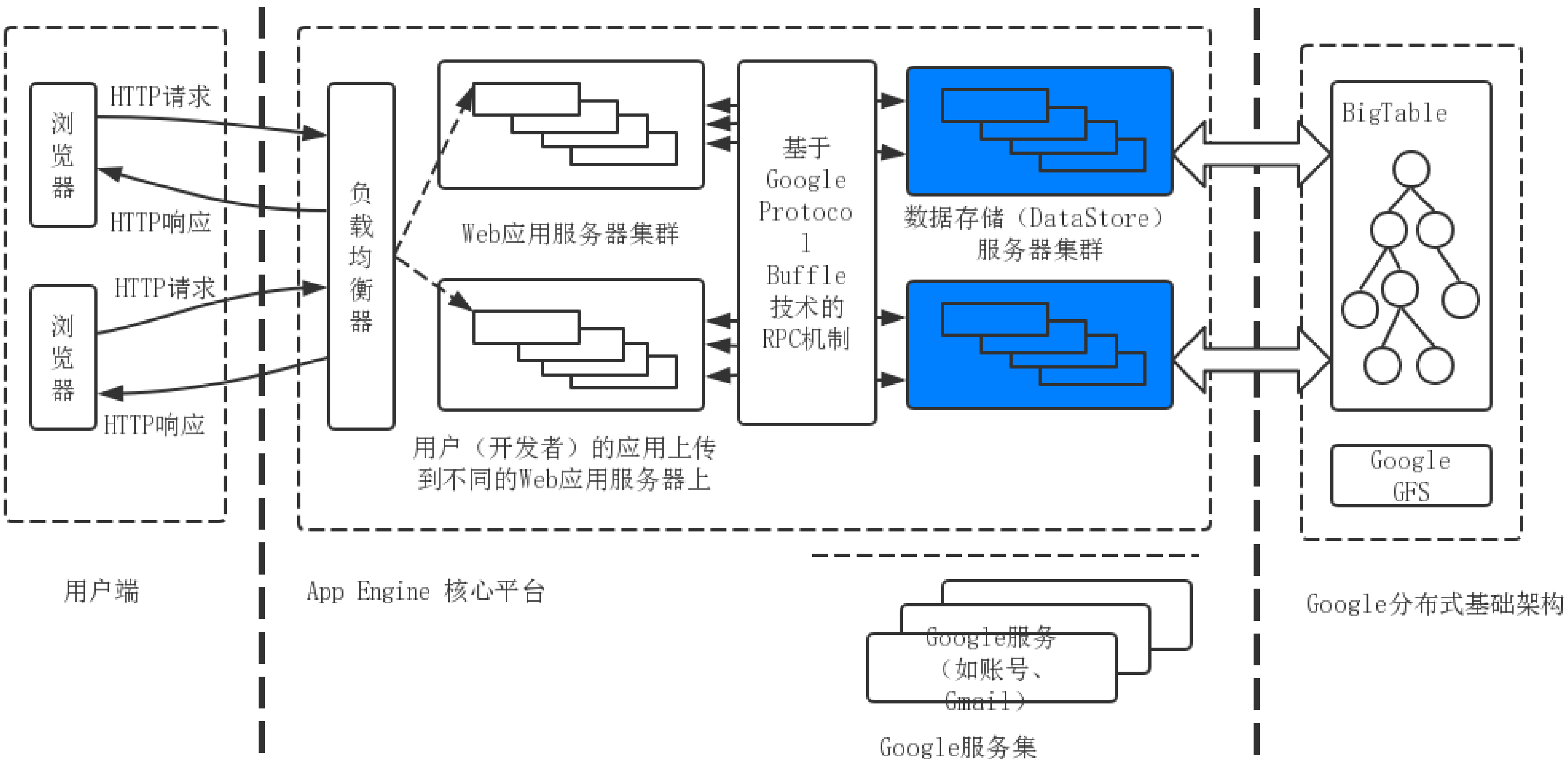
Chapter 4 大数据处理技术

§4.1分布式文件系统

§4.2 NoSQL数据库

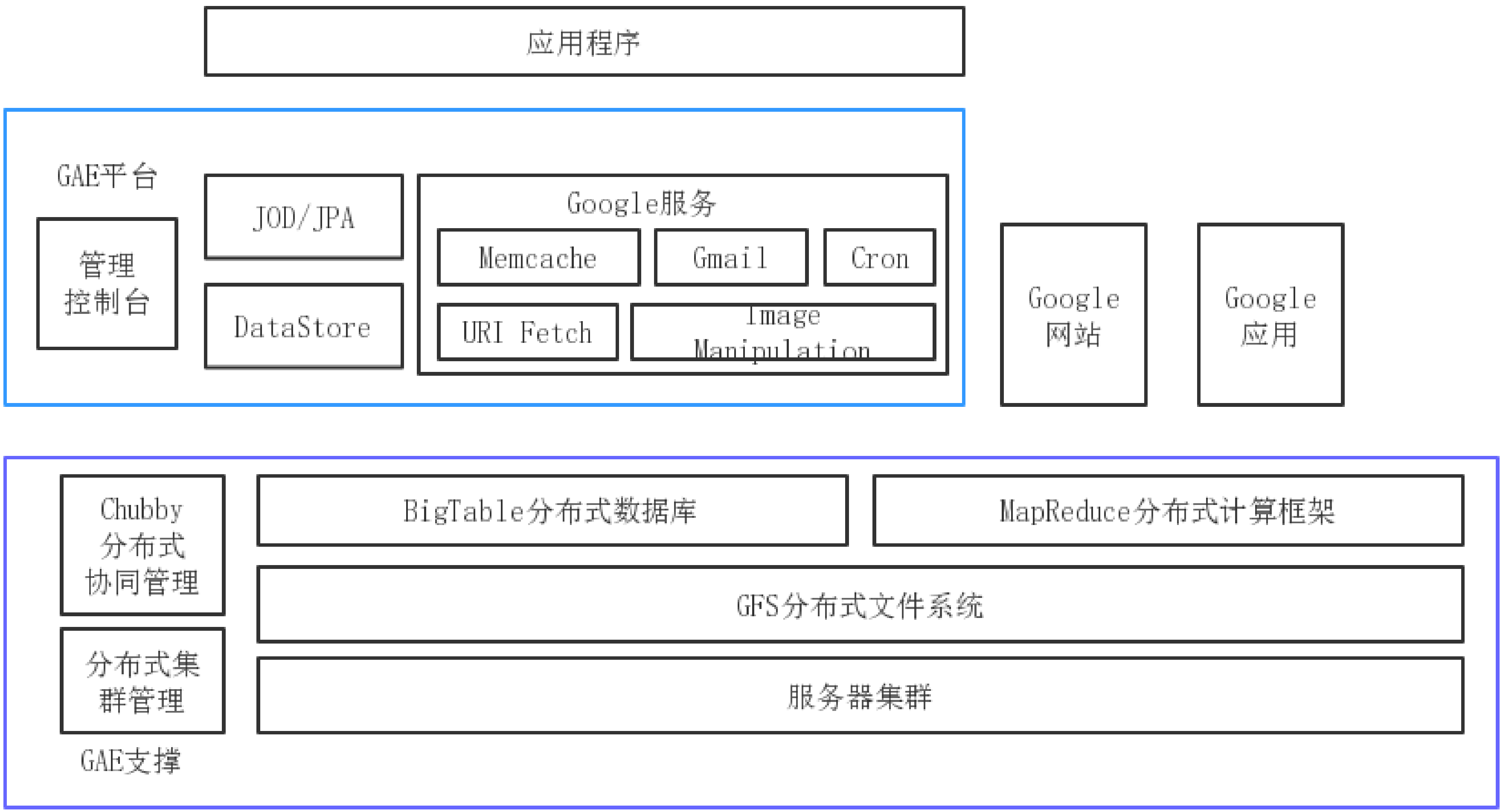
§4.3 MapReduce与Hadoop

云计算数据处理架构



浏览器
HTTP请求
HTTP响应
浏览器
HTTP请求
HTTP响应
负载均衡器
Web应用服务器集群
用户（开发者）的应用上传到不同的Web应用服务器上
基于Google Protocol Buffle技术的RPC机制
数据存储（DataStore）服务器集群
BigTable
Google GFS
用户端
App Engine 核心平台
Google服务（如账号、Gmail）
Google服务集
Google分布式基础架构

