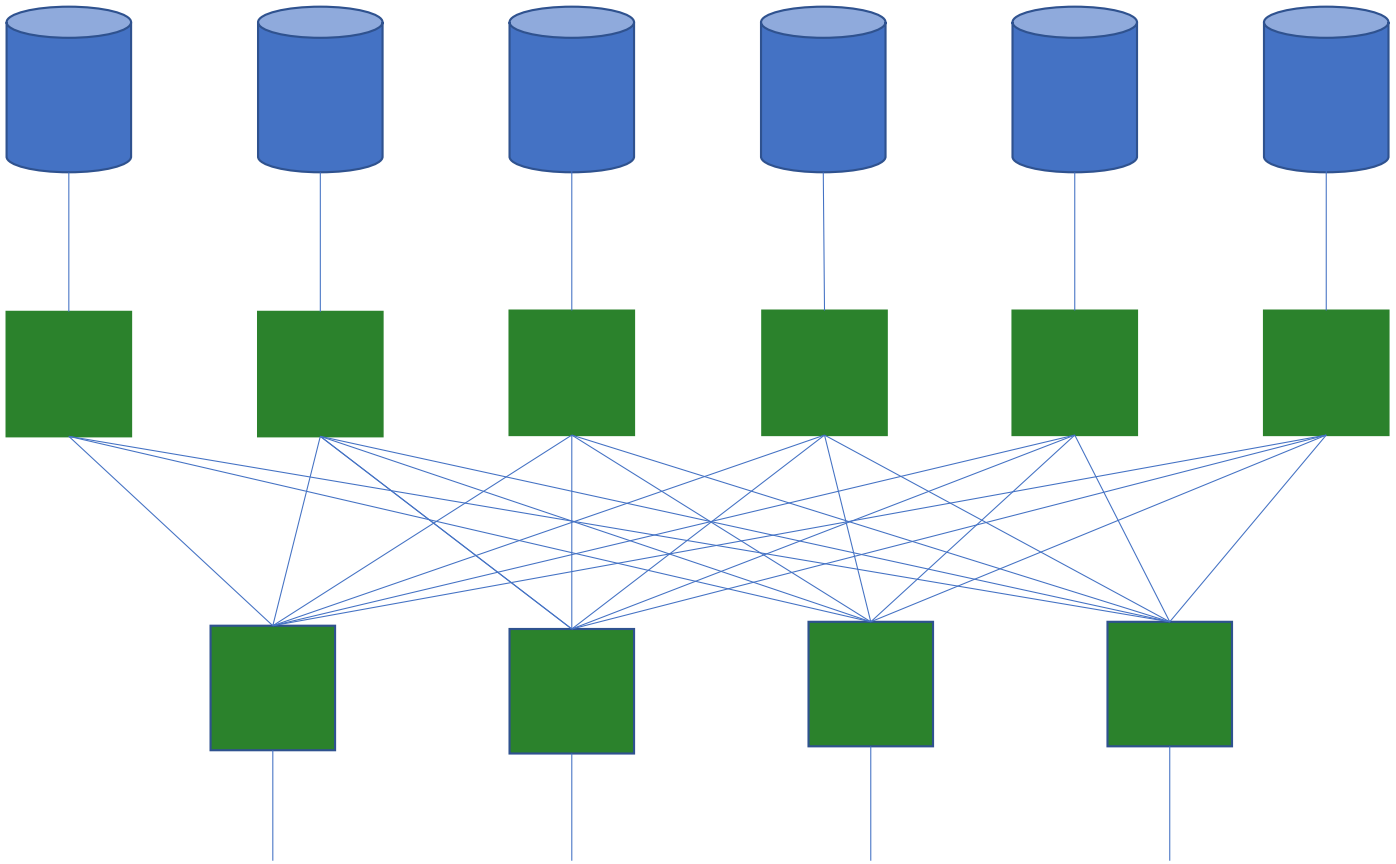
Today's Lecture

1. The MapReduce Abstraction
2. The MapReduce Programming Model
3. MapReduce Examples
4. Spark
5. NoSQL
6. MongoDB

1. The MapReduce Abstraction

The Map Reduce Abstraction for Distributed Algorithms



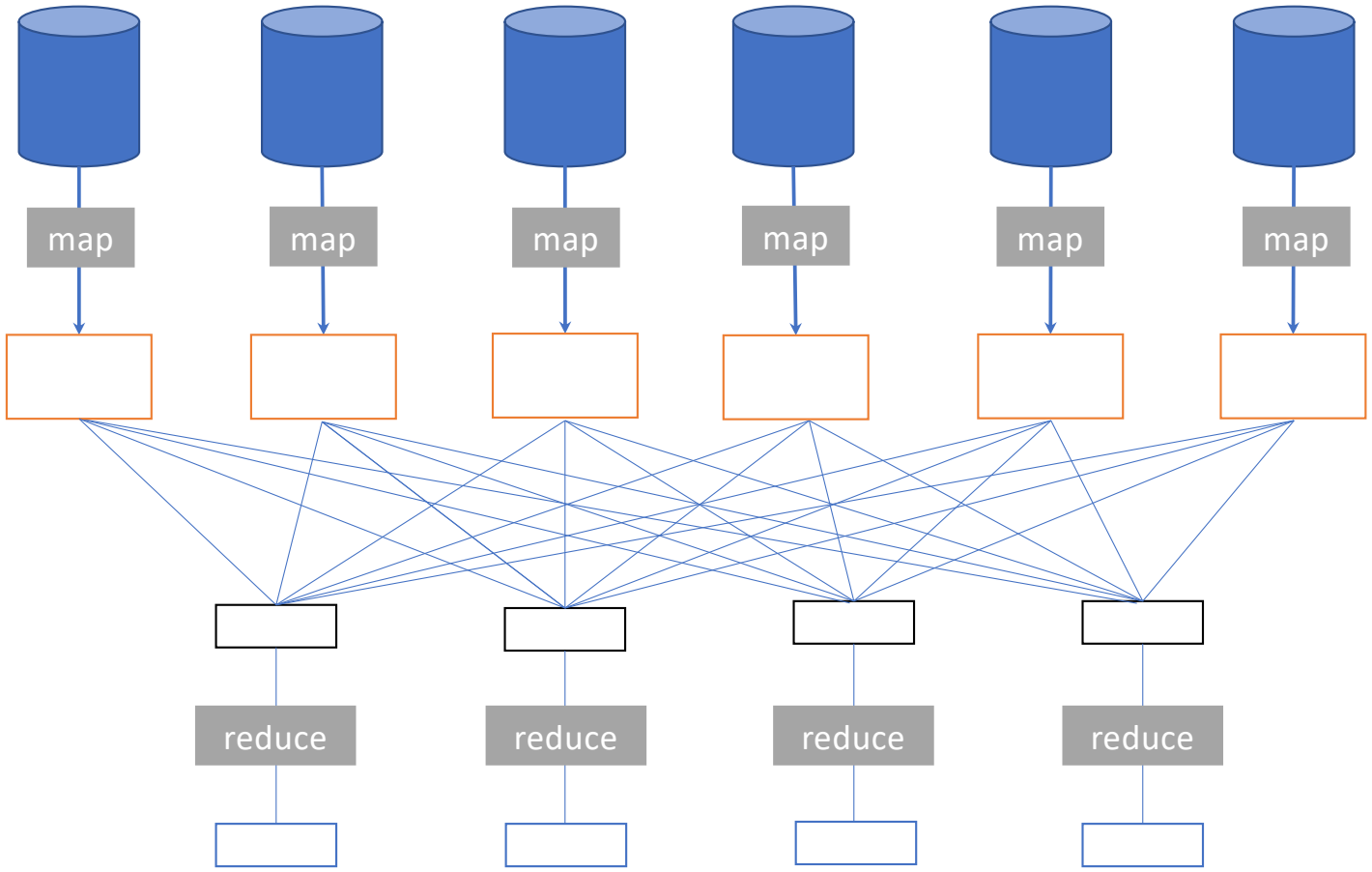
Distributed Data Storage

Map

(Shuffle)

Reduce

The Map Reduce Abstraction for Distributed Algorithms



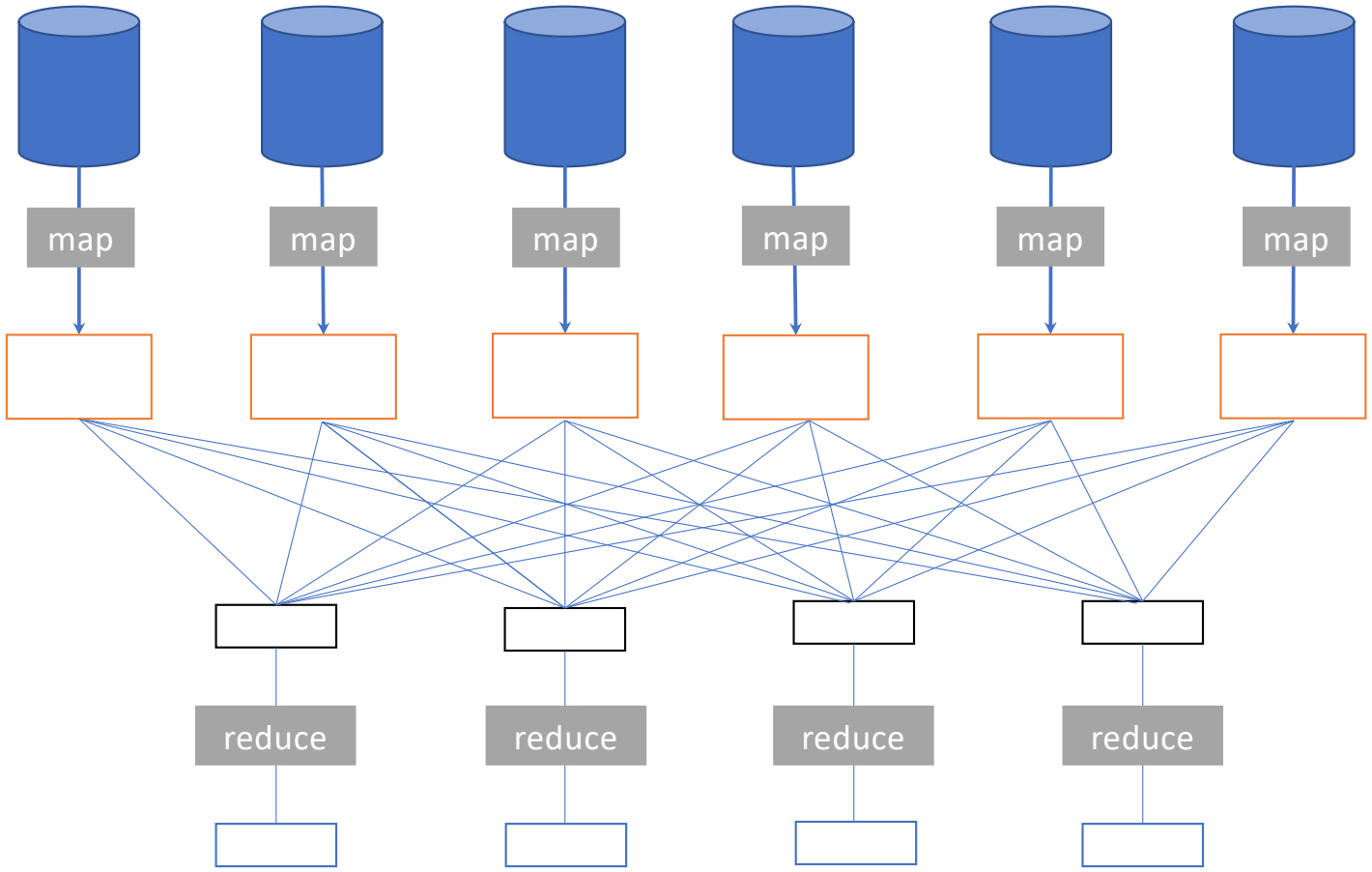
Distributed Data Storage

Map

(Shuffle)

Reduce

The Map Reduce Abstraction for Distributed Algorithms

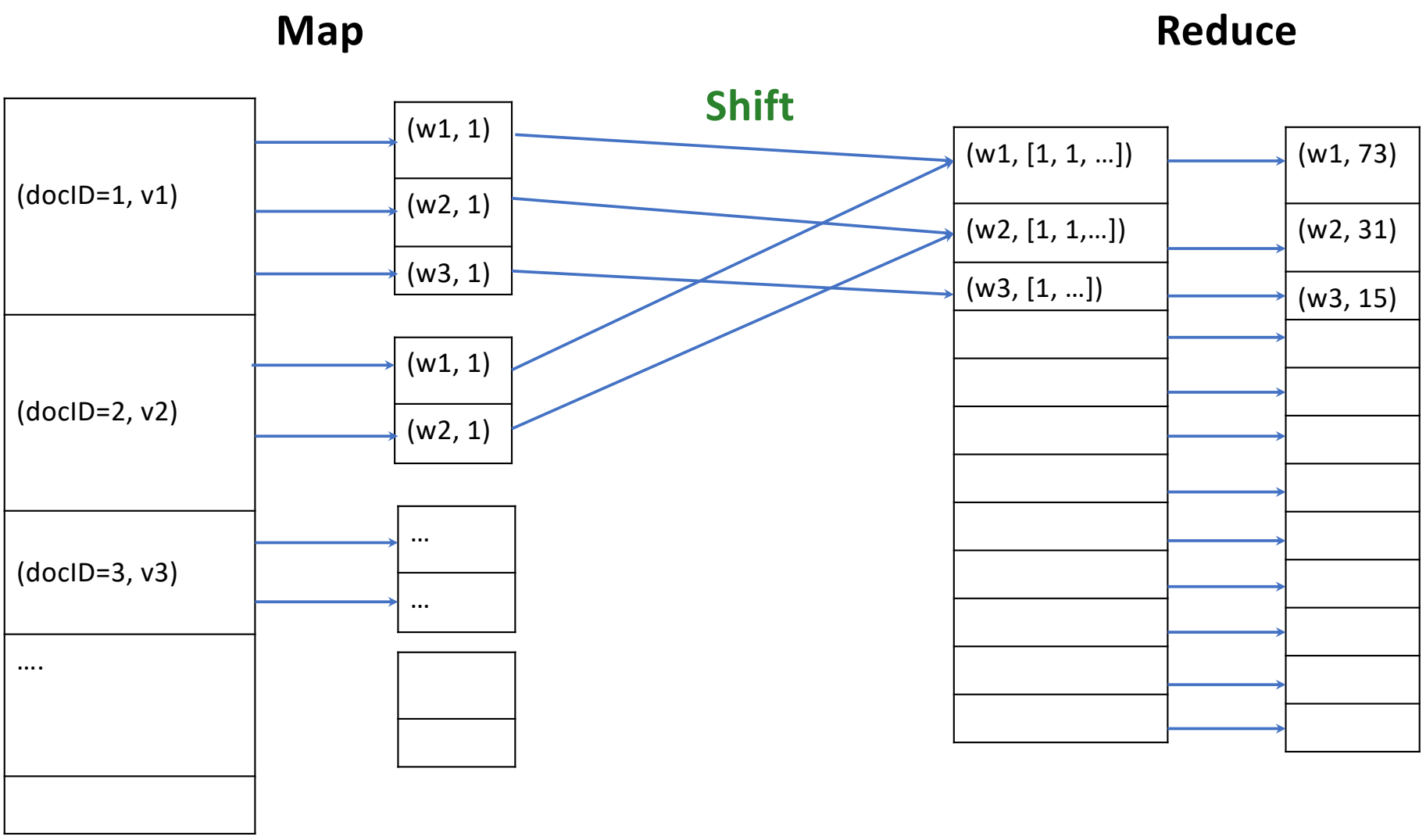


Distributed Data Storage

Map

(Shuffle)

