NoSQL

Based on slides by
Mike Franklin and Jimmy Lin

Big Data (some old numbers)

- **Facebook:**
  - 130TB/day: user logs
  - 200-400TB/day: 83 million pictures
- **Google: > 25 PB/day processed data**
- **Gene sequencing: 100M kilobases per day per machine**
  - Sequence 1 human cell costs Illumina $1k
  - Sequence 1 cell for every infant by 2015?
  - 10 trillion cells / human body
- **Total data created in 2010: 1 ZettaByte (1,000,000 PB)/year**
  - ~60% increase every year

Big data is not only databases

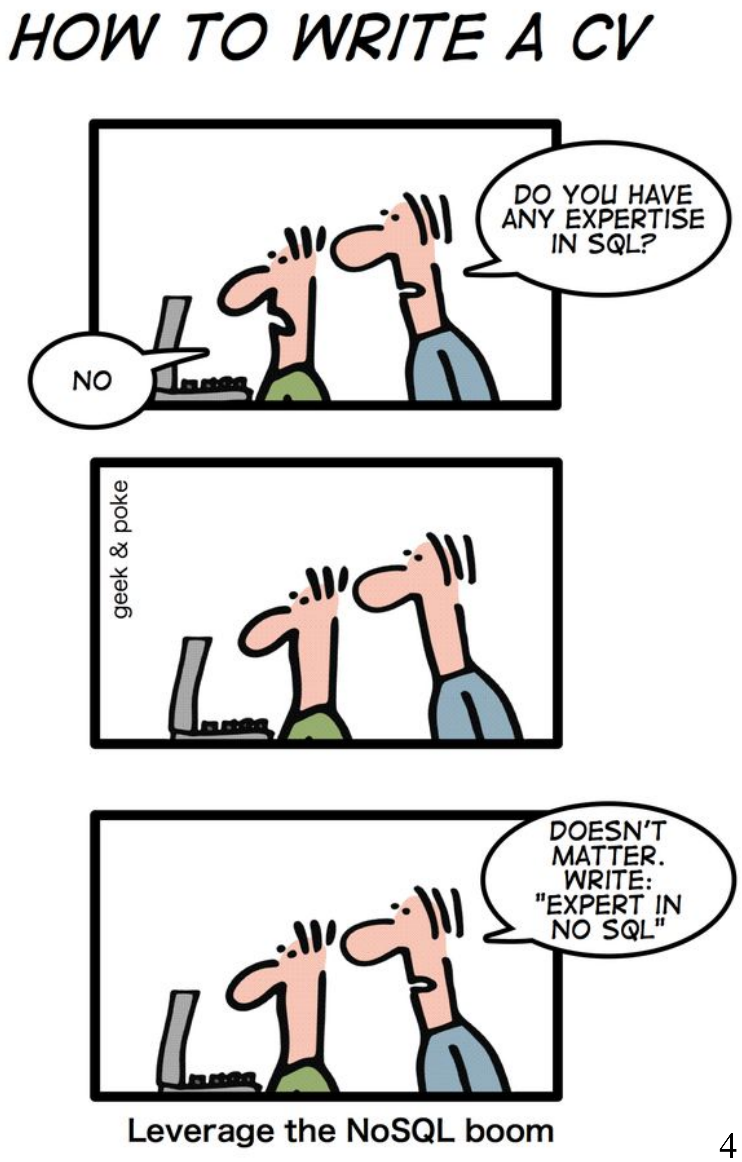
- **Big data is more about data analytics and on-line querying**

Many components:

- **Storage systems**
- **Database systems**
- **Data mining and statistical algorithms**
- **Visualization**

What is NoSQL?

from "Geek and Poke"

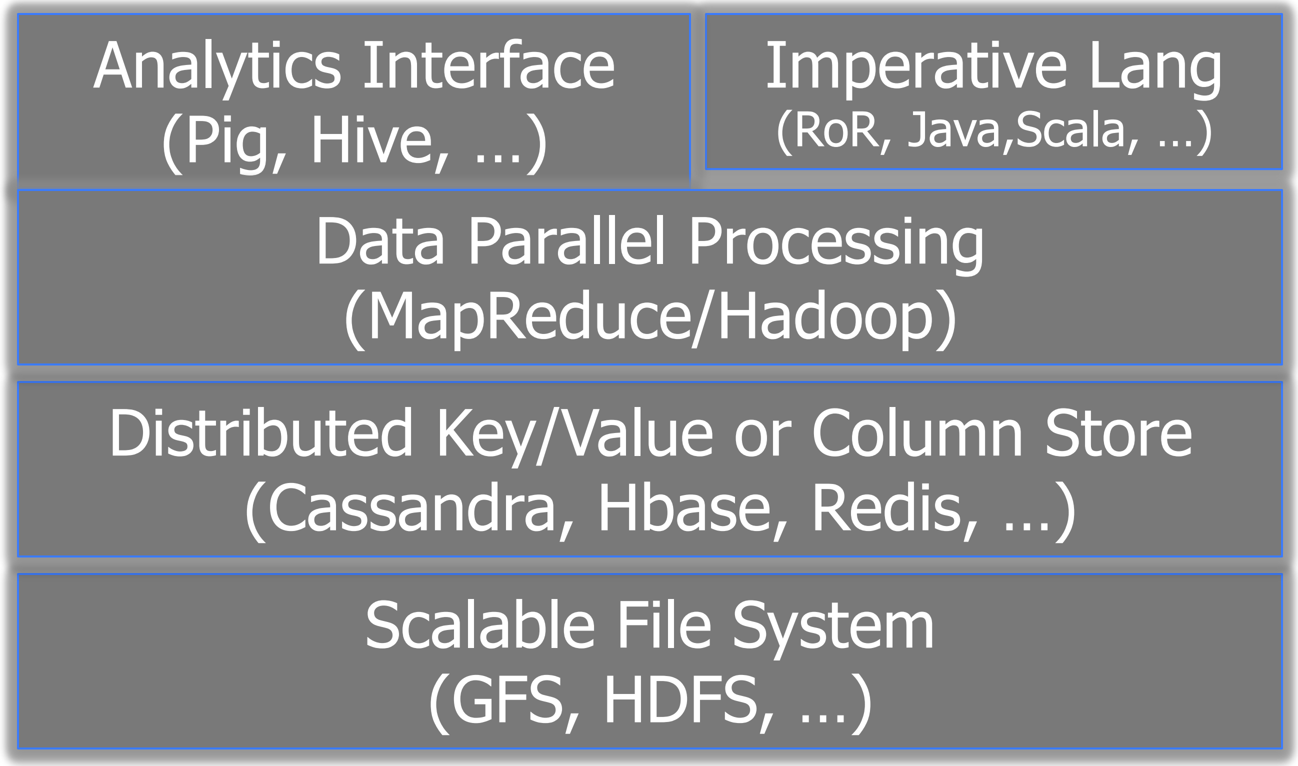

HOW TO WRITE A CV
DO YOU HAVE ANY EXPERTISE IN SQL?
NO
geek & poke
DOESN'T MATTER. WRITE: "EXPERT IN NO SQL"
Leverage the NoSQL boom

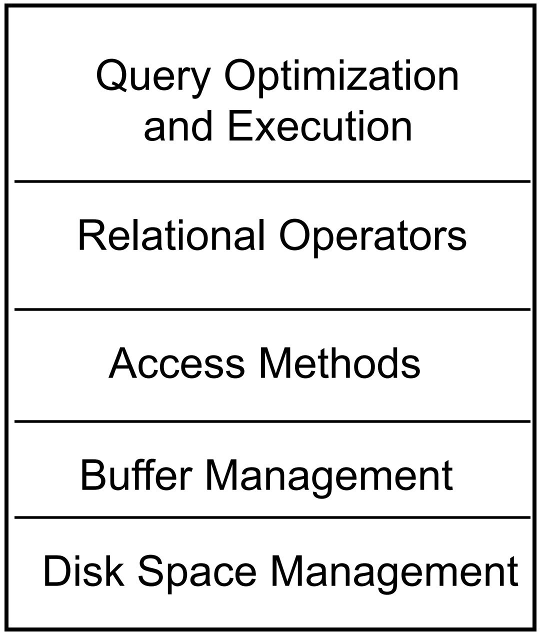
What is NoSQL?