Google 的大数据处理平台

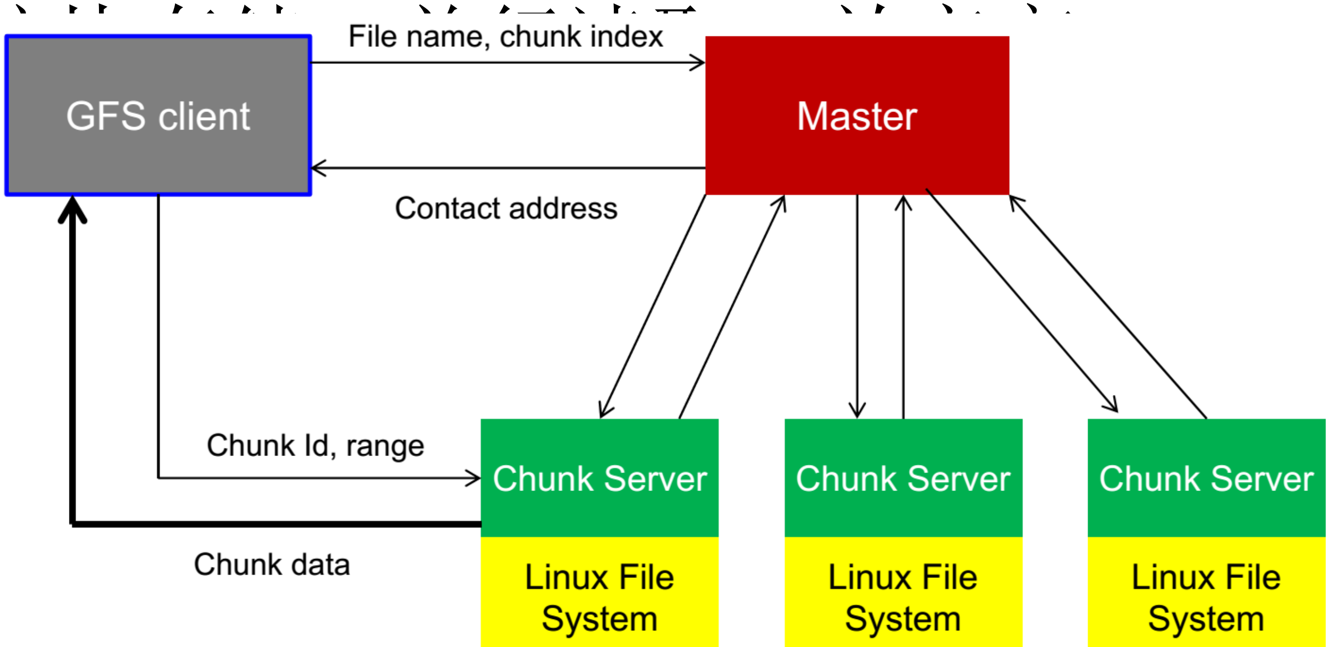


§4.1 分布式文件系统

- 基本特征
  - 透明性、并发访问、高可用性
- 基本需求
  - 数据冗余、异构性、一致性、高效性、安全性
- 基本架构
  - 按块存储、并行读取、效率高
  - 自动复制、多层次容错、原子操作保证一致性

基本架构

- 多层次容错
- 原子操作保证一致性
- 自动复制



HDFS

- 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
  - 大量的数据复制
  - 简单加入更多服务器
- HDFS是针对MapReduce设计
  - 数据尽可能根据其本地局部性进行访问与计算

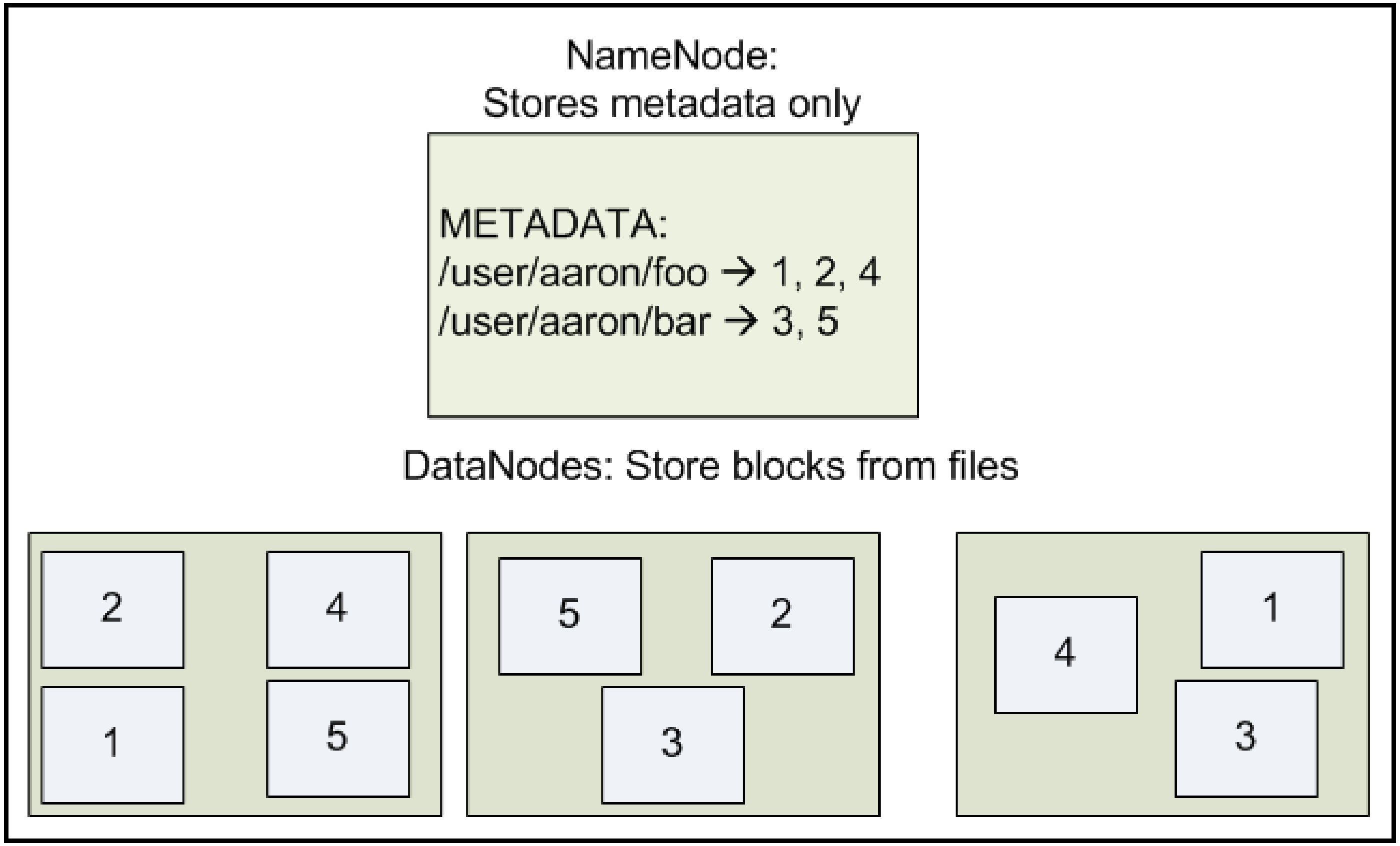
HDFS适应的场景

- HDFS是针对MapReduce设计
  - 数据尽可能根据其本地局部性进行访问与计算
- 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

