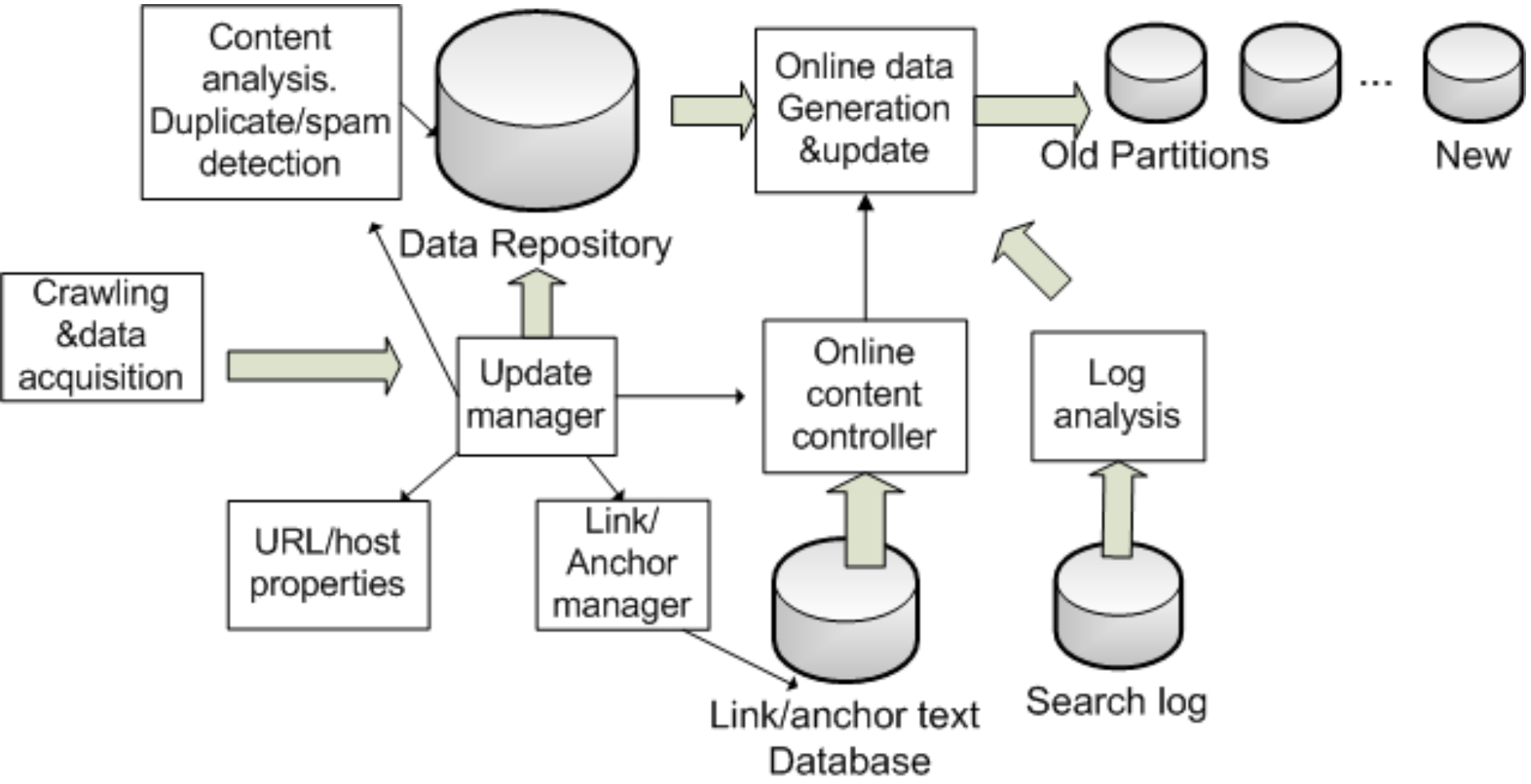
Offline Data Processing: Tasks and Infrastructure Support

T. Yang, UCSB 293S, 2022

Table of Content

- **Offline incremental data processing: case study**
  - Content management for large index
  - Text mining, knowledge graph
  - Example of content analysis
- **Duplicate content removal**
- **System support for offline data processing**

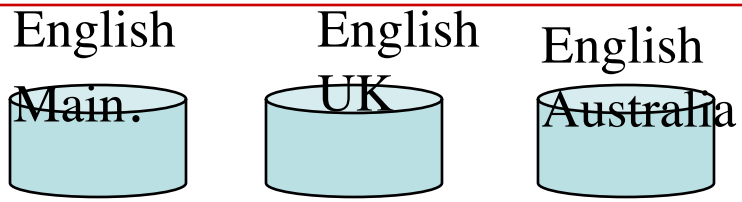
Offline Architecture for Ask.com Search



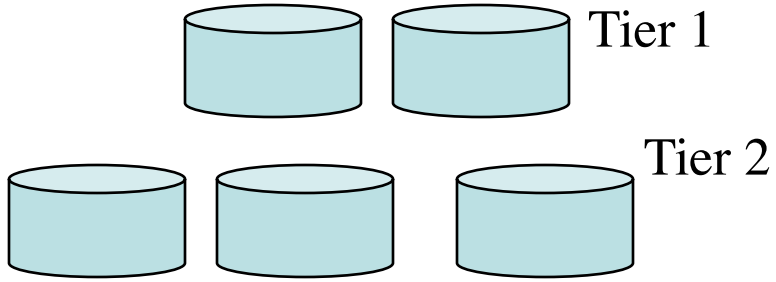
Content Management

- **Organize the vast amount of pages crawled to facilitate online search.**
  - Data preprocessing
  - Inverted index
  - Compression
  - Classify and partition data
- **Collect additional content and ranking signals.**
  - Link, anchor text, log data
- **Extract and structure content**
- **Duplicate detection**
- **Anti-spamming**

Classifying and Partitioning data



- **Classify**
  - Content quality. Language/country etc
- **Partition**
  - Based on languages and countries. Geographical distribution based on data center locations
  - Partition based on quality
    - First tier --- high chance that users will access
      - Quality indicator
      - Click feedback
    - Second tier – lower chance



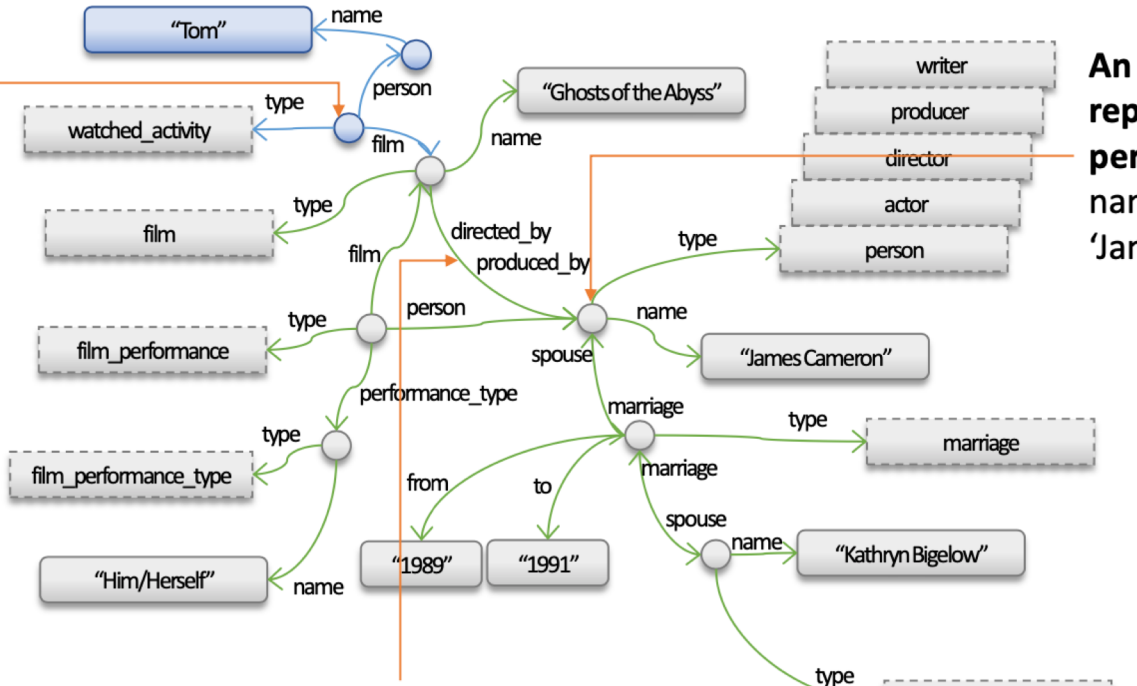
Text mining

- **“Text mining” is a cover-all marketing term**
- **A lot of what we’ve already talked about is actually the bread and butter of text mining:**
  - Text classification, clustering, and retrieval
- **But we will focus in on some of the higher-level text applications:**
  - Extracting document metadata
  - Entities/knowledge graphs
  - Topic tracking and new story detection
  - Cross document entity and event coreference
  - Text summarization

Knowledge Graph

A knowledge graph is represented as entities, edges and attributes

Personal entity showing that Tom watched Ghosts of the Abyss



An entity representing a person with name **attribute** 'James Cameron'

Key concepts

Entity	Represent something in the real world
Edge	Represent relationship
Attribute	Represent something about an entity
Ontology	Definition of possible types of entities, relationships and attributes

Edge (i.e. relationship) showing that "Ghosts of the Abyss" was 'directed_by' and 'produced_by' James Cameron

Knowledge Graphs and Challenges

General knowledge graphs

- Freebase Wikidata, Dbpedia
- Google Knowledge Vault, Google KG, Microsoft Satori KG

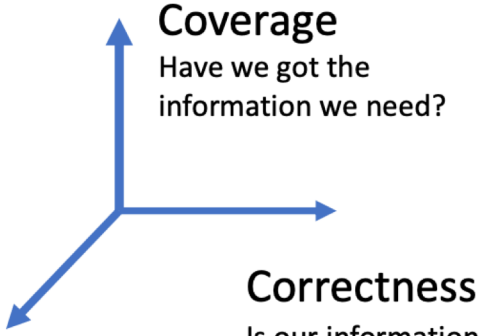
Large vertical KGs

- Facebook (social network), LinkedIn (people graph)
- Amazon (product graph)

