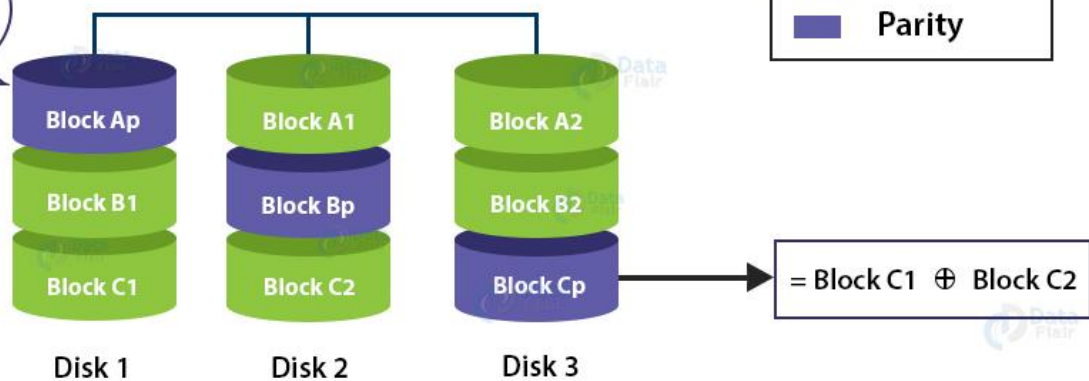
Space Used = $128 \times 3 = 384$ Mb
Therefore, 200% Storage Overhead



Therefore, 18 Blocks of Disk Space

Reduced storage overhead as 1 Parity Block stored for 2 Data Blocks

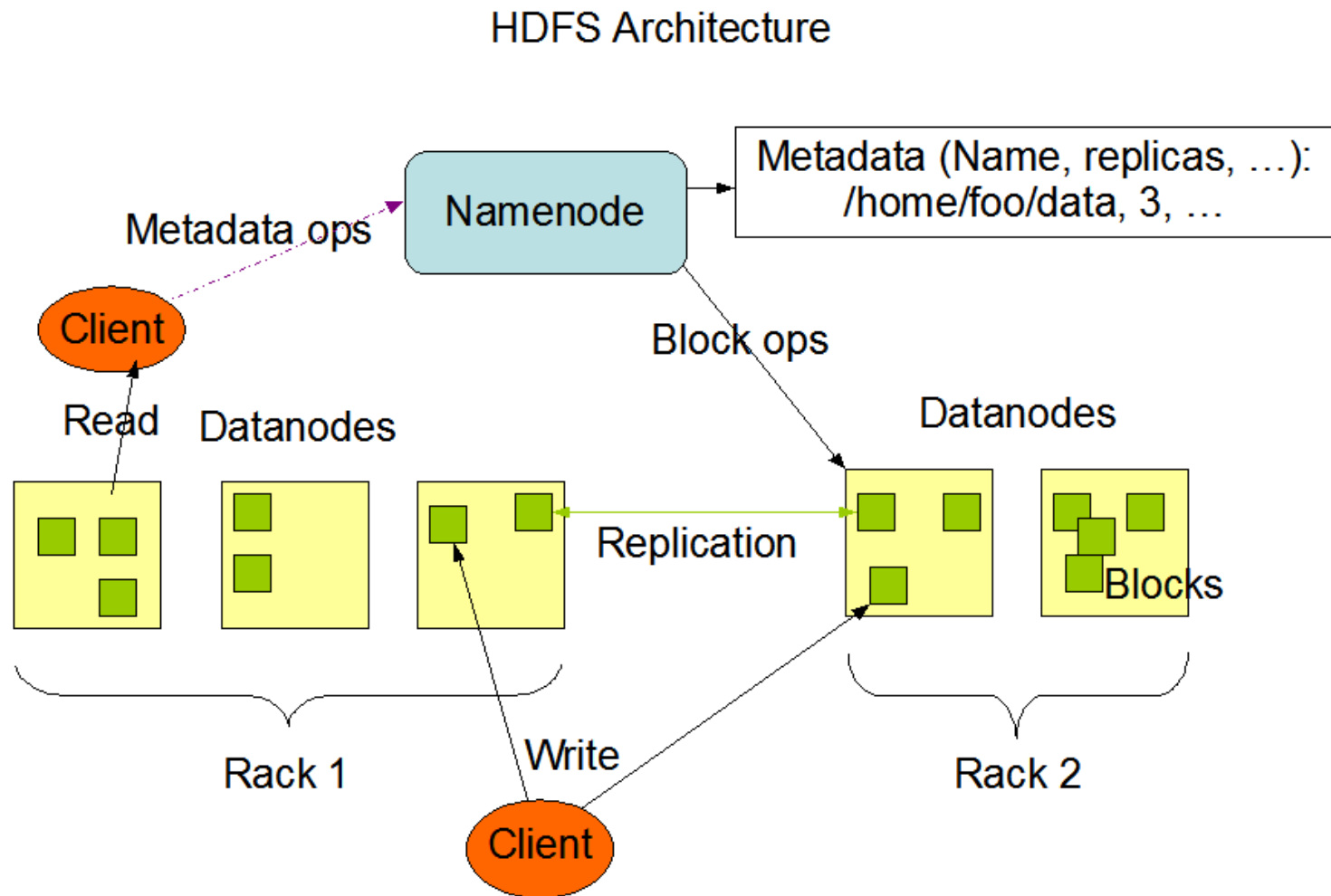
Erasure Coding



Limitations

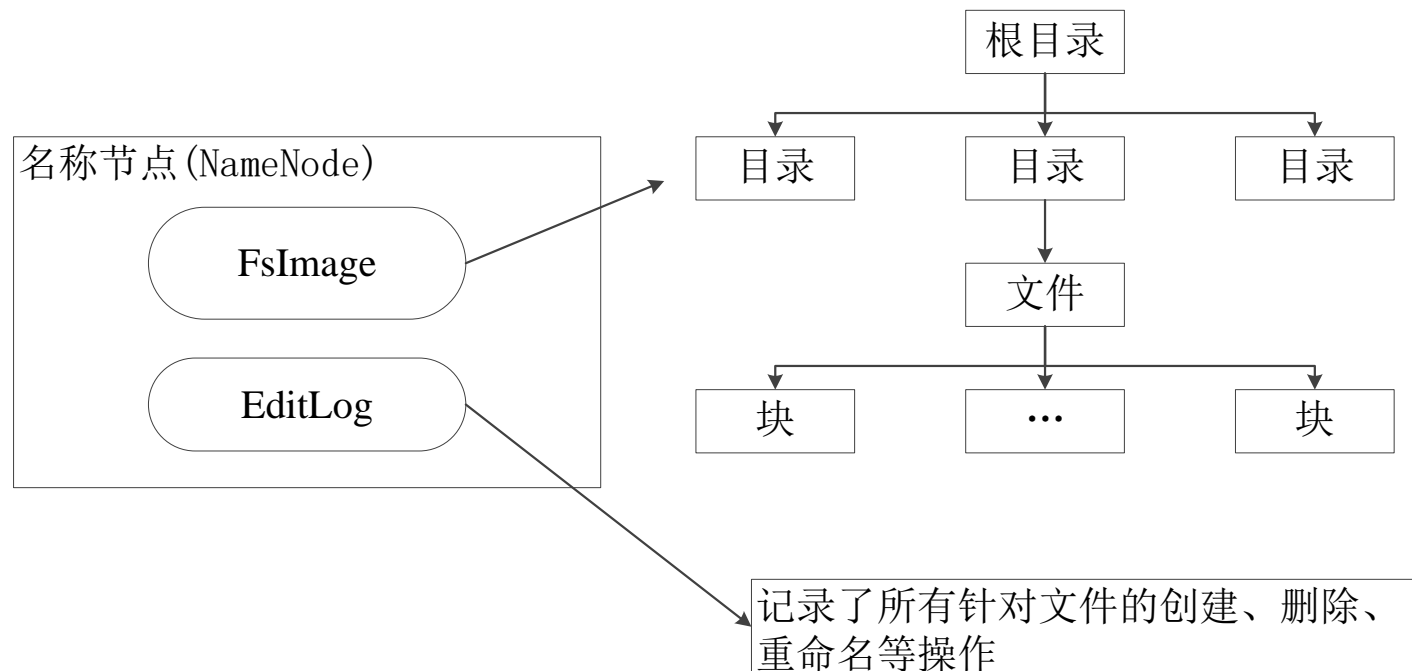
- Do not support certain file writes like hflush, hsync, and append.
- Need additional processing/computing
- Additional overhead in data reconstruction due to remote reads.