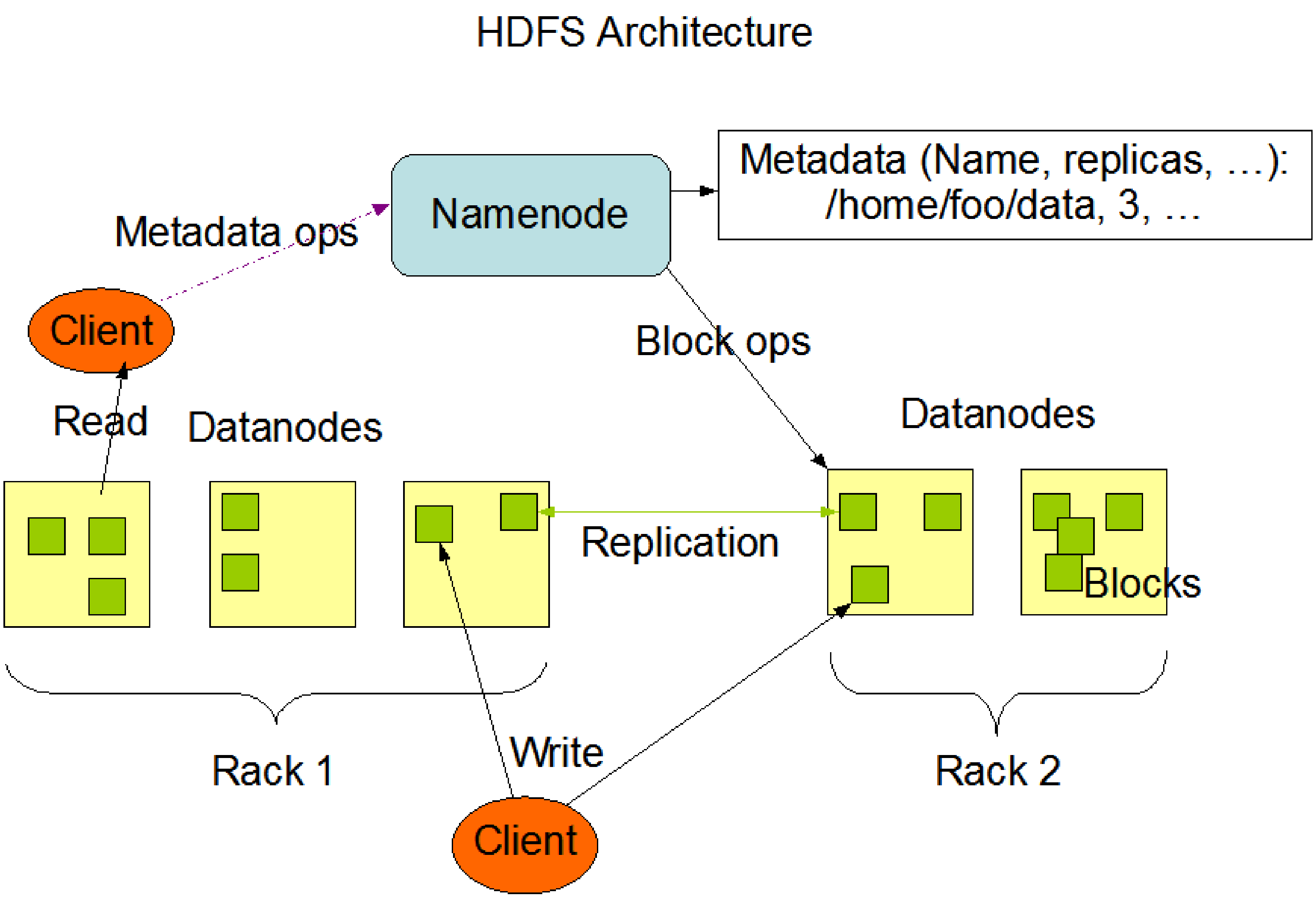
HDFS的设计

- 基于块的文件存储
- 块进行复制的形式放置，按照块的方式随机选择存储节点
- 副本的默认数目是3
- 默认的块的大小是64MB
  - 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数据分布设计


NameNode:
Stores metadata only
METADATA:
/user/aaron/foo → 1, 2, 4
/user/aaron/bar → 3, 5
DataNodes: Store blocks from files
2
4
1
5
5
2
3
4
1
3

HDFS体系结构



HDFS Architecture
Metadata (Name, replicas, …):
/home/foo/data, 3, …
Namenode
Metadata ops
Client
Block ops
Read
Datanodes
Datanodes
Replication
Blocks
Rack 1
Write
Rack 2
Client

HDFS设计要点

- 名字空间
- 副本选择
  - Rack Awareness
- 安全模式
  - 刚启动的时候，等待每一个DataMode报告情况
  - 退出安全模式的时候才进行副本复制操作
- NameNode有自己的FsImage和EditLog，前者有自己的文件系统状态，后者是还没有更新的记录

HDFS可靠性

- 磁盘数据错误
  - 心跳
  - 重新分布
- Cluster Rebalancing: not implemented
- Data Integrity: checksum
- Metadata Disk Failure: Multiple FsImage and EditLog, Checkpoint
- Snapshots: used for rollback, not implemented yet

HDFS程序接口

- 在MapReduce程序中使用HDFS
  - 通过fs.default.name的配置选项关联NameNode
- 在程序中使用HDFS接口
  - 命令行接口
  - Hadoop MapReduce Job的隐含的输入
  - Java程序直接操作
  - libhdfs从c/c++程序中操作

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
- 不是一个强的安全模型，不能保证操作的完全安全性
- 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

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

分布式拷贝

- bin/hadoop distcp hdfs://SomeNameNode:9000/foo/bar/ hdfs://OtherNameNode:2000/baz/quux/

- 目标也可以是s3://bucket-name/key

§4.2 NoSQL数据库

- Key features (advantages):
  - non-relational, don’t require schema
  - data are replicated to multiple nodes and can be partitioned:
    - down nodes easily replaced
    - no single point of failure
  - horizontal scalable
  - cheap, easy to implement
  - massive write performance
  - fast key-value access

NoSQL

- Disadvantages:
  - Don’t fully support relational features
    - no join, group by, order by operations (except within partitions)
    - no referential integrity constraints across partitions
  - No declarative query language (e.g., SQL) $\rightarrow$ more programming
  - Relaxed ACID (see CAP theorem) $\rightarrow$ fewer guarantees
  - No easy integration with other applications that support SQL

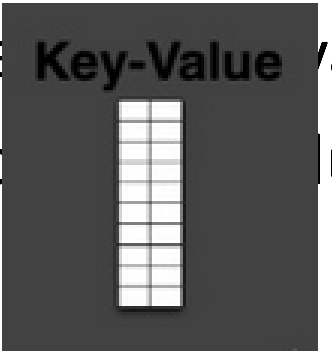
Who is using them?

NoSQL categories

1. Key-value
   - Example: DynamoDB, Voldermort, Scalaris
2. Document-based
   - Example: MongoDB, CouchDB
3. Column-based
   - Example: BigTable, Cassandra, Hbased
4. Graph-based
   - Example: Neo4J, InfoGrid
     - “No-schema” is a common characteristics of most NoSQL storage systems
     - Provide “flexible” data types

Key-value

- Focus on scaling to huge amounts of data
- Designed to handle massive load
- Data model: (global) collection of Key-value pairs
- *Dynamo ring partitioning* and *replication*
- Example: (DynamoDB)
  - *items* having one or more attribute alue)
  - An *attribute* can be single-valued c lued like set.
  - items are combined into a *table*



Basic API Access of Key-value

- get(key):
  - extract the value given a key
- put(key, value):
  - create or update the value given its key
- delete(key):
  - remove the key and its associated value
- execute(key, operation, parameters):
  - invoke an operation to the value (given its key) which is a special data structure (e.g. List, Set, Map .... etc.)

Key-value

Pros:

- very fast
- very scalable (horizontally distribution based on key)
- simple data model
- eventual consistency
- fault-tolerance

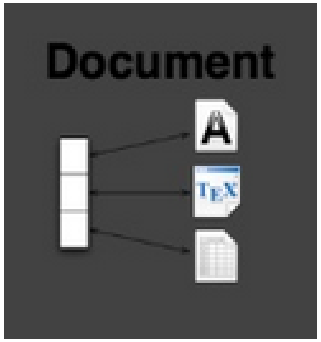
Cons:

- Can’t model more complex data structure such as objects

Key-value

Name	Producer	Data model	Querying
SimpleDB	Amazon	• set of couples (key, {attribute}) • attribute is a couple (name, value)	restricted SQL; select, delete, GetAttributes, and PutAttributes operations
Redis	Salvatore Sanfilippo	• set of couples (key, value), • value is simple typed value, list, ordered (according to ranking) or unordered set, hash value	primitive operations for each value type
Dynamo	Amazon	like SimpleDB	simple get operation and put in a context
Voldemort	LinkeId	like SimpleDB	similar to Dynamo

Document-based



- Can model more complex objects
- Inspired by Lotus Notes
- Data model: collection of documents
- Document:
  - JSON
    - **J**ava**S**cript **O**bject **N**otation, a data model
    - Key-value pairs, which supports objects, records, structs, lists, array, maps, dates, Boolean with **nesting**
  - XML
  - other semi-structured formats

Document-based

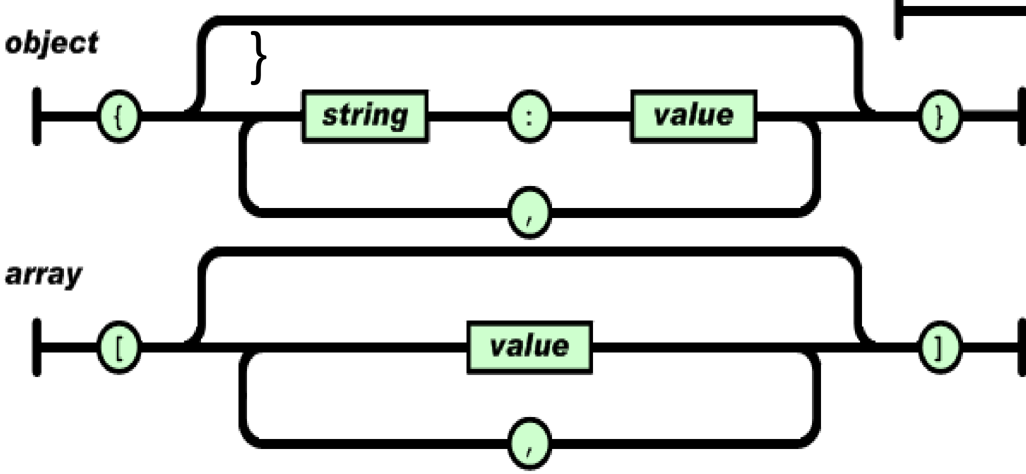
- Example: (MongoDB) document
  – {Name:"Jaroslav",