- **An emerging "movement" around non-relational software for Big Data**
- **Roots are in the Google and Amazon homegrown software stacks**

Wikipedia: "A NoSQL database provides a mechanism for storage and retrieval of data that use looser consistency models than traditional relational databases in order to achieve horizontal scaling and higher availability. Some authors refer to them as "Not only SQL" to emphasize that some NoSQL systems do allow SQL-like query language to be used."

Some NoSQL Components





NoSQL features

- **Scalability is crucial!**
  - load increased rapidly for many applications
- **Large servers are expensive**

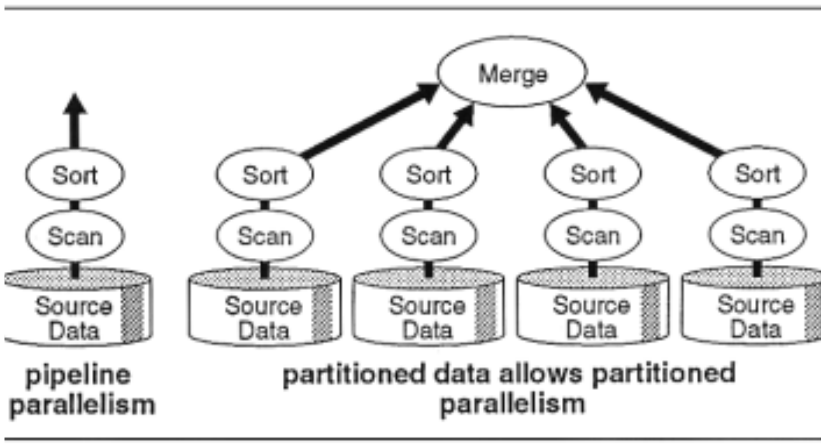
- **Solution: use clusters of small commodity machines**
  - need to partition the data and use replication (sharding)
  - cheap (usually open source!)
  - cloud-based storage

NoSQL features

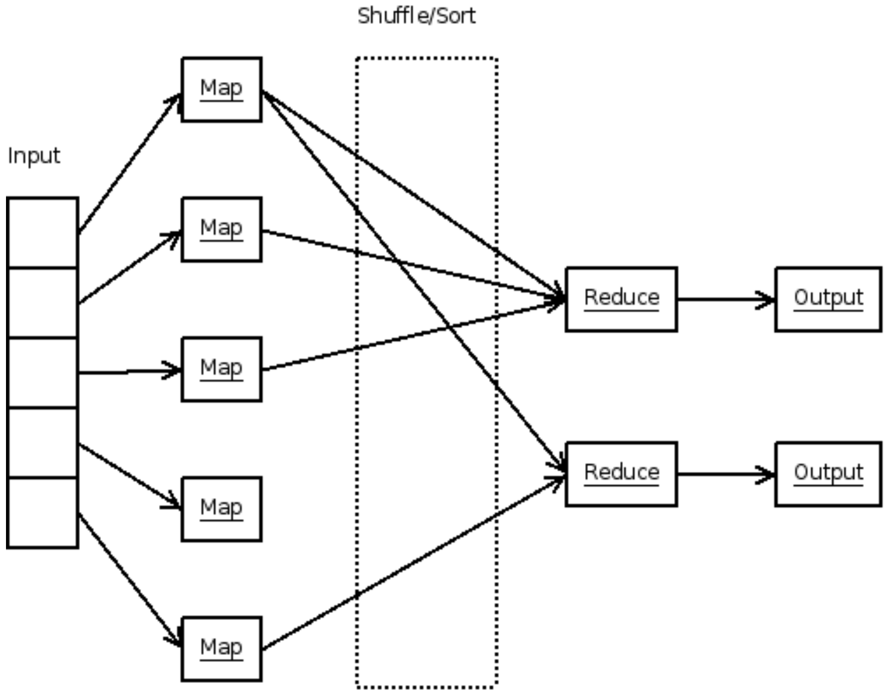
- **Sometimes not a well defined schema**

- **Allow for semi-structured data**
  - still need to provide ways to query efficiently (use of index methods)
  - need to express specific types of queries easily

Scalability



Parallel Database
(circa 1990)



Map Reduce
(circa 2005)

Scalability (continued)

- **Often cited as the main reason for moving from DB technology to NoSQL**
- **DB Position: there is no reason a parallel DBMS cannot scale to 1000's of nodes**
- **NoSQL Position: a) Prove it; b) it will cost too much anyway**

Flavors of NoSQL

Four main types:

- key-value stores
- document databases
- column-family (aka big-table) stores
- graph databases

=>Here we will talk more about "Document" databases (MongoDB)

Key-Value Stores

There are many systems like that: Redis, MemcacheDB, Amazon's DynamoDB, Voldemort

- **Simple data model: key/value pairs**
  - the DBMS does not attempt to interpret the value
- **Queries are limited to query by key**
  - get/put/update/delete a key/value pair
  - iterate over key/value pairs