HDFS读写



NameNode数据结构

- 两个关键数据结构：
 - FsImage: 维护文件目录树及文件和目录的元数据
 - EditLog: 记录文件的创建、删除、重命名等操作



NameNode

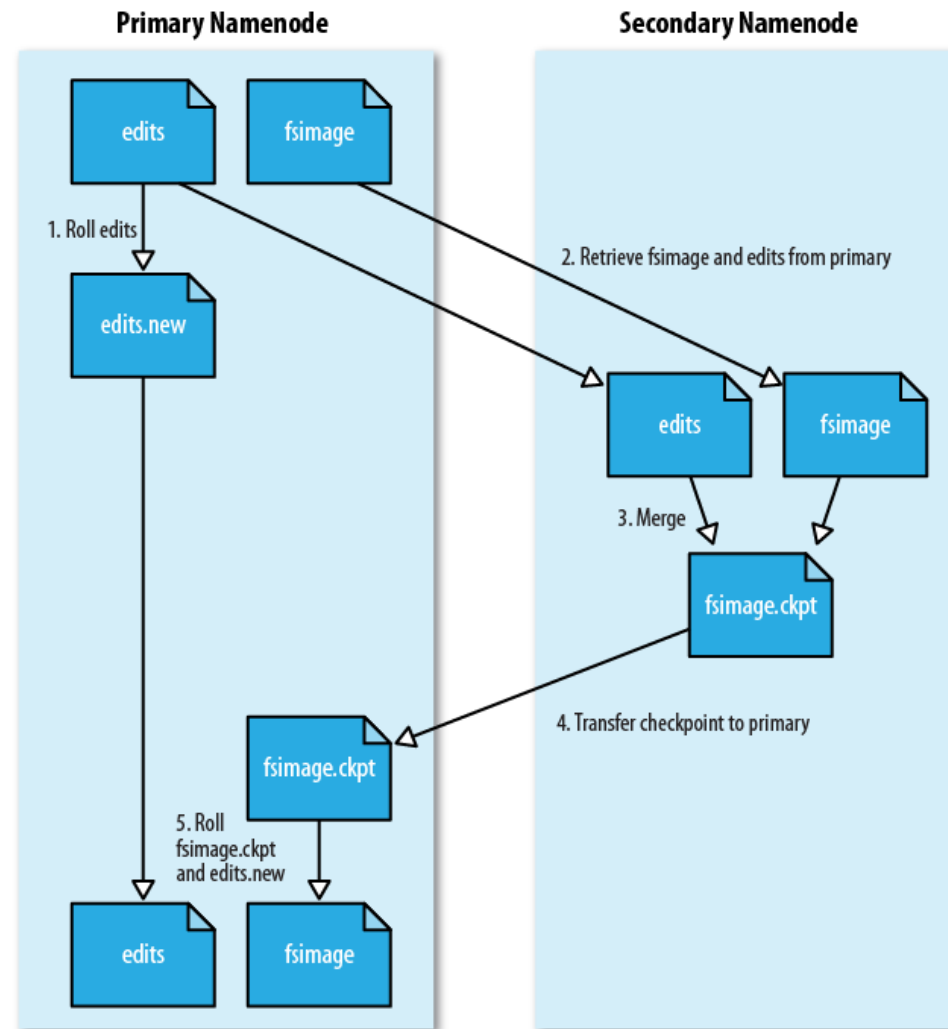
- 名称节点启动：
 - 先将FslImage文件加载到内存
 - 然后执行EditLog文件中的各项操作
 - 创建一个新的FslImage文件和一个空的EditLog文件
- 元数据更新：
 - NameNode启动后，更新操作写到EditLog文件
 - 因为FslImage文件一般很大（GB级别），直接更新太慢
 - 往EditLog文件里面写就会很快：EditLog 要小很多
 - 每次执行写操作之后，且在向客户端发送成功代码之前，edits文件都需要同步更新

Secondary NameNode

- 保存元数据备份
- 协助执行操作日志
 - 加速NameNode启动

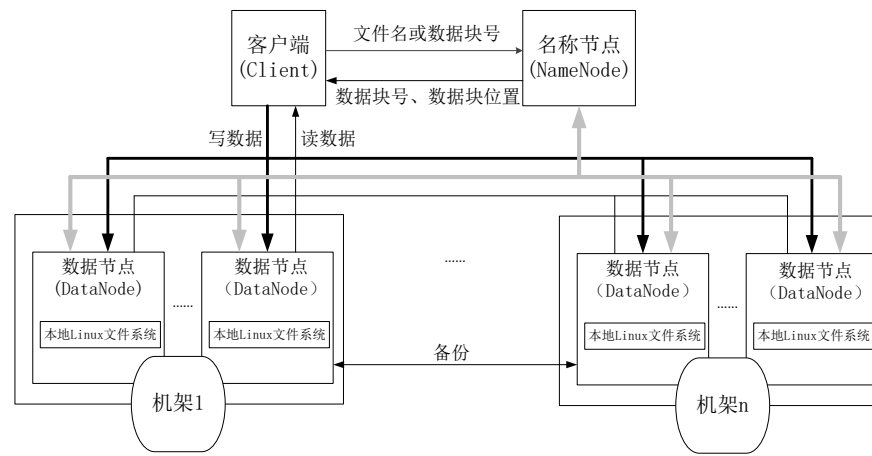
工作流程:

- 1) 请求: 定期与NameNode通信, 请求其停止使用EditLog文件, 暂时将新的写操作写到一个新的文件edit.new中;
(瞬间完成, 上层写日志的函数完全感觉不到)
- 1) 下载: 通过HTTP GET从NameNode下载FsImage和EditLog文件;
- 2) 执行: 将FsImage载入内存, 逐条执行EditLog中的操作, 更新FsImage, 即EditLog和FsImage文件合并;
- 3) 推送: 通过post方式将新的FsImage文件发送到NameNode节点。
- 4) 替换: NameNode将用新的FsImage替换旧的FsImage文件, 用edit.new替换EditLog文件。