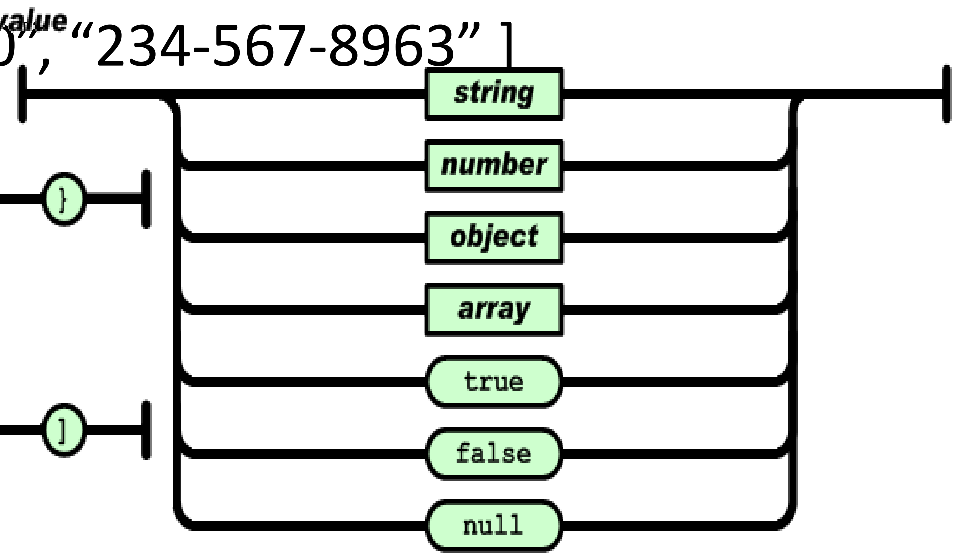
    Address:"Malostranske nám. 25, 118 00 Praha 1”,

    Grandchildren: {Claire: "7", Barbara: "6", "Magda: "3",
    "Kirsten: "1", "Otis: "3", Richard: "1“}

    Phones: [“123-456-7890”, “234-567-8963”]

    }



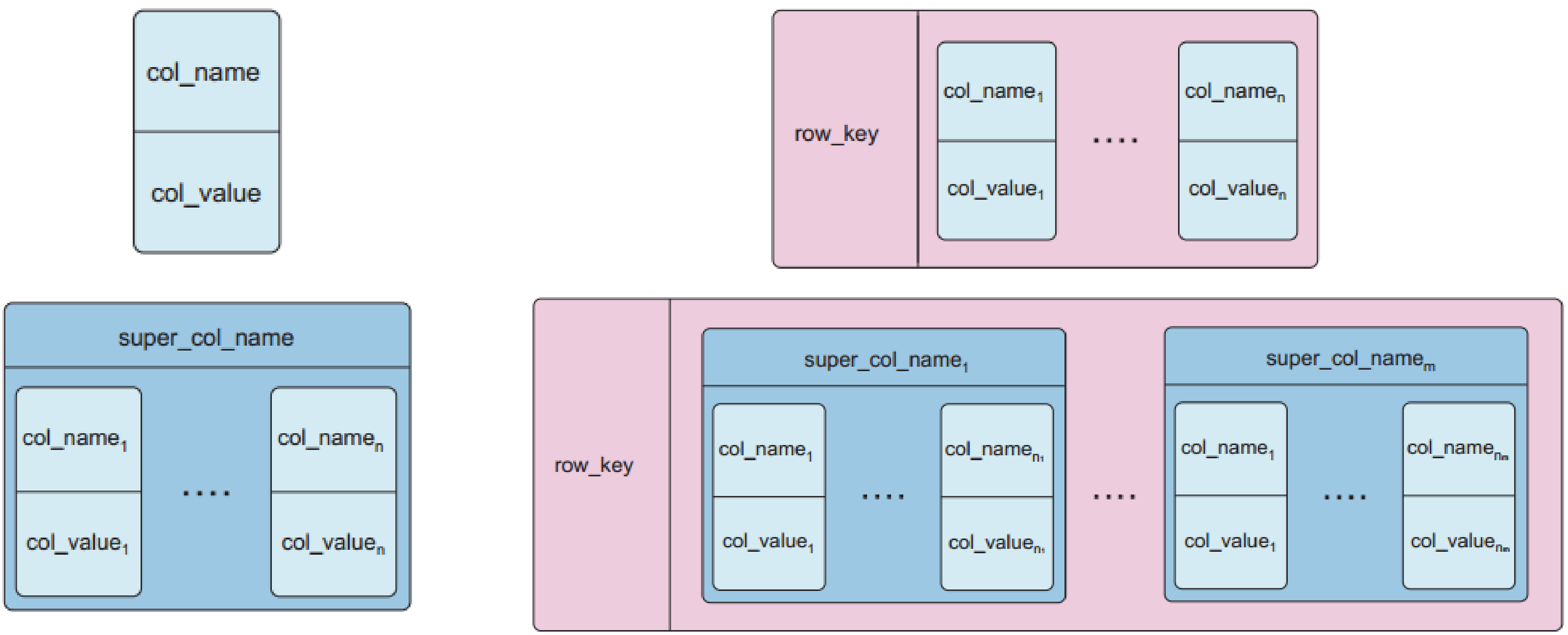


Document-based

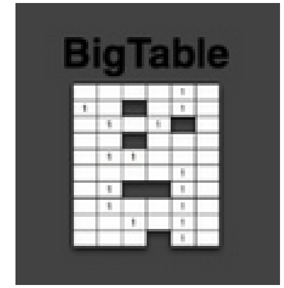
Name	Producer	Data model	Querying
MongoDB	10gen	• object-structured documents stored in collections • each object has a primary key called ObjectId	manipulations with objects in collections (find object or objects via simple selections and logical expressions, delete, update,)
Couchbase	Couchbase	• document as a list of named (structured) items (JSON document)	by key and key range, views via Javascript and MapReduce

Column-based

- Like column oriented relational databases (store data in column order) but with a twist
- Tables similarly to RDBMS, but handle semi-structured
- Data model:
  - Collection of Column Families
  - Column family = (key, value) where value = set of **related** columns (standard, super)
  - indexed by *row key*, *column key* and *timestamp*



Column-based



- One column family can have variable numbers of columns
- Cells within a column family are sorted “physically”
- Very sparse, most cells have null values

○ Query on multiple tables

  – **RDBMS:** must fetch data from several places on disk and glue together
  – **Column-based NOSQL:** only fetch column families of those columns that are required by a query

 (all columns in a column family are stored together on the disk → data locality)

Column-based

- Example:

(Cassandra column family--timestamps removed for simplicity)

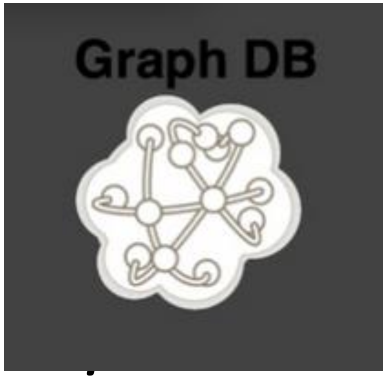
```
UserProfile = {
    Cassandra = { emailAddress:"casandra@apache.org" , age:"20"}
    TerryCho = { emailAddress:"terry.cho@apache.org" , gender:"male"}
    Cath = { emailAddress:"cath@apache.org" ,
             age:"20",gender:"female",address:"Seoul"}
}
```

Column-based

Name	Producer	Data model	Querying
BigTable	Google	set of couples (key, {value})	selection (by combination of row, column, and time stamp ranges)
HBase	Apache	groups of columns (a BigTable clone)	JRUBY IRB-based shell (similar to SQL)
Hypertable	Hypertable	like BigTable	HQL (Hypertext Query Language)
CASSANDRA	Apache (originally Facebook)	columns, groups of columns corresponding to a key (supercolumns)	simple selections on key, range queries, column or columns ranges
PNUTS	Yahoo	(hashed or ordered) tables, typed arrays, flexible schema	selection and projection from a single table (retrieve an arbitrary single record by primary key, range queries, complex predicates, ordering, top-k)