Challenges for building/maintaining a scalable large KG

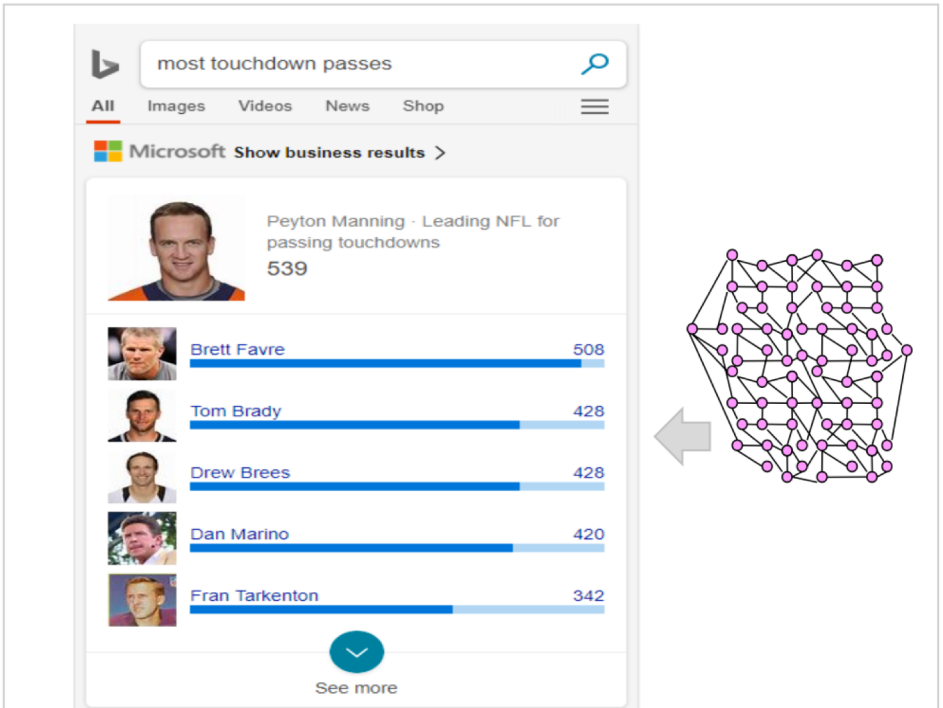
2B+ entities
130B+ Web pages
44+ languages

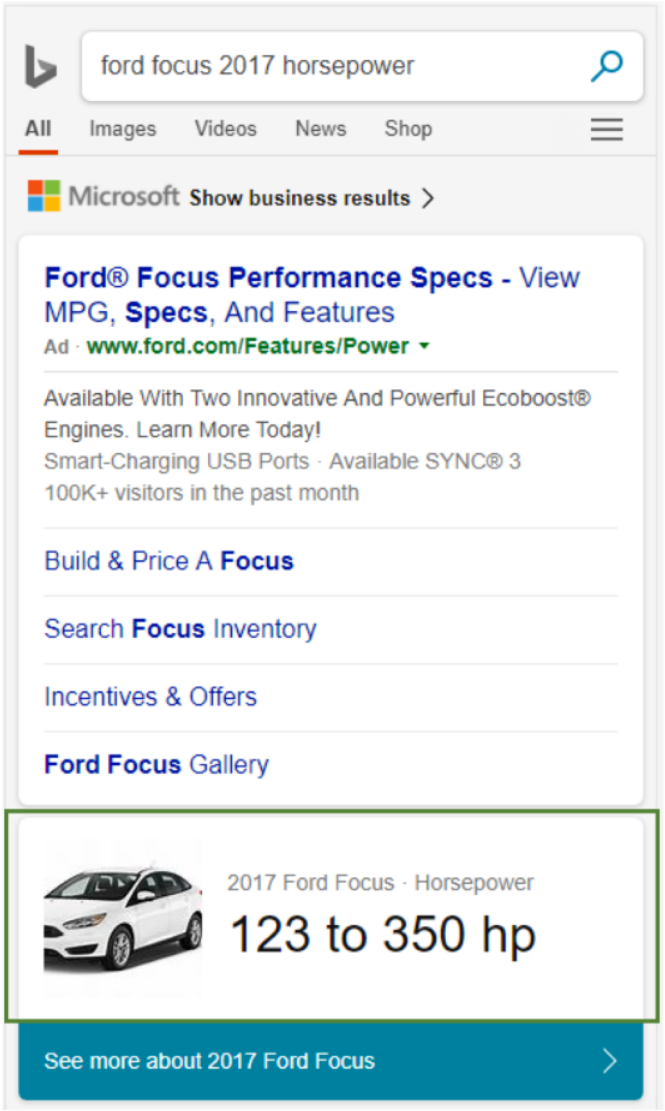
Will Smith: Single entity, 108K facts assembled from 41 web sites.
There are 200 Will Smiths on Wikipedia alone.



Usage of Knowledge Graphs for Search and Other Information Systems

- **Search and NLP questions**
  - Give direct answers
  - Enhance ranking
- **Recommendation**
- **Auto conversation**





Information extraction to enhance information on web pages and refine knowledge graphs

- **Getting semantic information out of textual data**
  - Understand more information on web pages
  - Refine knowledge entities and extract their relationship
  - Validation, association, duplicate removal/merging (entity linking), error correction, content refreshing
- **Look for specific types of web pages:**
  - E.g. an event web page:
    - What is the name of the event?
    - What date/time is it?
    - How much does it cost to attend
  - Home pages for persons, organizations,
- **Many vertical domains: resumes, health, products, …**

Examples of Context Extraction/Analysis

- **Getting semantic information out of textual data**
  - Identify key phrases that capture the meaning of this document. For example, title, section title, highlighted words.
  - Identify parts of a document representing the meaning of this document.
    - Many web pages contain a side-menu, which his less relevant to the main content of the documents
  - Identify entities and their relationships, attributes
  - Capture page content through Javascript analysis.
    - Page rendering and Javascript evaluation within a page

Example of Content Analysis

- **Identify content block related to the main content of a page**
  - Non-content text/link material is de-prioritized during indexing process

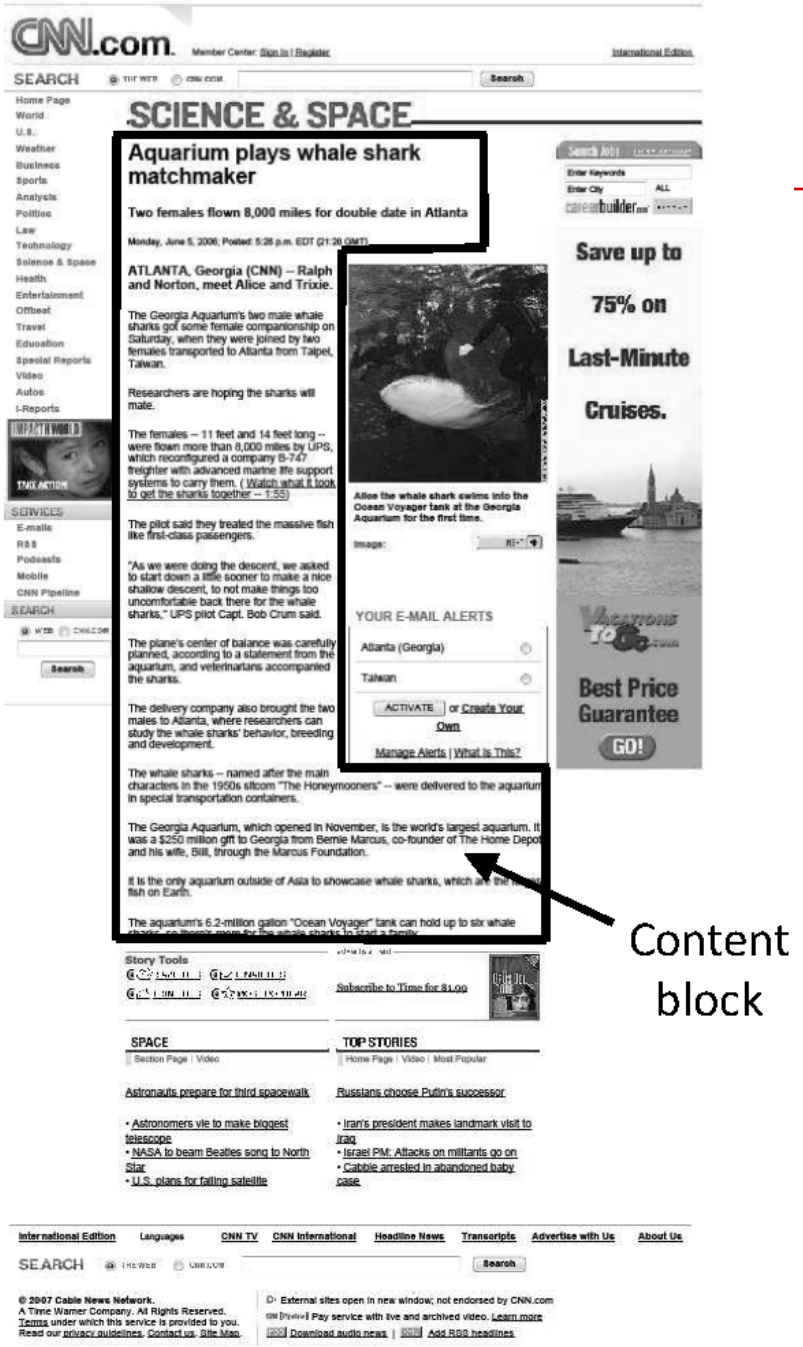
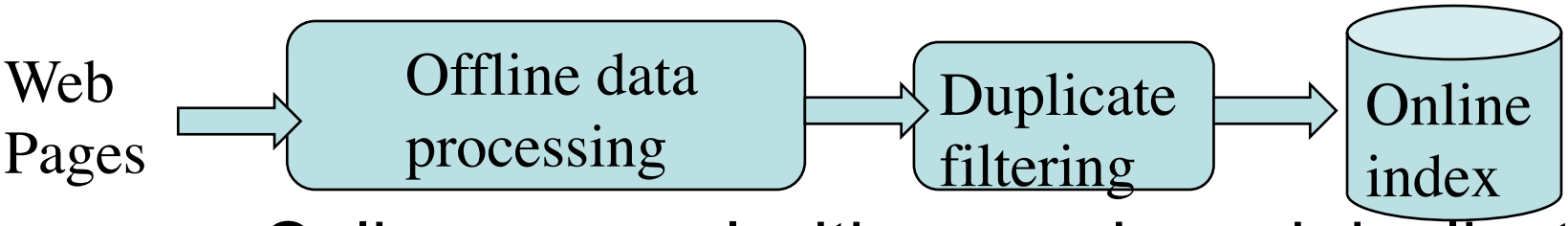