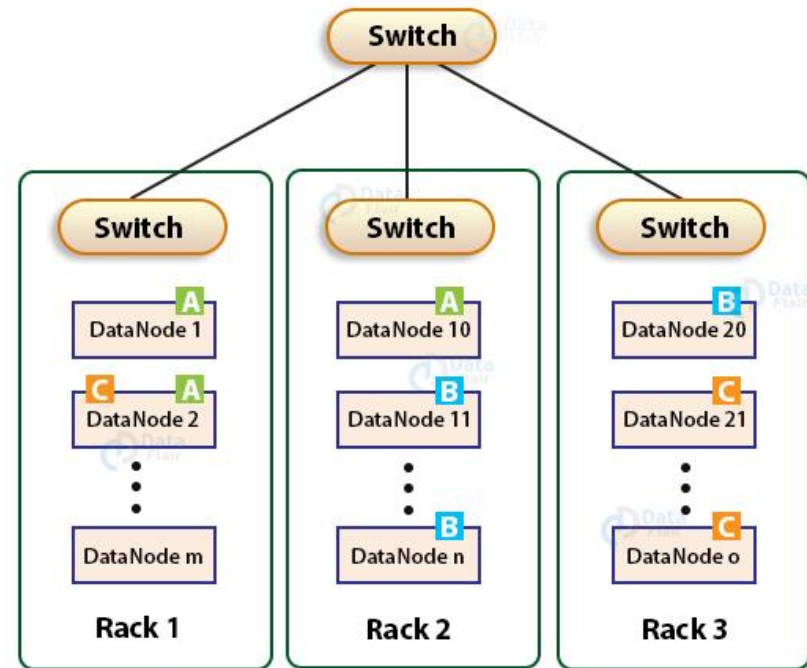
DataNode

- A Block Server
 - Stores data in local file system
 - Stores meta-data of a block: checksum
 - Serves data to clients
- Block Report
 - Periodically sends reports of existing blocks to NameNode
- Facilitate Pipelining of Data
 - Forwards data to other specified DataNodes



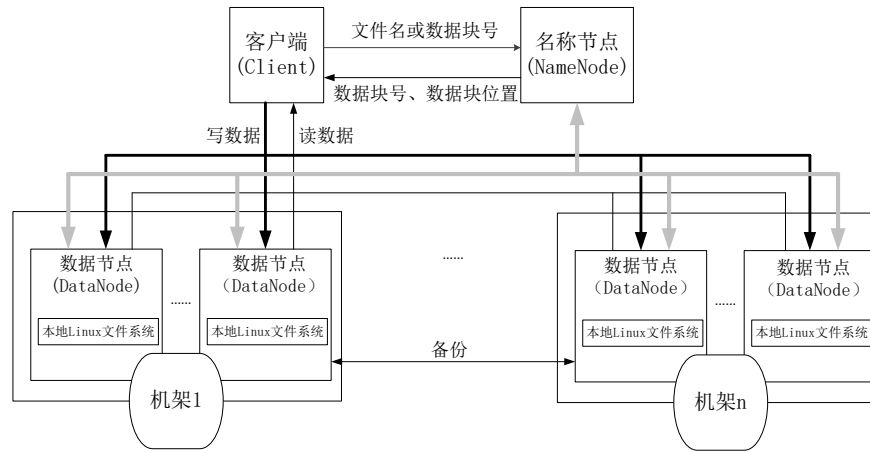
Block Placement

- Replication Strategy
 - One replica on local node
 - Second/third replica the same remote rack
 - Additional replicas are randomly placed
- Clients read from nearest replica
 - Based on rack id, ...
- Block Cache
 - A block may be cached by at most one DataNode



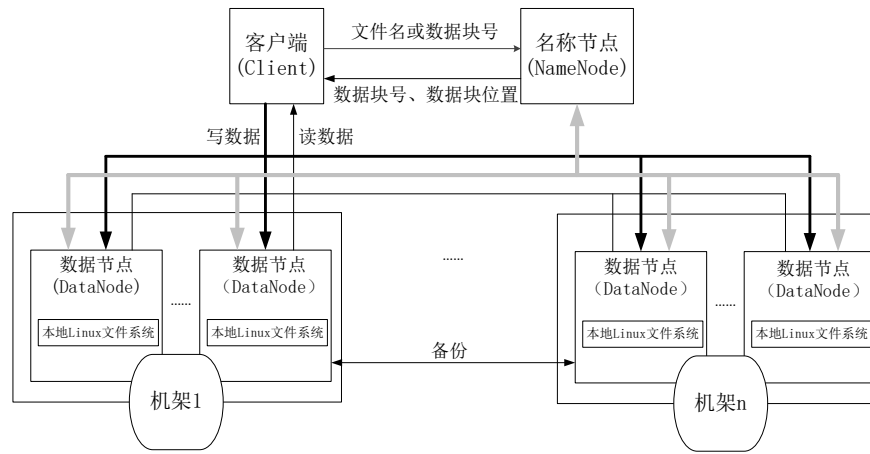
Data Correctness

- Use Checksums to validate data – CRC32
- File Creation
 - Client computes checksum per 512 byte
 - DataNode stores the checksum
- File Access
 - Client retrieves the data and checksum from DataNode
 - If validation fails, client tries other replicas



Data Pipelining

- Client retrieves a list of DataNodes on which to place replicas of a block
- Client writes block to the first DataNode
- The first DataNode forwards the data to the next DataNode in the Pipeline
- When all replicas are written, the client moves on to write the next block in file



HDFS FT and HA

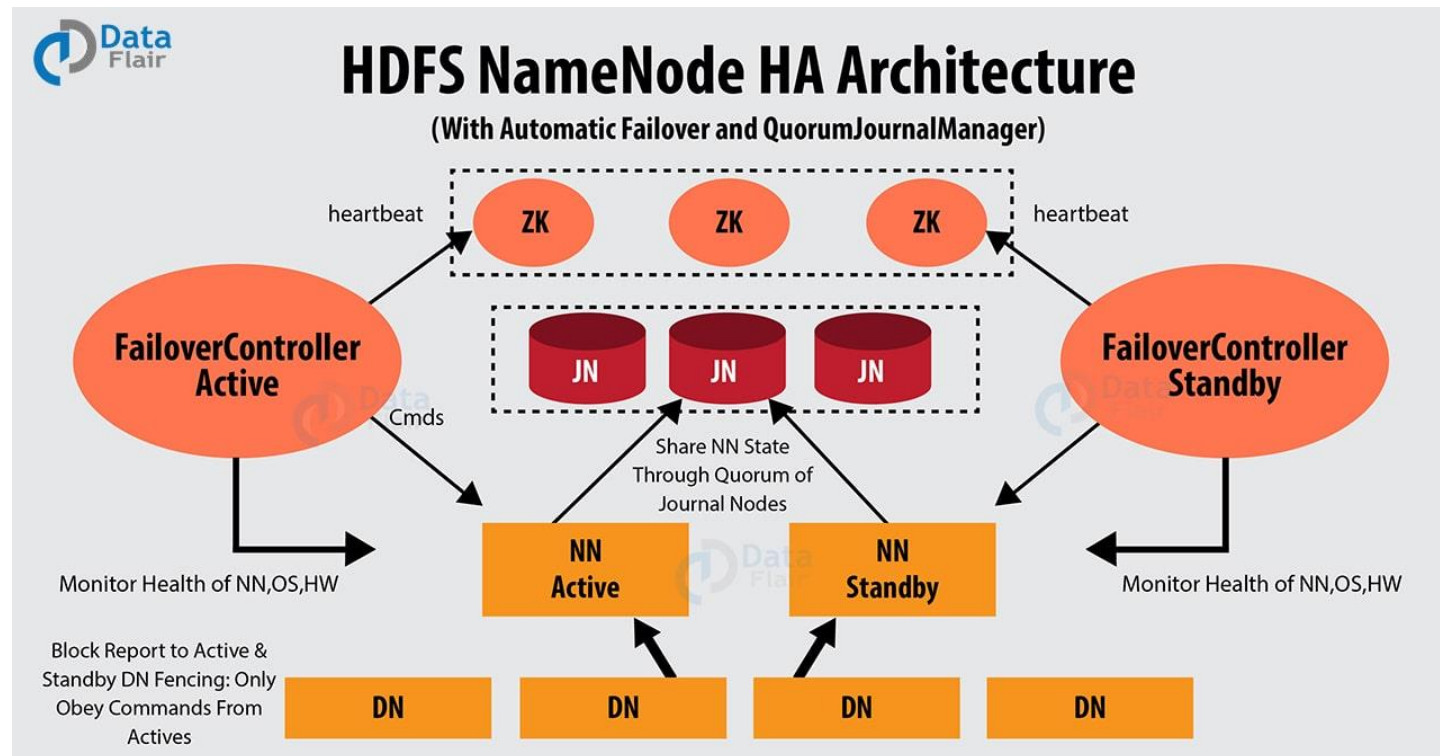
- DataNode Failure:
 - Replication, heartbeat
- Data Integrity:
 - Checksum
- Metadata Failure:
 - Multiple FsImage and EditLog, Checkpoint
- NameNode failure?