Graph-based

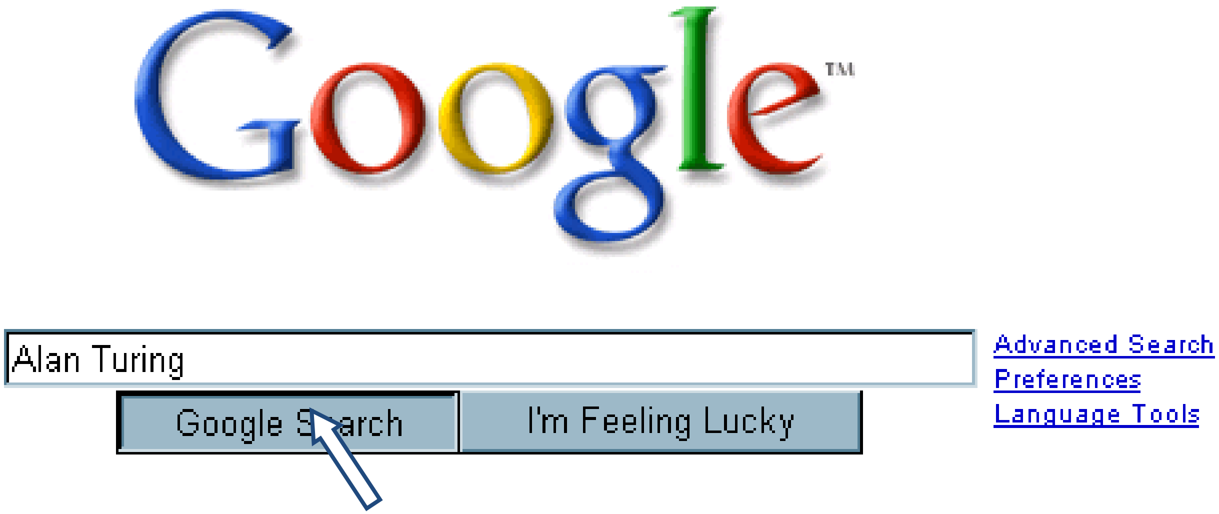
- Focus on modeling the structure of data (*interconnectivity*)
- Scales to the complexity of data
- Inspired by mathematical Graph Theory (G=(E,V))
- Data model:
  - (Property Graph) nodes and edges
    - Nodes may have properties (including ID)
    - Edges may have labels or roles
  - Key-value pairs on both
- Interfaces and query languages vary
- *Single-step* vs *path expressions* vs *full recursion*
- Example:
  - Neo4j, FlockDB, Pregel, InfoGrid …



§4.3 MapReduce与Hadoop

- MapReduce是一种编程模型和方法，用于大规模数据集（大于1TB）的并行运算。
- MapReduce是由Google提出的，初衷主要是为了解决其搜索引擎中大规模网页数据的并行化处理。
- 核心概念：Map（映射）和Reduce（归约）