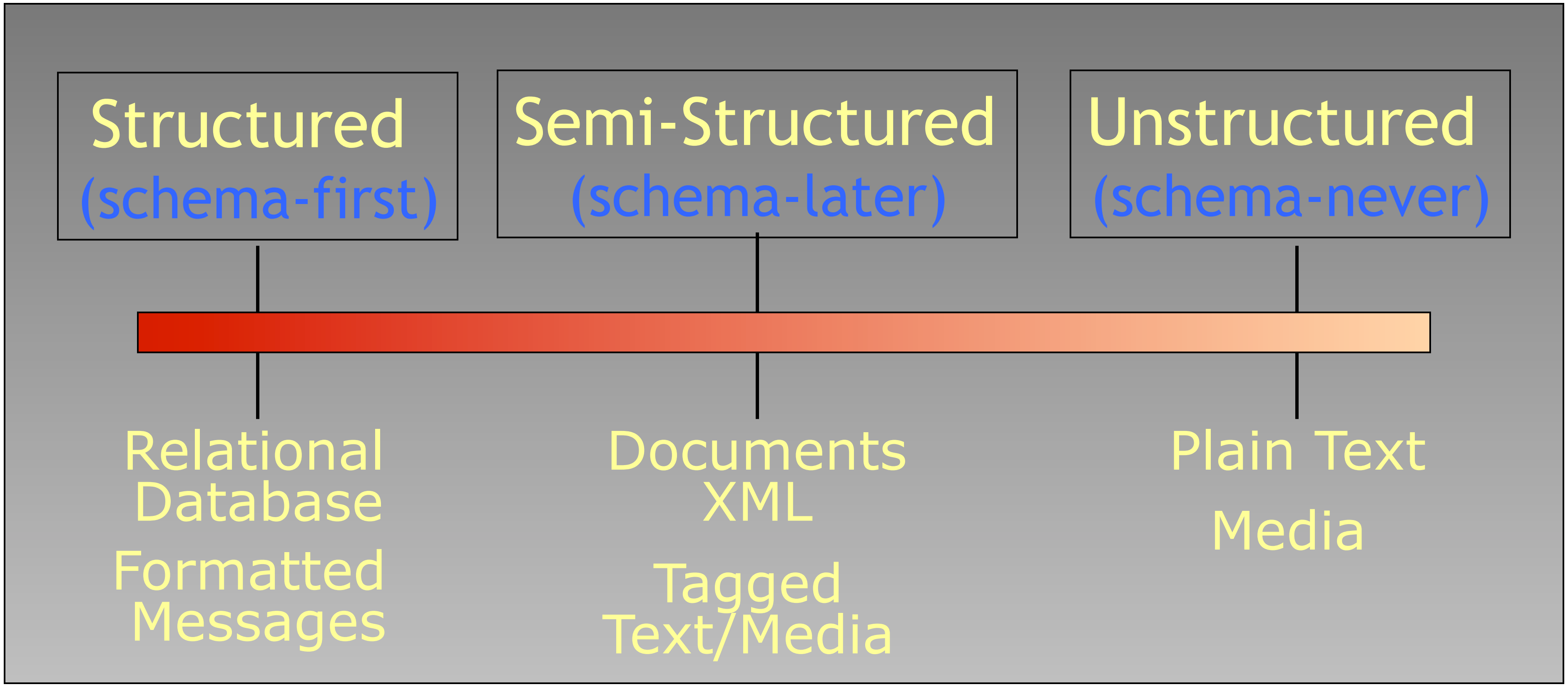
Document Databases

Examples include: MongoDB, CouchDB, Terrastore

- **Also store key/value pairs**
  - However, the value is a document.
    - expressed using some sort of semi-structured data model
      - XML
      - more often: JSON or BSON (JSON's binary counterpart)
  - the value can be examined and used by the DBMS (unlike in key/data stores)
- **Queries can be based on the key (as in key/value stores), but more often they are based on the contents of the document.**

- **Here again, there is support for sharding and replication.**
  - the sharding can be based on values within the document

The Structure Spectrum



MongoDB (An example of a Document Database)

-Data are organized in **collections.** A collection stores a set of **documents**.

- Collection like table and document like record
  - but: each document can have a different set of attributes even in the same collection
  - Semi-structured schema!
- Only requirement: every document should have an **"_id"** field

  - hu**mongo**us => Mongo

Example mongodb

```
{    "_id”:ObjectId("4efa8d2b7d284dad101e4bc9"),
      "Last Name": ” Cousteau",
      "First Name": ” Jacques-Yves",
      "Date of Birth": ”06-1-1910" },

  {    "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
     "Last Name": "PELLERIN",
     "First Name": "Franck",
     "Date of Birth": "09-19-1983",
     "Address": "1 chemin des Loges",
     "City": "VERSAILLES" }
```

Example Document Database: MongoDB

Key features include:

- **JSON-style documents**
  - actually uses BSON (JSON's binary format)
- **replication for high availability**
- **auto-sharding for scalability**
- **document-based queries**
- **can create an index on any attribute**
  - for faster reads

MongoDB Terminology

relational term <== >MongoDB equivalent

----

database <== > database

table <== > collection

row <== > document

attributes <== > fields (field-name:value pairs)

primary key <== > the _id field, which is the key associated with the document

JSON

- **JSON is an alternative data model for semi-structured data.**
  - JavaScript Object Notation
- **Built on two key structures:**
  - an object, which is a sequence of name/value pairs

```
{ ”_id": "1000",
  "name": "Sanders Theatre",
  "capacity": 1000 }
```

  - an array of values ["123", "222", "333"]
- A value can be:
  - an atomic value: string, number, true, false, null
  - an object
  - an array

The _id Field

Every MongoDB document must have an _id field.

- its value must be unique within the collection
- acts as the primary key of the collection
- it is the key in the key/value pair

- If you create a document without an _id field:
  - MongoDB adds the field for you
  - assigns it a unique BSON ObjectID
  - example from the MongoDB shell:

```
> db.test.save({ rating: "PG-13" })
> db.test.find() { "_id" :ObjectId("528bf38ce6d3df97b49a0569"),
"rating" : "PG-13" }
```

- Note: quoting field names is optional (see rating above)

Data Modeling in MongoDB

Need to determine how to map entities and relationships => collections of documents

- Could in theory give each type of entity:
    - its own (flexibly formatted) type of document
    - those documents would be stored in the same collection
- However, it can make sense to group different types of entities together.
    - create an aggregate containing data that tends to be accessed together

Capturing Relationships in MongoDB

- **Two options:**
  - 1. store references to other documents using their _id values
  - 2. embed documents within other documents

Example relationships

Consider the following documents examples:

```
{
  "_id":ObjectId("52ffc33cd85242f436000001"),
  "name": "Tom Hanks",
  "contact": "987654321",
  "dob": "01-01-1991"
}
```

```
{
  "_id":ObjectId("52ffc4a5d85242602e000000"),
  "building": "22 A, Indiana Apt",
  "pincode": 123456,
  "city": "Los Angeles",
  "state": "California"
}
```

Here is an example of embedded relationship:

```
{
  "_id":ObjectId("52ffc33cd85242f436000001"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "name": "Tom Benzamin",
  "address": [
    {
      "building": "22 A, Indiana Apt",
      "pincode": 123456,
      "city": "Los Angeles",
      "state": "California"
    },
    {
      "building": "170 A, Acropolis Apt",
      "pincode": 456789,
      "city": "Chicago",
      "state": "Illinois"
    }
  ]
}
```

And here an example of reference based

```
{
  "_id":ObjectId("52ffc33cd85242f436000001"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "name": "Tom Benzamin",
  "address_ids": [
    ObjectId("52ffc4a5d85242602e000000"),
    ObjectId("52ffc4a5d85242602e000001")
  ]
}
```

Queries in MongoDB

Each query can only access a single collection of documents.

- Use a method called

```
db.collection.find(<selection>, <projection>)
```

- Example: find the names of all R-rated movies:

```
> db.movies.find({ rating: 'R' }, { name: 1 })
```

Projection

- **Specify the name of the fields that you want in the output with 1 (0 hides the value)**

- Example:
  - `>db.movies.find({},{"title":1,_id:0})`

  (will report the title but not the id)

Selection

- **You can specify the condition on the corresponding attributes using the find:**

```
>db.movies.find({ rating: "R", year: 2000 },
{ name: 1, runtime: 1 })
```

- Operators for other types of comparisons:

MongoDB	SQL equivalent
$gt, $gte	>, >=
$lt, $lte	<, <=
$ne	!=

Example: find the names of movies with an earnings <= 200000

```
> db.movies.find({ earnings: { $lte: 200000 }})
```

- **For logical operators $and, $or, $nor**
  - use an array of conditions and apply the logical operator among the array conditions:

```
> db.movies.find({ $or: [ { rating: "R" }, { rating: "PG-13" } ] })
```

Aggregation

- **Recall the aggregate operators in SQL: AVG(), SUM(), etc.**
  More generally, aggregation involves computing a result
  from a collection of data.

- **MongoDB supports several approaches to aggregation:**
  - single-purpose aggregation methods
  - an aggregation pipeline
  - map-reduce

Aggregation pipelines are more flexible and useful (see next):
https://docs.mongodb.com/manual/core/aggregation-pipeline/

Simple Aggregations

- **db.collection.count(<selection>)**

  returns the number of documents in the collection

  that satisfy the specified selection document

Example: how may R-rated movies are shorter than 90 minutes?

```
>db.movies.count({ rating: "R”, runtime: { $lt: 90 }})
```

- **db.collection.distinct(<field>, <selection>)**

returns an array with the distinct values of the specified field

in documents that satisfy the specified selection document

if omit the query, get all distinct values of that field

- which actors have been in one or more of the top 10 grossing movies?

```
>db.movies.distinct("actors.name”, { earnings_rank: { $lte: 10 }})
```

Aggregation Pipeline

- A very powerful approach to write queries in MongoDB is to use pipelines.

- We execute the query in stages. Every stage gets as input some documents, applies filters/aggregations/projections and outputs some new documents. These documents are the input to the next stage (next operator) and so on

Aggregation Pipeline example

```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [
    -74.016323,
    40.710537
  ]
}
```

- Example for the zipcodes database:

```
 > db.zipcodes.aggregate( [
   { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
   { $match: { totalPop: { $gte: 10*1000*1000 } } }
] )
```

Here we use group_by to group documents per state, compute sum of population and output documents with _id, totalPop (_id has the name of the state). The next stage finds a match for all states the have more than 10M population and outputs the state and total population.

