第5a章 云计算与大数据处理

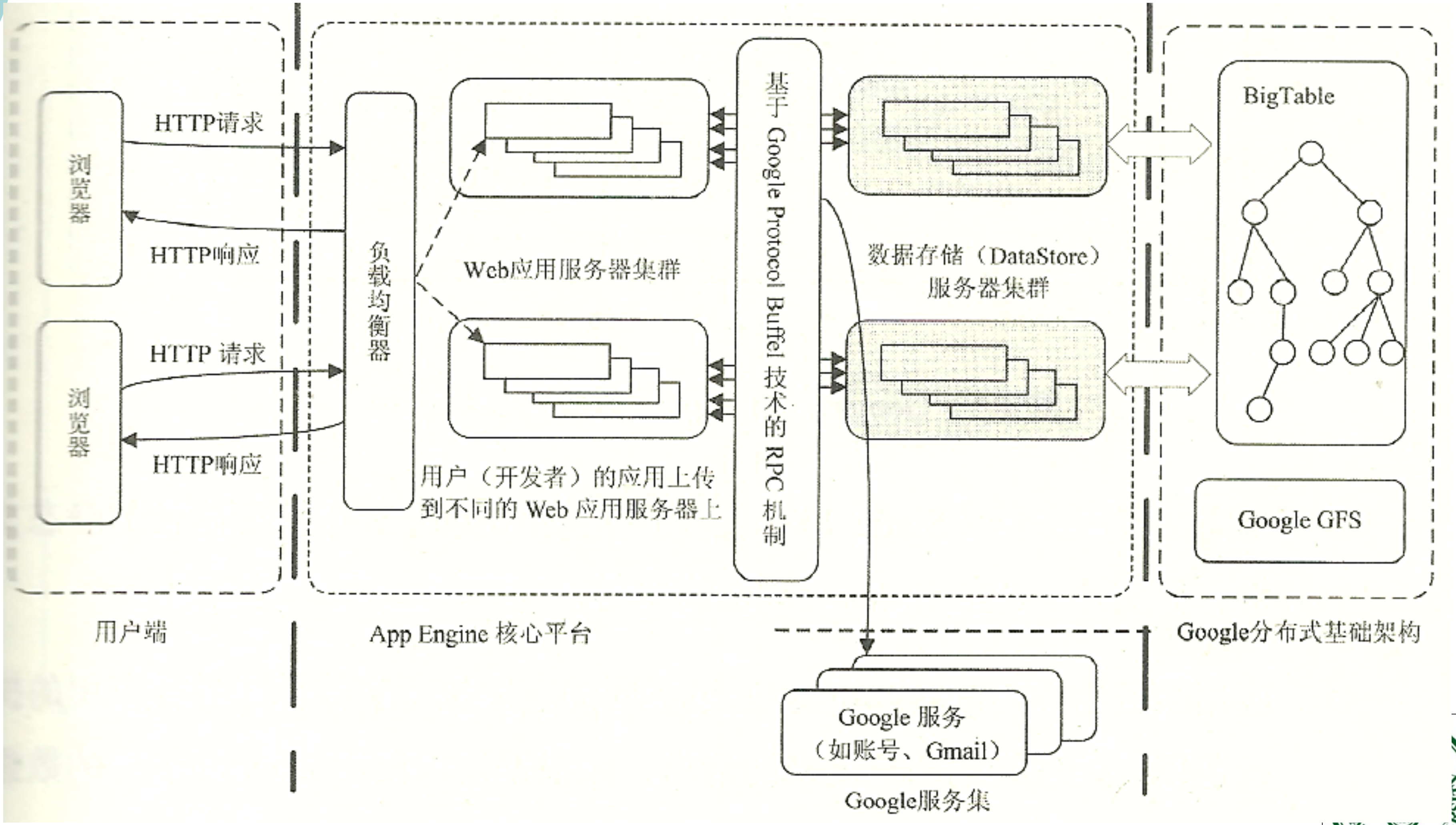
§5a.1 云计算数据处理架构

§5a.2 分布式文件系统

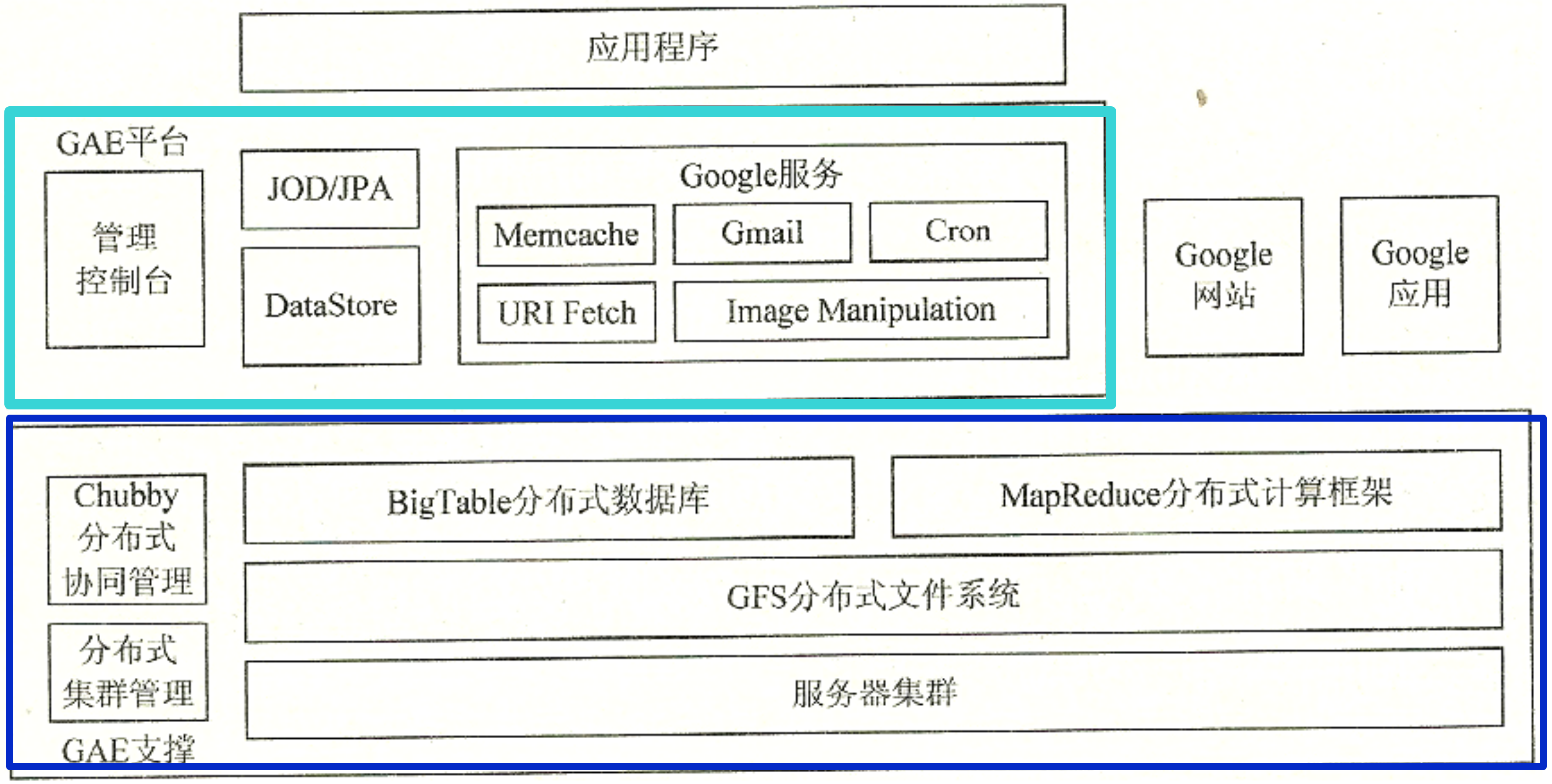
§5a.3 分布式数据库

§5a.4 MapReduce技术

§5a.1 云计算数据处理架构

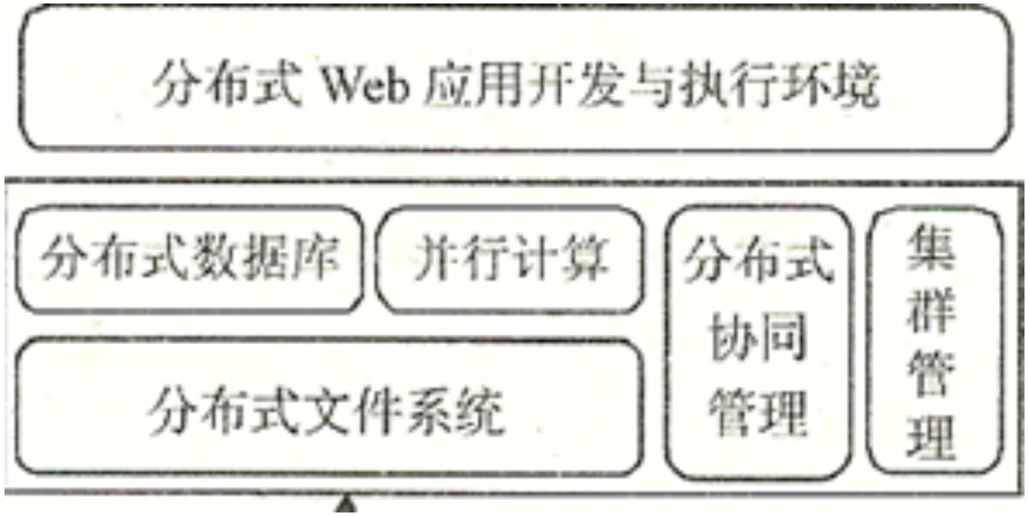


Google 的大数据处理平台



云数据处理关键技术

- 分布式数据管理
  - 分布式文件系统
  - 分布式数据库
- 云并行计算技术
- 分布式协同管理
- 集群管理



分布式计算技术

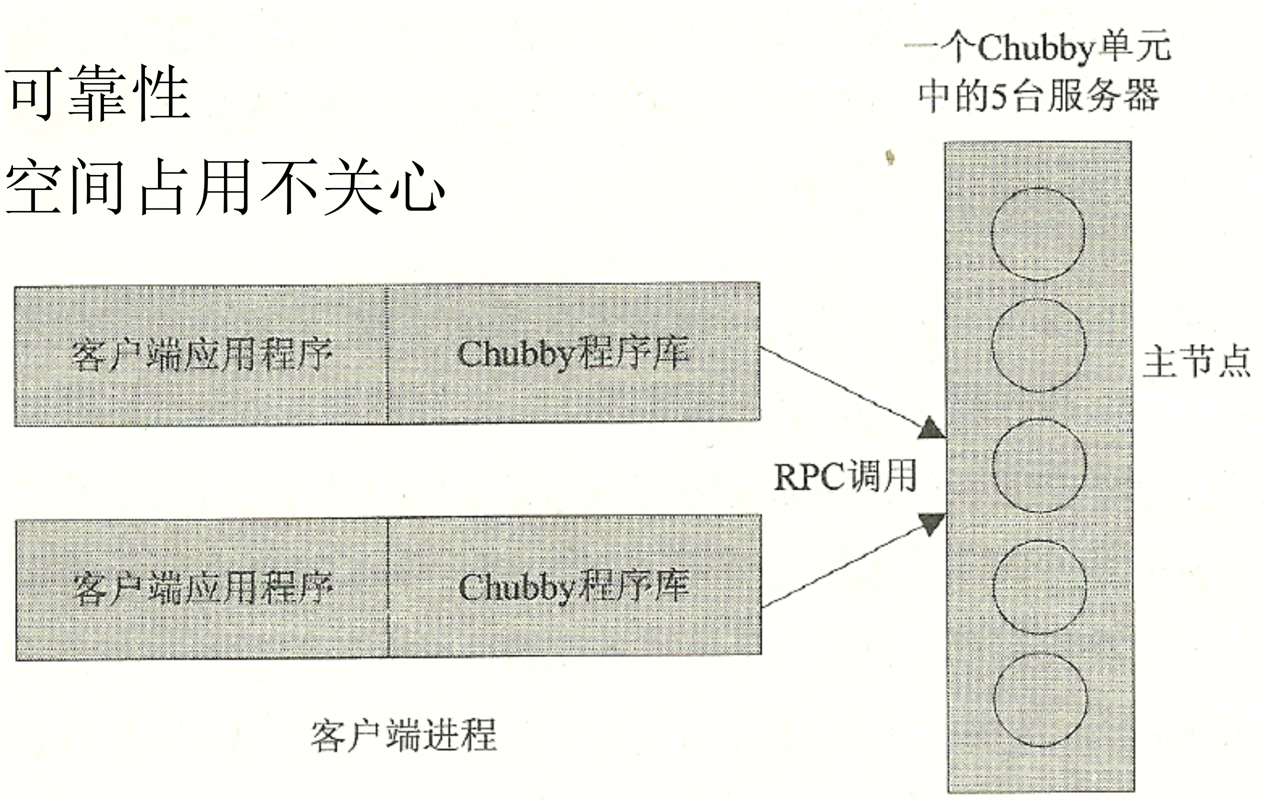
- 云平台的分布式计算特性
  - CAP理论：
    - Consistency、Availability、Partition-tolerance 不可兼得
  - 传统分布式计算注重”C”
  - 云计算平台注重”A”和”P”
- MapReduce计算模型
  - Google发明的分布式计算模型
  - 适于海量数据处理
    - 传统分布式计算适于“计算”密集型任务
  - 基于数据分割

分布式协同管理

- 共享资源的并行操作→数据不一致
- 通过同步机制控制并发操作
- 常用的并发控制方法
  - 基于锁的并发控制
    - 两阶段锁协议
    - 多副本的锁机制：读-写全法、多数法、主副本法、单一管理法
  - 基于时间戳的并发控制
    - 基于全局唯一的时间戳
  - 乐观并发控制
  - 基于版本的并发控制