Reduce


Map
Shift
Reduce
(docID=1, v1)
(docID=2, v2)
(docID=3, v3)
….
(w1, 1)
(w2, 1)
(w3, 1)
(w1, 1)
(w2, 1)
…
…
(w1, [1, 1, …])
(w2, [1, 1,…])
(w3, [1, …])
(w1, 73)
(w2, 31)
(w3, 15)

The Map Reduce Abstraction for Distributed Algorithms

- MapReduce is a high-level programming model and implementation for large-scale parallel data processing

- Like RDBMS adopt the the relational data model, MapReduce has a data model as well

MapReduce's Data Model

- Files!
- A File is a bag of **(key, value)** pairs
  - A bag is a **multiset**
- A map-reduce program:
  - Input: a bag of **(inputkey, value)** pairs
  - Output: a bag of **(outputkey, value)** pairs

2. The MapReduce Programming Model

User input

- All the user needs to define are the MAP and REDUCE functions
- Execute proceeds in multiple MAP – REDUCE rounds
  - MAP – REDUCE = MAP phase followed REDUCE

MAP Phase

Step 1: the MAP phase

- User provides a MAP-function:
  - Input: **(input key, value)**
  - Output: bag of **(intermediate key, value)**

- System applies the map function in parallel to all (input key, value) pairs in the input file

REDUCE Phase

Step 2: the REDUCE phase

- User provides a REDUCE-function:
  - Input: **(intermediate key, bag of values)**
  - Output: **(intermediate key, values)**

- The system will group all pairs with the same intermediate key, and passes the bag of values to the REDUCE function

MapReduce Programming Model

Input & Output: each a set of key/value pairs

Programmer specifies two functions:

map (in_key, in_value) -> list(out_key, intermediate_value)

- Processes input key/value pair
- Produces set of intermediate pairs

reduce (out_key, list(intermediate_value)) -> (out_key, list(out_values))

- Combines all intermediate values for a particular key
- Produces a set of merged output values (usually just one)

Example: what does the next program do?

```
map(String input_key, String input_value):
  //input_key: document id
  //input_value: document bag of words
  for each word w in input_value:
     EmitIntermediate(w, 1);

reduce(String intermediate_key, Iterator intermediate_values):
  //intermediate_key: word
  //intermediate_values: ????
  result = 0;
  for each v in intermediate_values:
    result += v;
  EmitFinal(intermediate_key, result);
```

Example: what does the next program do?

```
map(String input_key, String input_value):
  //input_key: document id
  //input_value: document bag of words
  word_count = {}
  for each word w in input_value:
     increase word_count[w] by one
  for each word w in word_count:
     EmitIntermediate(w, word_count[w]);

reduce(String intermediate_key, Iterator intermediate_values):
  //intermediate_key: word
  //intermediate_values: ????
  result = 0;
  for each v in intermediate_values:
    result += v;
  EmitFinal(intermediate_key, result);
```

3. MapReduce Examples

Word length histogram

How many big, medium, small, and tiny words are in a document?

Big = 10+ letters
Medium = 5..9 letters
Small = 2..4 letters
Tiny = 1 letter

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that

Word length histogram

Split the document into chunks and process each chunk on a different computer

Chunk 1

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Chunk 2

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that

Word length histogram

Map Chunk 1
(204words)

Output
(Big , 17)
(Medium, 77)
(Small, 107)
(Tiny, 3)

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Chunk 1

Chunk 2

Abstract

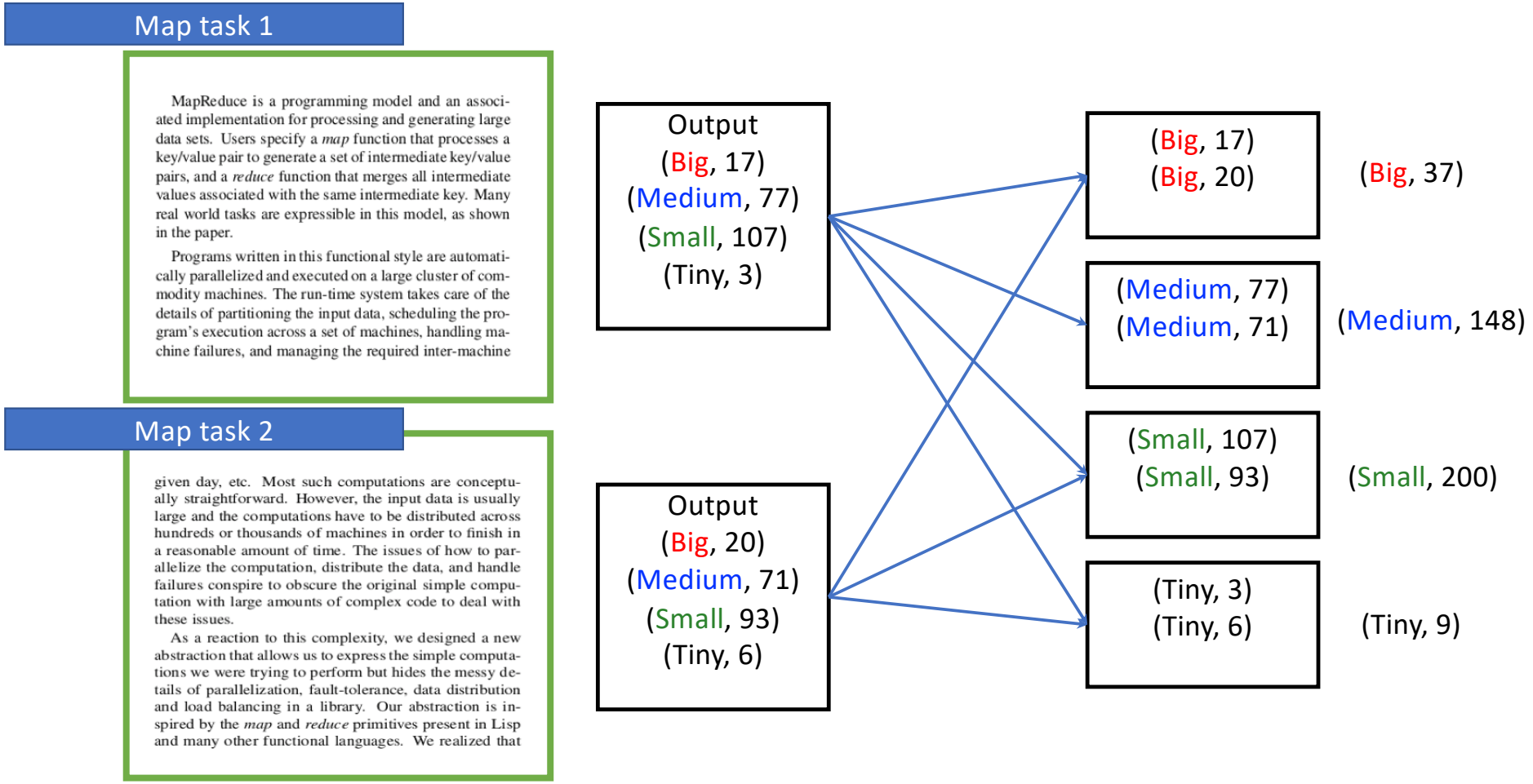
MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that

Word length histogram



Build an Inverted Index

Input:

doc1, (“I love medium roast coffee”)
doc2, (“I do not like coffee”)
doc3, (“This medium well steak is great”)
doc4, (“I love steak”)

Output:

“roast”, (doc1)
“coffee”, (doc1, doc2)
“medium”, (doc1, doc3)
“steak”, (doc3, do4)

Let’s design the solution!

Input:

doc1, (“I love medium roast coffee”)

doc2, (“I do not like coffee”)

doc3, (“This medium well steak is great”)

doc4, (“I love steak”)

Output:

“roast”, (doc1)

“coffee”, (doc1, doc2)

“medium”, (doc1, doc3)

“steak”, (doc3, do4)

2. More MapReduce Examples

Relational Join

Employee

Name	SSN
Sue	9999999999
Tony	7777777777

Assigned Departments

EmpSSN	DepName
9999999999	Accounts
7777777777	Sales
7777777777	Marketing

Employee $\bowtie_{SSN=EmpSSN}$ Assigned Departments

Name	SSN	EmpSSN	DepName
Sue	9999999999	9999999999	Accounts
Tony	7777777777	7777777777	Sales
Tony	7777777777	7777777777	Marketing

Relational Join

Employee

Name	SSN
Sue	9999999999
Tony	7777777777

Assigned Departments

EmpSSN	DepName
9999999999	Accounts
7777777777	Sales
7777777777	Marketing

Employee $\bowtie_{SSN=EmpSSN}$ Assigned Departments

Name	SSN	EmpSSN	DepName
Sue	9999999999	9999999999	Accounts
Tony	7777777777	7777777777	Sales
Tony	7777777777	7777777777	Marketing

Remember the semantics!

```
join_result = []
for e in Employee:
   for d in Assigned Departments:
      if e.SSN = d.EmpSSN
         r = <e.Name, e.SSN, d.EmpSSN, d.DepName>
         join_result.append(r)
rerun join_result
```

Relational Join

Employee

Name	SSN
Sue	9999999999
Tony	7777777777

Assigned Departments

EmpSSN	DepName
9999999999	Accounts
7777777777	Sales
7777777777	Marketing

Employee $\bowtie_{SSN=EmpSSN}$ Assigned Departments

Name	SSN	EmpSSN	DepName
Sue	9999999999	9999999999	Accounts
Tony	7777777777	7777777777	Sales
Tony	7777777777	7777777777	Marketing

Remember the semantics!

```
join_result = []
for e in Employee:
   for d in Assigned Departments:
      if e.SSN = d.EmpSSN
         r = <e.Name, e.SSN, d.EmpSSN, d.DepName>
         join_result.append(r)
rerun join_result
```

Imagine we have a huge number of records!
Let's use MapReduce!
We want the ***map*** phase to process each tuple.
Is there a problem?

Relational Join

Employee

Name	SSN
Sue	9999999999
Tony	7777777777

Assigned Departments

EmpSSN	DepName
9999999999	Accounts
7777777777	Sales
7777777777	Marketing

Employee $\bowtie_{SSN=EmpSSN}$ Assigned Departments

Name	SSN	EmpSSN	DepName
Sue	9999999999	9999999999	Accounts
Tony	7777777777	7777777777	Sales
Tony	7777777777	7777777777	Marketing

Remember the semantics!

```
join_result = []
for e in Employee:
   for d in Assigned Departments:
      if e.SSN = d.EmpSSN
         r = <e.Name, e.SSN, d.EmpSSN, d.DepName>
         join_result.append(r)
rerun join_result
```

Imagine we have a huge number of records!
Let's use MapReduce!
We want the ***map*** phase to process each tuple.
Is there a problem?

The **Relational Join** is a **binary** operation!
But **MapReduce** is a **unary** operation:
I operate on a single key

Can we approximate the join using MapReduce?

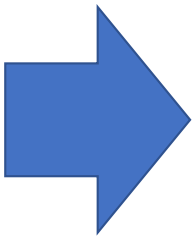
Relational Join in MapReduce: Preprocessing before the Map Phase

Employee

Name	SSN
Sue	9999999999
Tony	7777777777

Assigned Departments

EmpSSN	DepName
9999999999	Accounts
7777777777	Sales
7777777777	Marketing



Key idea: Flatten all tables and combine tuples from different tables in a single dataset

Employee	Sue	9999999999
Employee	Tony	7777777777
Assigned Departments	9999999999	Accounts
Assigned Departments	7777777777	Sales
Assigned Departments	7777777777	Marketing

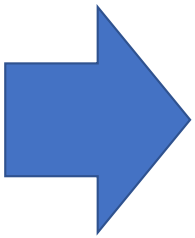
Relational Join in MapReduce: Preprocessing before the Map Phase

Employee

Name	SSN
Sue	9999999999
Tony	7777777777

Assigned Departments

EmpSSN	DepName
9999999999	Accounts
7777777777	Sales
7777777777	Marketing



Key idea: Flatten all tables and combine tuples from different tables in a single dataset

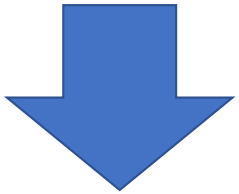
Employee	Sue	9999999999
Employee	Tony	7777777777
Assigned Departments	9999999999	Accounts
Assigned Departments	7777777777	Sales
Assigned Departments	7777777777	Marketing

We use the table name to keep track of "which table did the tuple come from"

This is a label that we've attached to every tuple so that we can know where that came from. We'll use it later!