Table of Content

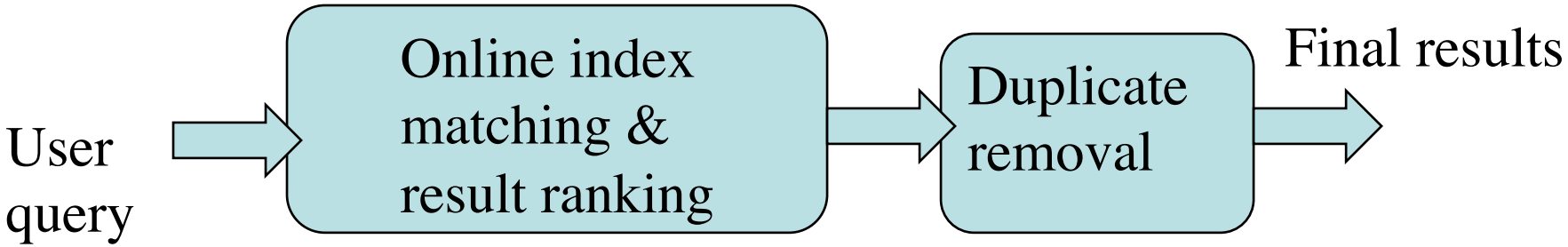
- **Offline incremental data processing: case study**
  - Content management for large index
  - Text mining, knowledge graph
  - Example of content analysis
- **Duplicate content removal** 
- **System support**

Redundant Content Removal in Search Engines

- **Over 1/3 of Web pages crawled are near duplicates**
- **When to remove near duplicates?**
  - Offline removal

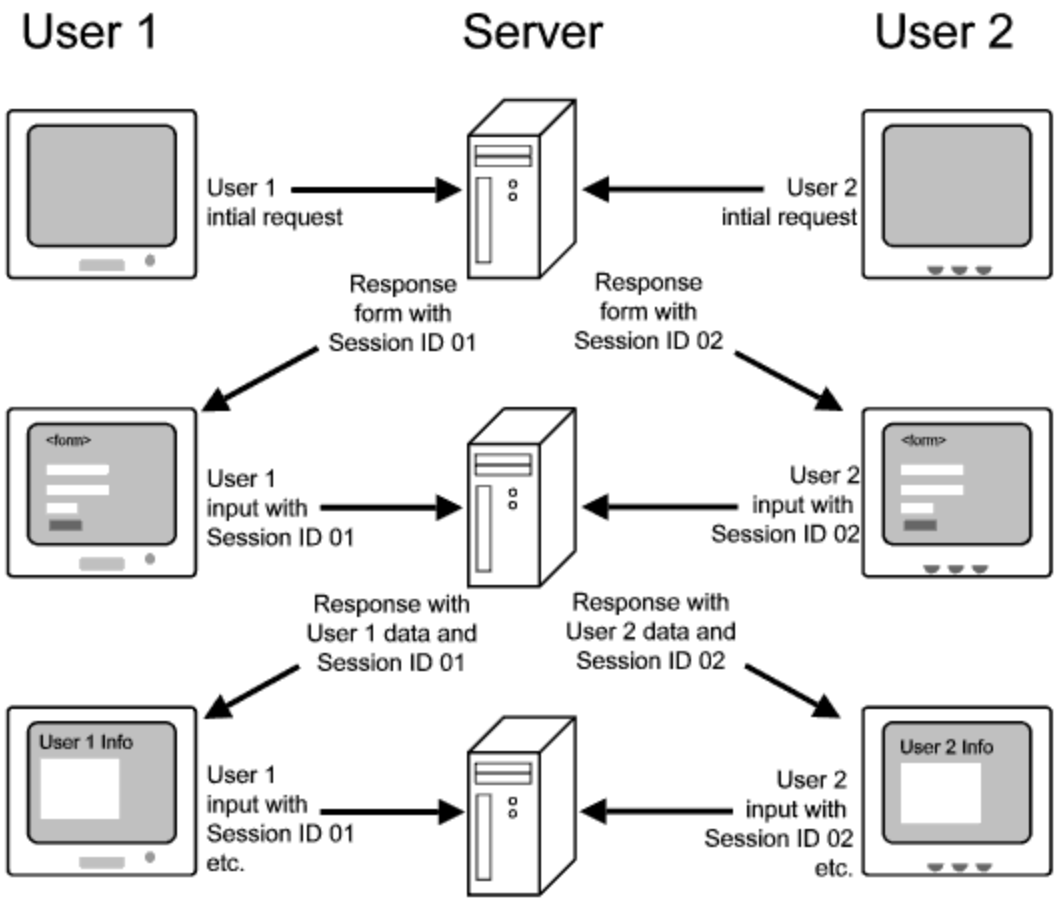


  - Online removal with query-based duplicate removal



Why there are so many duplicates?

- **Same content, different URLs, often with different session IDs.**
- **Crawling time difference**

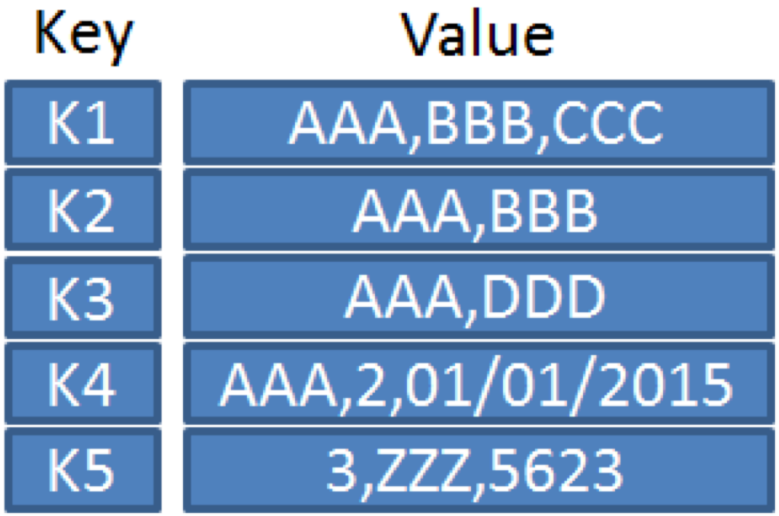


Tradeoff of online vs. offline removal

	Online-dominating approach	Offline-dominating approach
Impact to offline data processing design	High precision Low recall Remove fewer duplicates	High precision High recall Remove most of duplicates Higher offline burden
Impact to online system design	More burden to online deduplication	Less burden to online deduplication
Impact to overall cost	Higher serving cost	Lower serving cost

Key Value Stores/Storage

- **Handle huge volumes of data, e.g., PetaBytes!**
  - Store (key, value) tuples



- **Simple interface**
  - `put(key, value);` Insert/write "value" associated with "key"
  - `value = get(key);` Get/read data associated with "key"

Used sometimes as a simpler but more scalable "database"

Key Values: Examples

- **Web search: store documents, cache results, store URL properties**
  - Document server, image server, cache server, URL server
  - Neural embeddings for tokens and documents
- **Amazon shopping:**
  - Key: customerID
  - Value: customer profile (e.g., buying history, credit card, ..)

- **Facebook, Twitter accounts:**
  - Key: UserID
  - Value: user profile (e.g., posting history, photos, friends, …)

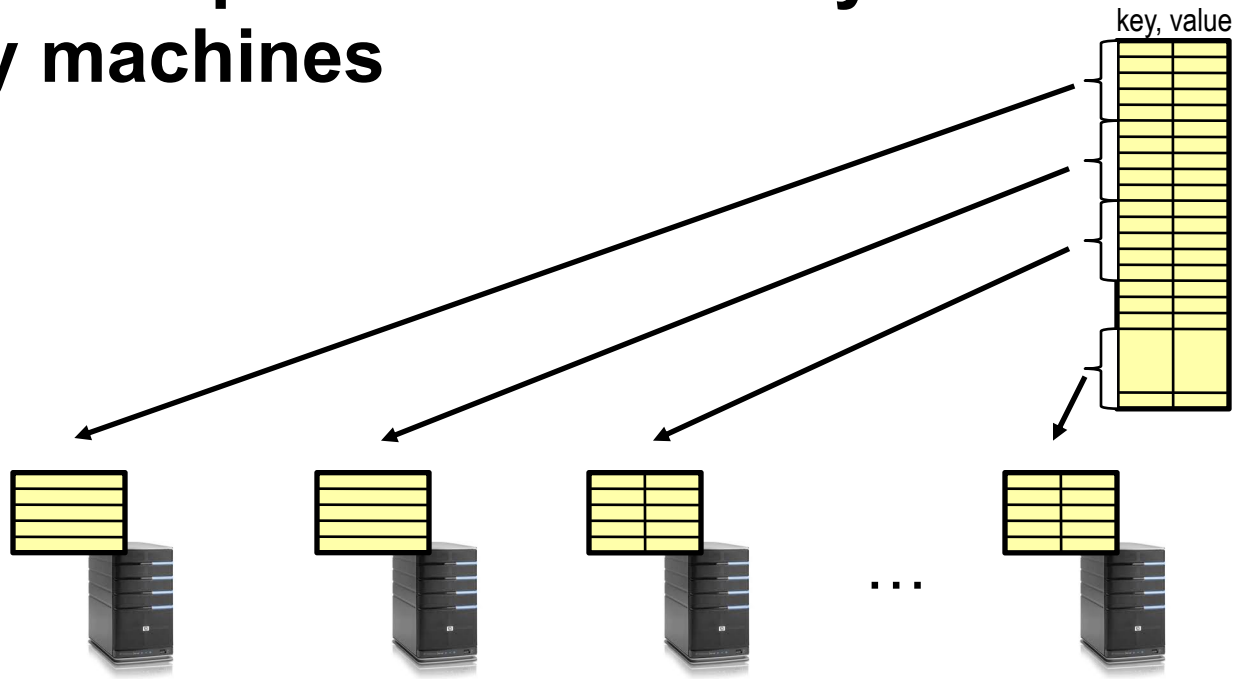
- **iCloud/iTunes:**
  - Key: Movie/song name
  - Value: Movie, Song

Key-Value Storage Systems in Real Life

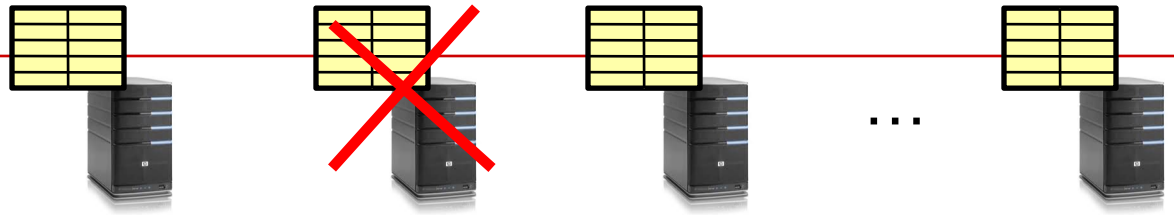
- **Amazon**
  - DynamoDB: internal key value store used for Amazon.com (cart)
  - Amazon SimpleDB. Simple Storage System (S3)
- **BigTable/HBase/Hypertable:** distributed, scalable data store
- **Cassandra:** "distributed data management system" (developed by Facebook)
- **Memcached:** in-memory key-value store for small chunks of arbitrary data (strings, objects)
- **BitTorrent distributed file location:** peer-to-peer sharing system
- **Redis, Oracle NSQL Database…**
- **Distributed file systems: set of (file block ID, file block)**