More here:
https://docs.mongodb.com/v3.0/tutorial/aggregation-zip-code-data-set/

continued:

In SQL:

```
SELECT state, SUM(pop) AS totalPop
FROM zipcodes
GROUP BY state
HAVING totalPop >= (10*1000*1000)
```

Output example:

```
{
  "_id" : "AK",
  "totalPop" : 550043
}
```

```
db.zipcodes.aggregate( [
   { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
   { $match: { totalPop: { $gte: 10*1000*1000 } } }
] )
```

more examples:

```
db.zipcodes.aggregate( [
  { $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
  { $group: { _id: "$_id.state", avgCityPop: { $avg: "$pop" } } }
] )
```

What we compute here?

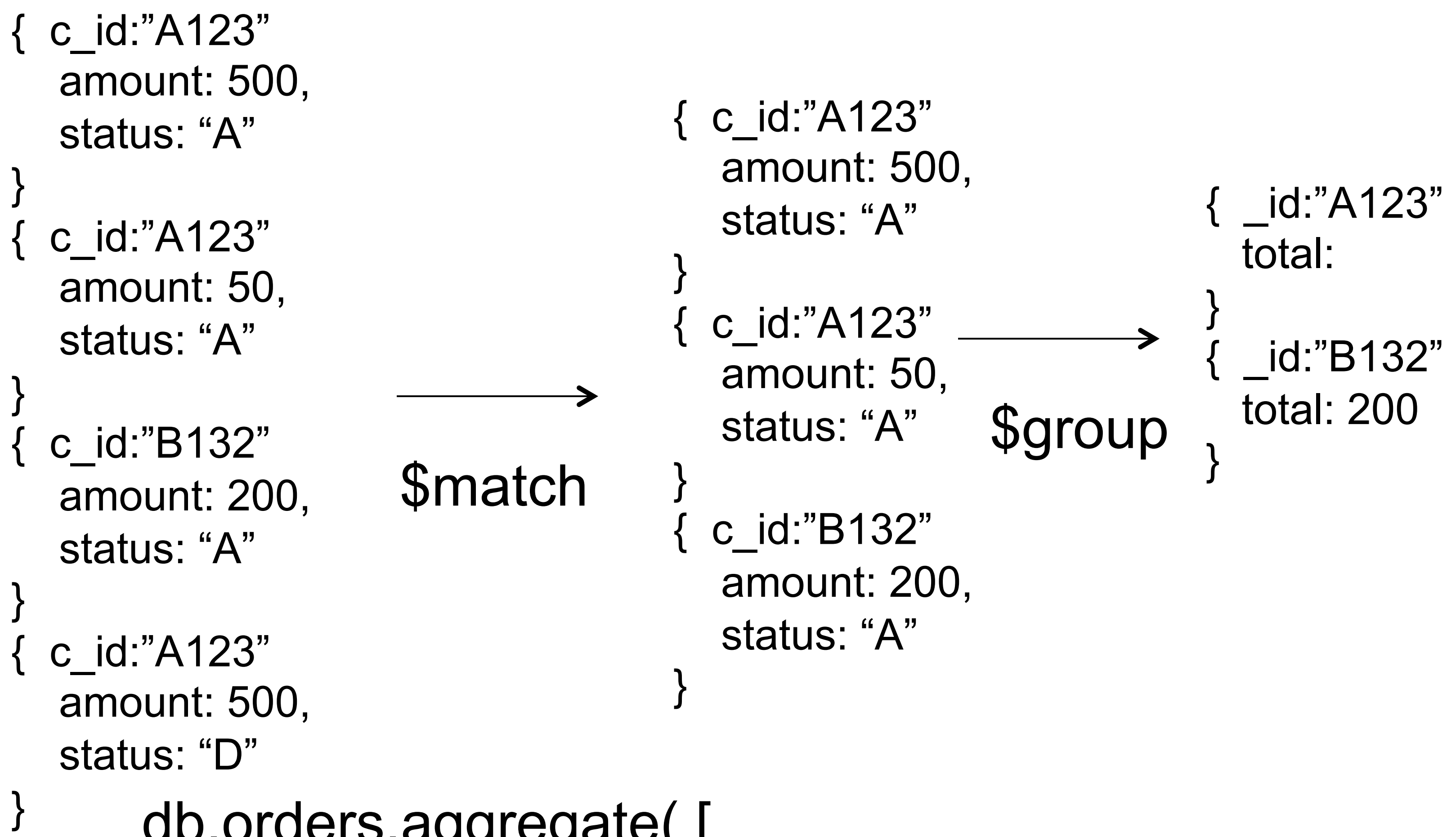
First we get groups by city and state and for each group we compute the population.
Then we get groups by state and compute the average city population

```
{
  "_id" : {
    "state" : "CO",
    "city" : "EDGEWATER"
  },
  "pop" : 13154
}
```

$\longrightarrow$

```
{
  "_id" : "MN",
  "avgCityPop" : 5335
}
```

Aggregation Pipeline example



```
{  c_id:"A123"
    amount: 500,
    status: "A"
}
{  c_id:"A123"
    amount: 50,
    status: "A"
}
{  c_id:"B132"
    amount: 200,
    status: "A"
}
{  c_id:"A123"
    amount: 500,
    status: "D"
}
```

$match

```
{  c_id:"A123"
    amount: 500,
    status: "A"
}
{  c_id:"A123"
    amount: 50,
    status: "A"
}
{  c_id:"B132"
    amount: 200,
    status: "A"
}
```

$group

```
{  _id:"A123"
   total:
}
{  _id:"B132"
   total: 200
}
```

```
db.orders.aggregate( [
                { $match: {status: "A"}}
              { $group: {_id:"c_id", total: {$sum: $amount}}
   ])
```

Other Structure Issues

- **NoSQL: a) Tables are unnatural, b) "joins" are evil, c) need to be able to "grep" my data**

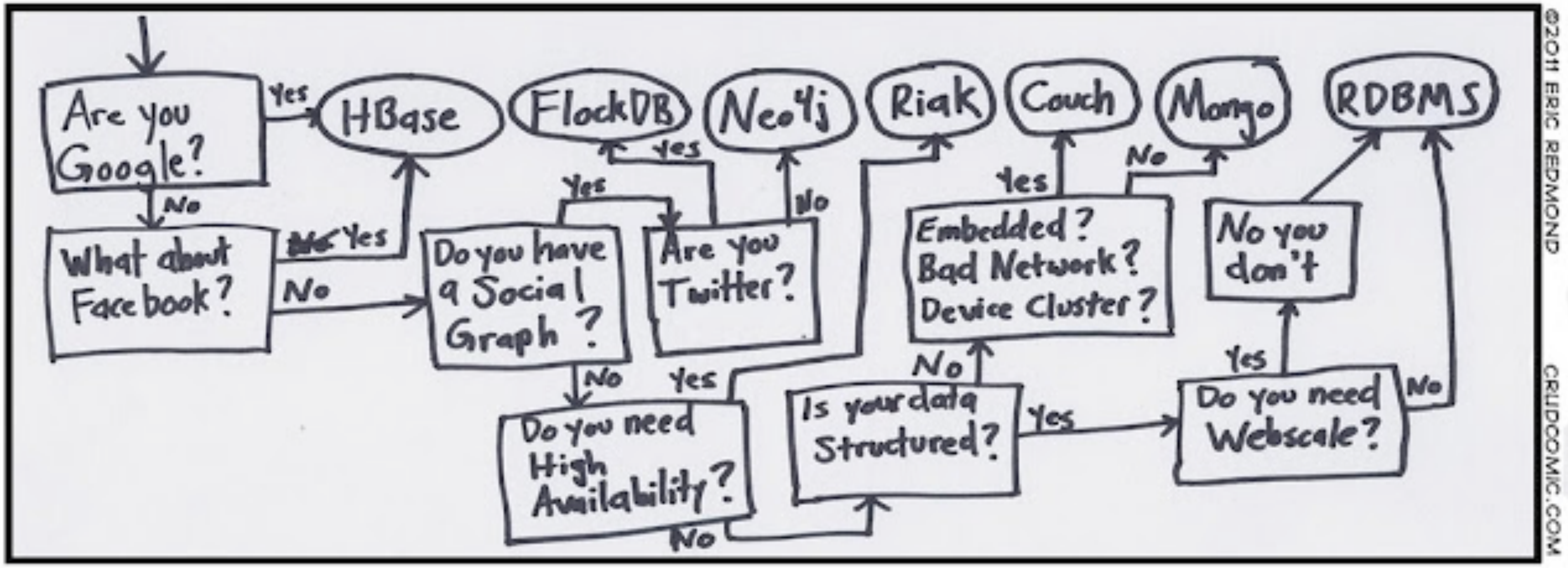
- **DB: a) Tables are a natural/neutral structure, b) data independence lets you precompute joins under the covers, c) this is a price of all the DBMS goodness you get**