NameNode HA

- Introduced in Hadoop 2.0
- Secondary NameNode: active-standby
- Quorum journal manager (QJM)
 - A group of nodes for sharing edit logs between NameNodes
 - At least 3
 - Can tolerate $N/2$ failures
- Normal operations
 - Active logs its modification operations to all QJM nodes
 - Standby reads the edits from the journal nodes and applies to its own namespace in a constant manner
 - Datanodes send block reports and heartbeat to both

NameNode HA

- Failover controller
 - Daemon at each NameNode
 - Trigger switch upon failure
 - Help from Zookeeper to guarantee at most one “active”



Fencing of NameNode

- Unreliable failure detection (Paxos, ZooKeeper)
 - Too slow speed or network partition
 - Two “living” Namenodes?
- Fencing (规避):
 - To guarantee only one NameNode can write
 - Journal nodes allowing only one namenode to be the writer at a time.
 - The standby namenode takes the responsibility of writing to the journal nodes and prohibit any other namenode to remain active.

Can this really “solve” the problem?

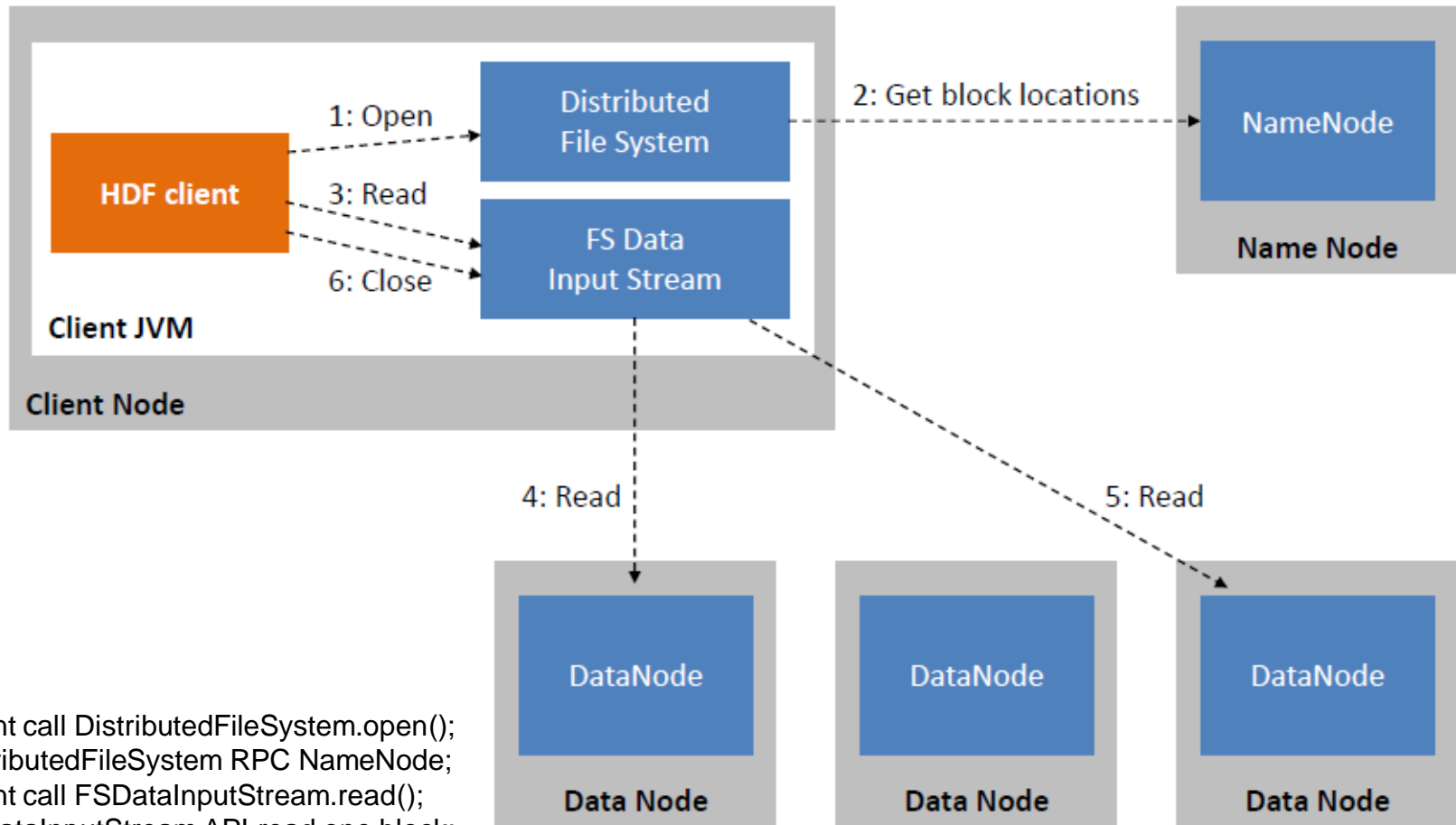
HDFS权限控制与安全特性

- 类似于POSIX的安全特性
 - 不完全，主要预防操作失误
 - 三类权限：只读、写入、执行
 - Owner、group、mode
- 用户：
 - 当前登录的用户名，即使用Linux自身设定的用户与组的概念
- 安全性弱：不能保证操作的完全安全性

负载均衡

- 加入一个新节点的步骤
 - 配置新节点上的hadoop程序
 - 在Master的slaves文件中加入新的slave节点
 - 启动slave节点上的DataNode，会自动去联系NameNode，加入到集群中
- Balancer类用来做负载均衡
 - 默认的均衡参数是10%范围内
 - `bin/start-balancer.sh -threshold 5`