Google Chubby 并发控制

- 分布式所服务
- 通过文件操作实现锁操作
  - 文件代表锁
- 主要目标
  - 高可用性、高可靠性
  - 性能、吞吐和空间占用不关心



集群和平台管理

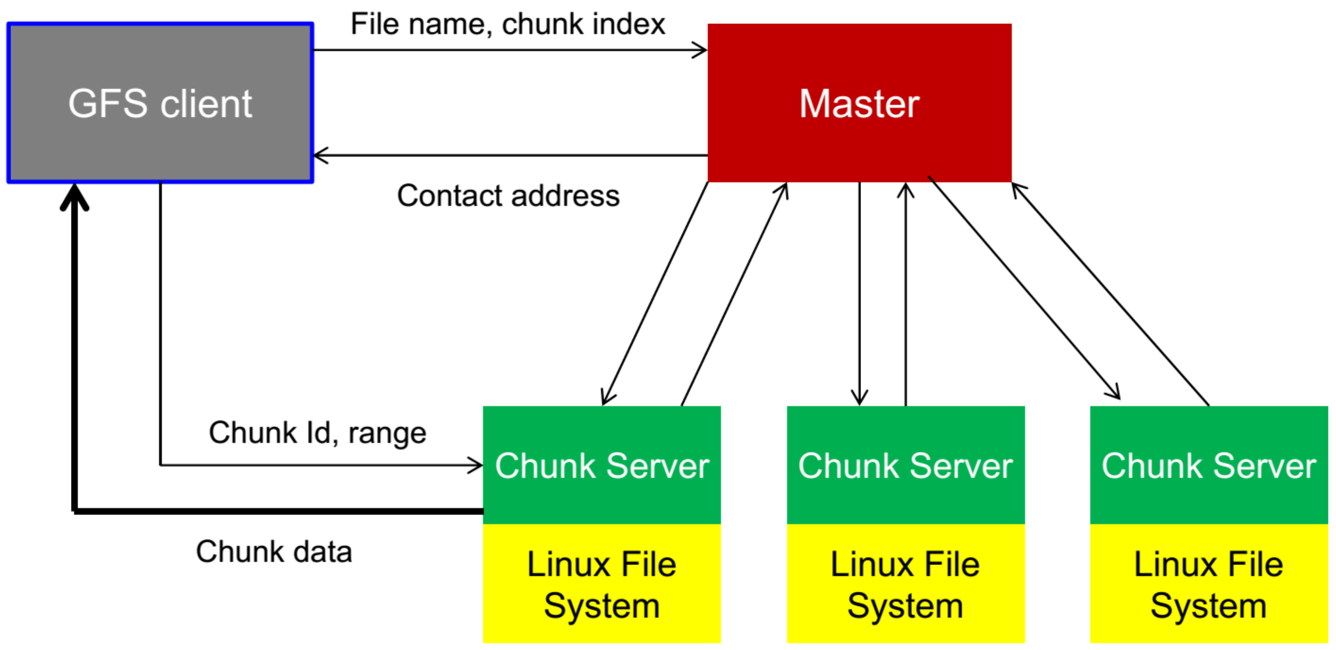
- 集群的自动化部署
  - 安装配置OS、DFS、分布式计算程序、作业管理软件、系统管理软件等。
- 集群作业调度
  - 基于资源管理器调度作业的运行节点和时间
  - Google Work Queue： 调度管理MapReduce任务
- 远程控制
  - 开关机控制
  - 远程控制台
- 应用管理与度量计费