Key Value Store on a Cluster of Machines

- Also called Distributed Hash Tables (DHT)
- Main idea: partition set of key-values across many machines



Challenges



- **Fault Tolerance:** handle machine failures without losing data and without degradation in performance
- **Scalability:**
  - Need to scale to thousands of machines
  - Need to allow easy addition of new machines
- **Consistency:** maintain data consistency in face of node failures and message losses
- **Heterogeneity (if deployed as peer-to-peer systems):**
  - Latency: 1ms to 1000ms
  - Bandwidth: 32Kb/s to 100Mb/s

Important Concepts on Fault Tolerance

- **Availability: the probability that the system can accept and process requests even there are failures**
  - Often measured in "nines" of probability. So, a 99.9% probability is considered "3-nines of availability"
  - Key idea here is independence of failures

Class	Uptime	Downtime per Year (maximum)	Examples
1	90.0%	36 d 12 h	personal clients, experimental systems
2	99.0%	87 h 36 m	entry-level business systems
3	99.9%	8 h 46 m	top Internet Service Providers, mainstream business systems
4	99.99%	52 m 33 s	high-end business systems, data centers
5	99.999%	5 m 15 s	carrier-grade telephony; health systems; banking
6	99.9999%	31.5 s	military defense systems

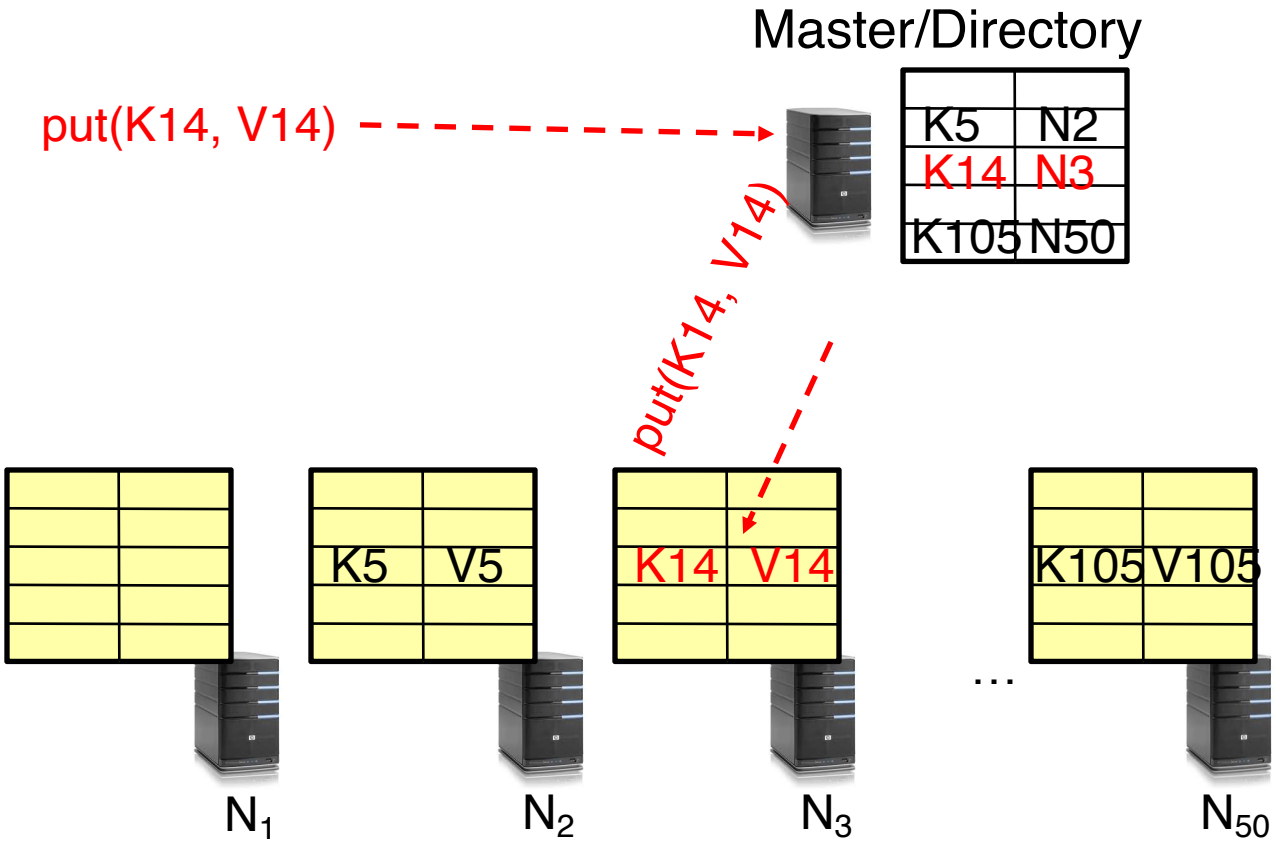
- **Reliability**: the ability to perform required functions correctly during failure
  - Usually stronger than simply availability: means that the system is not only "up", but also working correctly
  - Must make sure data survives system crashes, disk crashes, etc

Key Questions

- **`put(key, value)`: where to store a new (key, value) tuple?**
- **`get(key)`: where is the value associated with a given "key" stored?**
- **And, do the above while providing**
  - Fault Tolerance
  - Scalability
  - Consistency

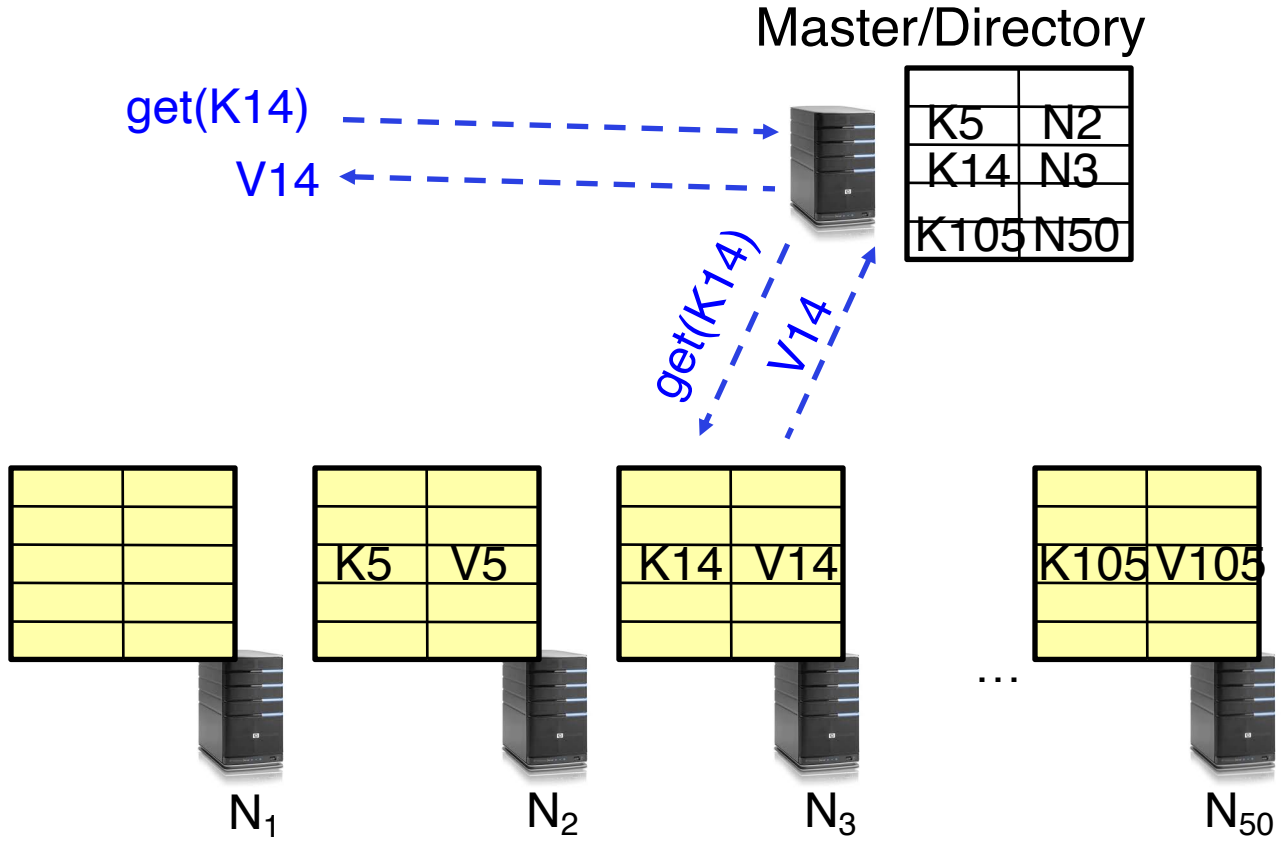
Directory-Based Architecture

- **Have a node maintain the mapping between keys and the machines (nodes) that store the values associated with the keys**