Workflow of Reading



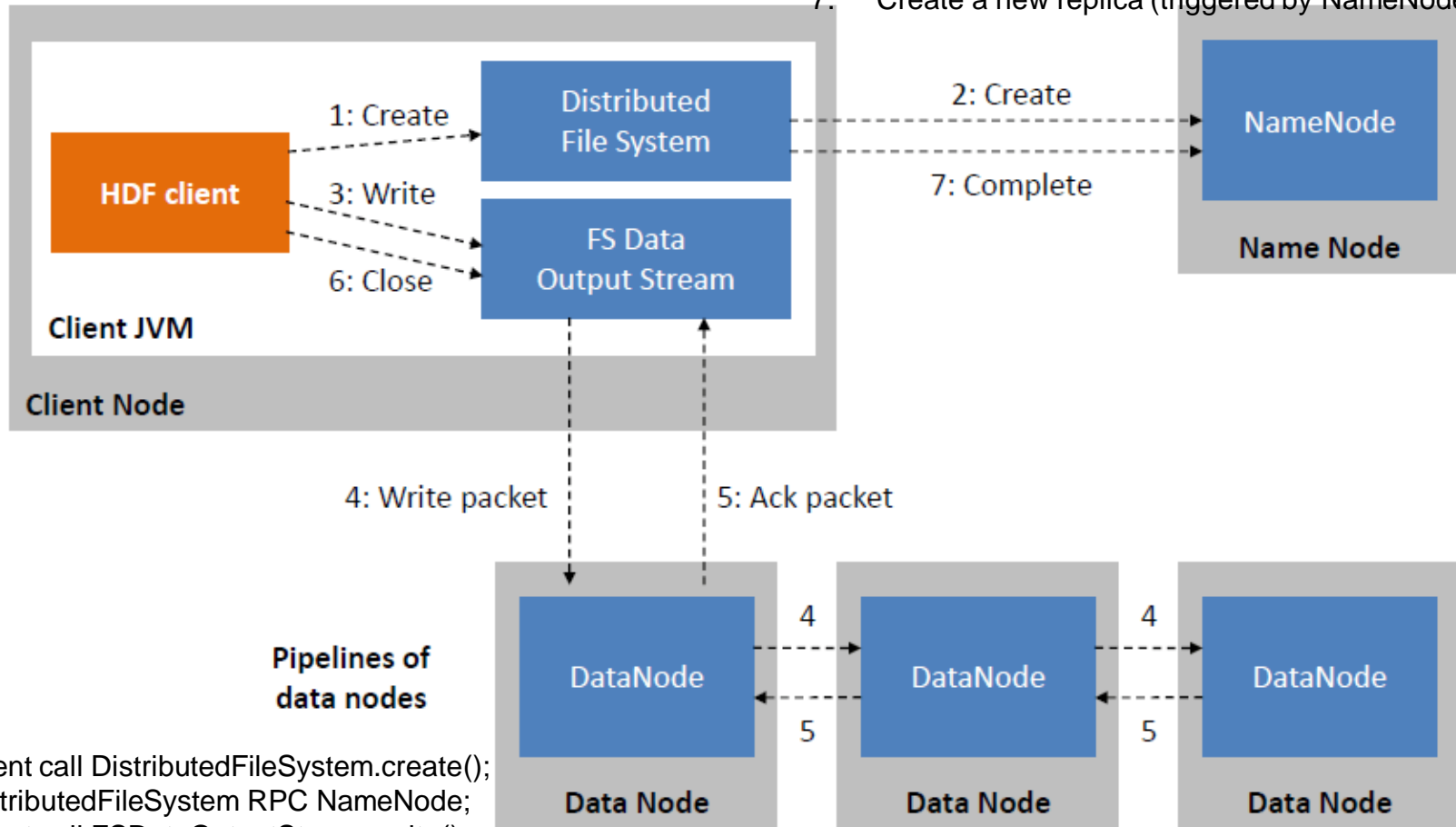
1. Client call `DistributedFileSystem.open();`
2. `DistributedFileSystem` RPC `NameNode`;
3. Client call `FSDaataInputStream.read();`
4. `FSDaataInputStream` API read one block;
5. `FSDaataInputStream` API read another block;
6. Client call `DistributedFileSystem.close();`

Workflow of Writing



Upon failure at DataNode during writing:

1. Close the pipeline;
2. Add packets back to queue;
3. Inform NameNode about failed block;
4. Delete failed DataNode from pipeline;
5. Construct new pipeline for the two correct DataNodes;
6. Write rest blocks to correct nodes;
7. Create a new replica (triggered by NameNode).



1. Client call DistributedFileSystem.create();
2. DistributedFileSystem RPC NameNode;
3. Client call FSDataOutputStream.write();
FSDataOutputStream split data into packets and write to data queue;
4. DataStreamer select DataNodes, request blocks from NameNode, and write via Pipeline;
5. DataNodes send Ack, FSDataOutputStream collects all Ack of the Pipeline;
6. Client call FSDataOutputStream.close();
7. DistributedFileSystem RPC NameNode to complete.



Outline

1. Basics
2. MapReduce
3. YARN
4. HDFS
5. ZooKeeper

ZooKeeper

- A highly-available service for distributed coordination and agreement
 - **designed** to relieve developers from writing coordination logic code
 - Coordination problems:
 - leader election, mutual exclusion,
 - group membership, barriers, etc.
- Developed at Yahoo! Research
- Started as sub-project of Hadoop
- Now a top-level Apache project



Design Principles

- API is wait-free
 - No blocking primitives in ZooKeeper
 - Blocking can be implemented by a client
 - No deadlocks
- Guarantees
 - Client requests are processed in FIFO order
 - Writes to ZooKeeper are linearizable
- Clients receive notifications of changes before the changed data becomes visible

High Performance and Reliability

- Server replication
 - Typically 5 nodes
- Client side cache
 - E.g., caching ID of the current leader to avoid probing every time
- New leader elected?
 - Naive solution: polling (not optimal)
 - Watch mechanism: clients can watch for an update of a given data object

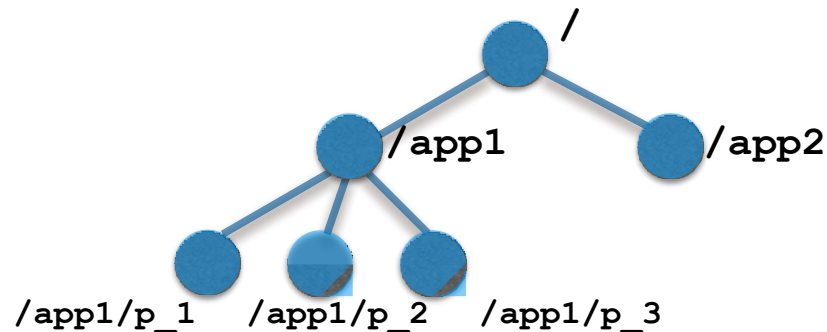
ZooKeeper is optimised for read-dominant operations!

ZooKeeper Terminology

- **Client:** user of the ZooKeeper service
- **Server:** process providing the ZooKeeper service
- **znode: in-memory** data node in ZooKeeper
- **Update/Write:** any operation which modifies the state of znodes
- Clients establish a **session** when connecting to ZooKeeper

Data Model

- znodes organised in a hierarchical namespace, i.e., a data tree
- It is a filesystem
 - znodes are referred to by UNIX style file system paths
 - Manipulated by clients through API



All znodes store data **(file like)** &
can have children **(directory like)**.

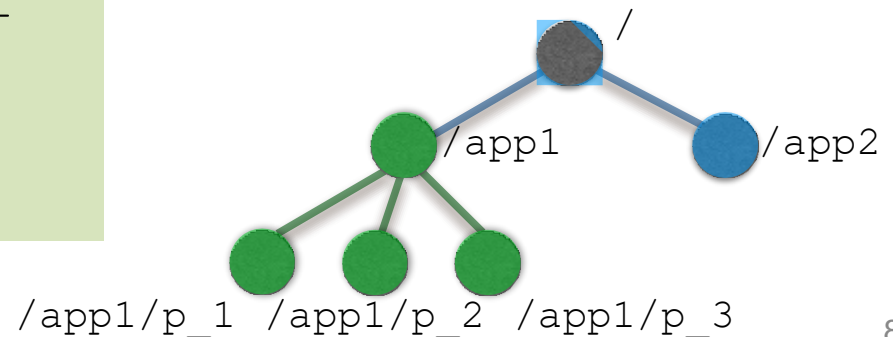
znode

- znode is not designed for general data storage (usually require storage in the order of kilobytes)
- znodes map to abstractions of the client application
- Clients manipulate znodes by creating and deleting them

Group membership protocol:

Client process p_i creates znode p_i under $/app1$.