Relational Join in MapReduce: Map Phase

Employee	Sue	9999999999
Employee	Tony	7777777777
Assigned Departments	9999999999	Accounts
Assigned Departments	7777777777	Sales
Assigned Departments	7777777777	Marketing



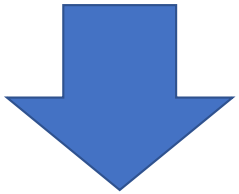
For each tuple in the flattened input we will generate a key value pair!

key=9999999999, value=(Employee, Sue, 9999999999)
key=7777777777, value=(Employee, Tony, 7777777777)
key=9999999999, value=(Assigned Departments, 9999999999, Accounts)
key=7777777777, value=(Assigned Departments, 7777777777, Sales)
key=7777777777, value=(Assigned Departments, 7777777777, Marketing)

Relational Join in MapReduce: Map Phase

Employee	Sue	9999999999
Employee	Tony	7777777777
Assigned Departments	9999999999	Accounts
Assigned Departments	7777777777	Sales
Assigned Departments	7777777777	Marketing

Why use this value as the key?

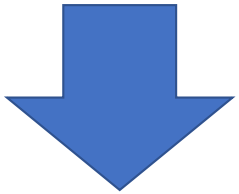


For each tuple in the flattened input we will generate a key value pair!

key=9999999999, value=(Employee, Sue, 9999999999)
key=7777777777, value=(Employee, Tony, 7777777777)
key=9999999999, value=(Assigned Departments, 9999999999, Accounts)
key=7777777777, value=(Assigned Departments, 7777777777, Sales)
key=7777777777, value=(Assigned Departments, 7777777777, Marketing)

Relational Join in MapReduce: Map Phase

Employee	Sue	9999999999
Employee	Tony	7777777777
Assigned Departments	9999999999	Accounts
Assigned Departments	7777777777	Sales
Assigned Departments	7777777777	Marketing



Why use this value as the key?
We are joining on SSN (for Employee)
and EmpSSN (for Assigned Depts)

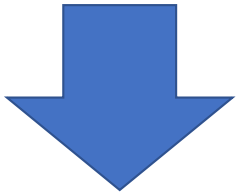
For each tuple in the flattened input we will generate a key value pair!

key=9999999999, value=(Employee, Sue, 9999999999)
key=7777777777, value=(Employee, Tony, 7777777777)
key=9999999999, value=(Assigned Departments, 9999999999, Accounts)
key=7777777777, value=(Assigned Departments, 7777777777, Sales)
key=7777777777, value=(Assigned Departments, 7777777777, Marketing)

Relational Join in MapReduce: Map Phase (Two Tricks so far)

Employee	Sue	9999999999
Employee	Tony	7777777777
Assigned Departments	9999999999	Accounts
Assigned Departments	7777777777	Sales
Assigned Departments	7777777777	Marketing

Trick 1: Flattened and combined tables in a single input file.



Why use this value as the key?
We are joining on SSN (for Employee)
and EmpSSN (for Assigned Depts)

For each tuple in the flattened input we will generate a key value pair!

Trick 2: Produce a key value pair where the key is the join attribute.

key=9999999999, value=(Employee, Sue, 9999999999)
key=7777777777, value=(Employee, Tony, 7777777777)
key=9999999999, value=(Assigned Departments, 9999999999, Accounts)
key=7777777777, value=(Assigned Departments, 7777777777, Sales)
key=7777777777, value=(Assigned Departments, 7777777777, Marketing)

Relational Join in MapReduce: Reduce Phase (after the magic Shuffle)

Input to Reducer 1

key=9999999999, value=[(Employee, Sue, 9999999999),
(Assigned Departments, 9999999999, Accounts)]

Input to Reducer 2

key=7777777777, value=[(Employee, Tony, 7777777777),
(Assigned Departments, 7777777777, Sales),
(Assigned Departments, 7777777777, Marketing)]

After the shuffle phase all inputs with the same key will end up in the same reducer!

It does not matter which relation the different tuples came from!

Relational Join in MapReduce: Reduce Phase (after the magic Shuffle)

Input to Reducer 1

key=9999999999, value=[(Employee, Sue, 9999999999),
(Assigned Departments, 9999999999, Accounts)]

Input to Reducer 2

key=7777777777, value=[(Employee, Tony, 7777777777),
(Assigned Departments, 7777777777, Sales),
(Assigned Departments, 7777777777, Marketing)]

We have all the information we need to perform the join for a each key in a single machine.

This is how we scale.

Relational Join in MapReduce: Reduce Phase (after the magic Shuffle)

Input to Reducer 1

key=9999999999, value=[(Employee, Sue, 9999999999),
(Assigned Departments, 9999999999, Accounts)]

Input to Reducer 2

key=7777777777, value=[(Employee, Tony, 7777777777),
(Assigned Departments, 7777777777, Sales),
(Assigned Departments, 7777777777, Marketing)]

Desired output of reduce function

Output of Reduce Function (Reducer 1)

Sue, 9999999999, 9999999999, Accounts

Output of Reduce Function (Reducer 2)

Tony, 7777777777, 7777777777, Sales
Tony, 7777777777, 7777777777, Marketing

Relational Join in MapReduce: Reduce Phase (after the magic Shuffle)

Input to Reducer 1

key=9999999999, value=[(Employee, Sue, 9999999999),
(Assigned Departments, 9999999999, Accounts)]

Input to Reducer 2

key=7777777777, value=[(Employee, Tony, 7777777777),
(Assigned Departments, 7777777777, Sales),
(Assigned Departments, 7777777777, Marketing)]

Desired output of reduce function

Output of Reduce Function (Reducer 1)

Sue, 9999999999, 9999999999, Accounts

Output of Reduce Function (Reducer 2)

Tony, 7777777777, 7777777777, Sales
Tony, 7777777777, 7777777777, Marketing

This part came from the Employees table
This part came from the Assigned Departments table

What is the reduce function implementation?

Relational Join in MapReduce: Reduce Phase Implementation

Input to Reducer 1

key=9999999999, value=[(Employee, Sue, 9999999999),
(Assigned Departments, 9999999999, Accounts)]

Input to Reducer 2

key=7777777777, value=[(Employee, Tony, 7777777777),
(Assigned Departments, 7777777777, Sales),
(Assigned Departments, 7777777777, Marketing)]

Desired output of reduce function

Output of Reduce Function (Reducer 1)

Sue, 9999999999, 9999999999, Accounts

Output of Reduce Function (Reducer 2)

Tony, 7777777777, 7777777777, Sales
Tony, 7777777777, 7777777777, Marketing

Simple Pseudo-code for Reduce

```
reduce(String key, Iterator tuples):
 //intermediate_key: join key
 //intermediate_values: tuples with the same join key
 join_result = [];
 for t1 in tuples:
    for t2 in tuples:
       if t1[0] <> t2[0]:
          output tuple = (t1[1:], t2[1:])
          join_result.append(t)
rerun (key, join_result)
```

This is a cross-product operation!
Relational algebra is everywhere!
Notice that we need to keep track of where each tuple came from.

Social Network Analysis: Count Friends

Input

Jim	Sue
Sue	Jim
Lin	Joe
Joe	Lin
Jim	Kai
Kai	Jim
Jim	Lin
Lin	Jim

Symmetric friendship edges

Desired Output

Jim, 3
Lin, 2
Sue, 1
Kai, 1
Joe, 1

Social Network Analysis: Count Friends

Input

Jim	Sue
Sue	Jim
Lin	Joe
Joe	Lin
Jim	Kai
Kai	Jim
Jim	Lin
Lin	Jim

Symmetric friendship edges

MAP

key, value
Jim, 1
Sue, 1
Lin, 1
Joe, 1
Jim, 1
Kai, 1
Jim, 1
Lin, 1

Emit one for each left-hand value

SHUFFLE

Jim, (1, 1, 1)
Sue, 1
Lin, (1,1)
Joe, 1
Kai, 1

REDUCE

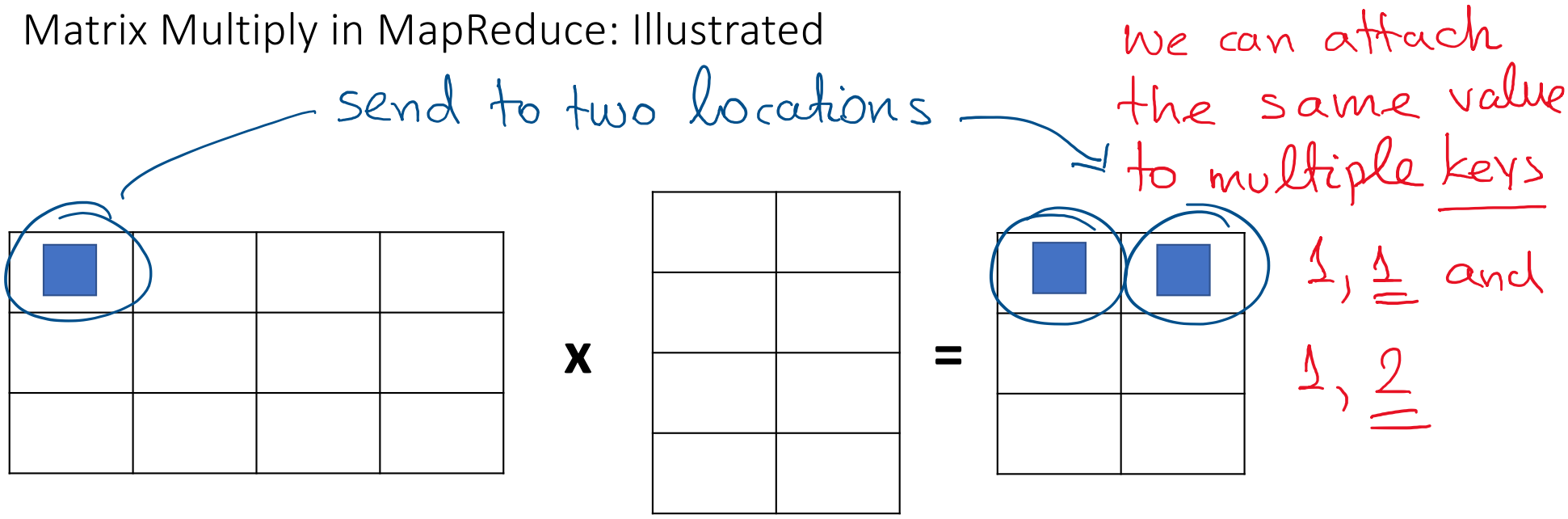
Desired Output

Jim, 3
Lin, 2
Sue, 1
Kai, 1
Joe, 1

Matrix Multiply in MapReduce

- C = A x B
- A dimensions m x n, B dimensions n x l
- In the map phase:
  - for each element (i,j) of A, emit ((i,k),A[i,j]) for k in **1..l**
    - **Key = (i,k) and value = A[i,j]**
  - for each element (i,j) of B, emit ((i,k),B[i,j]) for k in **1..m**
    - **Key = (i,k) and value = B[i,j]**
- In the reduce phase, emit
  - key = (i,k)
  - Value = Sum_j (A[i,j] * B[j,k])

Matrix Multiply in MapReduce: Illustrated

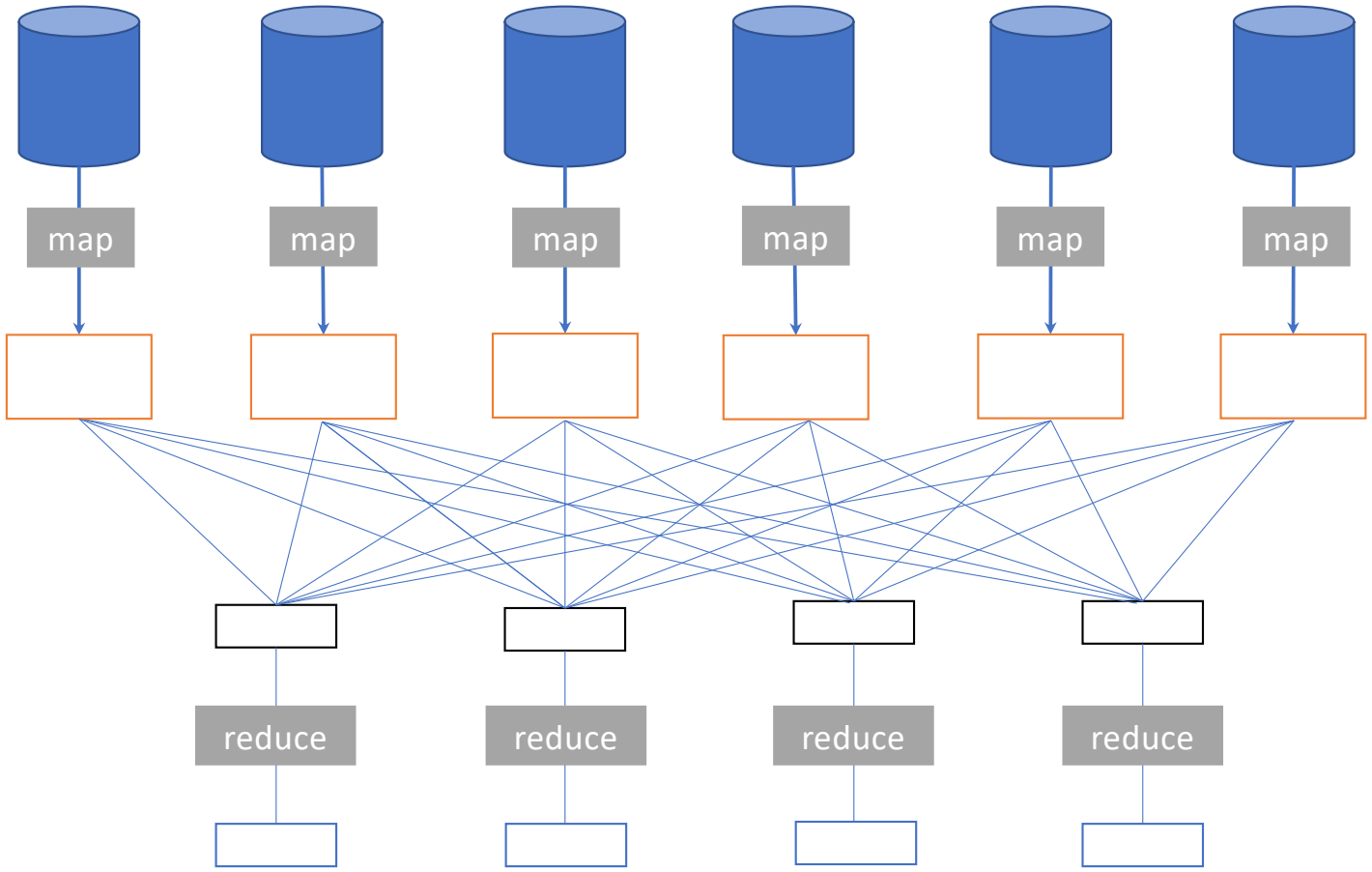


We have one reducer per output cell

Each reducer computes Sum_j (A[i,j] * B[j,k])

1. MapReduce Implementation

Recall: The Map Reduce Abstraction for Distributed Algorithms



Distributed Data Storage

Map

(Shuffle)

Reduce

MapReduce: what happens in between?