§5a.2 分布式文件系统

- 分布式数据处理的基础
- 分布式数据库的基础
- 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

- 多层次容错
- 原子操作保证一致性
- 自动复制
- 按块存储，并行读取，效率高



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

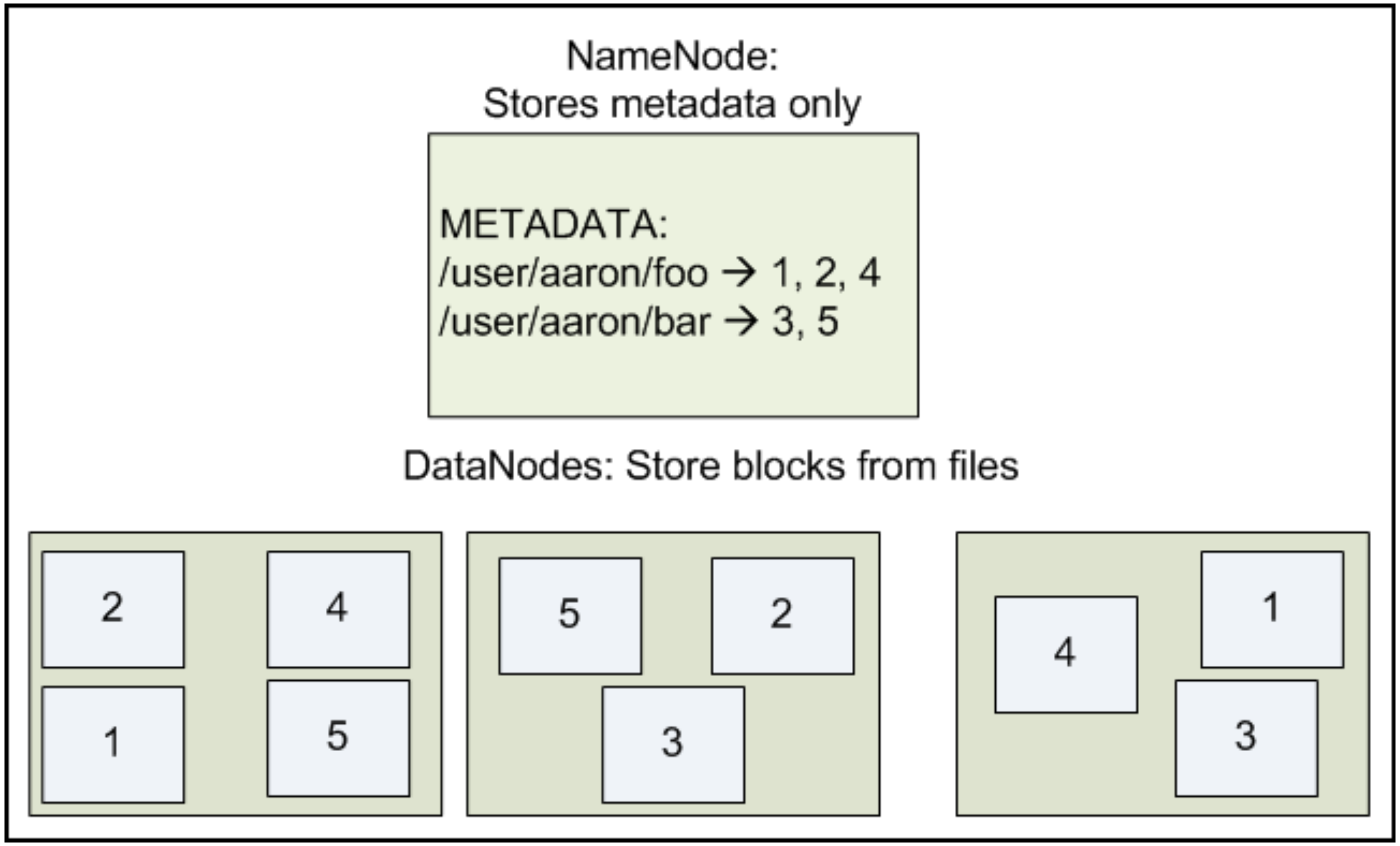
HDFS适应的场景

- 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

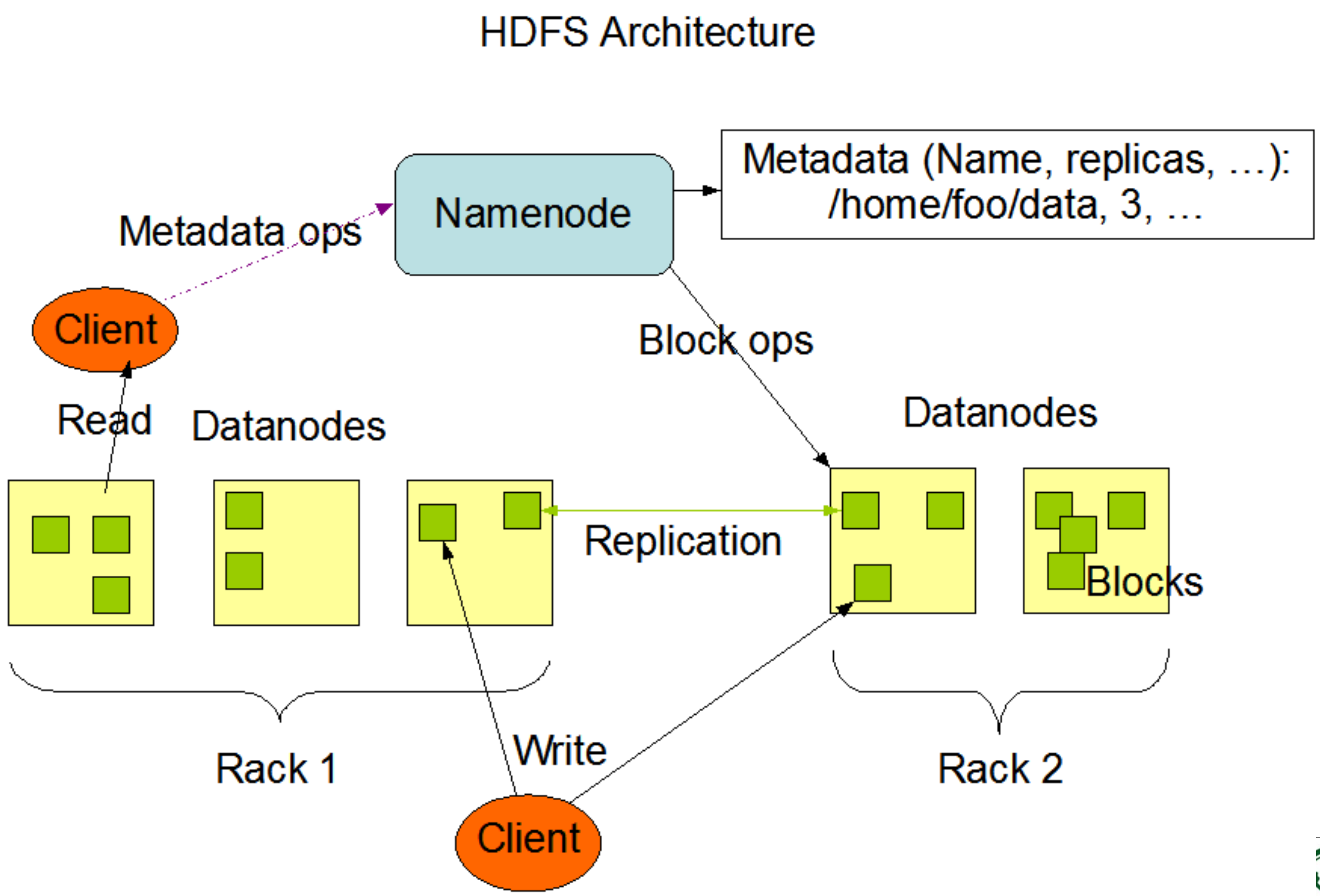
HDFS的设计

- 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

§5a.3 分布式数据库

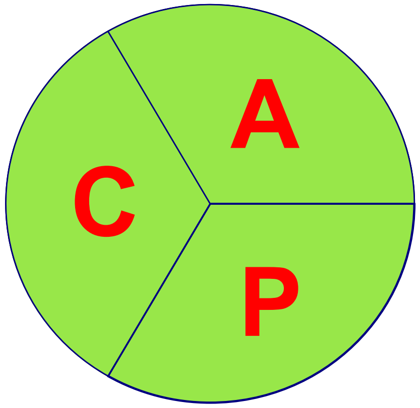
- Huge demands on data storage volume and transaction rate
- Scalability of app servers is easy, but database?
  - Approach 1: memcache or other caching
  - Limited in scalability
  - Approach 2: use existing parallel databases
  - Expensive, and most parallel databases were designed for decision support not OLTP
  - Approach 3: build parallel stores with databases underneath

Scaling RDBMS - Partitioning

- "Sharding"
  - Divide data amongst many cheap databases (MySQL/ PostgreSQL)
  - Manage parallel access in the application
  - Scales well for both reads and writes
  - Not transparent, application needs to be partition-aware

CAP Theorem

- Three properties in sharing data)
  - **Consistency:**
    - all copies have the same value
  - **Availability:**
    - reads and writes always succeed
  - **Partition-tolerance:**
    - consistency and/or availability hold even when network failures prevent communicating among some machines



CAP Theorem

- **Brewer's CAP Theorem:**
  - *For any system sharing data, it is "**impossible**" to guarantee **simultaneously** all of the **three properties***
  - You can have at most two of these three
  - Traditional DBMS prefers **C** over **A** and **P**
- Cloud systems, partition is unavoidable:
  - That leaves either **C** or **A** to choose from ()
  - In almost all cases, you would choose **A** over **C** (except in specific applications such as order processing)

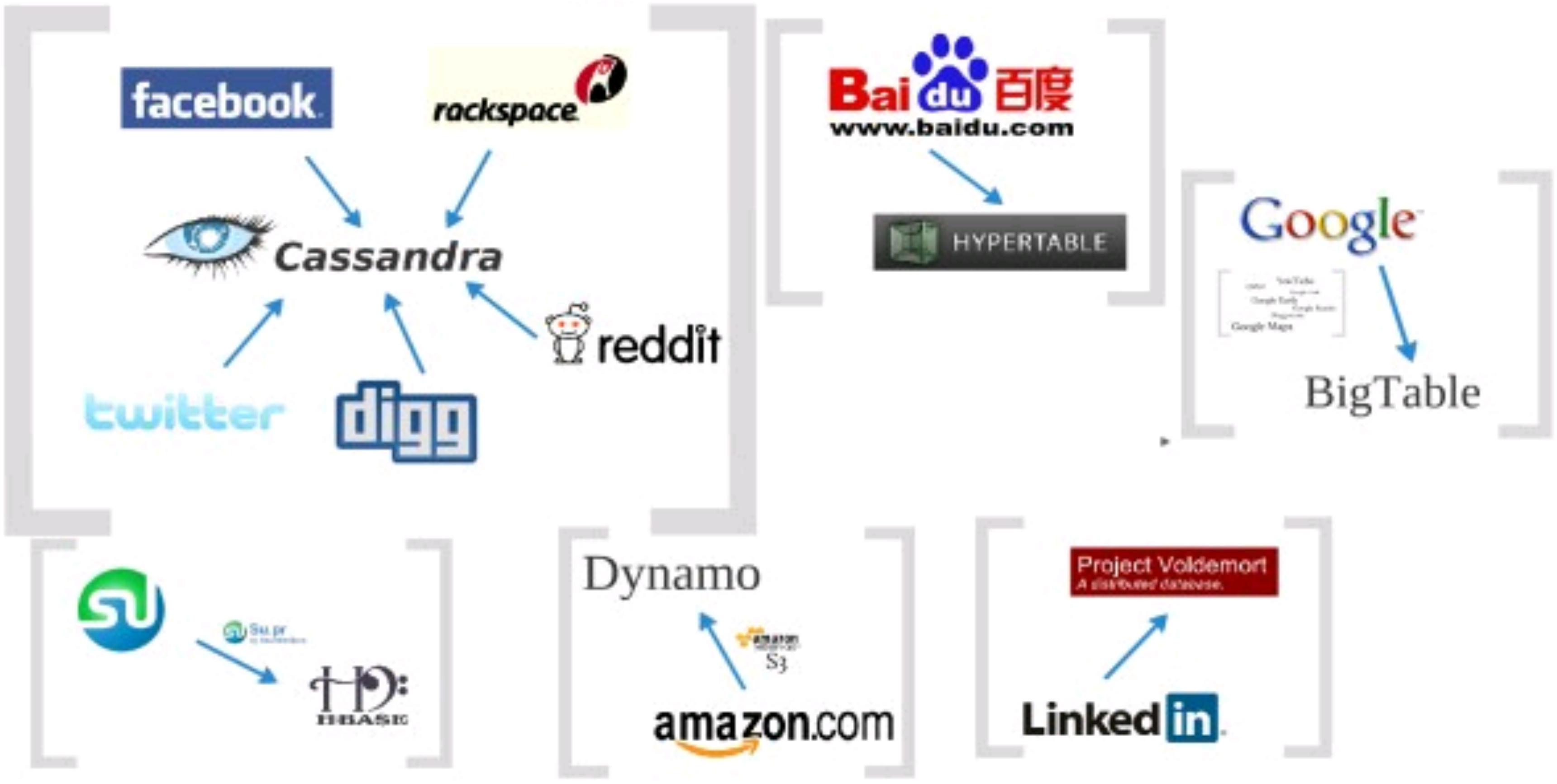
What is NoSQL?