$/app1$ persists as long as the process is running.



APIs

. String create(path, data, flags)

- creates a znode with path name path, stores data in it and sets flags (ephemeral, sequential)

. void delete(path, version)

- deletes the anode if it is at the expected version

. Stat exists(path, watch)

- watch flag enables the client to set a watch on the znode

. (data, Stat) getData(path, watch)

- returns the data and meta-data of the znode

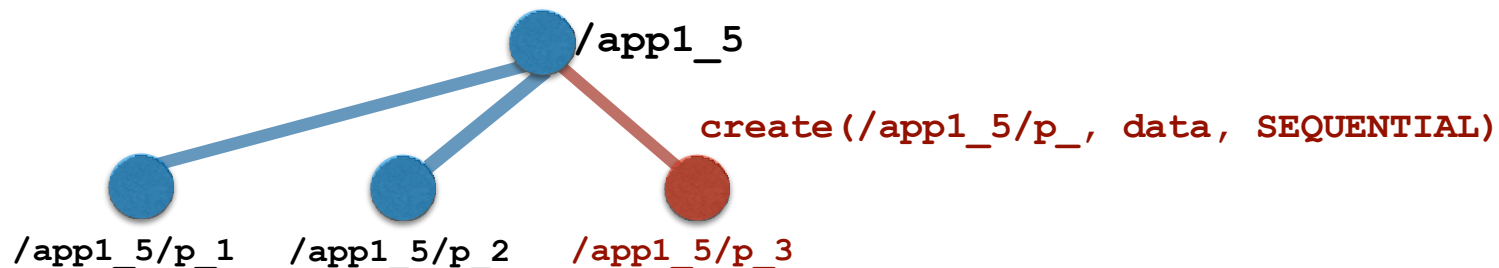
. Stat setData(path, data, version)

- writes data if the version number is the current version of the znode

. String[] getChildren(path, watch)

znode Flags

- **EPHEMERAL** flag:
 - znode will be deleted at the end of the client's session
- **SEQUENTIAL** flag:
 - monotonically increasing counter appended to a znode's path
 - counter value of a new znode under a parent is always larger than value of existing children



znode Watch Flag

- Clients can issue read operations on znodes with a watch flag (true or false)
- Server **notifies** the client when the information on the znode has changed
- Watches are **one-time** triggers associated with a session (unregistered once triggered or session closes)
- Watch notification indicates the change, not new data

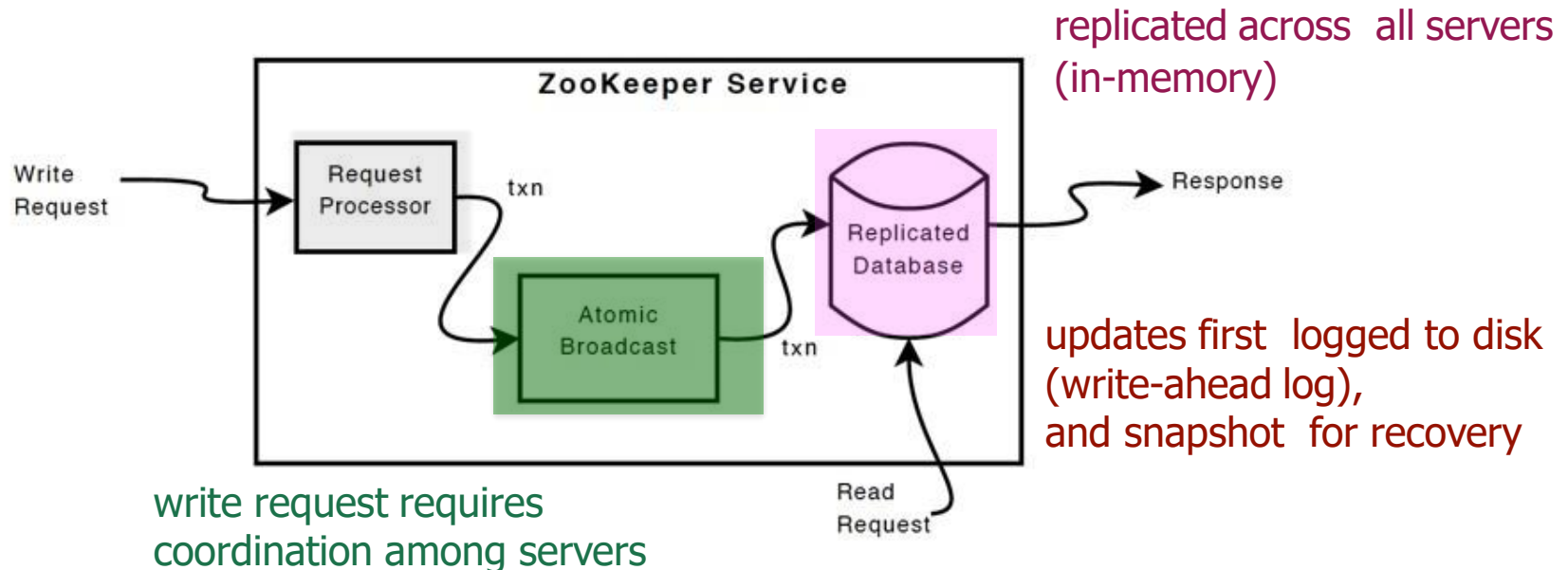
```
getData (/app1/lock1, true)
```

Session Operations

- Initiation:
 - A client connects to ZooKeeper and initiates a session
- Timeout:
 - session has an associated **timeout**
 - ZooKeeper considers a client faulty if nothing received from its session upon timeout
- End:
 - faulty client or explicitly ended by client

Major Operations

- Clients connect to exactly one server to submit requests
- read requests served from the local replica
- write requests are processed by an consensus protocol (leader initiates processing of the write request)



Example: group membership

Questions:

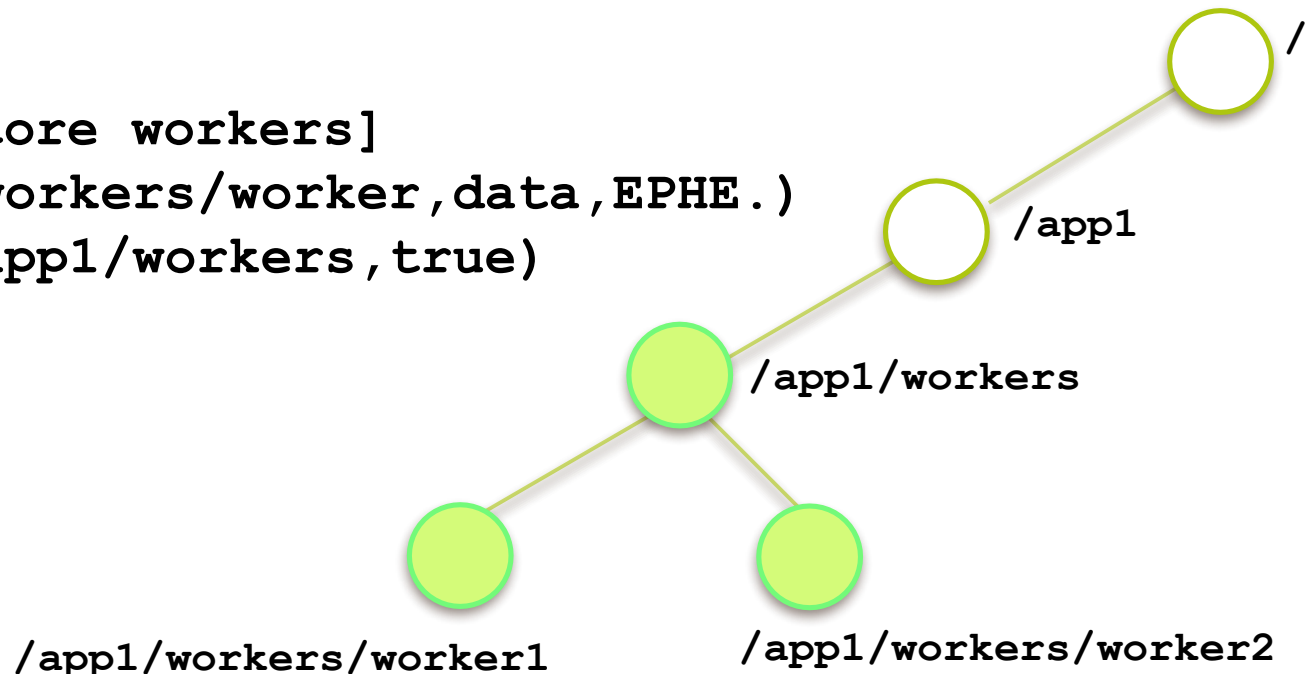
1. How can all workers (slaves) of an application **register themselves** on ZK?
2. How can a process find out about all **active** workers of an application?

- `String create(path, data, flags)`
- `void delete(path, version)`
- `Stat exists(path, watch)`
- `(data, Stat) getData(path, watch)`
- `Stat setData(path, data, version)`
- `String[] getChildren(path, watch)`

[a znode to store workers]

```
create (/app1/workers/worker, data, EPHE.)
```

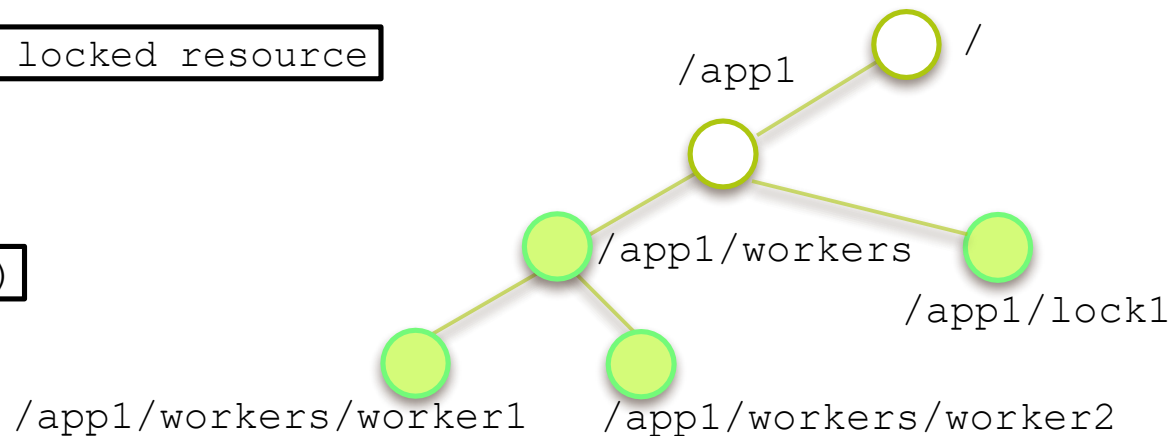
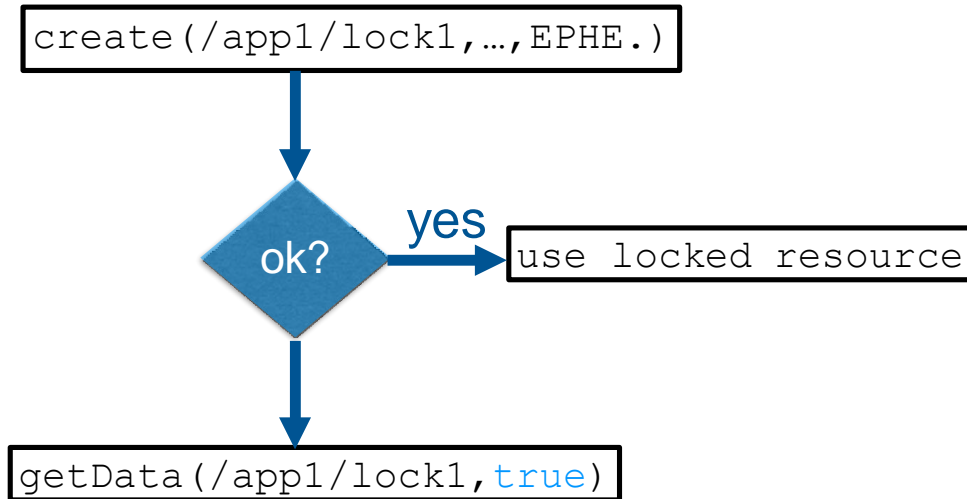
```
getChildren (/app1/workers, true)
```



Example: simple lock

Question:

How can all workers of an application use a single resource through a lock?

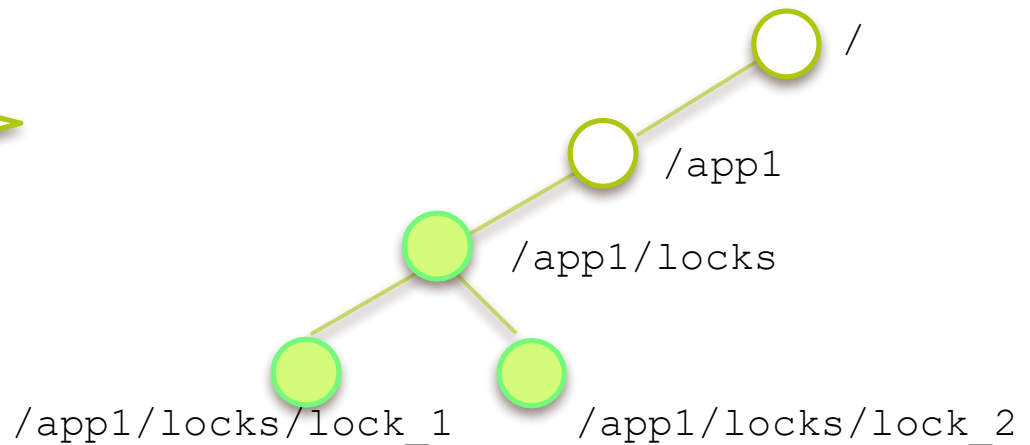
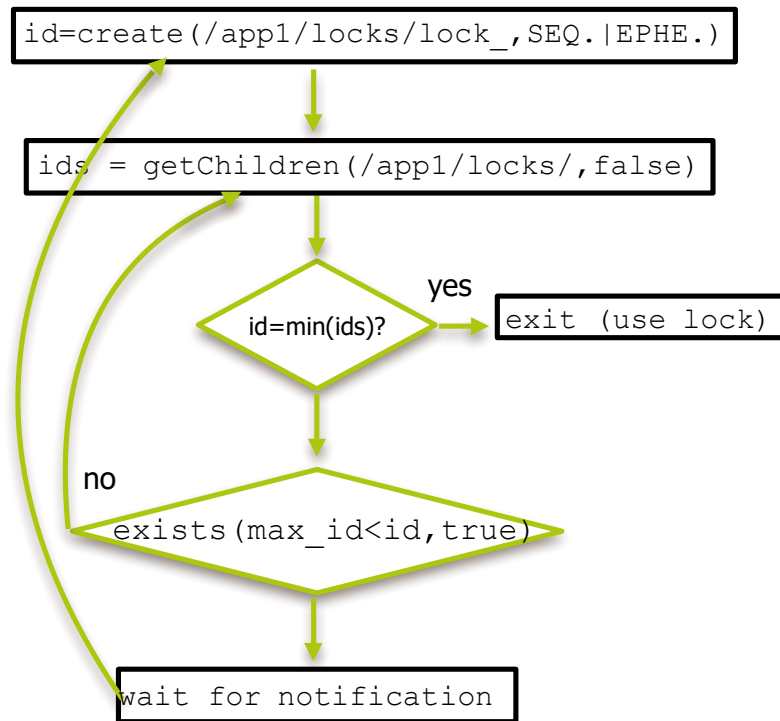