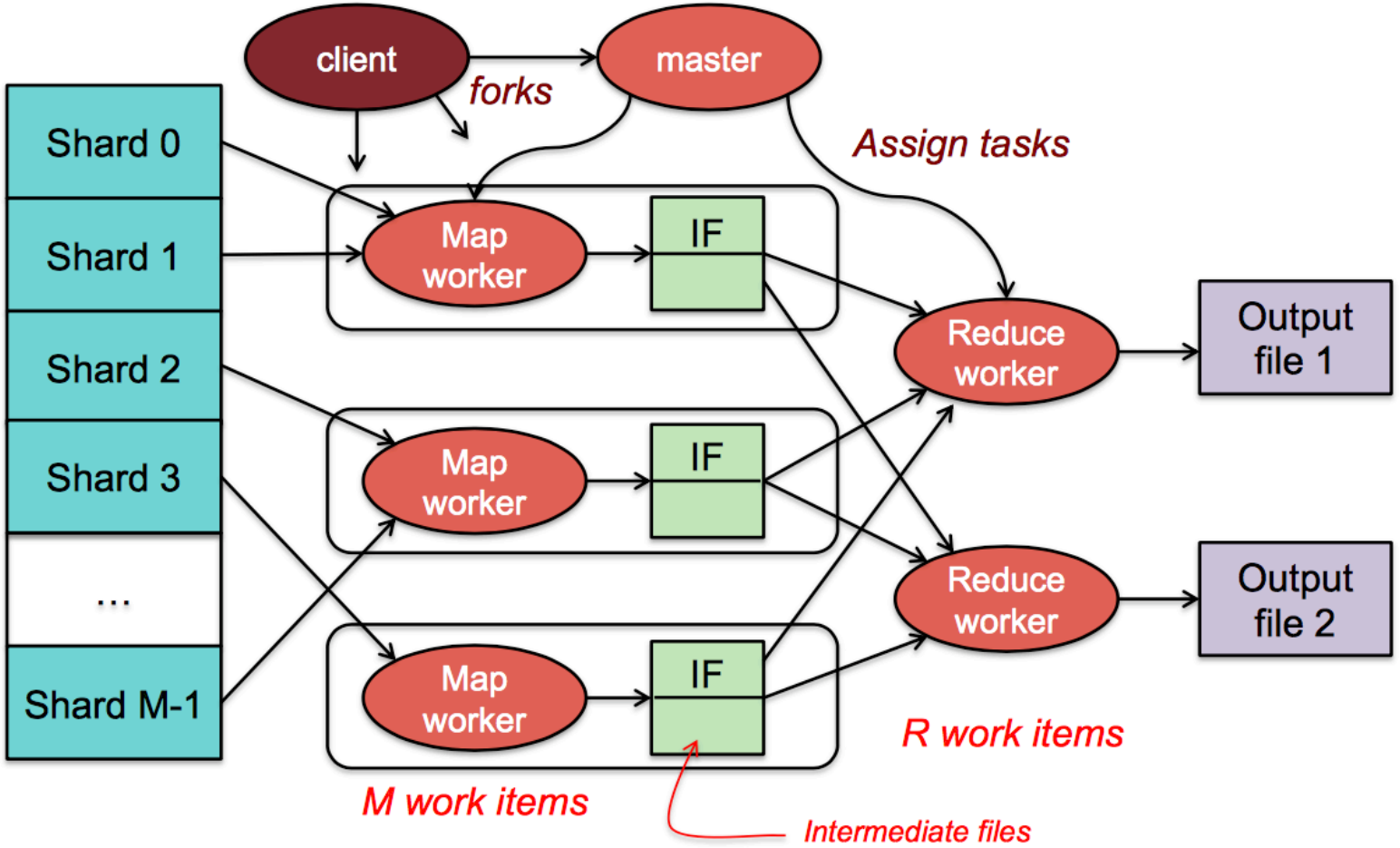
Map Worker

- **Map**
  - Grab the relevant data from the source (parse into key, value)
  - Write it to an intermediate file
- **Partition**
  - Partitioning: identify which of *R* reducers will handle which keys
  - Map partitions data to target it to one of *R* Reduce workers based on a partitioning function (both *R* and partitioning function user defined)

Reduce Worker

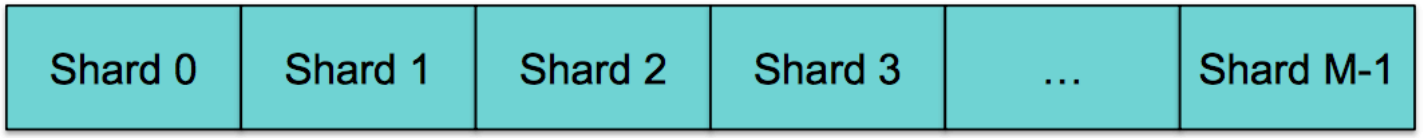
- **Shuffle & Sort**
  - Shuffle: Fetch the relevant partition of the output from all mappers
  - Sort by keys (different mappers may have sent data with the same key)
- **Reduce**
  - Input is the sorted output of mappers
  - Call the user *Reduce* function per key with the list of values for that key to aggregate the results

MapReduce: the complete picture



Step 1: Split input files into chunks (shards)

- Break up the input data into *M* pieces (typically 64 MB)

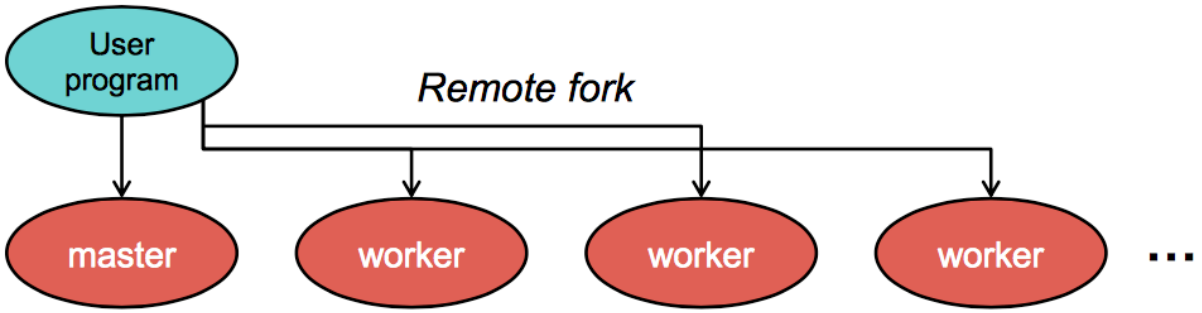


Input files

Divided into *M* shards

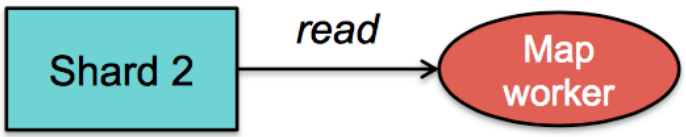
Step 2: Fork processes

- Start up many copies of the program on a cluster of machines
  - **One master**: scheduler & coordinator
  - Lots of workers
- Idle workers are assigned either:
  - map tasks (each works on a shard) – there are *M* map tasks
  - reduce tasks (each works on intermediate files) – there are *R*
    - *R* = # partitions, defined by the user



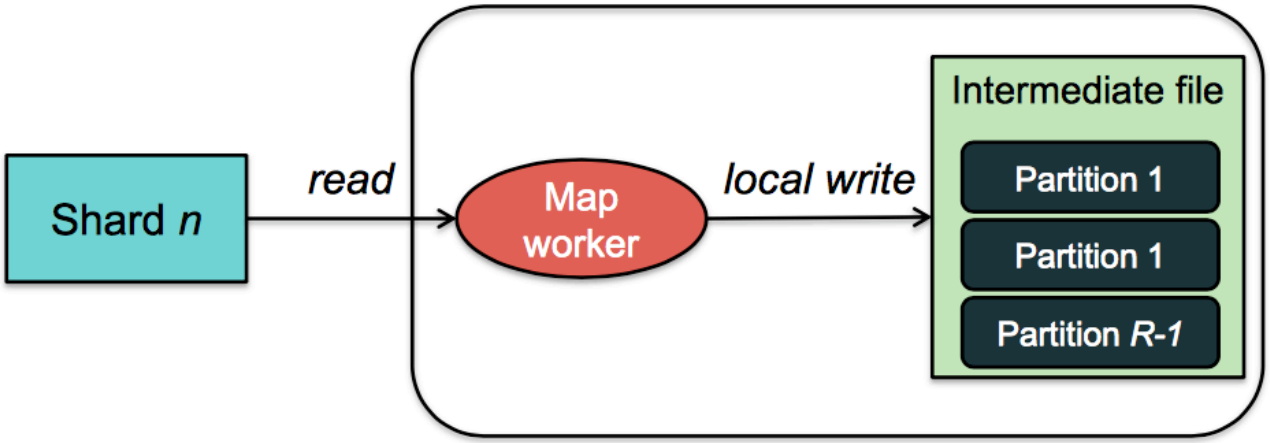
Step 3: Run Map Tasks

- Reads contents of the input shard assigned to it
- Parses key/value pairs out of the input data
- Passes each pair to a user-defined *map* function
  - Produces intermediate key/value pairs
  - These are buffered in memory



Step 4: Create intermediate files

- Intermediate key/value pairs produced by the user's *map* function buffered in memory and are periodically written to the local disk
  - Partitioned into *R* regions by a partitioning function

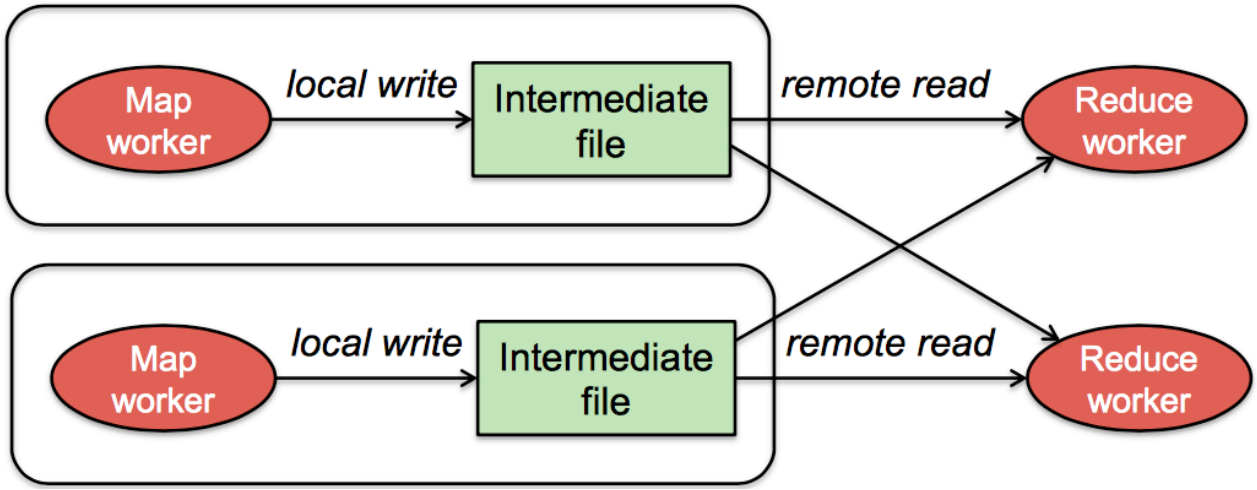


Step 4a: Partitioning

- Map data will be processed by Reduce workers
  - User's *Reduce* function will be called once per unique key generated by *Map*.
- We first need to sort all the (*key, value*) data by keys and decide which Reduce worker processes which keys
  - The Reduce worker will do the sorting
- **Partition function**
  Decides which of *R* reduce workers will work on which key
  - Default function: *hash(key) mod R*
  - Map worker partitions the data by keys
- Each Reduce worker will later read their partition from every Map worker