- 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

What is 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) ? more programming
  - Relaxed ACID (see CAP theorem) ? fewer guarantees
  - No easy integration with other applications that support SQL

Who is using them?



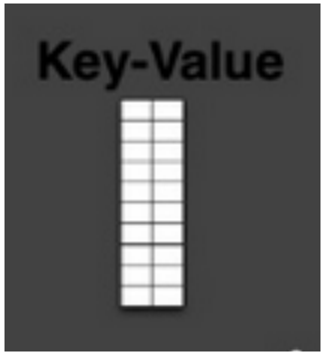
NoSQL categories

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 attributes (name, value)
  - An *attribute* can be single-valued or multi-valued 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
A
TEX

Document-based

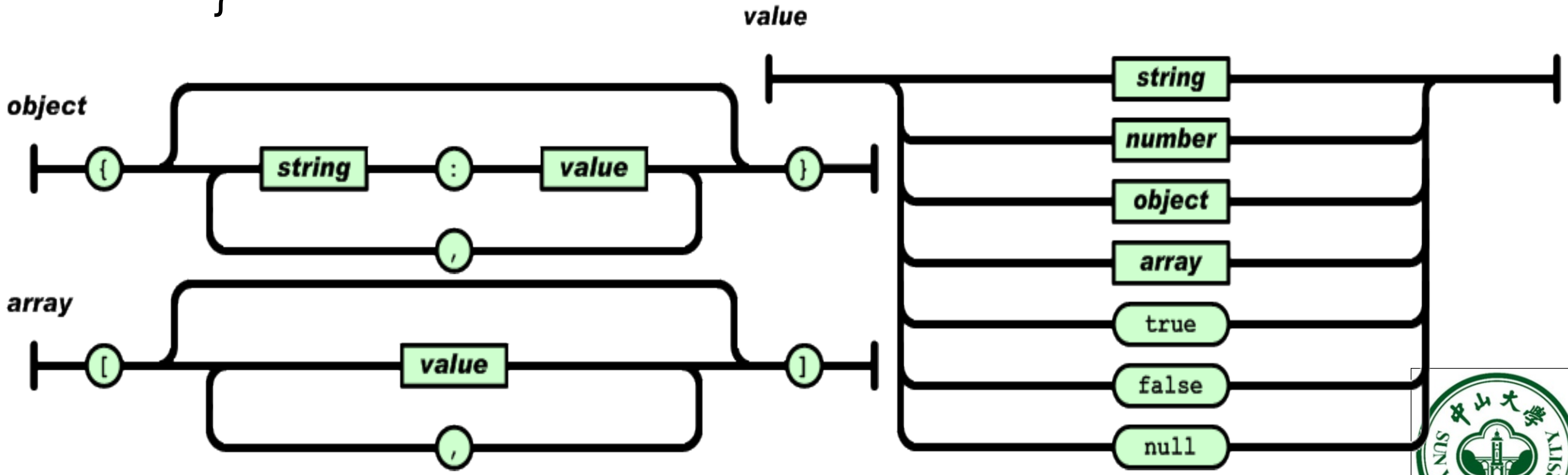
- Example: (MongoDB) document
  - {Name:"Jaroslav",

    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"]