This is an Old Debate – Object-oriented databases, XML DBs, Hierarchical, …

Fault Tolerance

- **DBs: coarse-grained FT – if trouble, restart transaction**
  - Fewer, Better nodes, so failures are rare
  - Transactions allow you to kill a job and easily restart it

- **NoSQL: Massive amounts of cheap HW, failures are the norm and massive data means long running jobs**
  - So must be able to do mini-recoveries
  - This causes some overhead (file writes)


Are you Google?
Yes
HBase
No
What about Facebook?
Yes
No
Do you have a Social Graph?
Yes
Are you Twitter?
Yes
FlockDB
No
Neo4j
No
Do you need High Availability?
Yes
Riak
No
Is your data Structured?
No
Embedded? Bad Network? Device Cluster?
Yes
Couch
No
Mongo
Yes
Do you need Webscale?
Yes
No you don't
No
RDBMS
©2011 ERIC REDMOND
CRUDCOMIC.COM

Cloud Computing Computation Models

- **Finding the right level of abstraction**
  - von Neumann architecture vs cloud environment
- **Hide system-level details from the developers**
  - No more race conditions, lock contention, etc.
- **Separating the *what* from *how***
  - Developer specifies the computation that needs to be performed
  - Execution framework ("runtime") handles actual execution

Similar to SQL!!

Typical Large-Data Problem

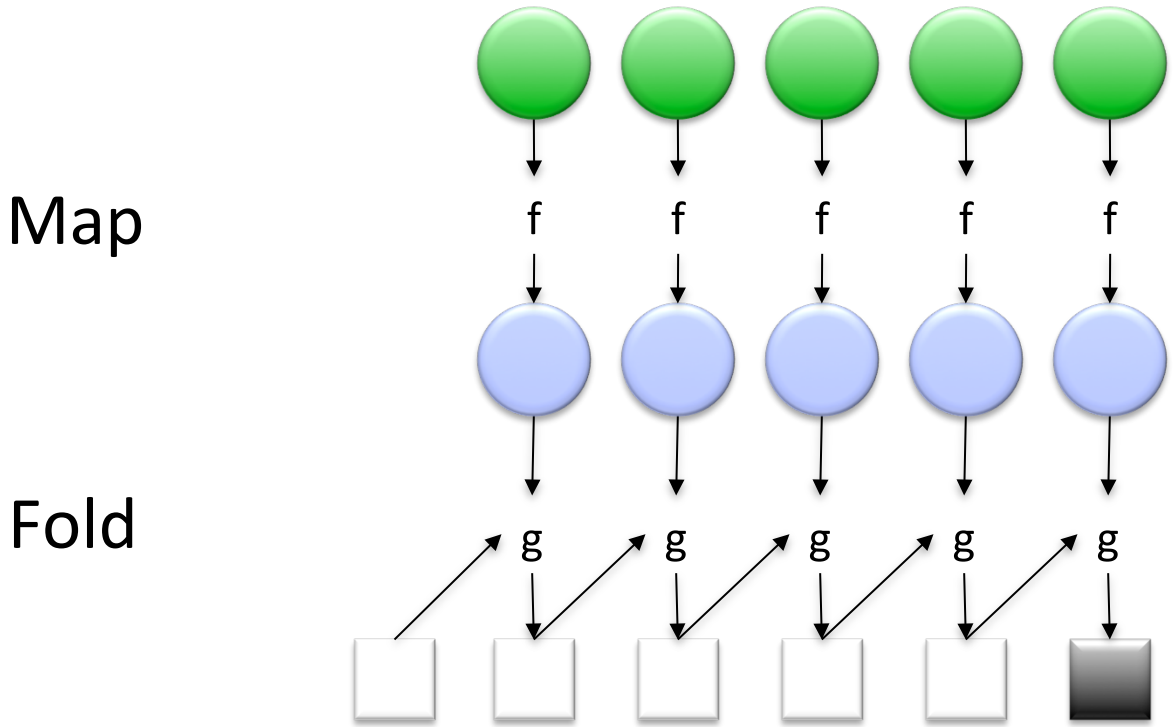
- **Iterate over a large number of records**
- **Extract something of interest from each**
- **Shuffle and sort intermediate results**
- **Aggregate intermedi**
- **Generate final outpu**