Step 5: Reduce Task - sorting

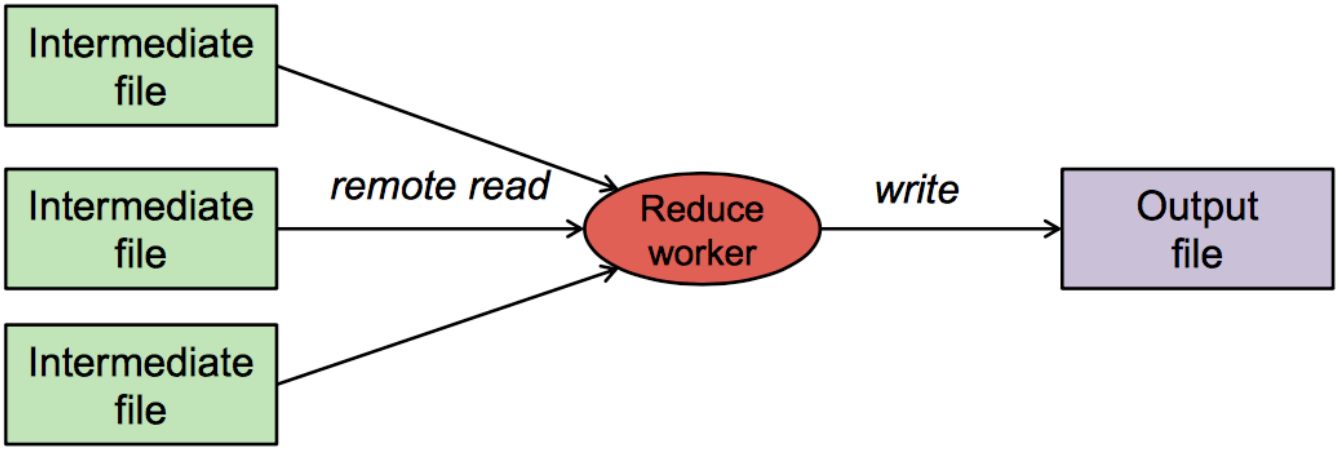
- Reduce worker gets notified by the master about the location of intermediate files for its partition
- Shuffle: Uses RPCs to read the data from the local disks of the map workers
- Sort: When the *reduce* worker reads intermediate data for its partition
  - It sorts the data by the intermediate keys
  - All occurrences of the same key are grouped together



Step 6: Reduce Task - reduce

- The sort phase grouped data with a unique intermediate key
- User's ***Reduce*** function is given the key and the set of intermediate values for that key

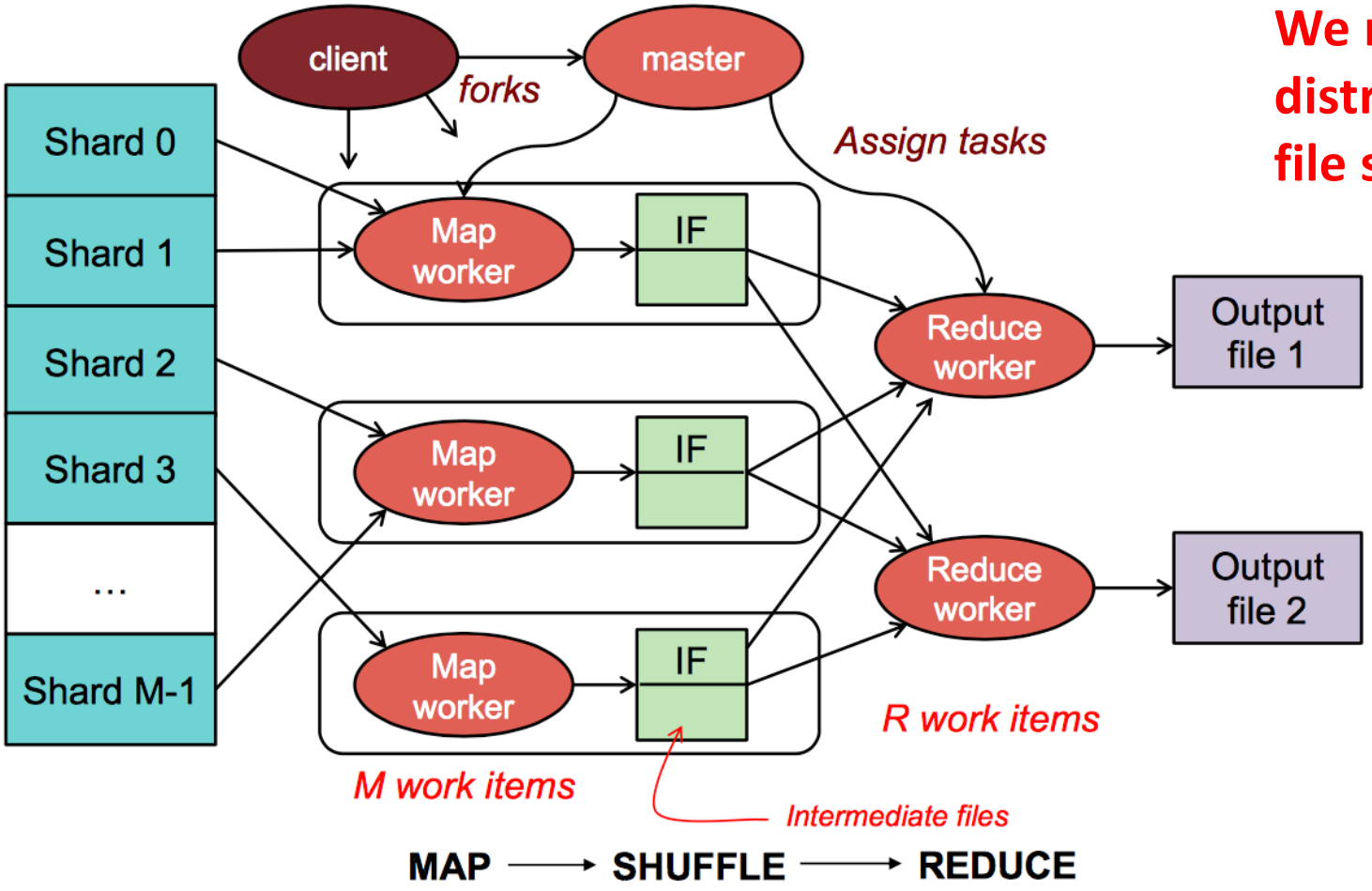
  < key, (value1, value2, value3, value4, …) >
- The output of the *Reduce* function is appended to an output file



Step 7: Return to user

- When all *map* and *reduce* tasks have completed, the master wakes up the user program
- The *MapReduce* call in the user program returns and the program can resume execution.
  - Output of *MapReduce* is available in *R* output files

MapReduce: the complete picture



We need a distributed file system!

2. Spark

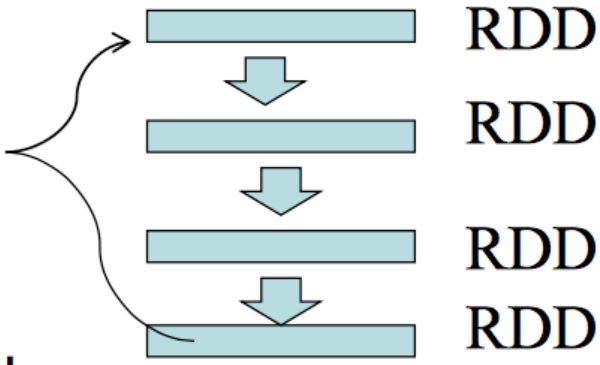
Intro to Spark

- Spark is really a different implementation of the MapReduce programming model
- What makes Spark different is that it operates on Main Memory
- Spark: we write programs in terms of operations on resilient distributed datasets (RDDs).
- RDD (simple view): a collection of elements partitioned across the nodes of a cluster that can be operated on in parallel.
- RDD (complex view): RDD is an interface for data transformation, RDD refers to the data stored either in persisted store (HDFS) or in cache (memory, memory+disk, disk only) or in another RDD

RDDs in Spark

RDD: Resilient Distributed Datasets

- **Like a big list:**
  - Collections of objects spread across a cluster, stored in RAM or on Disk
- **Built through parallel transformations**
- **Automatically rebuilt on failure**

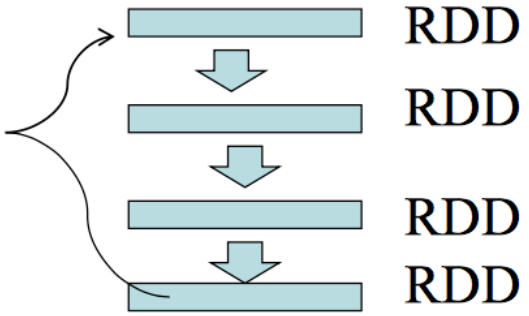


Operations

- **Transformations (e.g. map, filter, groupBy)**
- **Make sure input/output match**

MapReduce vs Spark

Map and reduce tasks operate on key-value pairs

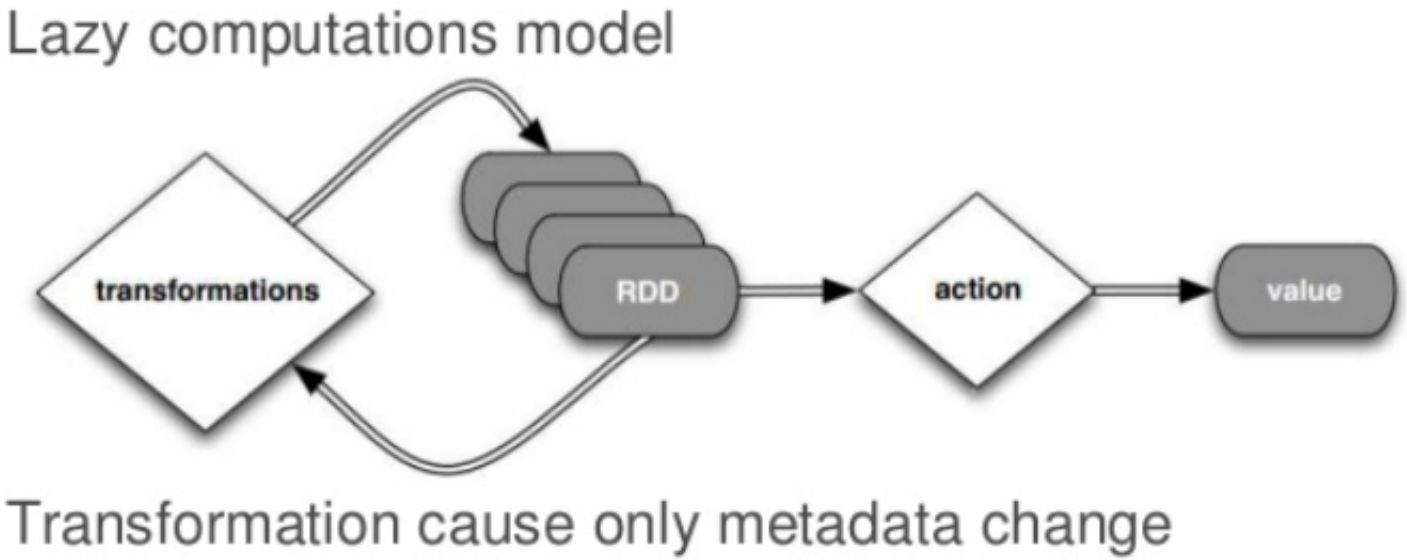


Spark operates on **RDD**

RDDs

- Partitions are recomputed on failure or cache eviction
- Metadata stored for interface:
  - Partitions – set of data splits associated with this RDD
  - Dependencies – list of parent RDDs involved in computation
  - Compute – function to compute partition of the RDD given the parent partitions from the Dependencies
  - Preferred Locations – where is the best place to put computations on this partition (data locality)
  - Partitioner – how the data is split into partitions

RDDs

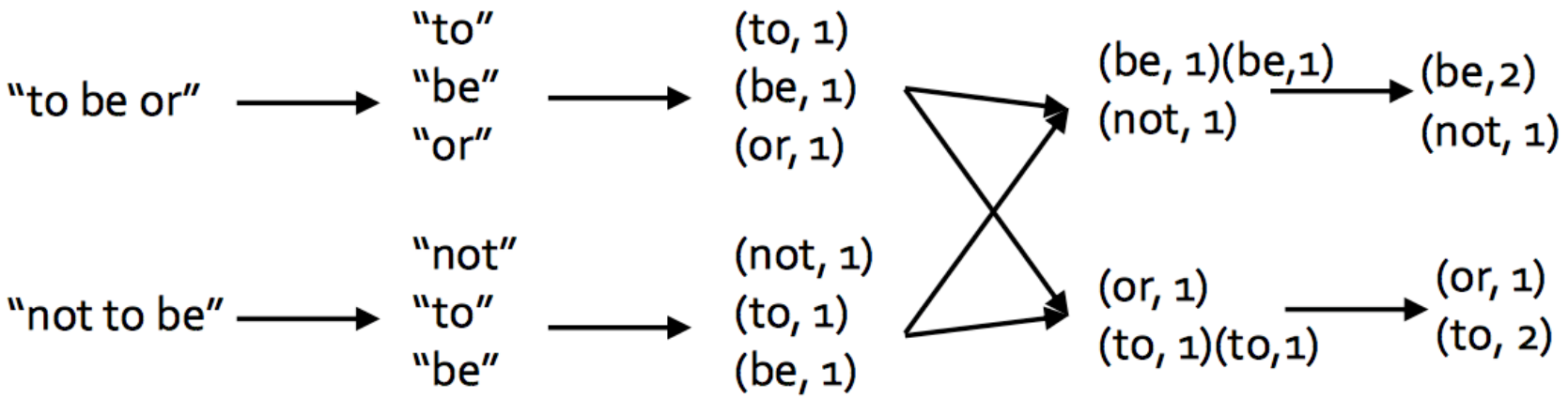

Lazy computations model
transformations
RDD
action
value
Transformation cause only metadata change