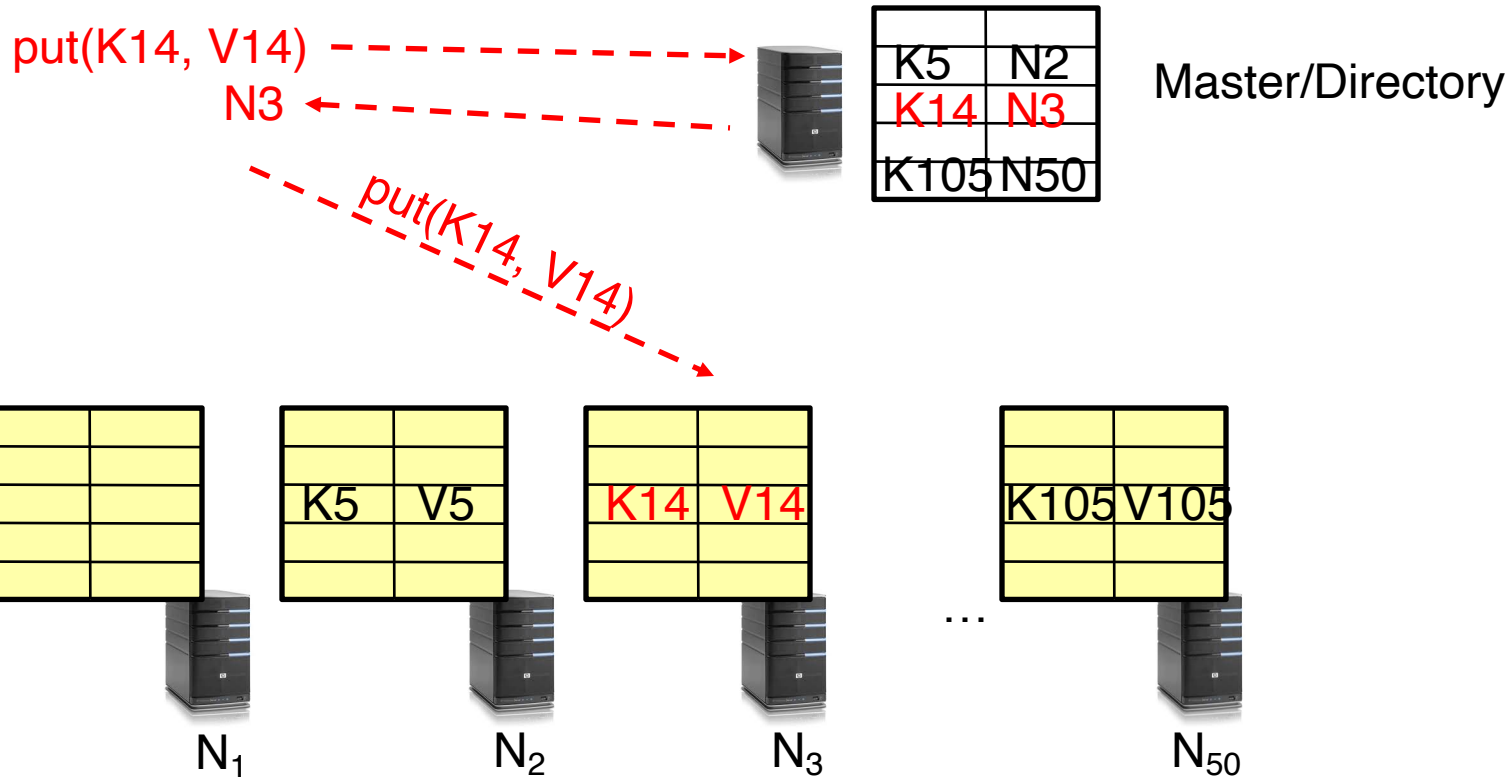
Directory-Based Architecture

- **Have a node maintain the mapping between keys and the machines (nodes) that store the values associated with the keys**



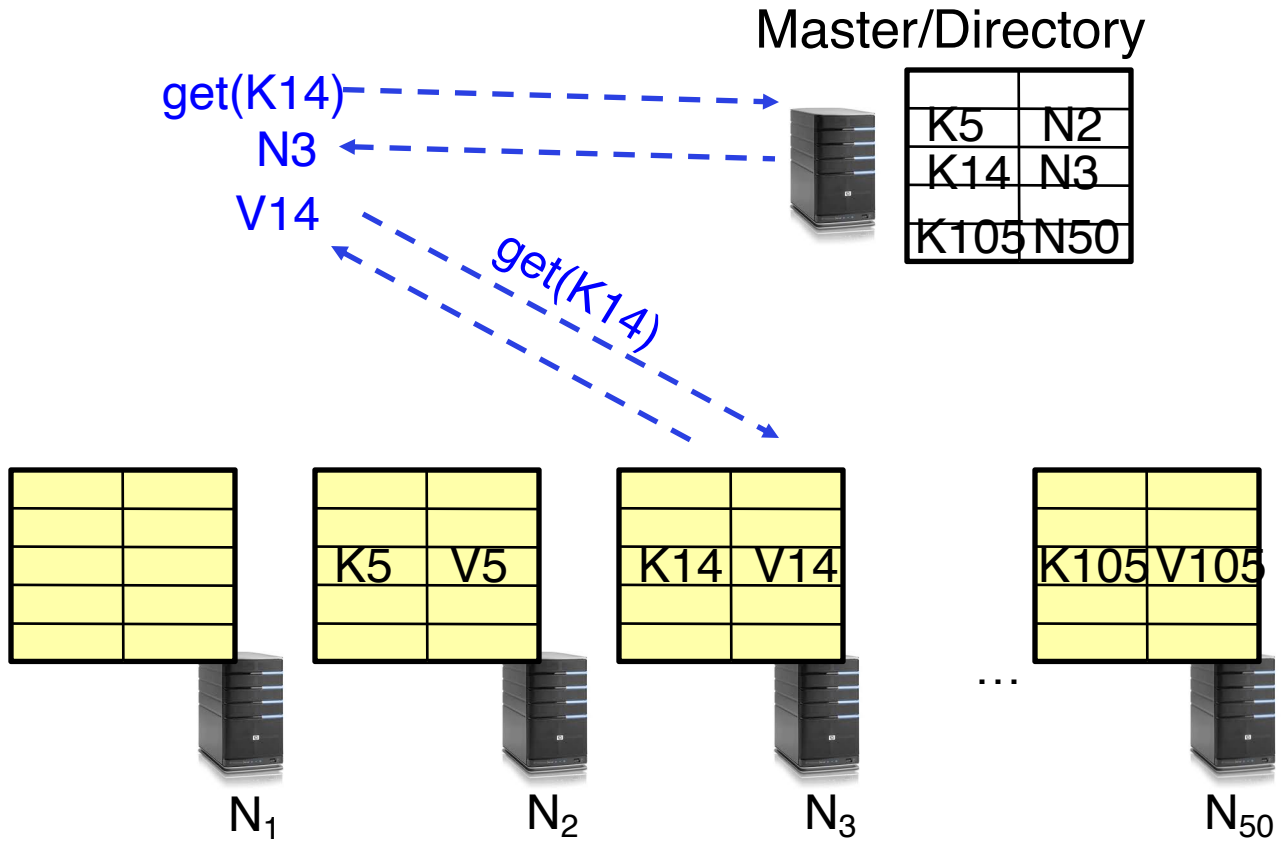
Directory-Based Architecture

- ~~Having the master relay the requests → recursive query~~
- Another method: **iterative query** (this slide)
  - Return node to requester and let requester contact node



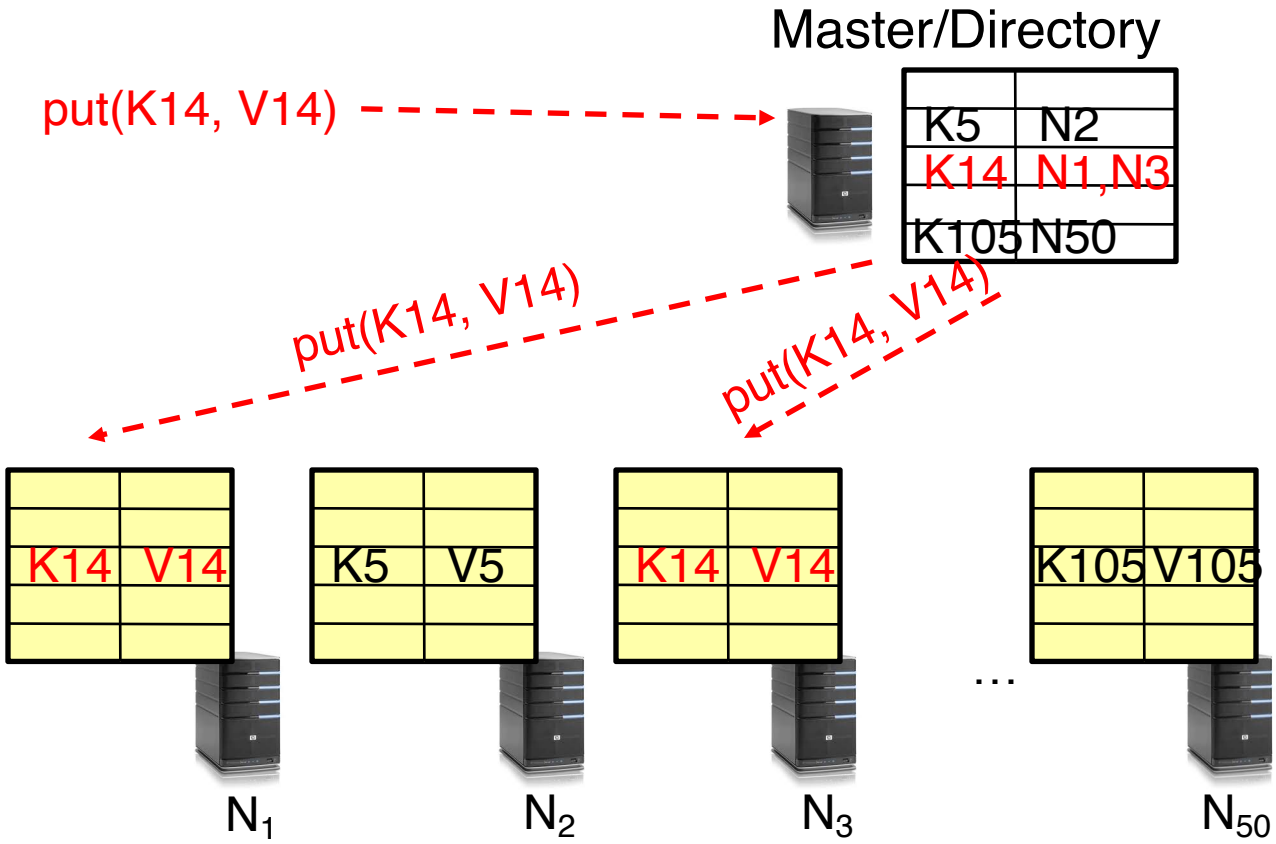
Directory-Based Architecture

- ~~**Having the master relay the requests → recursive query**~~
- **Another method: iterative query**
  - Return node to requester and let requester contact node

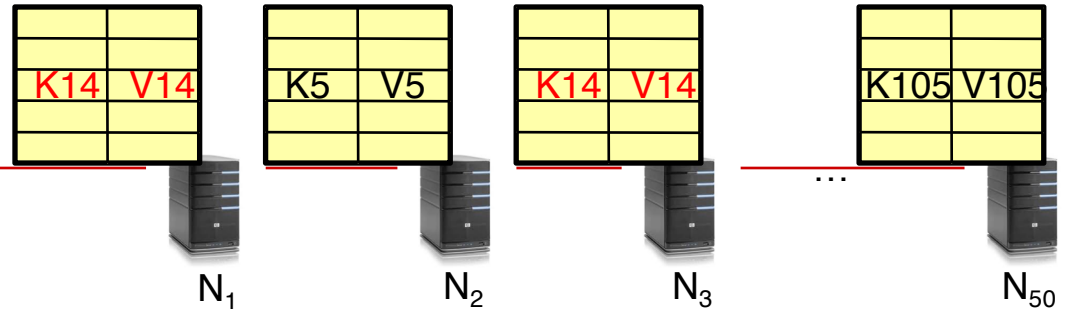


Fault Tolerance

- Replicate key-value pairs on several nodes
- Usually, place replicas on different racks in a datacenter to guard against rack failures