Map

Reduce

Key idea: provide a functional abstraction for these two operations – MapReduce

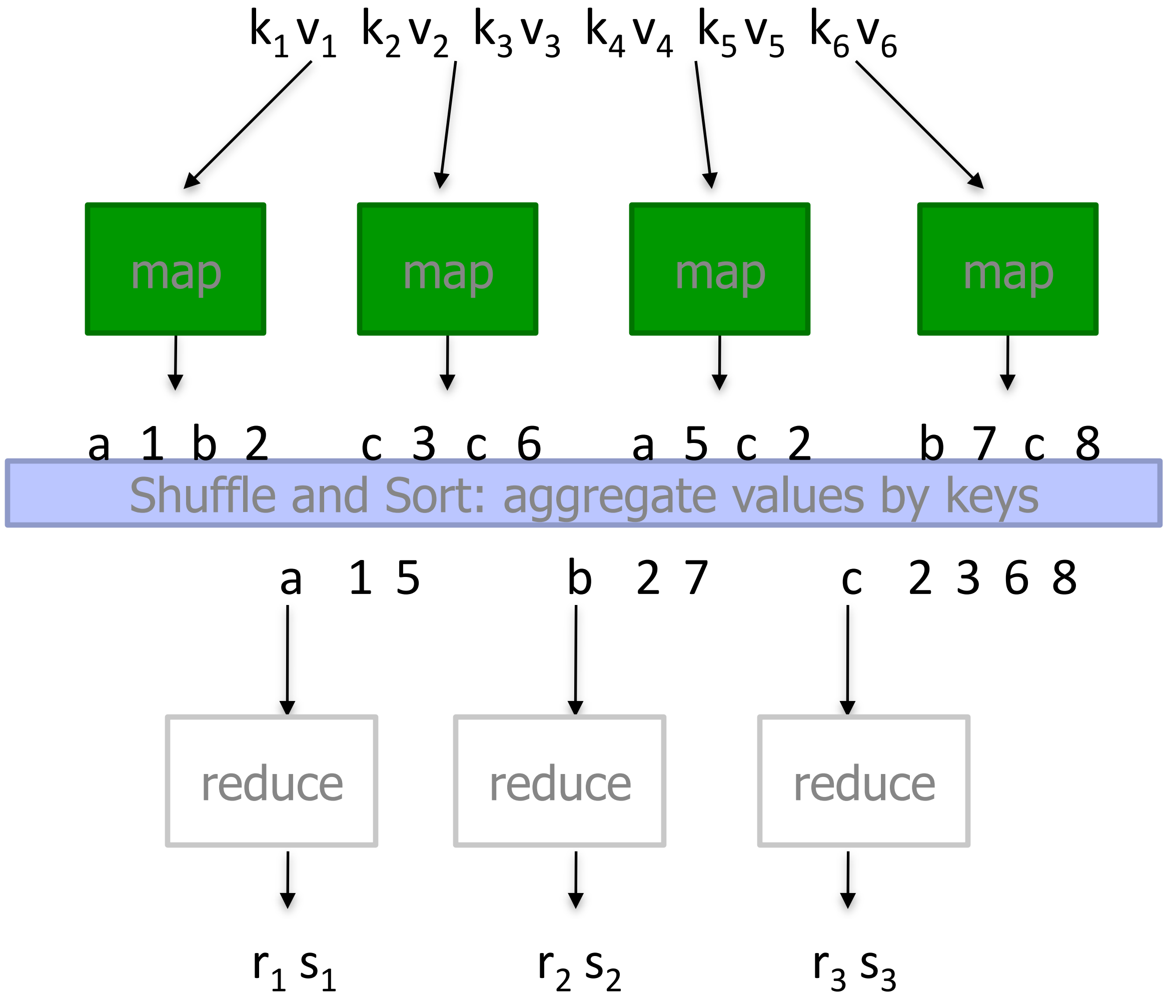
Roots in Functional Programming



MapReduce

- **Programmers specify two functions:**
  map (k, v) → <k’ , v’ >*
  reduce (k’ , v’) → <k’ , v’ ’>*
  - All values with the same key are sent to the same reducer
- **The execution framework handles everything else…**

MapReduce



$k_1 v_1$ $k_2 v_2$ $k_3 v_3$ $k_4 v_4$ $k_5 v_5$ $k_6 v_6$
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
$r_1 s_1$
$r_2 s_2$
$r_3 s_3$

MapReduce

- **Programmers specify two functions:**

  map (k, v) → <k', v'>*

  reduce (k', v') → <k', v'>*

  - All values with the same key are sent to the same reducer
- **The execution framework handles everything else…**

What's "everything else"?

MapReduce “Runtime”

- **Handles scheduling**
  - Assigns workers to map and reduce tasks
- **Handles “data distribution”**
  - Moves processes to data
- **Handles synchronization**
  - Gathers, sorts, and shuffles intermediate data
- **Handles errors and faults**
  - Detects worker failures and automatically restarts
- **Handles speculative execution**
  - Detects “slow” workers and re-executes work
- **Everything happens on top of a distributed FS (later)**

Sounds simple, but many challenges!

MapReduce

- **Programmers specify two functions:**

  map (k, v) → <k', v'>*

  reduce (k', v') → <k', v'>*

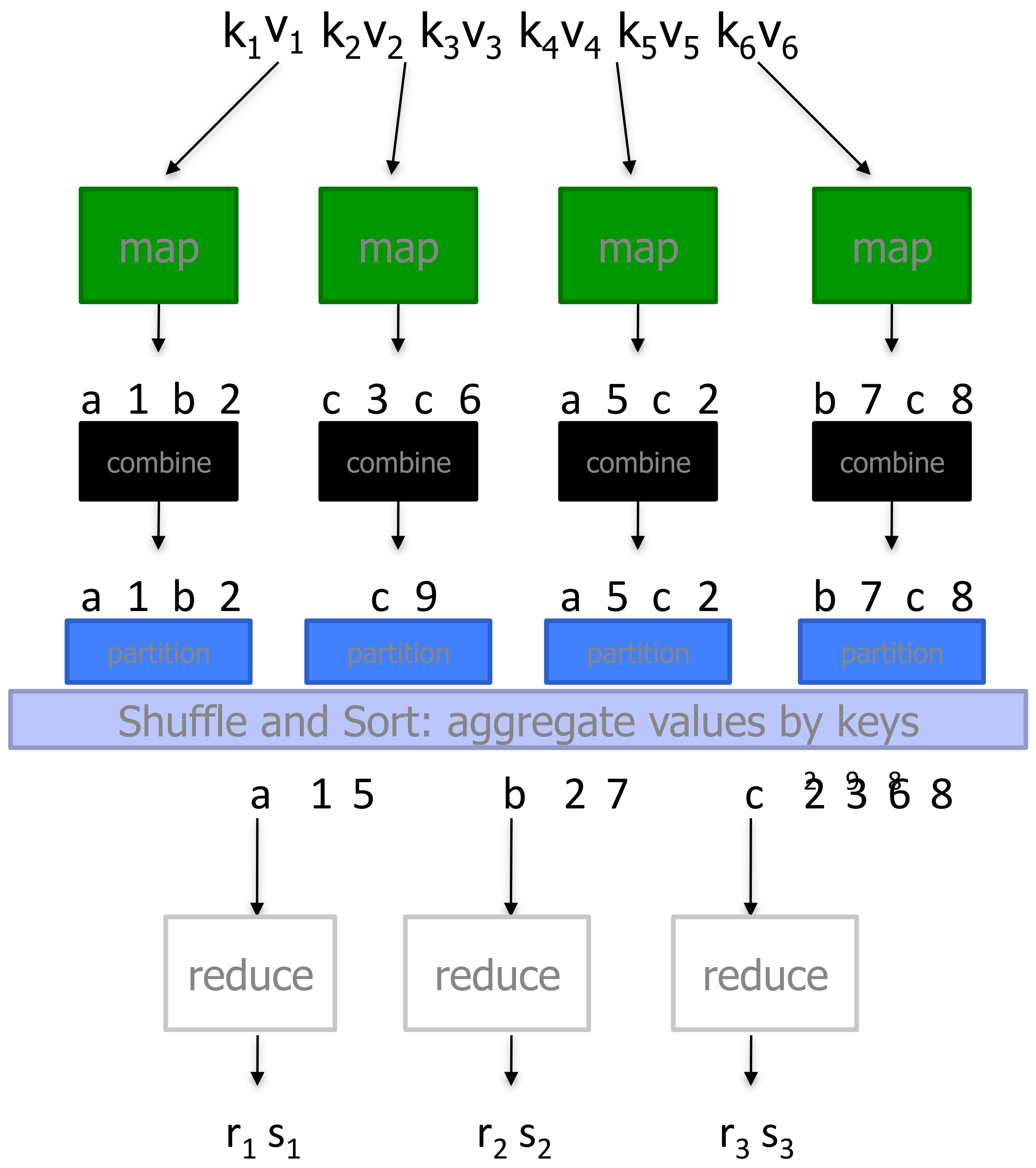
  - All values with the same key are reduced together
- **The execution framework handles everything else…**
- **Not quite…usually, programmers also specify:**

  partition (k', number of partitions) → partition for k'

  - Often a simple hash of the key, e.g., hash(k') mod R
  - Divides up key space for parallel reduce operations