DAG

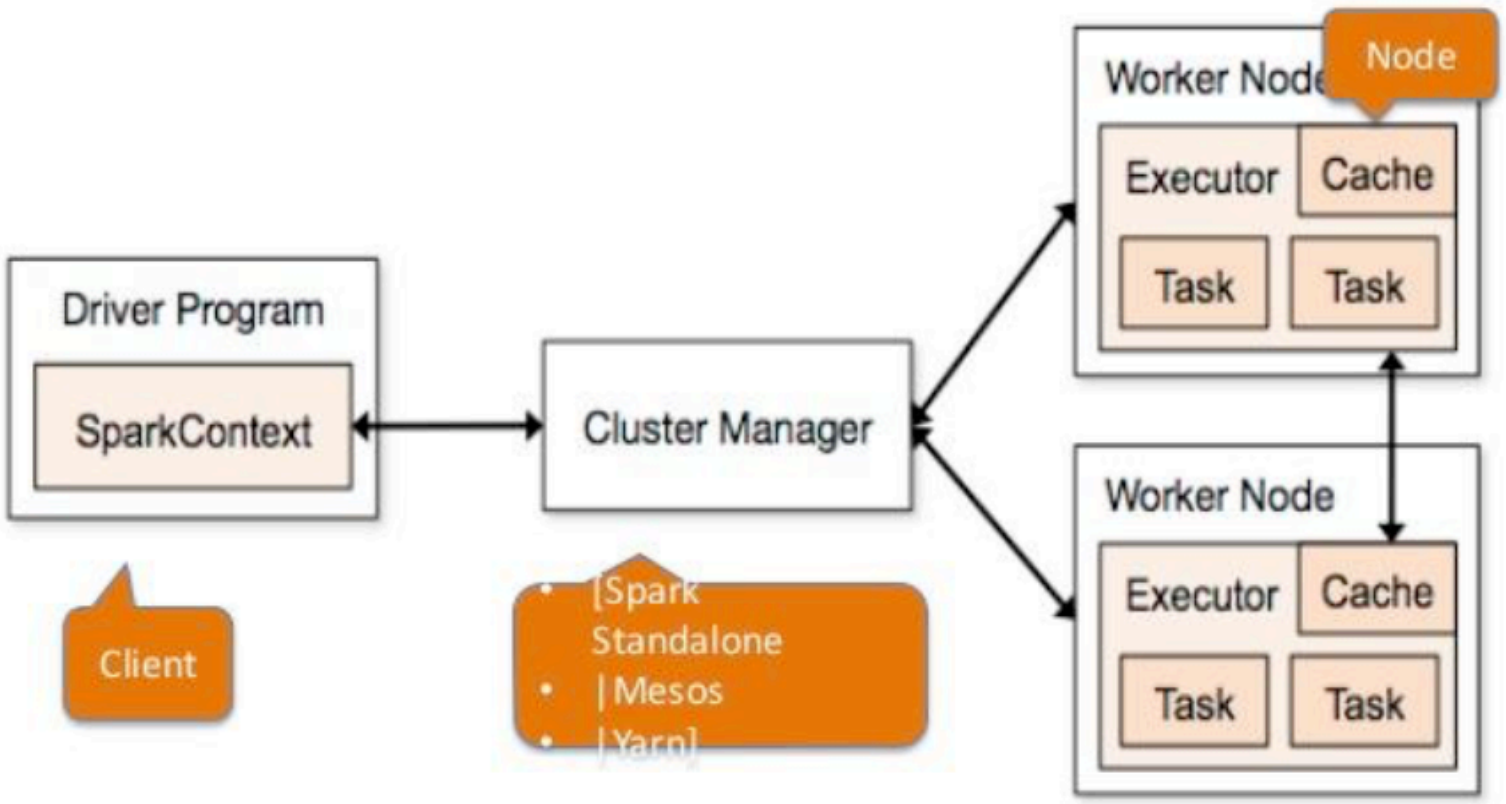
- Directed Acyclic Graph – sequence of computations performed on data
- Node – RDD partition
- Edge – transformation on top of the data
- Acyclic – graph cannot return to the older partition
- Directed – transformation is an action that transitions data partitions state (from A to B)

Example: Word Count

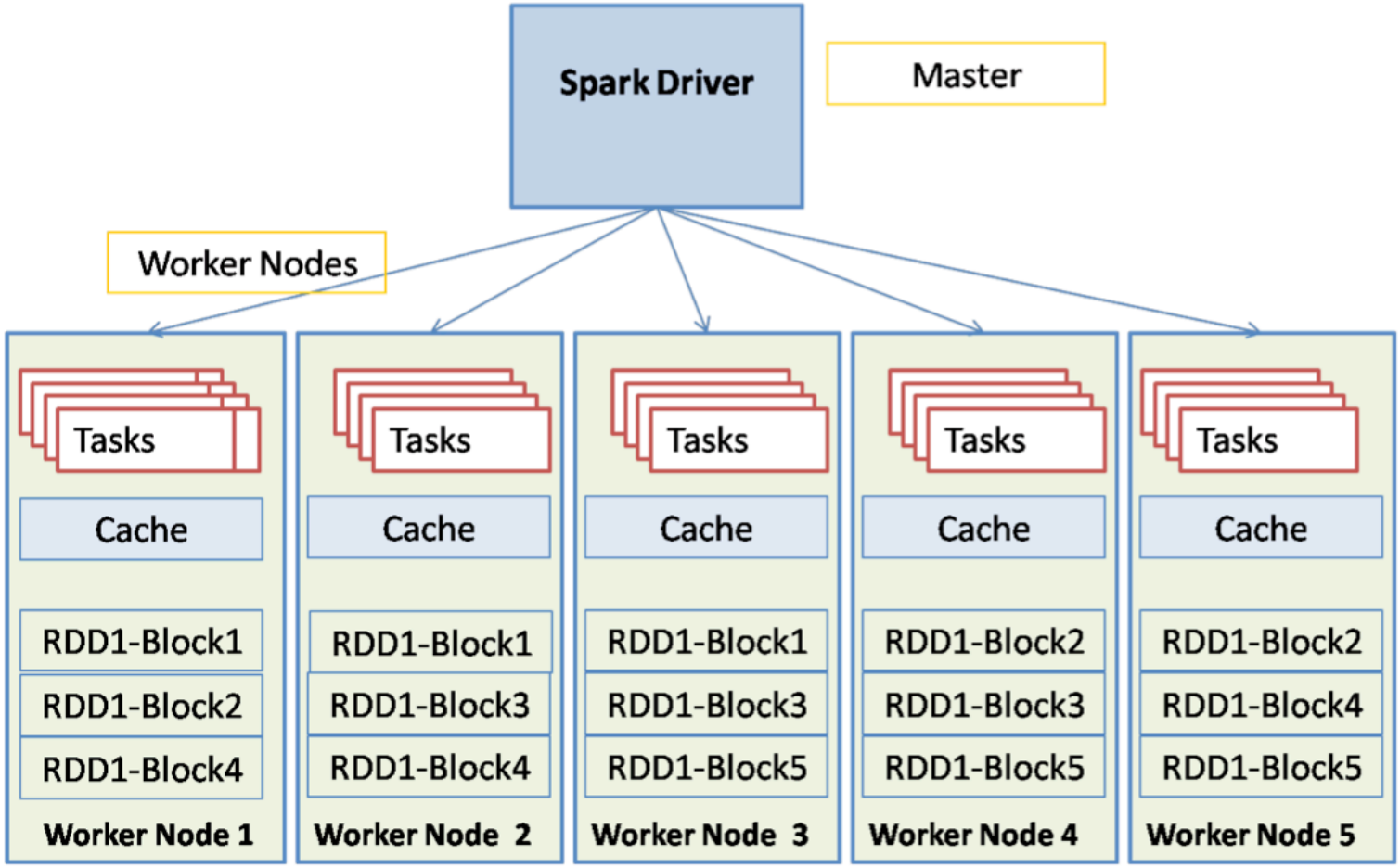
```
> lines = sc.textFile("hamlet.txt")
> counts = lines.flatMap(lambda line: line.split(" "))
                .map(lambda word: (word, 1))
                .reduceByKey(lambda x, y: x + y)
```



Spark Architecture



Spark Components



Spark Driver

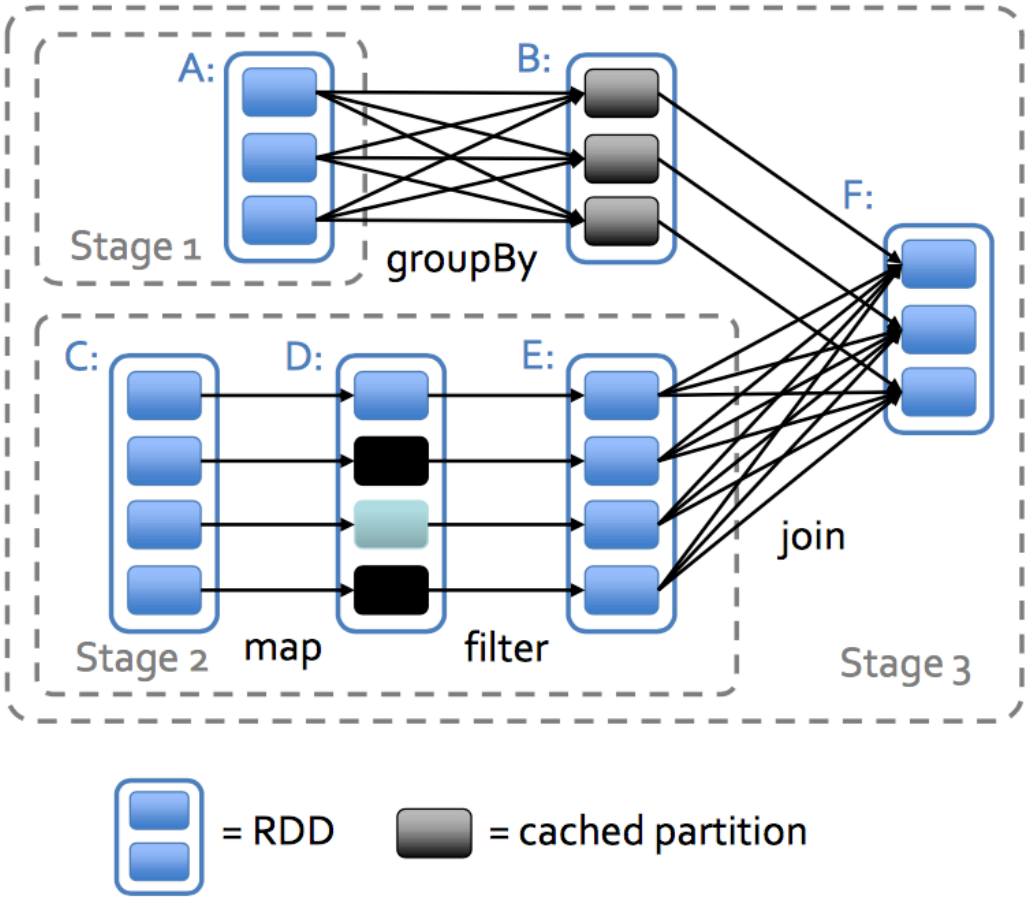
- Entry point of the Spark Shell (Scala, Python, R)
- The place where SparkContext is created
- Translates RDD into the execution graph
- Splits graph into stages
- Schedules tasks and controls their execution
- Stores metadata about all the RDDs and their partitions
- Brings up Spark WebUI with job information

Spark Executor

- Stores the data in cache in JVM heap or on HDDs
- Reads data from external sources
- Writes data to external sources
- Performs all the data processing

Dag Scheduler

- **General task graphs**
- **Automatically pipelines functions**
- **Data locality aware**
- **Partitioning aware to avoid shuffles**



More RDD Operations

- map
- filter
- groupBy
- sort
- union
- join
- leftOuterJoin
- rightOuterJoin
- reduce
- count
- fold
- reduceByKey
- groupByKey
- cogroup
- cross
- zip

sample

take

first

partitionBy

mapWith

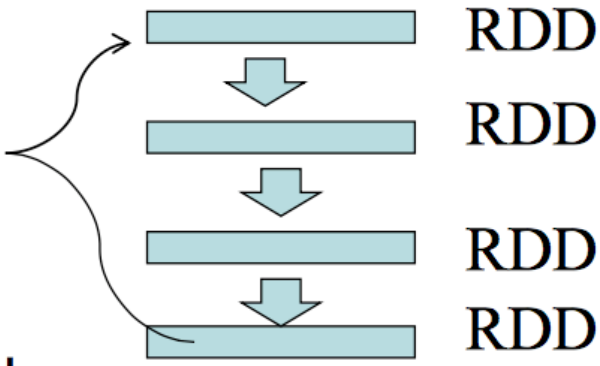
pipe

save ...