    }

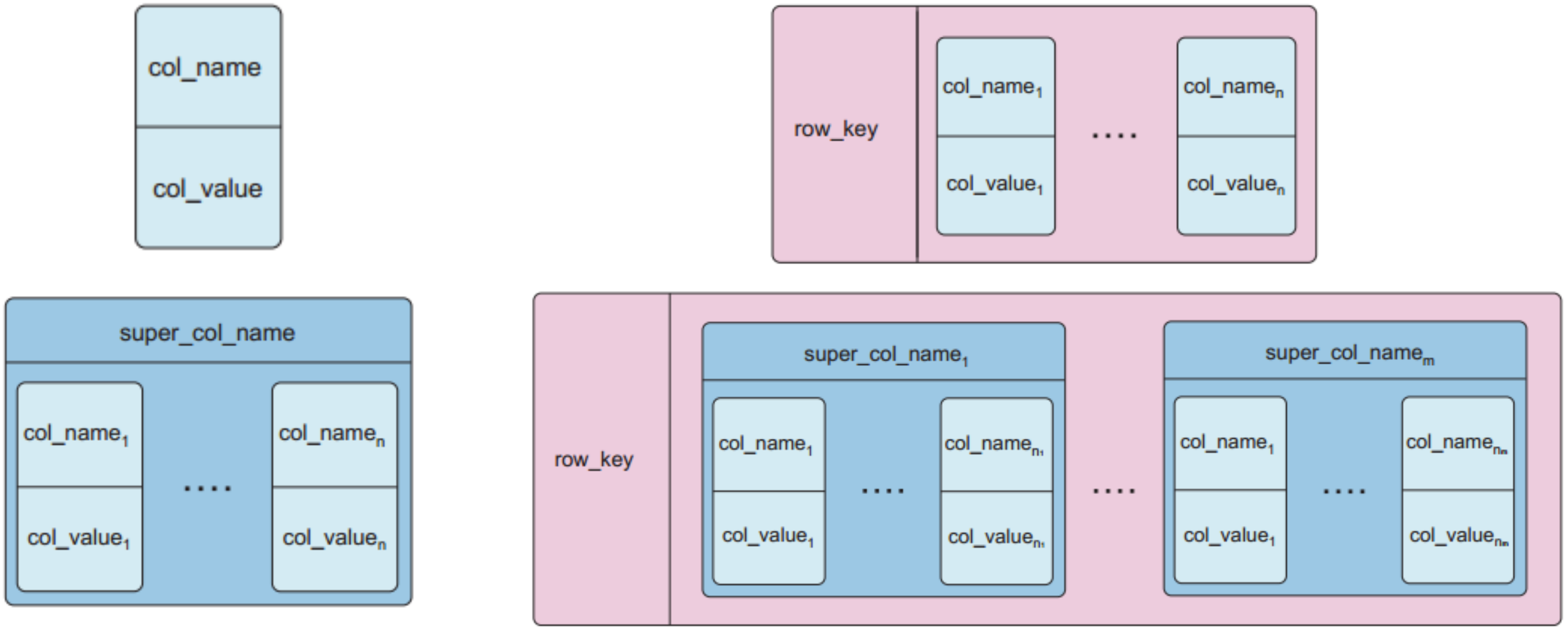


Document-based

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"}

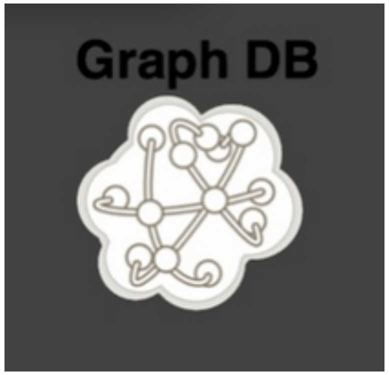
}

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

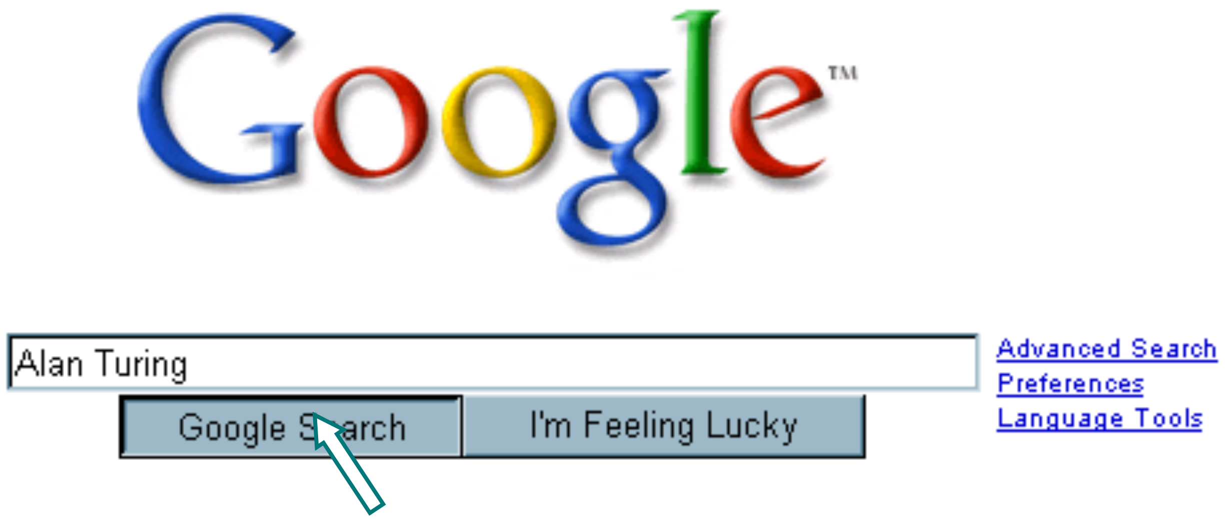
- Focus on modeling the structure of data (*interconnectivity*)
- Scales to the complexity of data
- Inspired by mathematical Graph Theory (G=(E,V))
- Data model:
  - (Property Graph) nodes and edges
    - Nodes may have properties (including ID)
    - Edges may have labels or roles
  - Key-value pairs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