  combine (k', v') → <k', v'>*

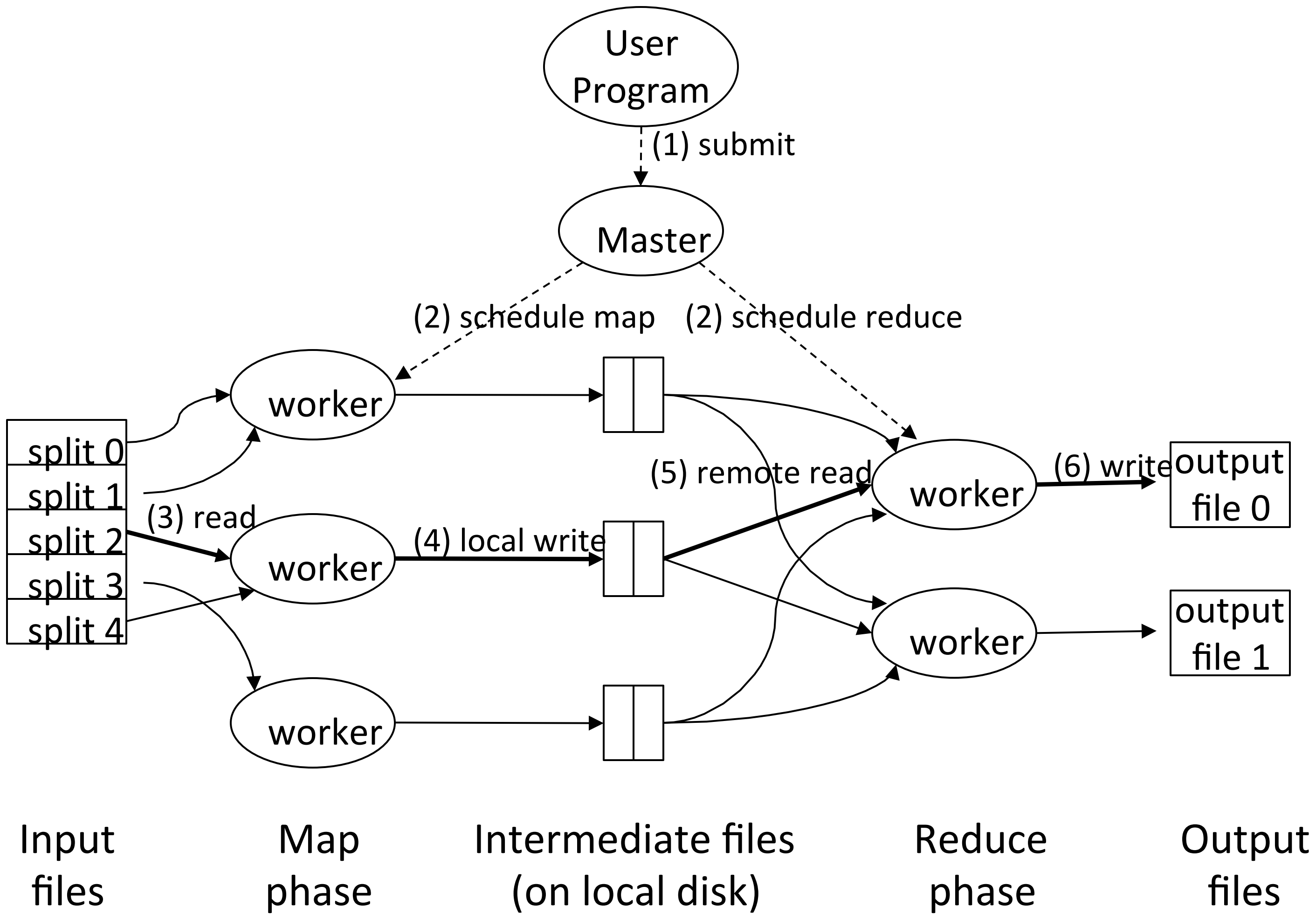
  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic


k_1v_1 k_2v_2 k_3v_3 k_4v_4 k_5v_5 k_6v_6
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 3 6 8
2 9 8
reduce
reduce
reduce
r_1 s_1
r_2 s_2
r_3 s_3

Two more details…

- **Barrier between map and reduce phases**
  - But we can begin copying intermediate data earlier
- **Keys arrive at each reducer in sorted order**
  - No enforced ordering *across* reducers

MapReduce Overall Architecture



Adapted from (Dean and Ghemawat, OSDI 2004)