Motivation

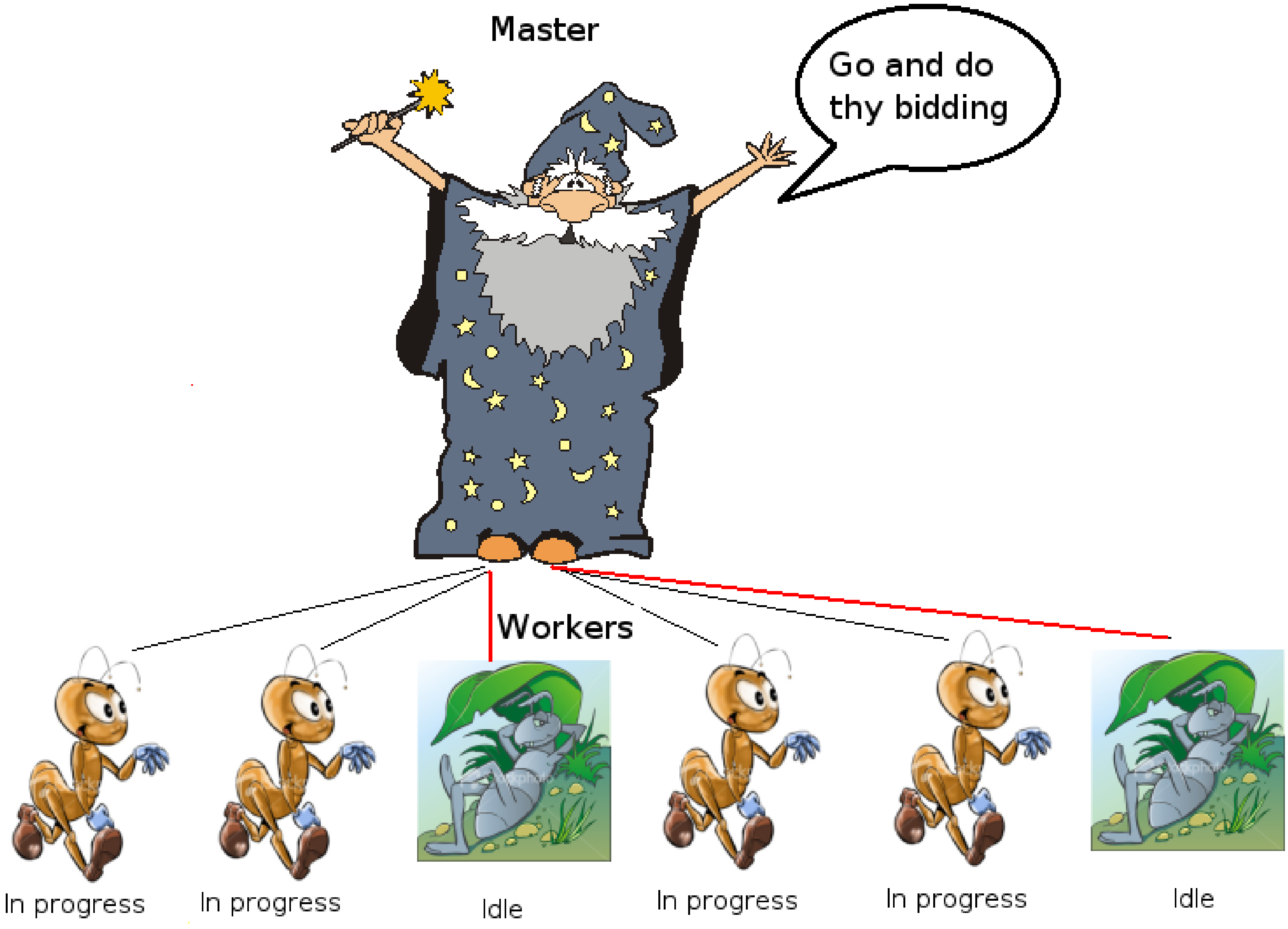


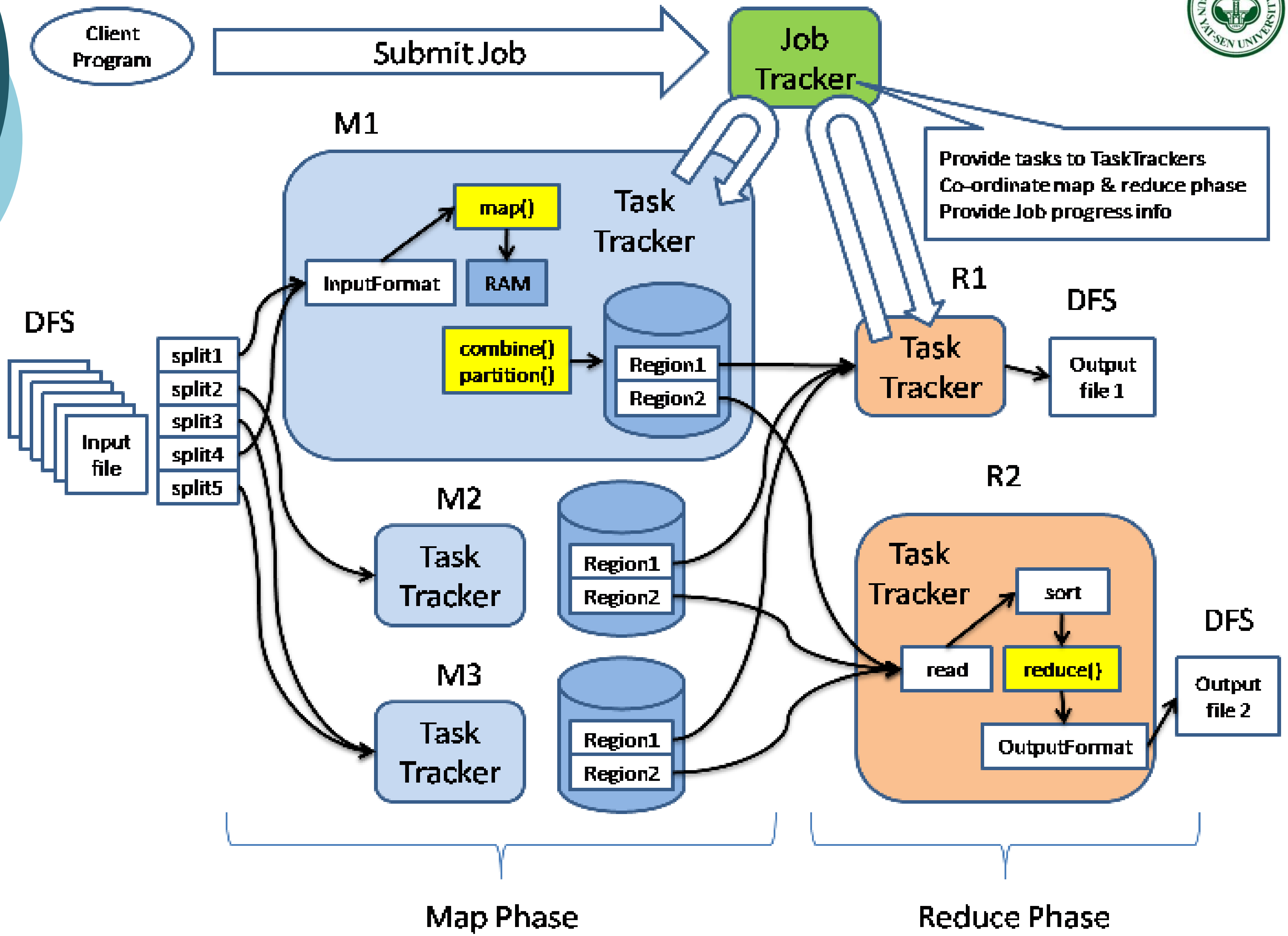
- 200+ processors
- 200+ terabyte database
- 10^{10} total clock cycles
- 0.1 second response time
- 5¢ average advertising revenue

Motivation: 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

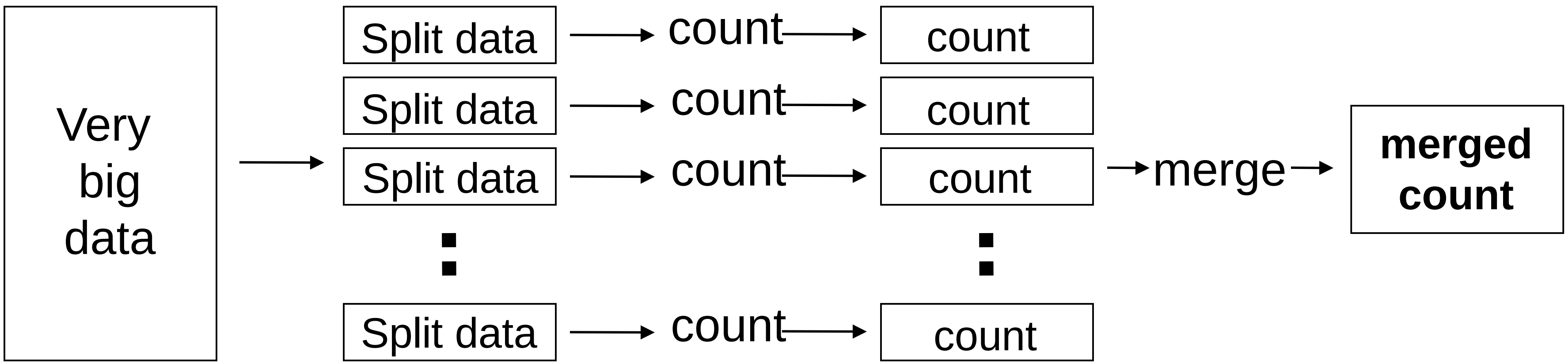
"Google Earth uses **70.5 TB**: 70 TB for the raw imagery and 500 GB for the index data."-2016


Master
Go and do thy bidding
Workers
In progress
In progress
Idle
In progress
In progress
Idle

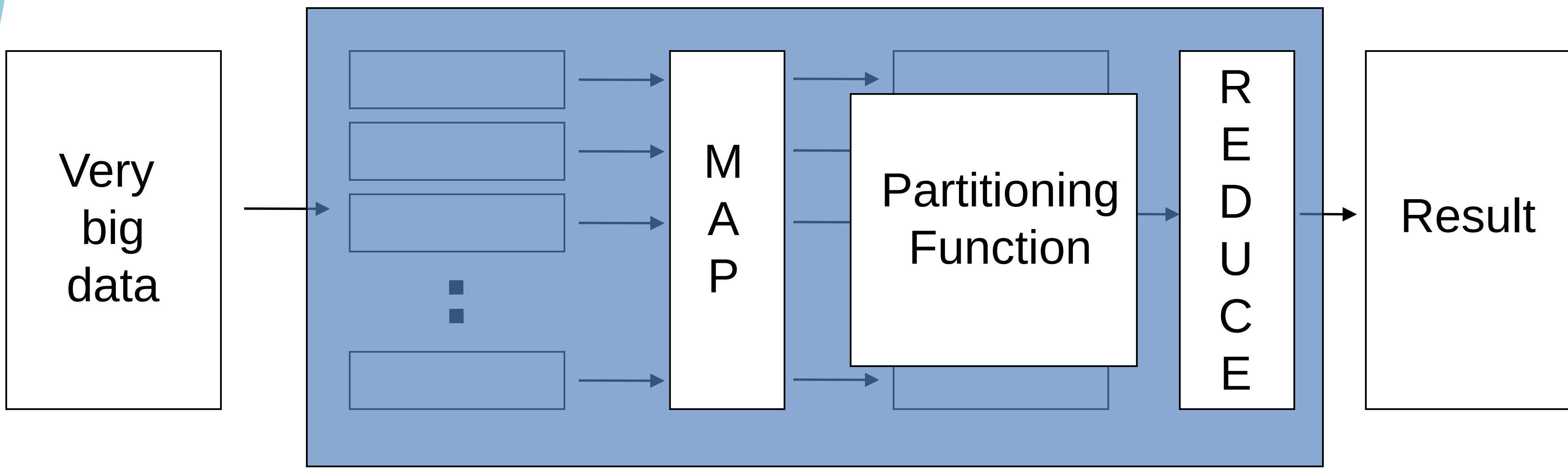


Client Program
Submit Job
Job Tracker
Provide tasks to TaskTrackers
Co-ordinate map & reduce phase
Provide Job progress info
M1
Task Tracker
map()
InputFormat
RAM
combine()
partition()
Region1
Region2
DFS
Input file
split1
split2
split3
split4
split5
M2
Task Tracker
Region1
Region2
M3
Task Tracker
Region1
Region2
R1
Task Tracker
DFS
Output file 1
R2
Task Tracker
sort
read
reduce()
OutputFormat
DFS
Output file 2
Map Phase
Reduce Phase

Distributed Word Count



Map Reduce



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

InputSplits

- InputSplit定义了输入到单个Map任务的输入数据
- 一个MapReduce程序被统称为一个Job，可能有上百个任务构成
- InputSplit将文件分为64MB的大小
  - hadoop-site.xml中的mapred.min.split.size参数控制这个大小
- mapred.tasktracker.map.taks.maximum用来控制某一个节点上所有map任务的最大数目

RecordReader

- InputSplit定义了一项工作的大小，但是没有定义如何读取数据
- RecordReader实际上定义了如何从数据上转化为一个(key,value)对，从而输出到Mapper类中
- TextInputFormat提供了LineRecordReader

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.