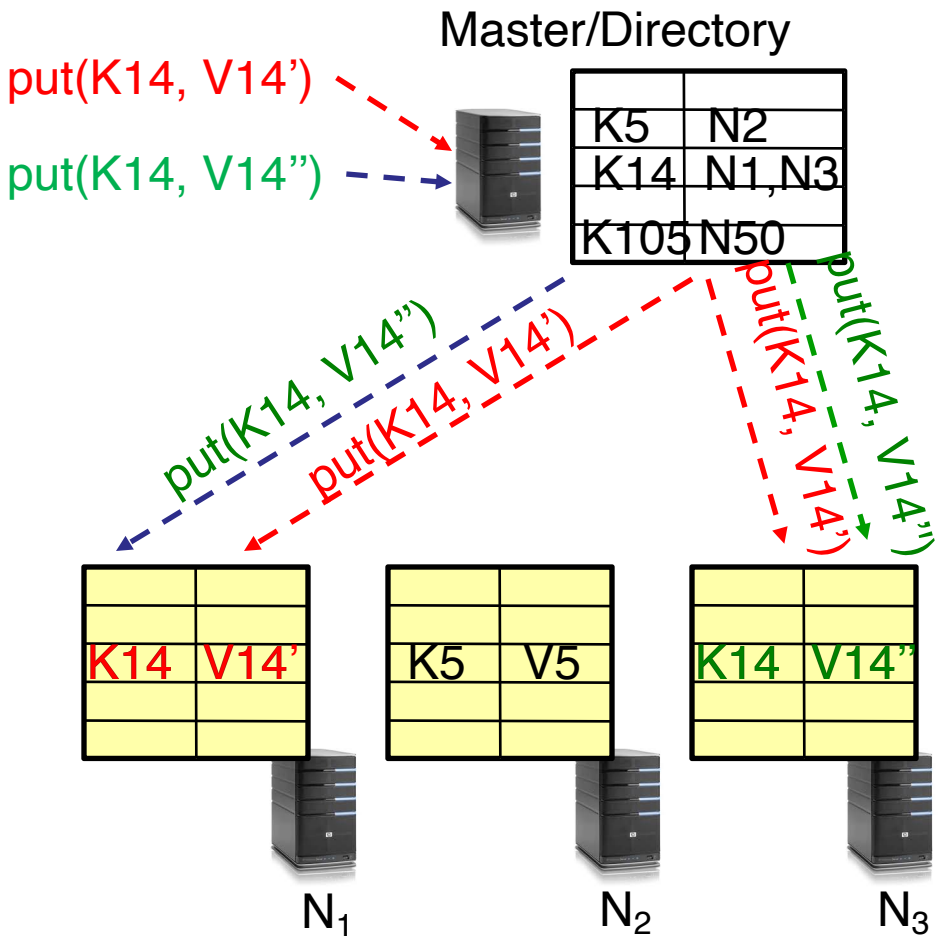
Consistency among replicas



- **Need to make sure that a value is replicated correctly**
  - How do you know a value has been replicated on every node?
  - Wait for acknowledgements from every node
- **What happens if a node fails during replication?**
  - Pick another node and try again
- **What happens if a node is slow?**
  - Slow down the entire put()? Pick another node?
- **In general, with multiple replicas**
  - Slow puts and fast gets

Consistency (cont'd)

- **If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order**



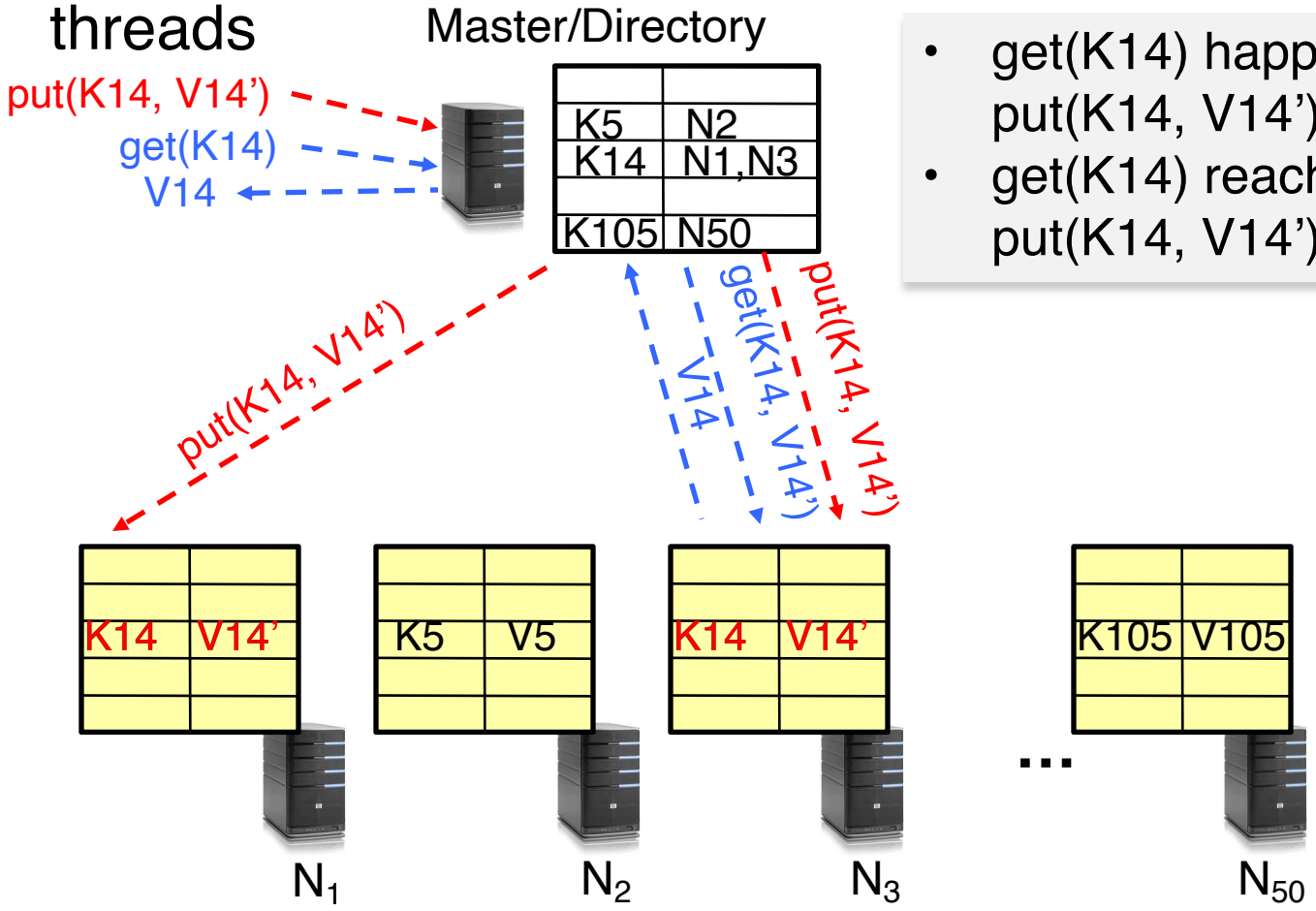
- put(K14,V14') and put(K14,V14") reach N_1 & N_3 in reverse order
- What does get(K14) return?
  - Undefined!

Inconsistent write/write

from UCB CS162

Read after Write

- **Read not guaranteed to return value of latest write**
  - Can happen if Master processes requests in different threads



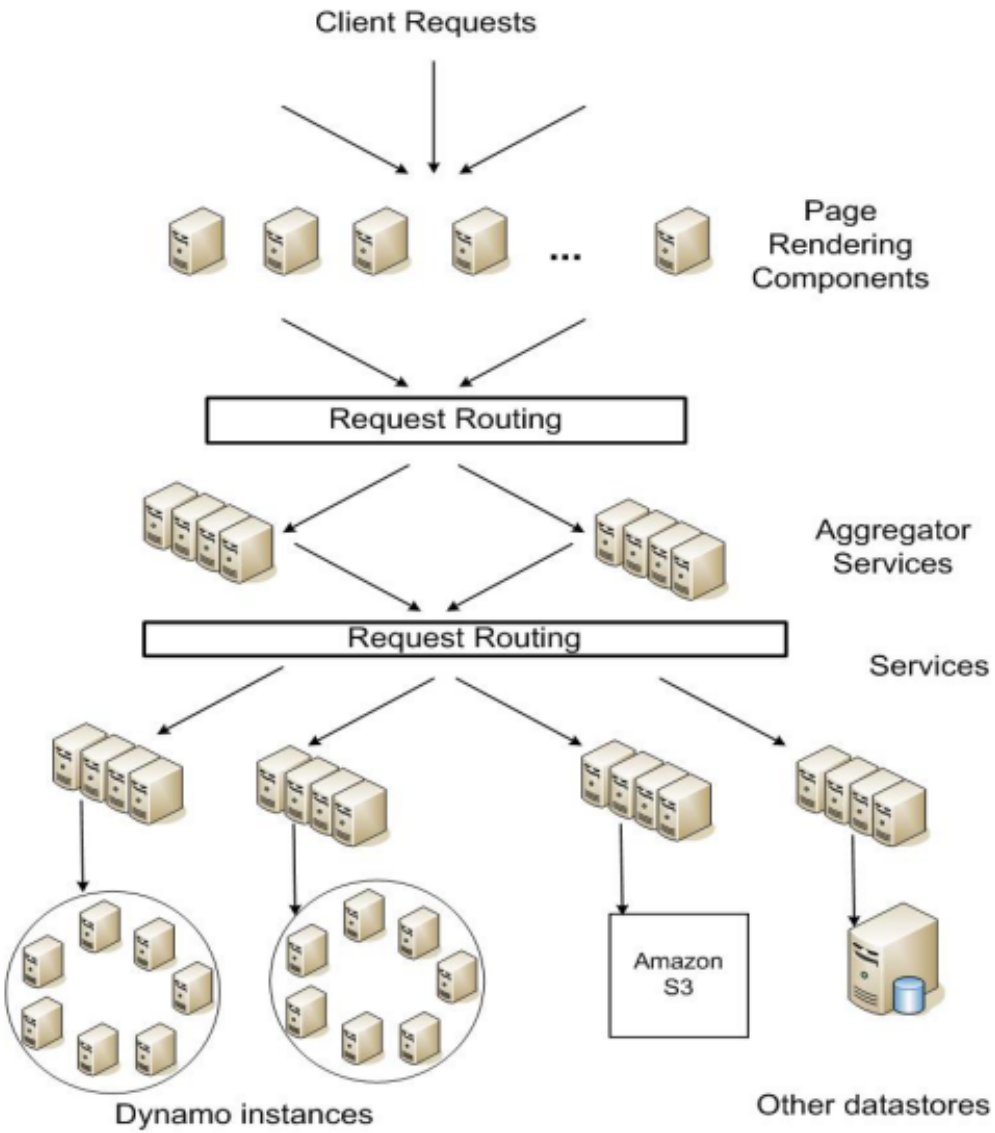
- get(K14) happens right after put(K14, V14')
- get(K14) reaches N3 before put(K14, V14')!