“Hello World” Example: 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);
```

MapReduce can refer to…

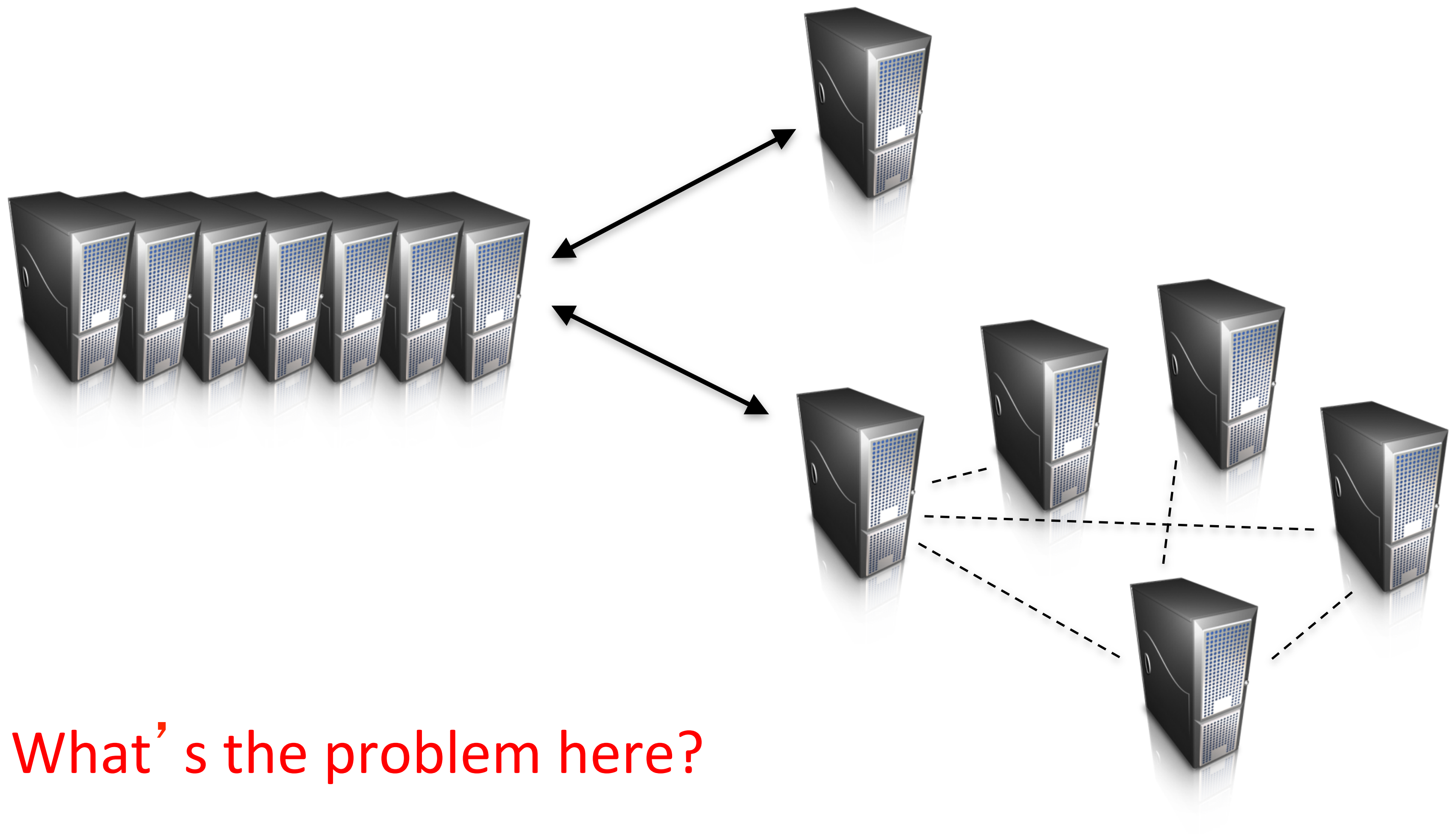
- **The programming model**
- **The execution framework (aka “runtime”)**
- **The specific implementation**

Usage is usually clear from context!

MapReduce Implementations

- **Google has a proprietary implementation in C++**
  - Bindings in Java, Python
- **Hadoop is an open-source implementation in Java**
  - Development led by Yahoo, used in production
  - Now an Apache project
  - Rapidly expanding software ecosystem, but still lots of room for improvement
- **Lots of custom research implementations**
  - For GPUs, cell processors, etc.

Cloud Computing Storage, or how do we get data to the workers?



What's the problem here?

Distributed File System

- **Don't move data to workers… move workers to the data!**
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- **Why?**
  - Network bisection bandwidth is limited
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
- **A distributed file system is the answer**
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

GFS: Assumptions

- **Choose commodity hardware over "exotic" hardware**
  - Scale "out", not "up"
- **High component failure rates**
  - Inexpensive commodity components fail all the time
- **"Modest" number of huge files**
  - Multi-gigabyte files are common, if not encouraged
- **Files are write-once, mostly appended to**
  - Perhaps concurrently
- **Large streaming reads over random access**
  - High sustained throughput over low latency

GFS: Design Decisions

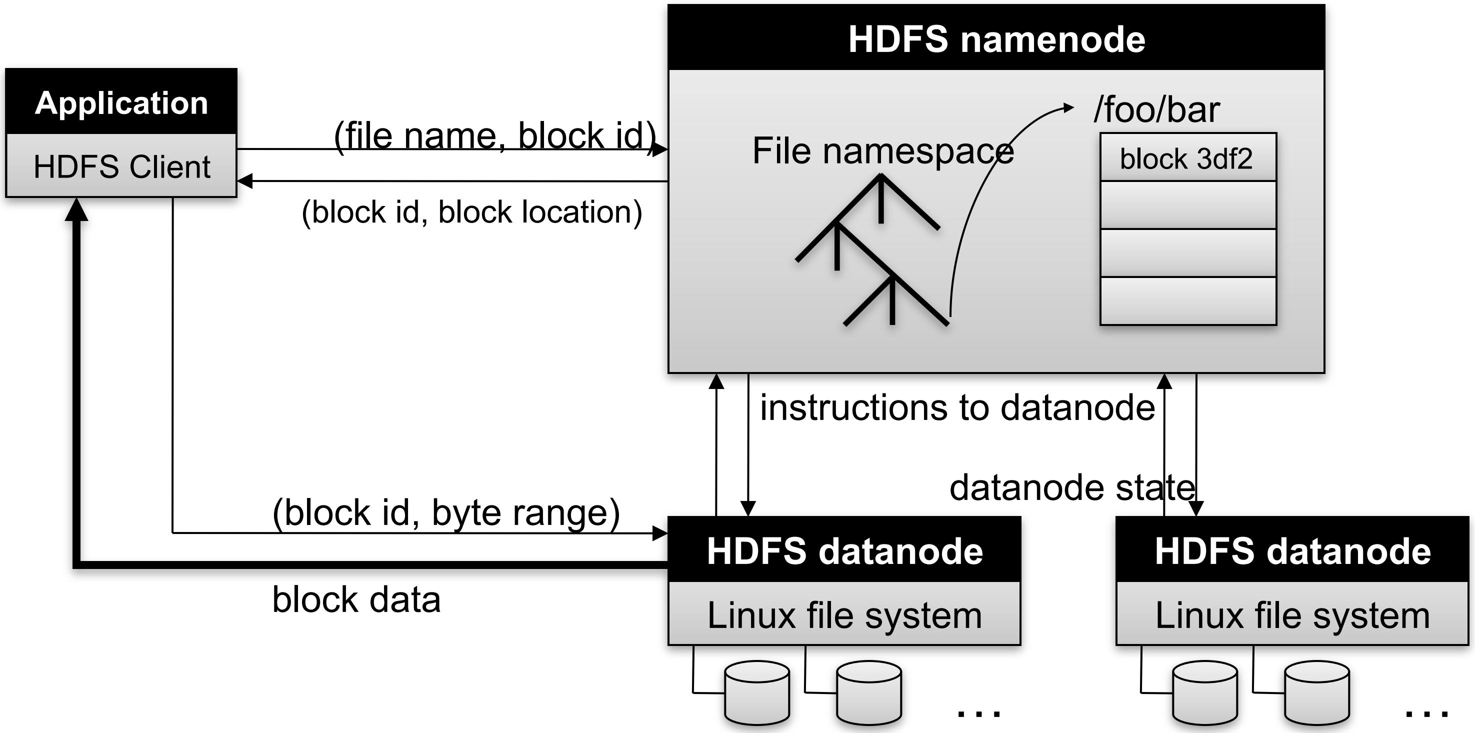
- **Files stored as chunks**
  - Fixed size (64MB)
- **Reliability through replication**
  - Each chunk replicated across 3+ chunkservers
- **Single master to coordinate access, keep metadata**
  - Simple centralized management
- **No data caching**
  - Little benefit due to large datasets, streaming reads
- **Simplify the API**
  - Push some of the issues onto the client (e.g., data layout)

HDFS = GFS clone (same basic ideas implemented in Java)

From GFS to HDFS

- **Terminology differences:**
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- **Functional differences:**
  - No file appends in HDFS (was planned)
  - HDFS performance is (likely) slower

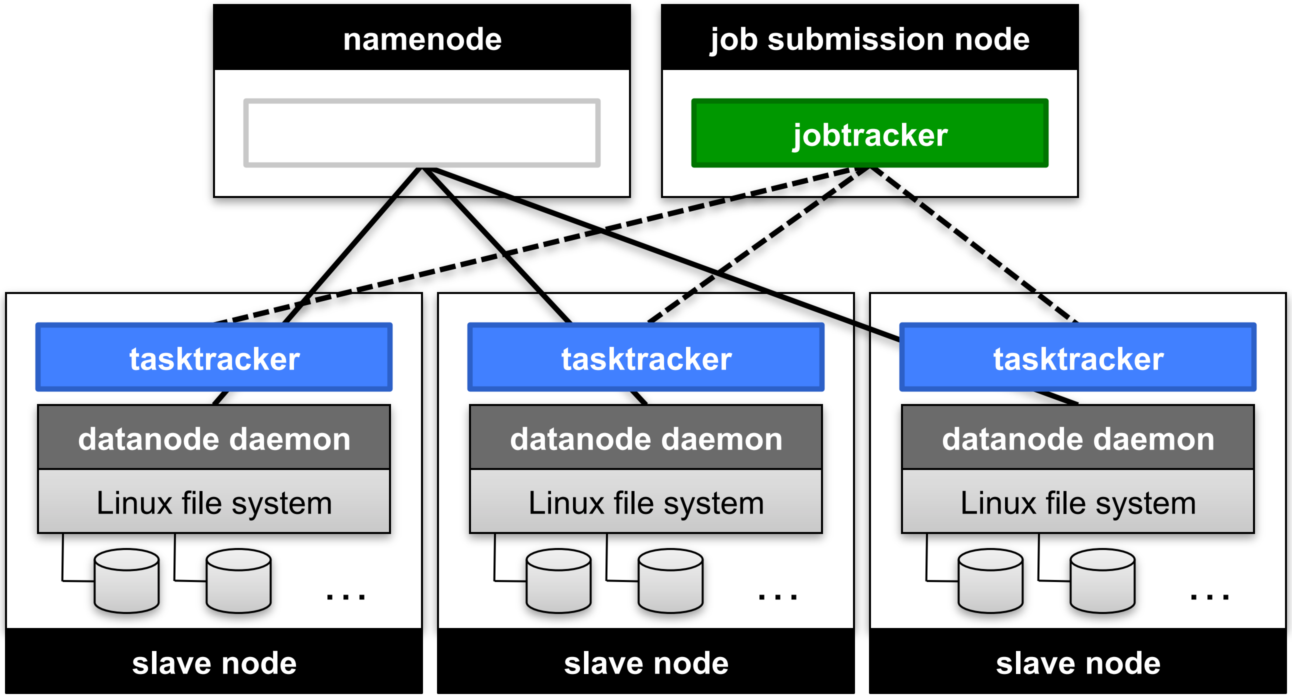
HDFS Architecture



Namenode Responsibilities

- **Managing the file system namespace:**
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- **Coordinating file operations:**
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- **Maintaining overall health:**
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

Putting everything together…



MapReduce/GFS Summary

- **Simple, but powerful programming model**
- **Scales to handle petabyte+ workloads**
  - Google: six hours and two minutes to sort 1PB (10 trillion 100-byte records) on 4,000 computers
  - Yahoo!: 16.25 hours to sort 1PB on 3,800 computers
- **Incremental performance improvement with more nodes**
- **Seamlessly handles failures, but possibly with performance penalties**