all processes compete at all times for the lock

Example: locking without herd effect

Question:

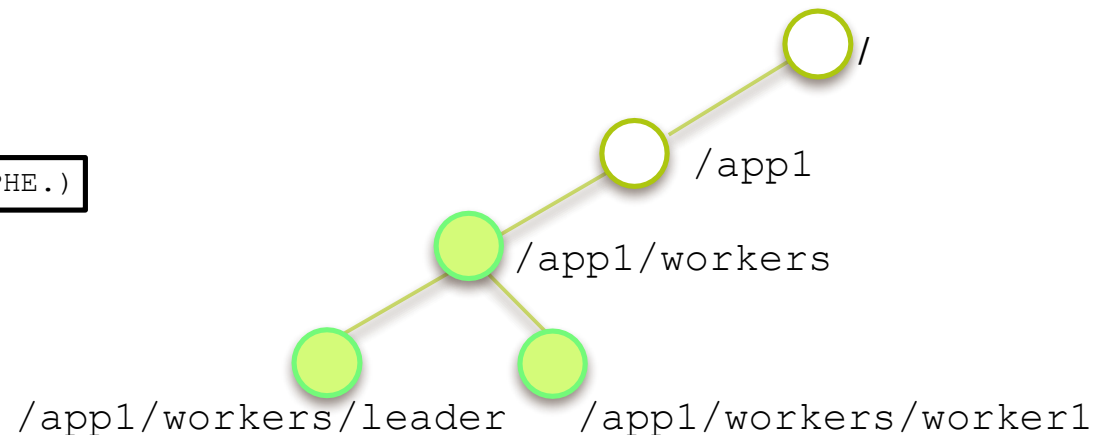
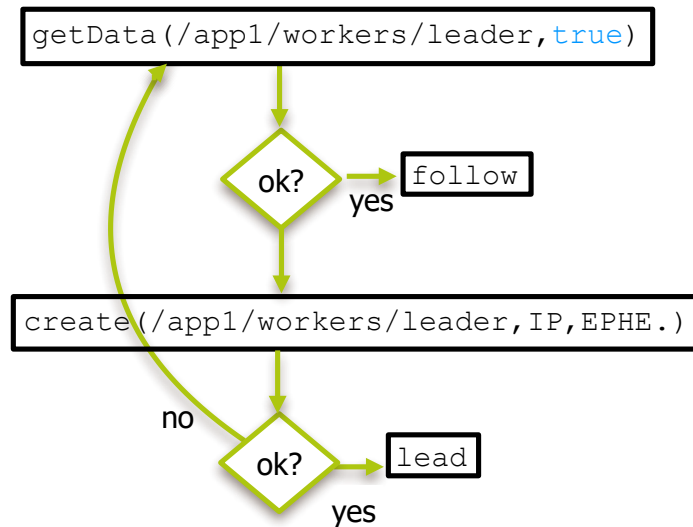
1. How can all workers of an application use a single resource through a lock with queuing?



Example: leader election

Question:

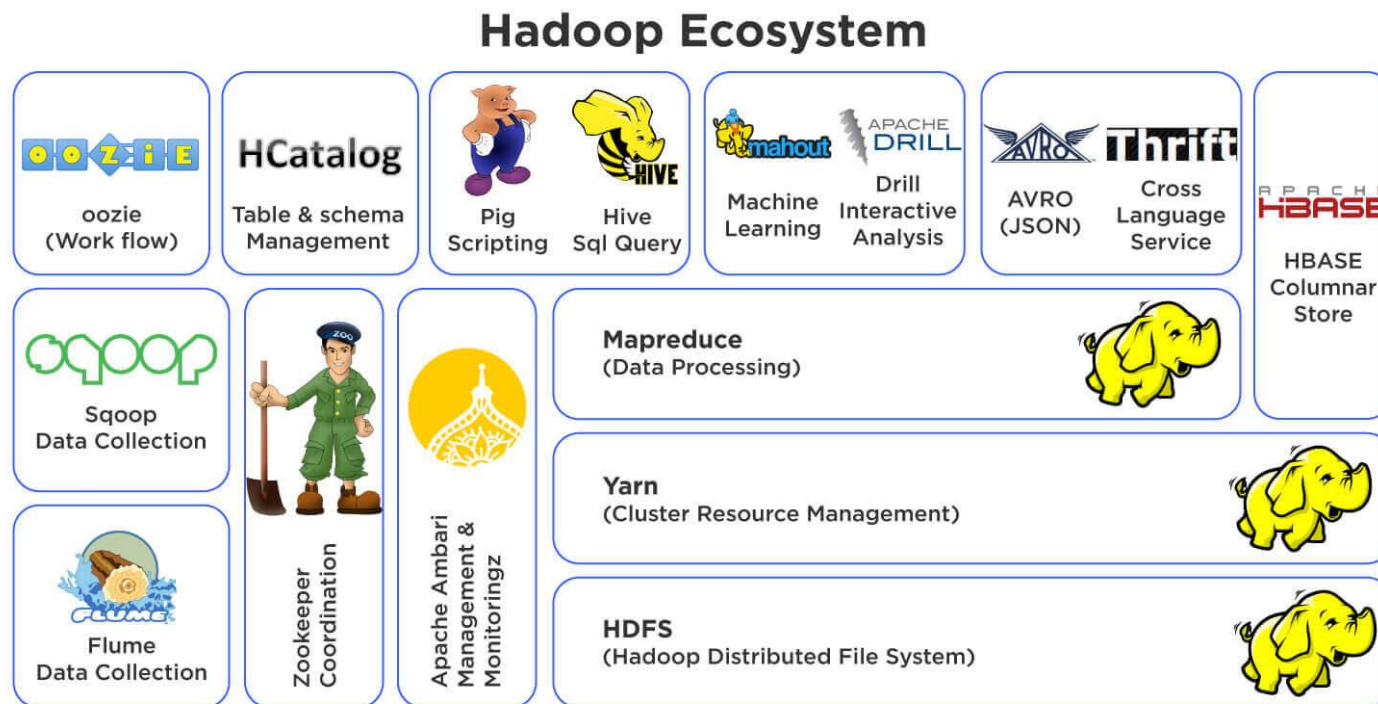
How can all workers of an application elect a leader among themselves?



if the leader dies, elect again ("herd effect")

A Summary

- Hadoop is a great distributed system
- Well developed and still developing
- MapReduce, YARN, HDFS, ZooKeeper, etc.





谢谢!