Inconsistent write/read

from UCB CS162

API Interface vs Amazon Store Architecture

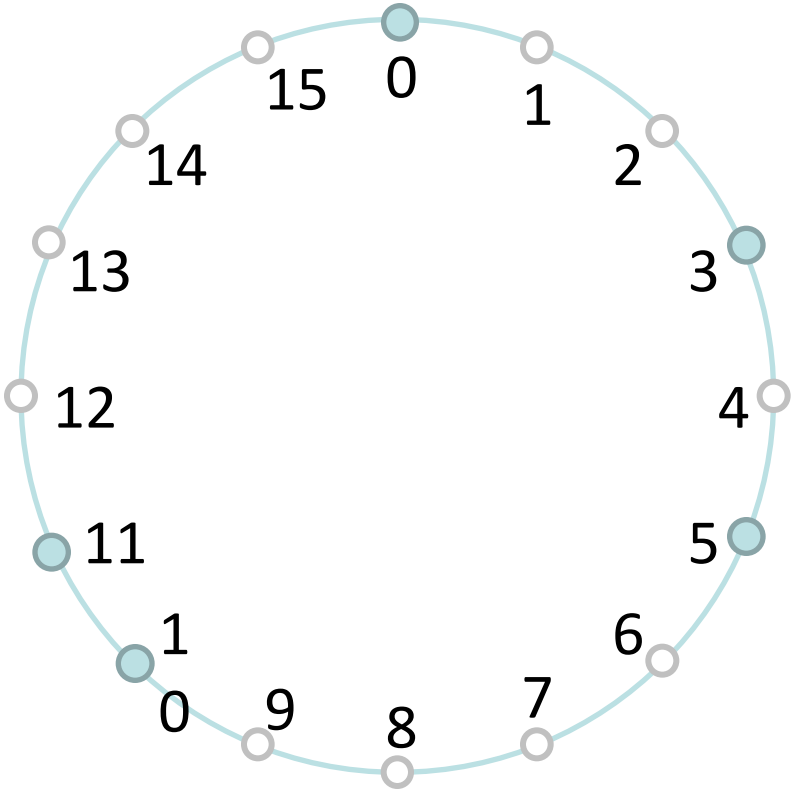
- **get(key)**
  - return single object or list of objects with conflicting version and context
- **put(key, context, object)**
  - store object and context under key
- **Context encodes system meta-data, e.g. version number**

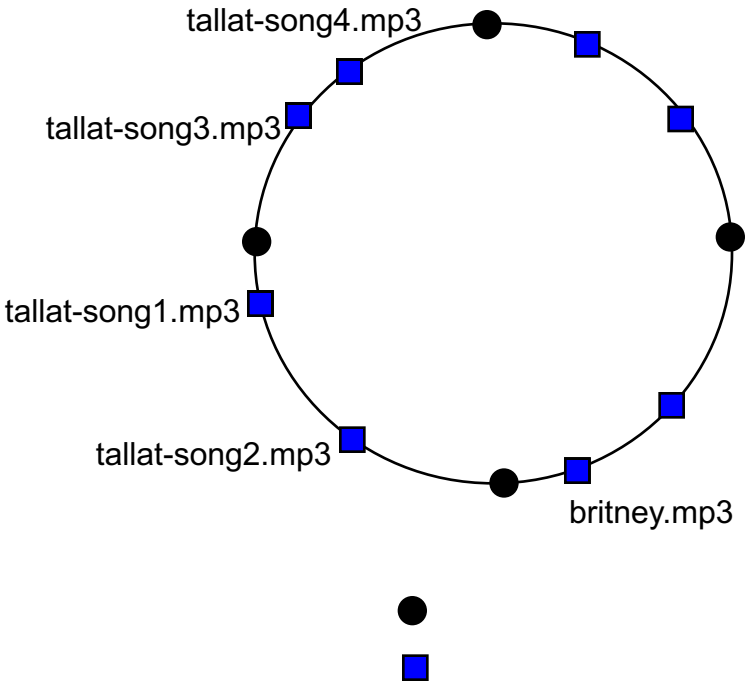


Dynamo: Amazon's Highly Available Key-value Store. SOSP 2007

Data partitioning: How to assign data to machines

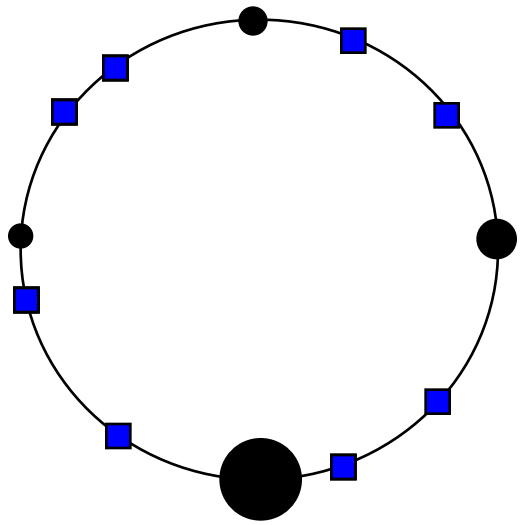
- **Assign data to machines with hashing**
  - View machines as a ring
  - Consistent hashing: the output range of a hash function is treated as a fixed circular space or “ring”.





Load imbalance caused by simple hashing

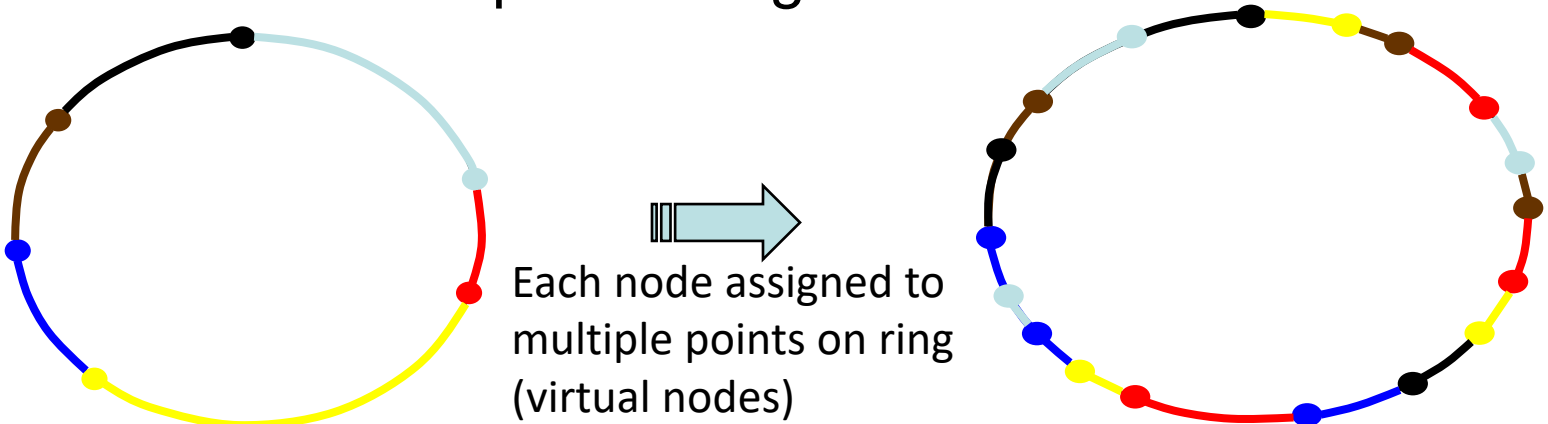
- Node identifiers may not be balanced
- Data identifiers may not be balanced
- Hot spots
- Heterogeneous nodes
- Nodes may be added or deleted periodically