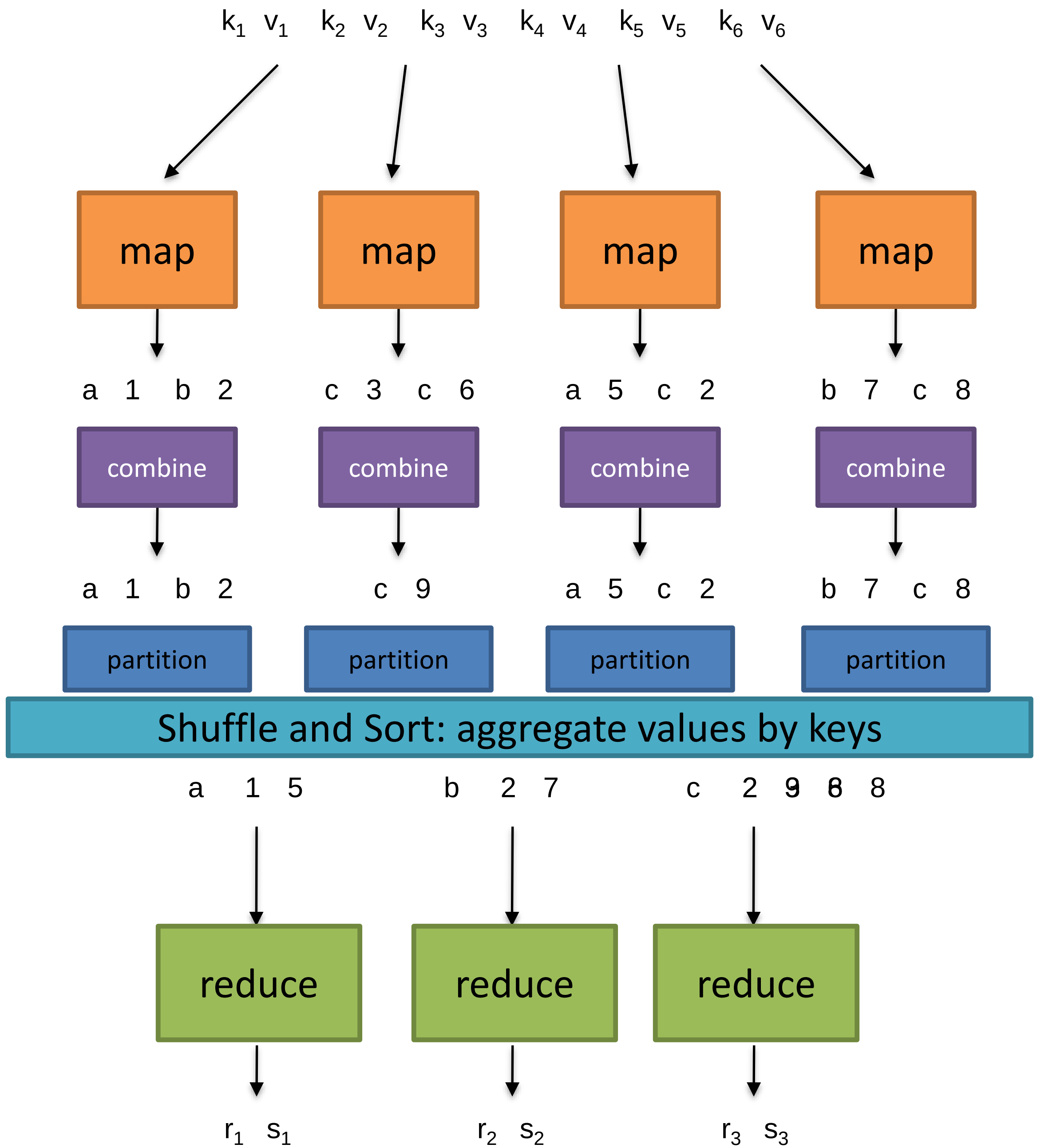
Mapper

- 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)



k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

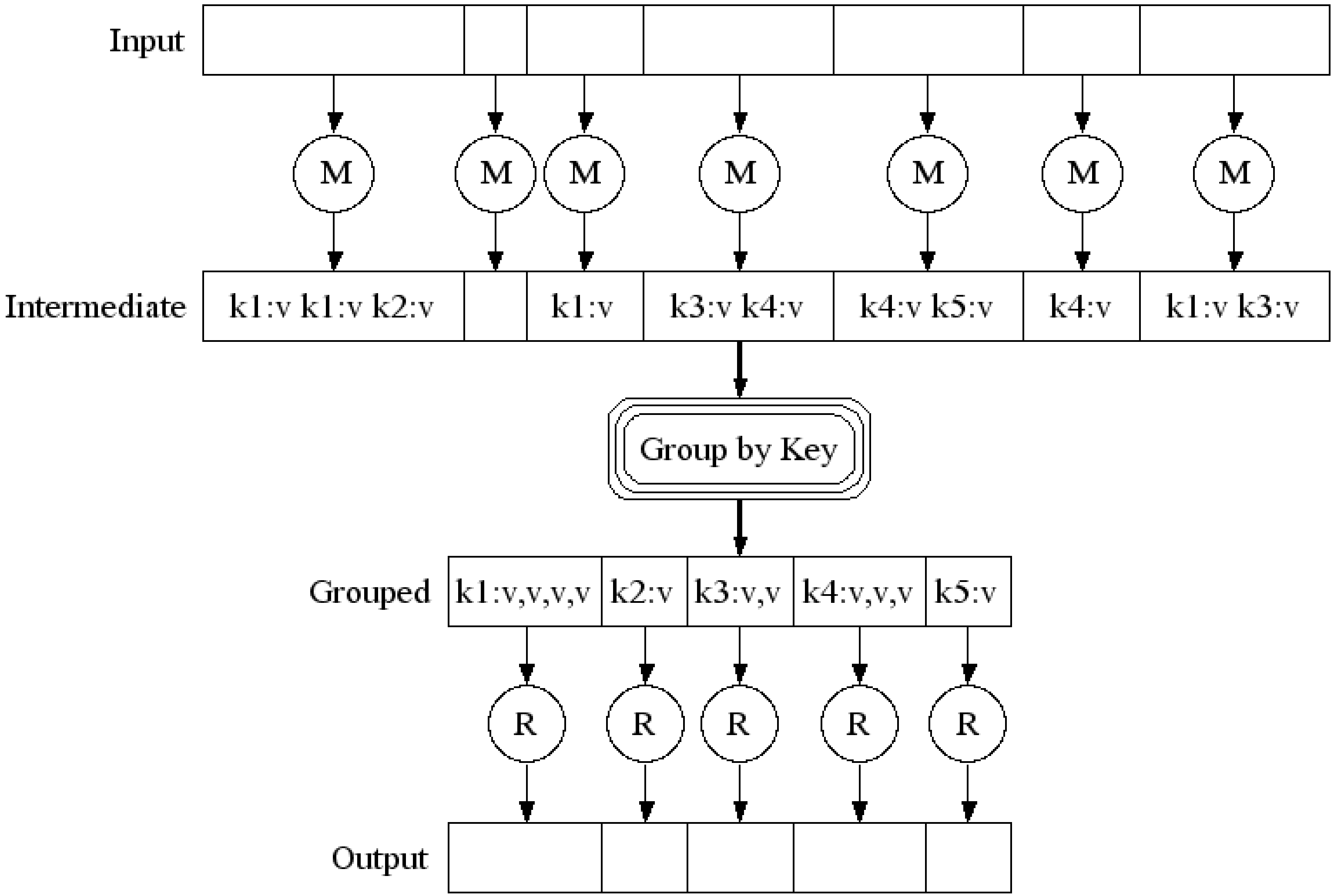
Partition&Shuffle

- 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函数接收到得Key,value对会被Hadoop自动排序（即Map生成的结果传送到某一个节点的时候，会被自动排序）
- Default : `hash(key) mod R`
- Guarantee:
  - Relatively well-balanced partitions
  - Ordering guarantee within partition

Partitioning Function



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

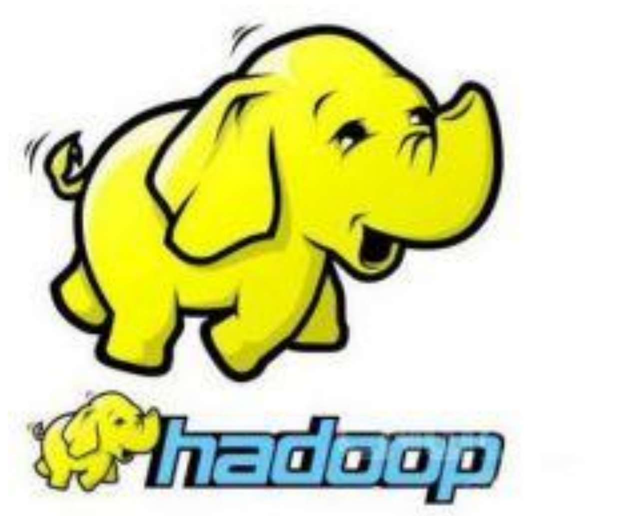
- Worker 1:
  - (the 1)
- Worker 2:
  - (is 1), (is 1), (is 1)
- Worker 3:
  - (weather 1), (weather 1)
- Worker 4:
  - (today 1)
- Worker 5:
  - (good 1), (good 1), (good 1), (good 1)