§5a.4 MapReduce技术

- 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

Motivation

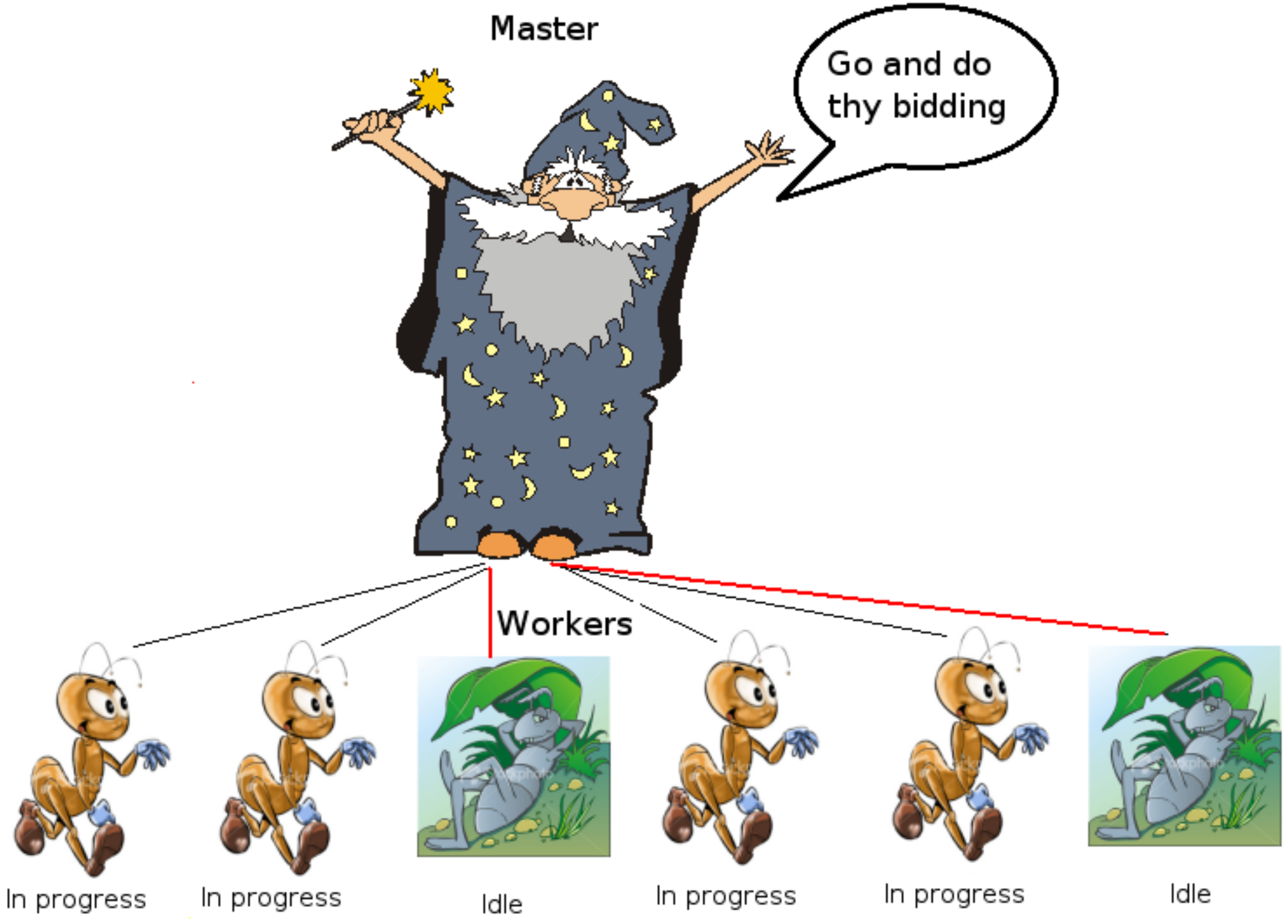


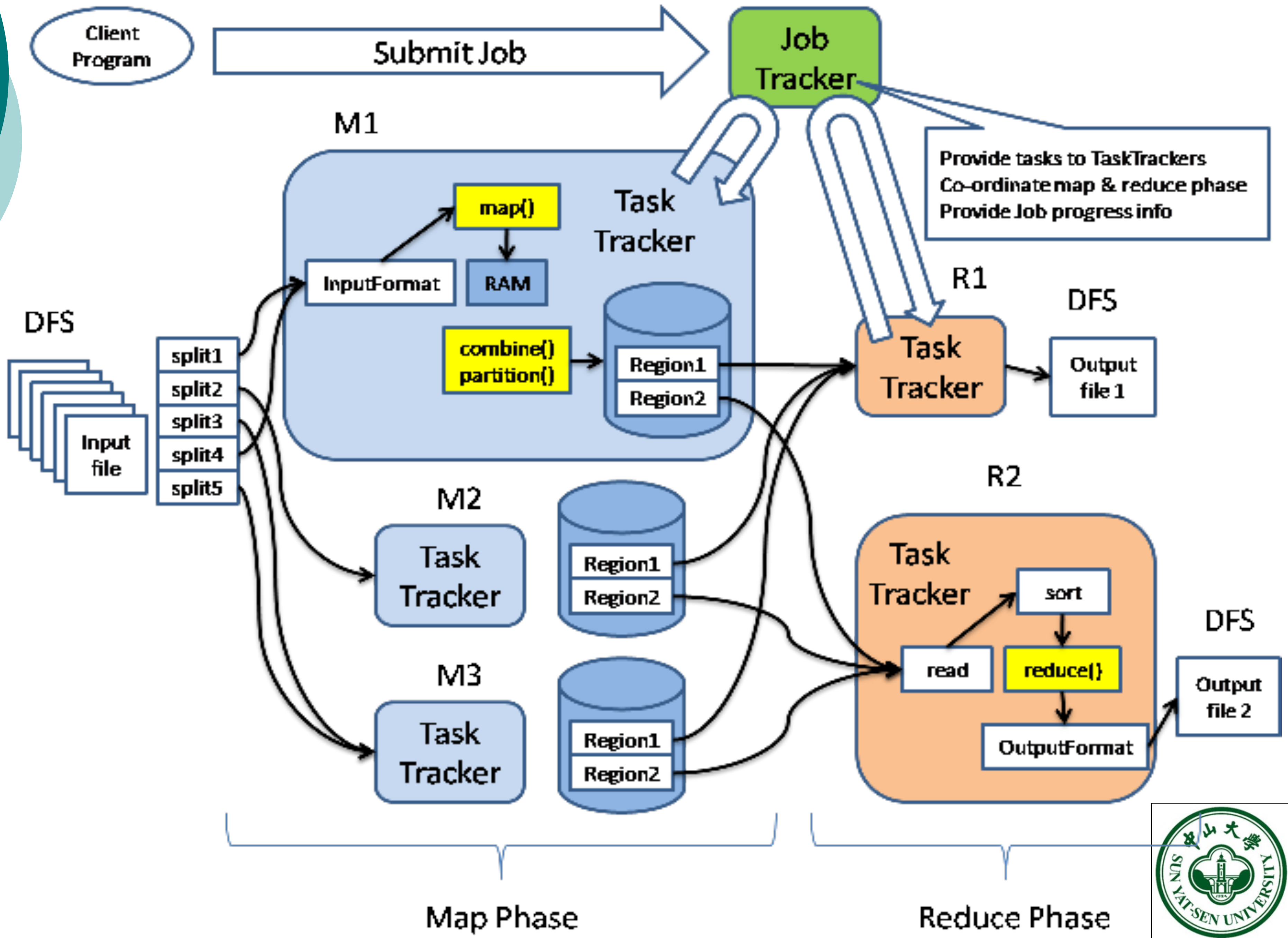
- 200+ processors
- 200+ terabyte database
- 10^{10} total clock cycles
- 0.1 second response time
- 5¢ average advertising revenue

Motivation: Large Scale Data Processing

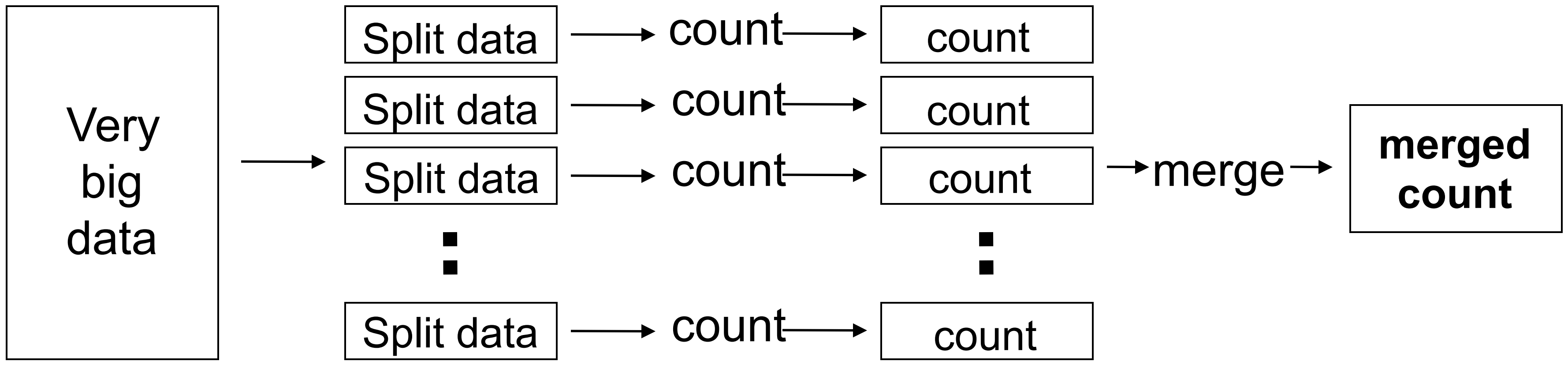
- Want to process lots of data (> 1 TB)
- Want to parallelize across hundreds /thousands of CPUs
- … Want to make this easy

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

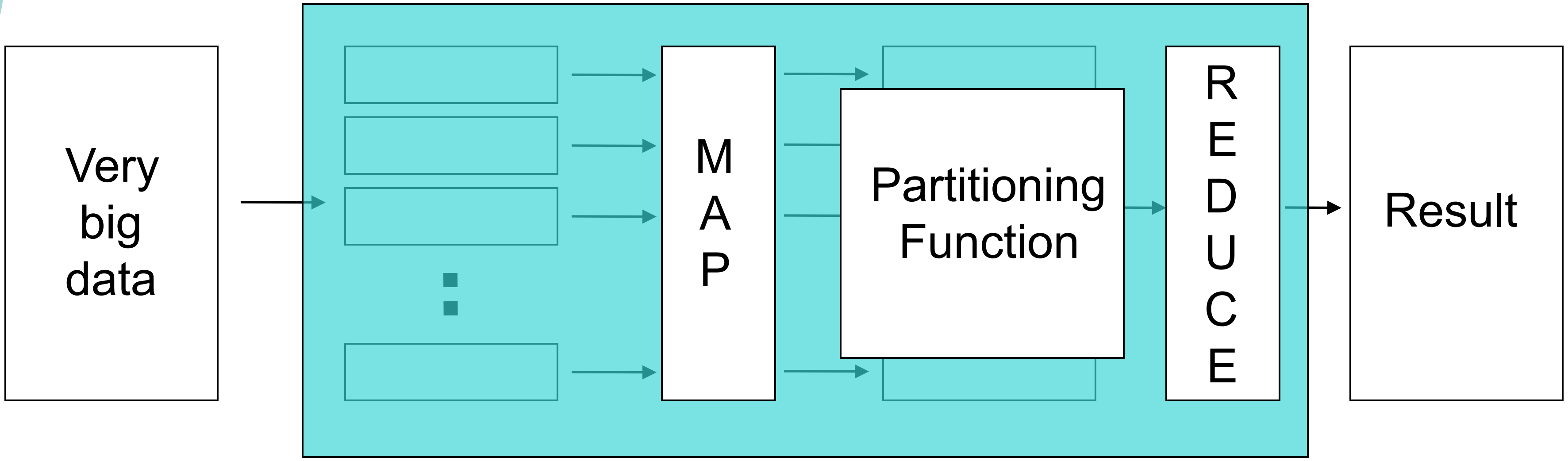

Master
Go and do thy bidding
Workers
In progress
In progress
Idle
In progress
In progress
Idle


Client Program
Submit Job
Job Tracker
Provide tasks to TaskTrackers
Co-ordinate map & reduce phase
Provide Job progress info
M1
Task Tracker
map()
RAM
InputFormat
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.