Spark's secret is really the RDD abstraction

RDD: Resilient Distributed Datasets

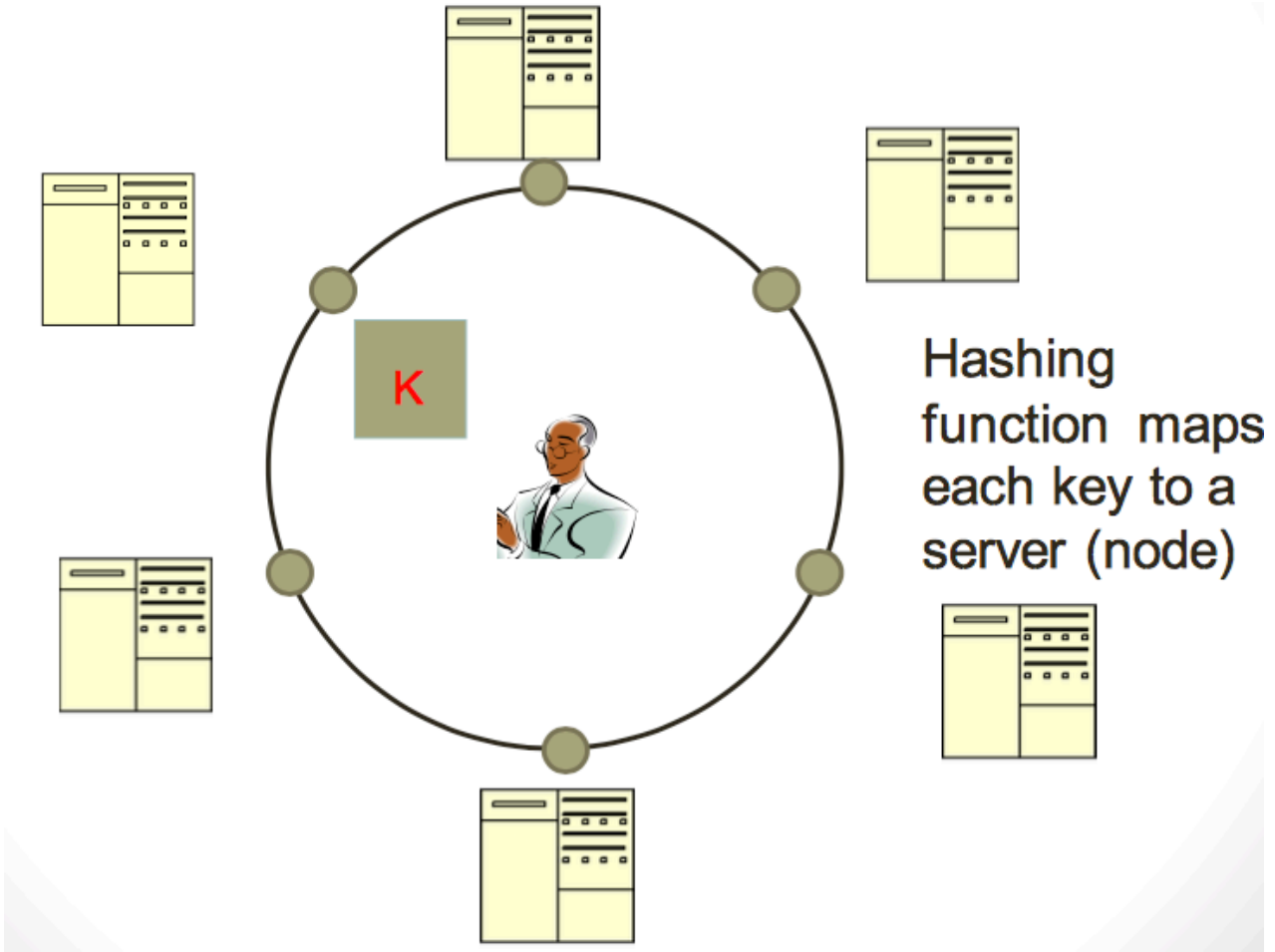
- **Like a big list:**
  - Collections of objects spread across a cluster, stored in RAM or on Disk
- **Built through parallel transformations**
- **Automatically rebuilt on failure**



Operations

- **Transformations (e.g. map, filter, groupBy)**
- **Make sure input/output match**

Typical NoSQL architecture



CAP theorem for NoSQL

- What the CAP theorem really says: If you cannot limit the number of faults and requests can be directed to any server and you insist on serving every request you receive then you cannot possibly be consistent

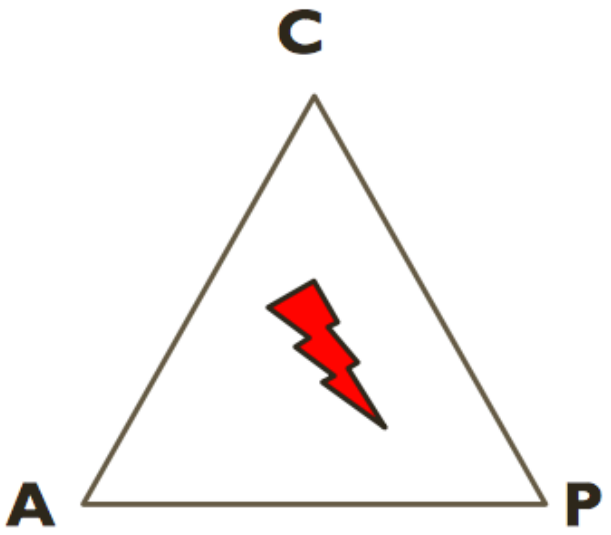
- How it is interpreted: You must always give something up: consistency, availability or tolerance to failure and reconfiguration

CAP theorem for NoSQL

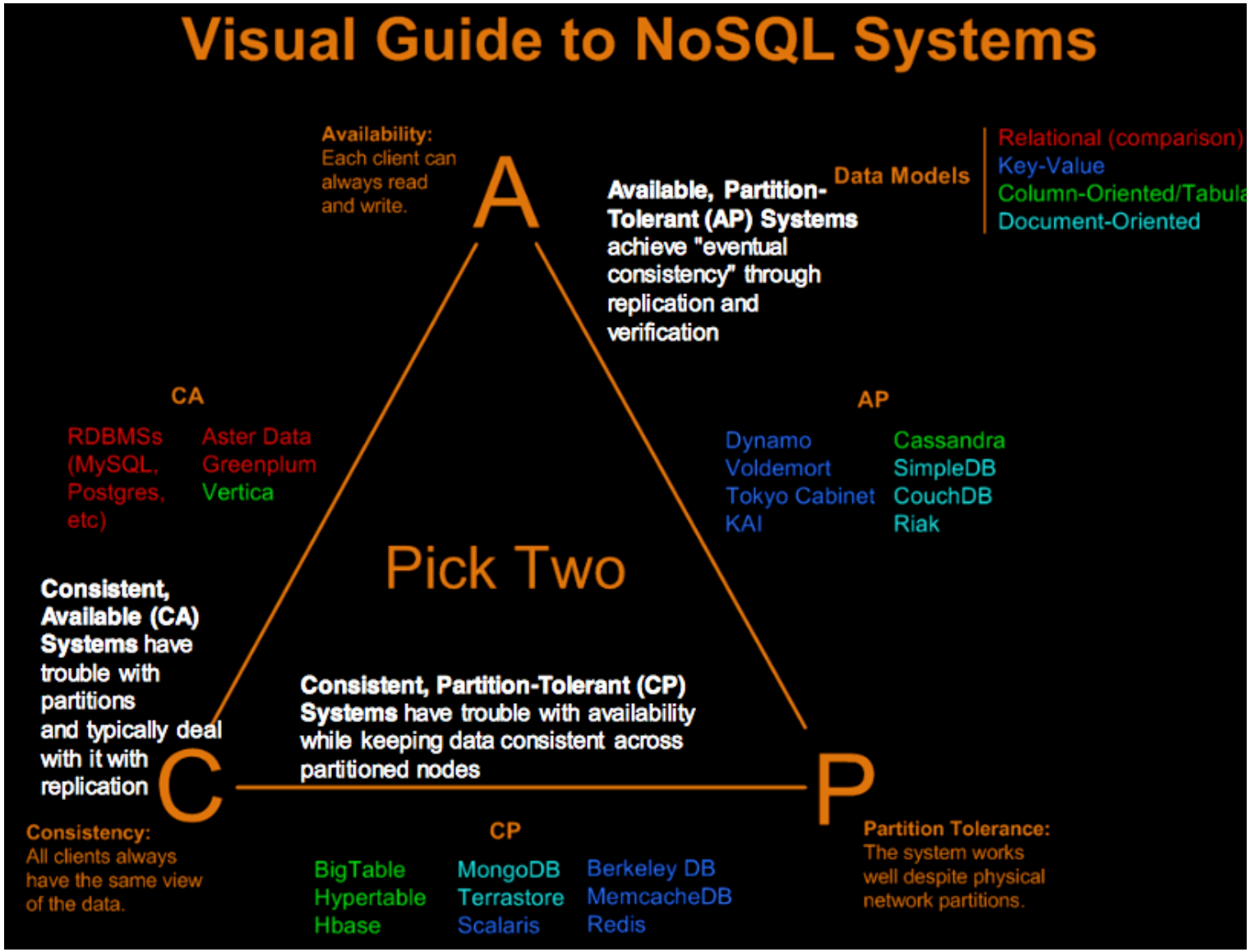
GIVEN:

- Many nodes
- Nodes contain ***replicas of partitions*** of the data

- **Consistency**
  - All replicas contain the same version of data
  - Client always has the same view of the data (no matter what node)
- **Availability**
  - System remains operational on failing nodes
  - All clients can always read and write
- **Partition tolerance**
  - multiple entry points
  - System remains operational on system split (communication malfunction)
  - System works well across physical network partitions



CAP Theorem: satisfying all three at the same time is impossible

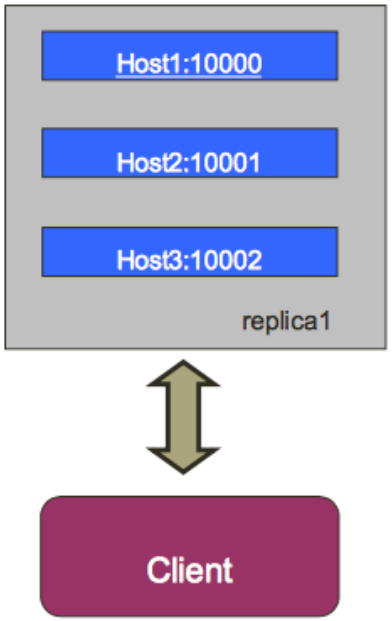

Visual Guide to NoSQL Systems
Availability:
Each client can always read and write.
A
Available, Partition-Tolerant (AP) Systems achieve "eventual consistency" through replication and verification
Data Models
Relational (comparison)
Key-Value
Column-Oriented/Tabula
Document-Oriented
CA
RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica
AP
Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak
Pick Two
Consistent, Available (CA) Systems have trouble with partitions and typically deal with it with replication
Consistent, Partition-Tolerant (CP) Systems have trouble with availability while keeping data consistent across partitioned nodes
C
P
Consistency:
All clients always have the same view of the data.
CP
BigTable
Hypertable
Hbase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis
Partition Tolerance:
The system works well despite physical network partitions.

Sharding of data

- Distributes a single logical database system across a cluster of machines
- Uses range-based partitioning to distribute documents based on a specific shard key
- Automatically balances the data associated with each shard
- Can be turned on and off per collection (table)

Replica Sets

- Redundancy and Failover
- Zero downtime for upgrades and maintenance
- Master-slave replication
  - Strong Consistency
  - Delayed Consistency
- Geospatial features



How does NoSQL vary from RDBMS?

- Looser schema definition
- Applications written to deal with specific documents/ data
  - Applications aware of the schema definition as opposed to the data
- Designed to handle distributed, large databases
- Trade offs:
  - No strong support for ad hoc queries but designed for speed and growth of database
    - Query language through the API
  - Relaxation of the ACID properties

Benefits of NoSQL

Elastic Scaling

- RDBMS scale up – bigger load , bigger server
- NO SQL scale out – distribute data across multiple hosts seamlessly

DBA Specialists

- RDMS require highly trained expert to monitor DB
- NoSQL require less management, automatic repair and simpler data models

Big Data

- Huge increase in data RDMS: capacity and constraints of data volumes at its limits
- NoSQL designed for big data

Benefits of NoSQL

Flexible data models

- Change management to schema for RDMS have to be carefully managed
- NoSQL databases more relaxed in structure of data
  - Database schema changes do not have to be managed as one complicated change unit
  - Application already written to address an amorphous schema

Economics

- RDMS rely on expensive proprietary servers to manage data
- No SQL: clusters of cheap commodity servers to manage the data and transaction volumes
- Cost per gigabyte or transaction/second for NoSQL can be lower than the cost for a RDBMS

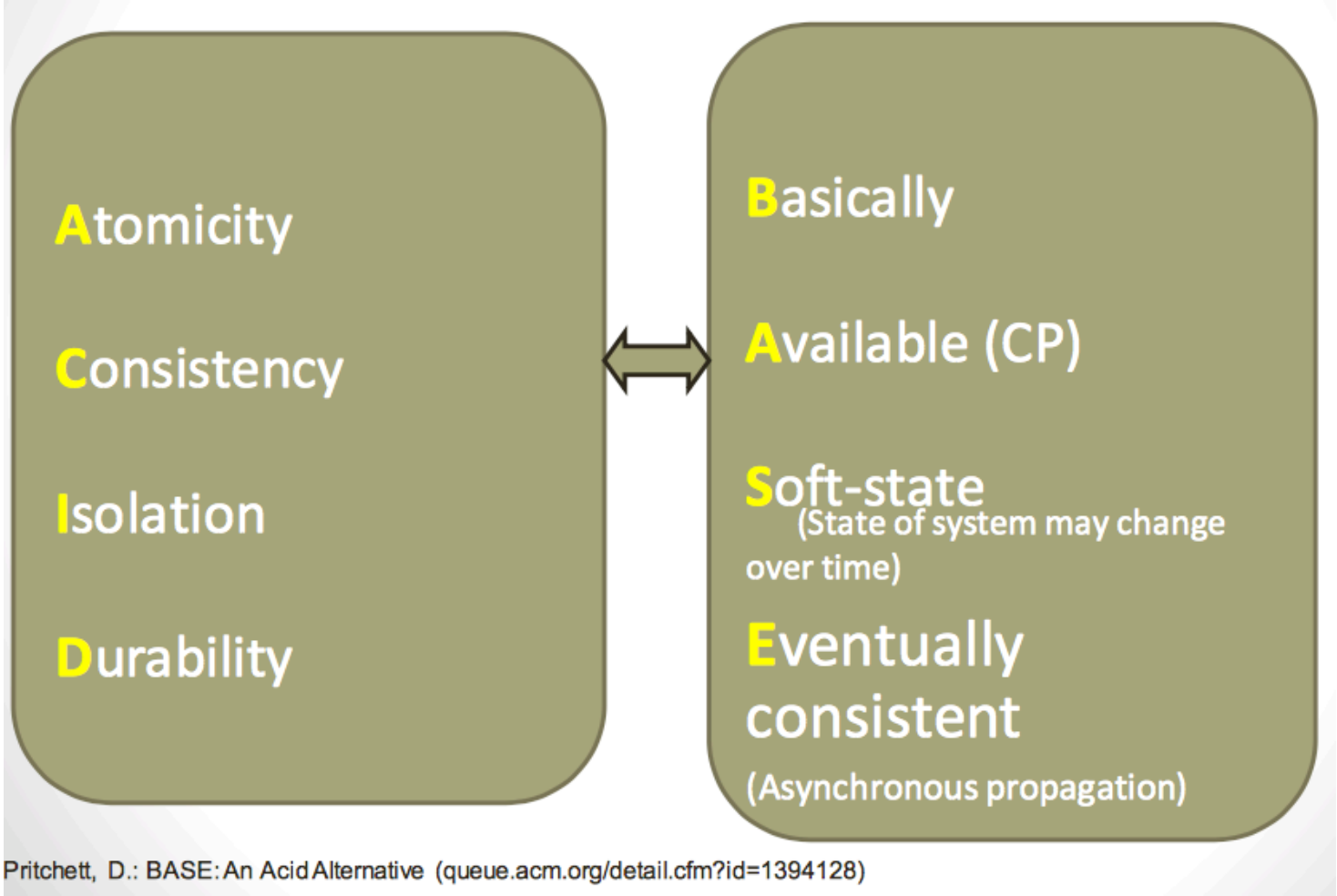
Drawbacks of NoSQL

- Support
  - RDBMS vendors provide a high level of support to clients
    - Stellar reputation
  - NoSQL – are open source projects with startups supporting them
    - Reputation not yet established
- Maturity
  - RDMS mature product: means stable and dependable
    - Also means old no longer cutting edge nor interesting
  - NoSQL are still implementing their basic feature set