Reduce Output

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

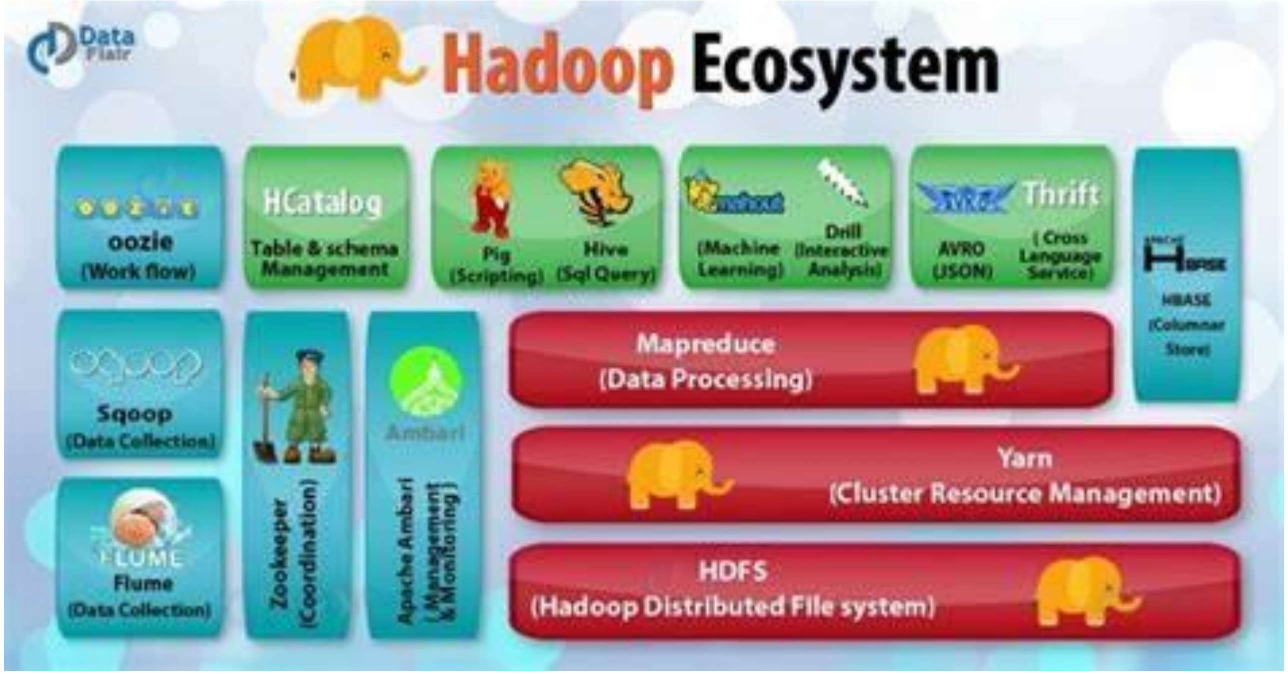
Hadoop

- 起源于开源网络搜索引擎Apache Nutch项目
- 谷歌的大数据处理技术的开源实现
  - GFS->HDFS
  - BigTable->HBase
  - MapReduce



Hadoop核心组件

- HDFS：分布式文件系统
- MapReduce：大规模数据并行计算框架
- Yarn：集群资源管理与任务调度



Yarn

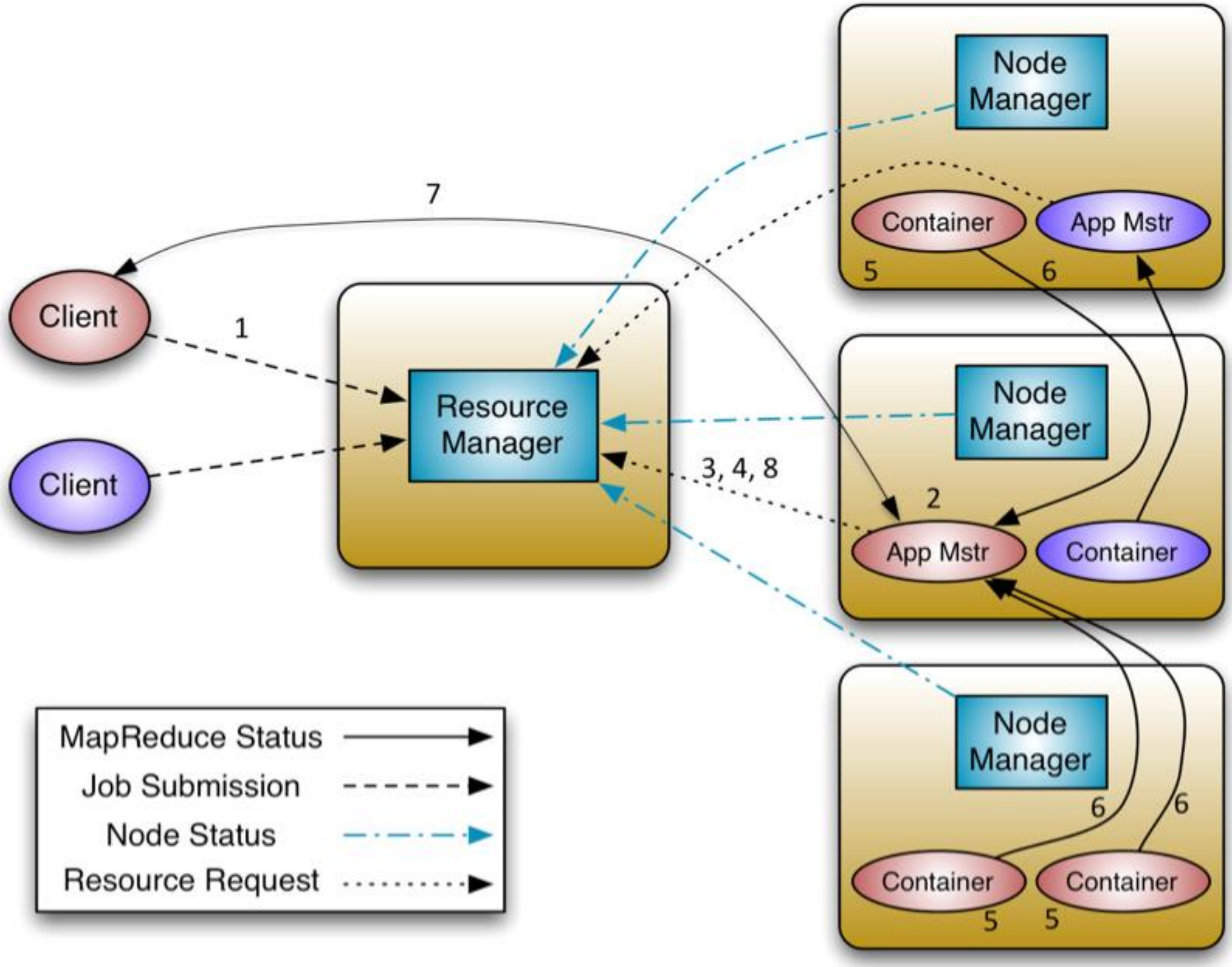
- Yarn(Yet Another Resource Negotiator)
- 负责资源分配和任务调度管理，Hadoop2引进
  - ResourceManager：负责系统中所有应用的资源管理和分配。
  - NodeManager：对应每个计算节点，负责监控节点的资源使用情况（CPU,内存,磁盘,网络）,并将报告给RM。
  - ApplicationMaster：对应每个应用程序，负责监控、管理这个Application的所有任务的执行，同时负责向RM申请资源。

MapReduce 1	YARN
Jobtracker	Resource manager, application master, timeline server
Tasktracker	Node manager
Slot	Container

Yarn

- Container
  - 基本的资源单位（CPU，内存），是Yarn资源的抽象，封装了某个节点的一定量的资源。
  - Container可以加载任意程序，运行ApplicationMaster或执行应用中的某个任务。
  - 一个节点可以包含多个Container
  - ApplicationMaster根据需要，动态申请和释放Container。

Yarn资源分配流程


Node
Manager
Container
App Mstr
5
6
7
Client
1
Resource
Manager
Client
Node
Manager
3, 4, 8
2
App Mstr
Container
Node
Manager
6
6
Container
Container
5
5
MapReduce Status
Job Submission
Node Status
Resource Request

Yarn执行过程

1. 客户端程序向RM 提交应用并请求一个 AM 实例。
2. RM 找到一个可以运行一个 Container 的 NM，并在这个 Container 中启动AM实例。
3. AM 向 RM 进行注册，注册之后客户端就可以查询 RM 获得自己 AM的详细信息，以后就可以和自己的 AM直接交互。
4. AM向RM发送resource-request请求。
5. 当Container被成功分配后，AM告诉NM启动Contrainer。
6. Task在启动的Container中运行，并把运行的状态信息发送给AM。
7. 提交的客户端与AM交流获取应用运行状态。
8. 应用程序执行完毕后，AM向RM取消注册然后关闭，归还Container。

小结

- 并行数据处理的基本架构
- 分布式数据存储技术
  - 文件系统
  - 数据库
    - SQL, NoSQL
- 分布式数据处理编程框架
  - MapReduce