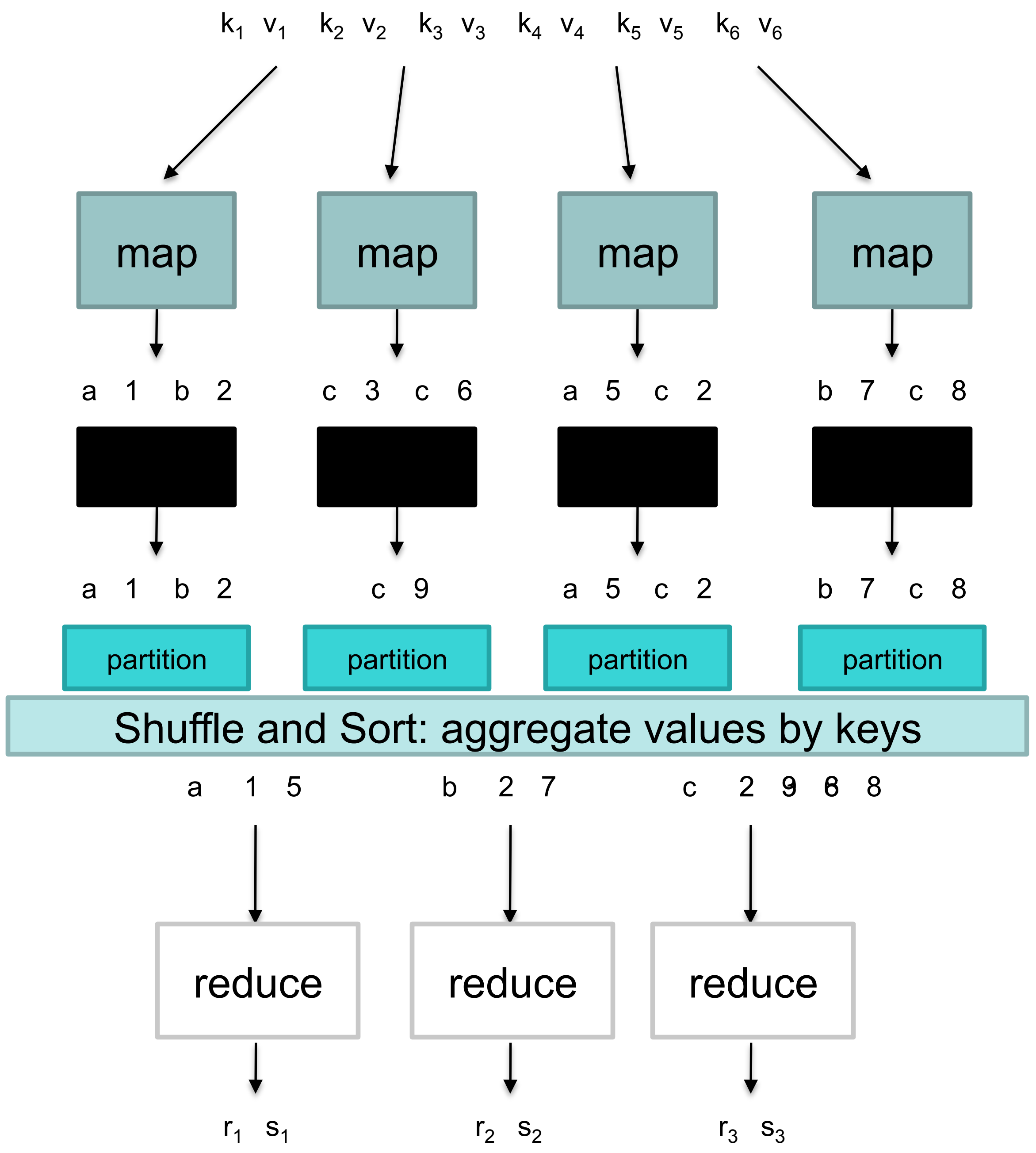
Mapper

- 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)


k_1 v_1 k_2 v_2 k_3 v_3 k_4 v_4 k_5 v_5 k_6 v_6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
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
reduce
reduce
reduce
r_1 s_1
r_2 s_2
r_3 s_3

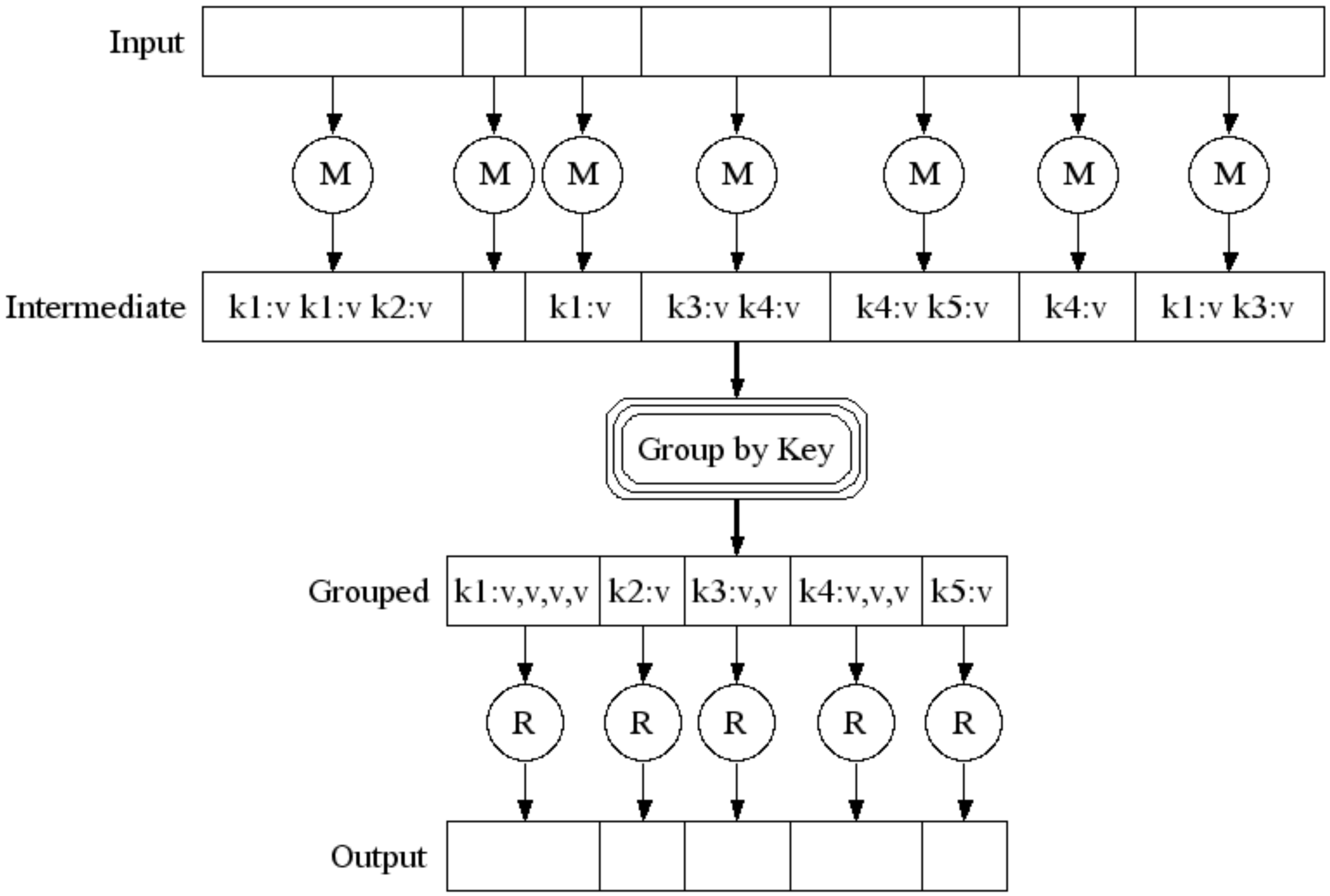
Partition&Shuffle

- 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)

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