Drawbacks of NoSQL

- **Administration**
  - RDMS administrator well defined role
  - No SQL's goal: no administrator necessary however NO SQL still requires effort to maintain
- **Lack of Expertise**
  - Whole workforce of trained and seasoned RDMS developers
  - Still recruiting developers to the NoSQL camp
- **Analytics and Business Intelligence**
  - **RDMS designed to address this niche**
  - NoSQL designed to meet the needs of an Web 2.0 application - not designed for ad hoc query of the data
    - Tools are being developed to address this need

ACID or BASE



Pritchett, D.: BASE: An Acid Alternative (queue.acm.org/detail.cfm?id=1394128)

Taxonomy of NoSQL

- **Key-value**

- **Graph database**

- **Document-oriented**

- **Column family**

4. MongoDB

What is MongoDB?

- Developed by 10gen
  - Founded in 2007
- A document-oriented, NoSQL database
  - Hash-based, *schema-less database*
    - No Data Definition Language
    - In practice, this means you can store hashes with any keys and values that you choose
      - Keys are a basic data type but in reality stored as strings
      - Document Identifiers (_id) will be created for each document, field name reserved by system
    - Application tracks the schema and mapping
    - Uses BSON format
      - Based on JSON – B stands for Binary
- Written in C++
- Supports APIs (drivers) in many computer languages
  - JavaScript, Python, Ruby, Perl, Java, Java Scala, C#, C++, Haskell, Erlang

Functionality of MongoDB

- Dynamic schema
  - No DDL
- Document-based database
- Secondary indexes
- Query language via an API
- Atomic writes and fully-consistent reads
  - If system configured that way
- Master-slave replication with automated failover (replica sets)
- Built-in horizontal scaling via automated range-based partitioning of data (sharding)
- No joins nor transactions

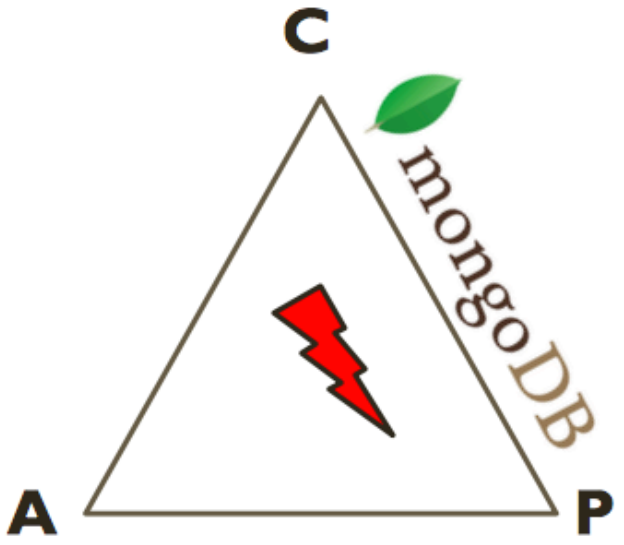
Why use MongoDB?

- Simple queries
- Functionality provided applicable to most web applications
- Easy and fast integration of data
  - No ERD diagram
- Not well suited for heavy and complex transactions systems

MongoDB: CAP approach

Focus on Consistency and Partition tolerance

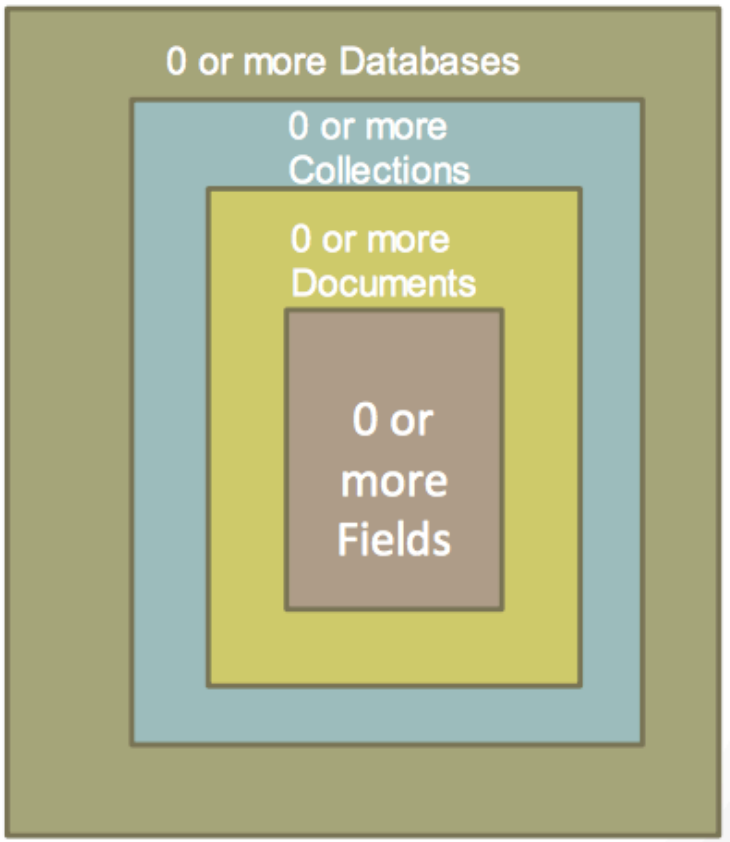
- **C**onsistency
  - all replicas contain the same version of the data
- **A**vailability
  - system remains operational on failing nodes
- **P**artition tolarence
  - multiple entry points
  - system remains operational on system split



CAP Theorem:
satisfying all three at the same time is impossible

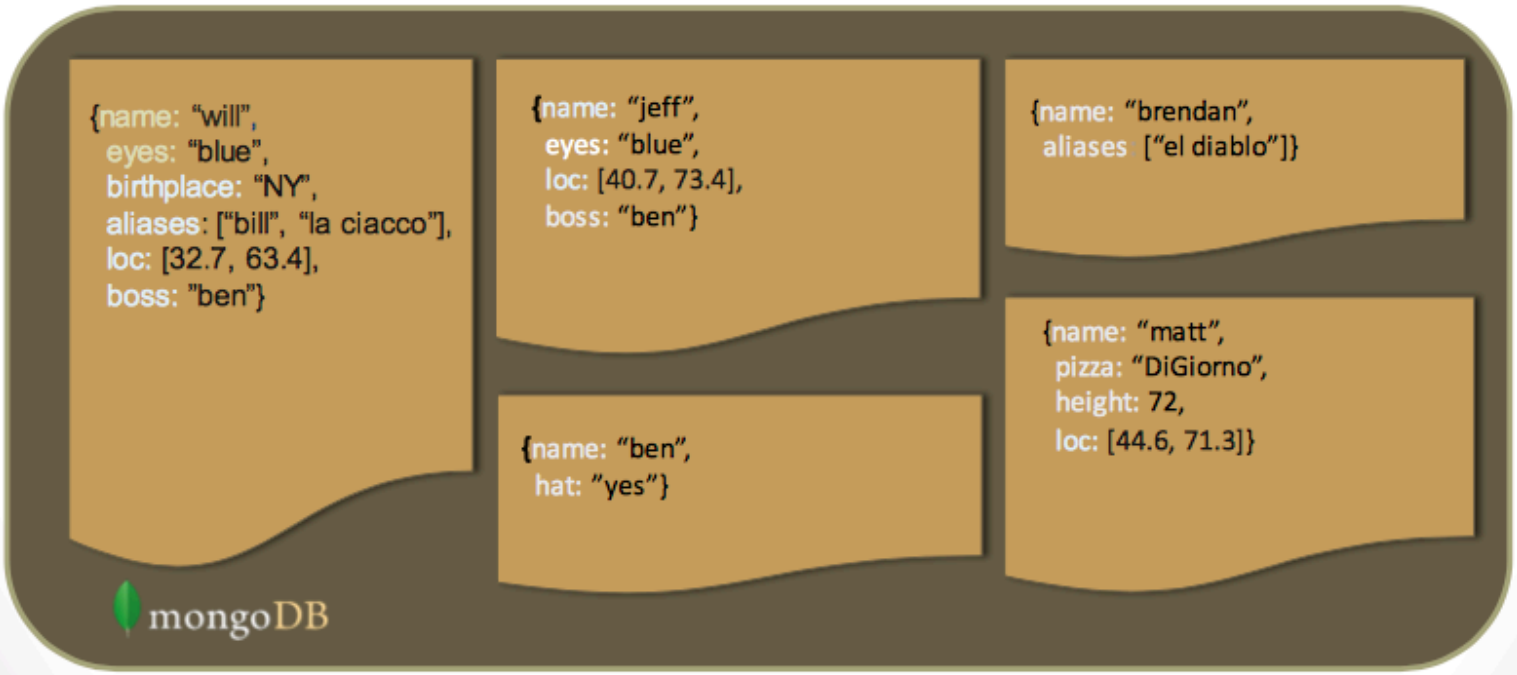
MongoDB Data model: Hierarchical Objects

- A MongoDB instance may have zero or more 'databases'
- A database may have zero or more 'collections'.
- A collection may have zero or more 'documents'.
- A document may have one or more 'fields'.
- MongoDB 'Indexes' function much like their RDBMS counterparts.



Schema Free

- MongoDB does not need any pre-defined data schema
- Every document in a collection could have different data
  - Addresses NULL data fields



MongoDB Features

- Document-Oriented storage
- Full Index Support
- Replication & High Availability
- Auto-Sharding
- Querying
- Fast In-Place Updates
- Map/Reduce functionality

Agile

Scalable

Index Functionality

- B+ tree indexes
- An index is automatically created on the _id field (the primary key)
- Users can create other indexes to improve query performance or to enforce Unique values for a particular field
- Supports single field index as well as Compound index
  - Like SQL order of the fields in a compound index matters
  - If you index a field that holds an array value, MongoDB creates separate index entries for *every* element of the array
- Sparse property of an index ensures that the index only contain entries for documents that have the indexed field. (so ignore records that do not have the field defined)
- If an index is both unique and sparse – then the system will reject records that have a duplicate key value but allow records that do not have the indexed field defined

CRUD operations

- Create
  - db.collection.insert(<document>)
  - db.collection.save(<document>)
  - db.collection.update(<query>, <update>, { upsert: true })
- Read
  - db.collection.find(<query>, <projection>)
  - db.collection.findOne(<query>, <projection>)
- Update
  - db.collection.update(<query>, <update>, <options>)
- Delete
  - db.collection.remove(<query>, <justOne>)

Query operations

Name	Description
$eq	Matches value that are equal to a specified value
$gt, $gte	Matches values that are greater than (or equal to a specified value
$lt, $lte	Matches values less than or (equal to) a specified value
$ne	Matches values that are not equal to a specified value
$in	Matches any of the values specified in an array
$nin	Matches none of the values specified in an array
$or	Joins query clauses with a logical OR returns all
$and	Join query clauses with a loginal AND
$not	Inverts the effect of a query expression
$nor	Join query clauses with a logical NOR
$exists	Matches documents that have a specified field

Aggregated functionality

Aggregation framework provides SQL-like aggregation functionality

- Pipeline documents from a collection pass through an aggregation pipeline, which transforms these objects as they pass through
- Expressions produce output documents based on calculations performed on input documents
- Example db.**parts**.aggregate ({$group : {_id: type, totalquantity : { $sum: quanity} } })

Map reduce functionality

- Performs complex aggregator functions given a collection of keys, value pairs
- Must provide at least a map function, reduction function and a name of the result set
- db.collection.mapReduce(<mapfunction>, <reducefunction>, { out: <collection>, query: <document>, sort: <document>, limit: <number>, finalize: <function>, scope: <document>, jsMode: <boolean>, verbose: <boolean> })
- More description of map reduce next lecture

Summary

- NoSQL built to address a distributed database system
  - Sharding
  - Replica sets of data
- CAP Theorem: consistency, availability and partition tolerant
- MongoDB
  - Document oriented data, schema-less database, supports secondary indexes, provides a query language, consistent reads on primary sets
  - Lacks transactions, joins