● - node
■ - data

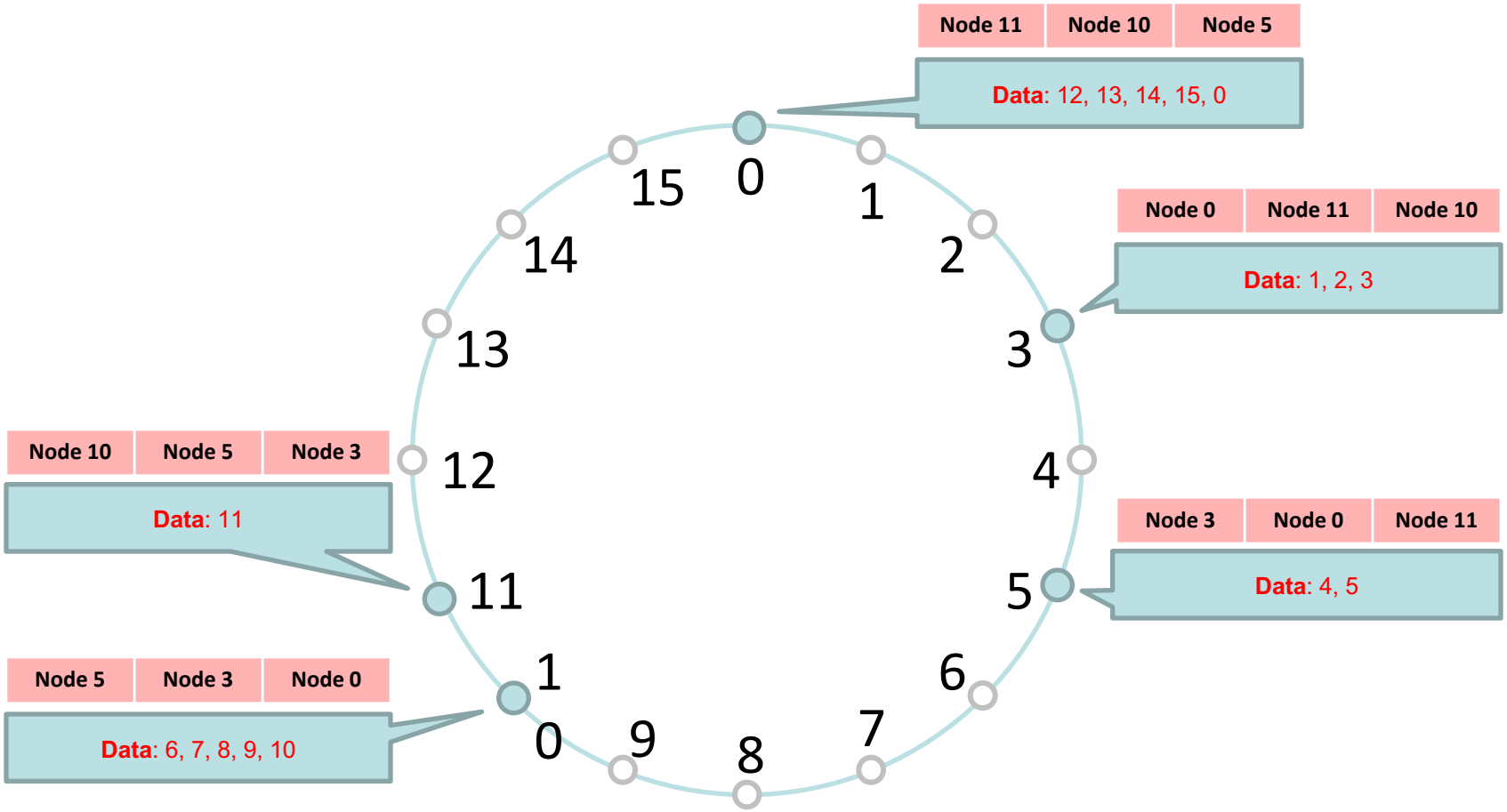
Load balancing via Virtual Servers

- **"Virtual Nodes": Each node can run and responsible for multiple virtual nodes.**
  - Each physical node picks multiple random identifiers
  - Each identifier represents a virtual server
- **Better load balancing**
  - Evenly dispersed on node failure
  - New node takes load from all others
- **#virtual nodes allows for heterogenity**
- **Problems:** Slow repartitioning



Data Replication for Better Availability

- Map the same copy of data to *N* nodes for a replication factor of *N*

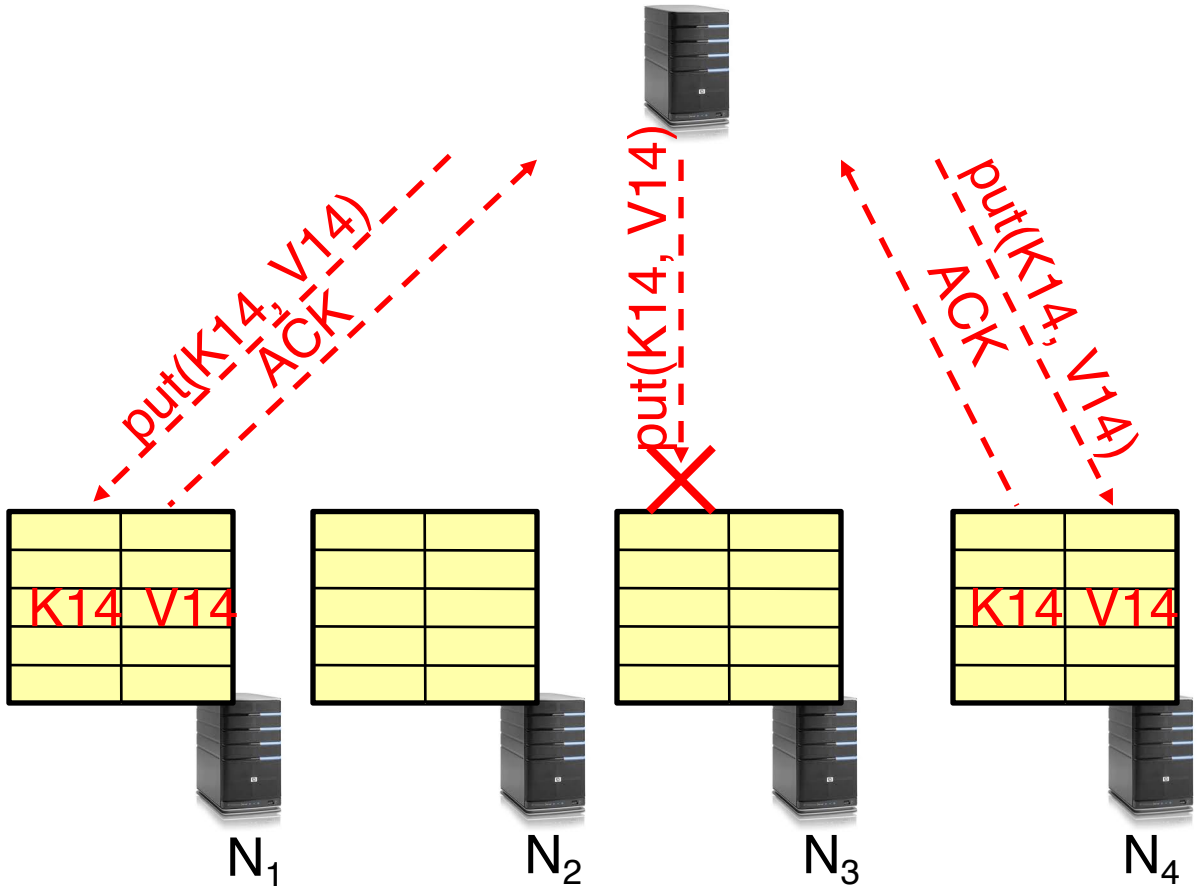


Quorum Consensus for Consistency

- **Reading or writing involves multiple replicas.**
  - But not wait for all replicas : improve put() and get() operation performance
- **Define a replica set of size N**
  - put() yields writes to all replicas, and waits for acknowledgements from at least W replicas. The writer returns after it hears form these replicas.
    - Ensure sufficient replicas have right versions.
  - get() asks a response from all replicas and waits for responses from at least R replicas. Use timestamp to get the latest version.
  - W+R > N
- **Why does it work?**
  - There is at least one node that contains the latest update

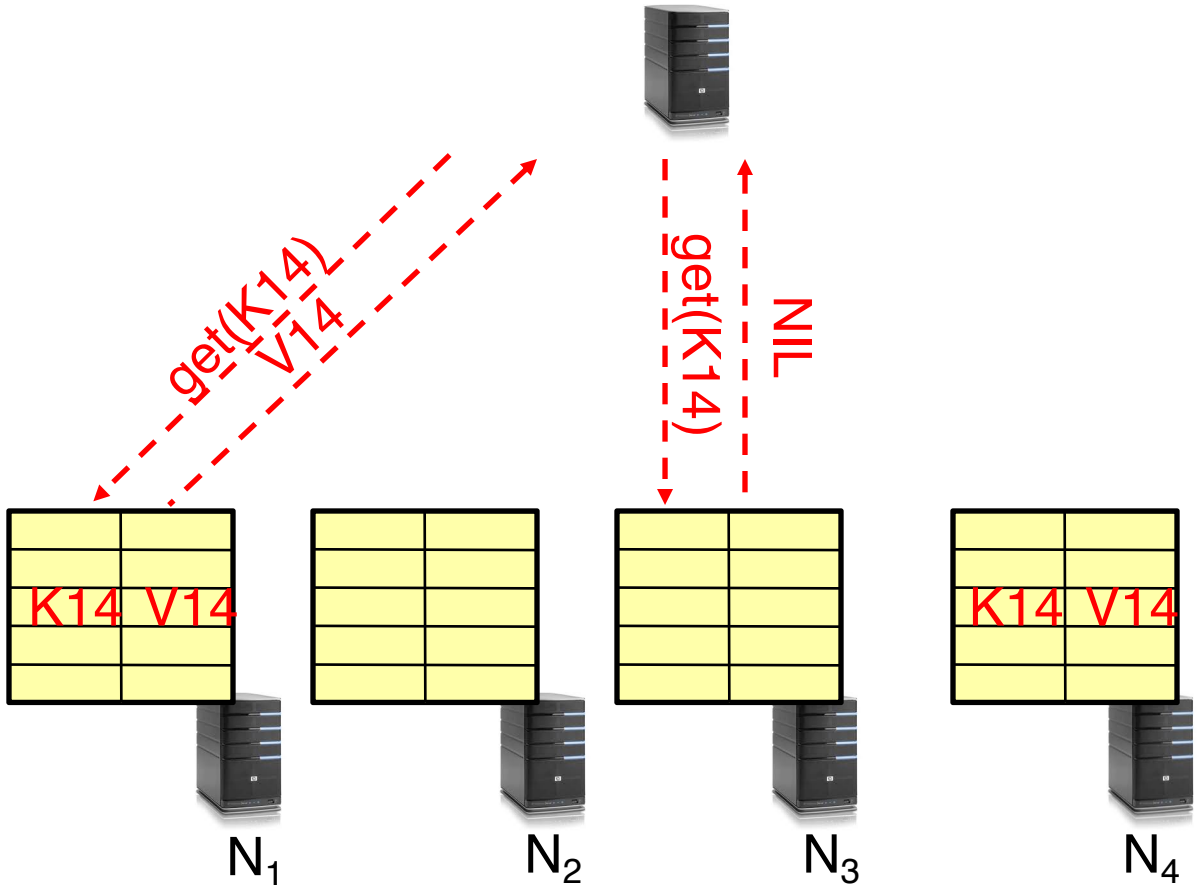
Quorum Consensus Example

- N=3, W=2, R=2
- Replica set for K14: {N_1, N_3, N_4}
- Assume put() on N_3 fails. But



Quorum Consensus Example

- **Now, issuing get() to any two nodes out of three will return the answer**



from UCB CS162

Questions : Key Value Stores

- Q1: True X False _ On a single machine, key-value store can be implemented as a hash table.

- Q2: 1 _ 2 X 3_ machine failures can be tolerated with the number of replicas as 3 in a distribued key-value store.

- Q3: 1 _ 2 X 3_ replicas must respond for a read in a quorum-based scheme when # of relicas =3 and # of replicas to respond a write operation is 2.

Remember W+R >N where

W=2 and N=3. Thus R>N-W=1

Performance Metrics and Optimization Goals

- **Each request is put/get operation**
- **Throughput** – # of requests that can be handled per second or by a cluster by one machine or by the service with a cluster of machines
  - High traffic → high throughput requirement. Typically a few thousand requests per second
- **Turnaround time** – amount of time to execute a request
  - Completion time – arrival time
- **Response time** – amount of time it takes for each request.
  - Similar to turnaround time. But if a partial response is conducted, then turnaround time is for completion of the entire request.
- **Possible bottlenecks:** CPU utilization, memory consumption, disk latency, network I/O

Evaluation of Dynamo DB

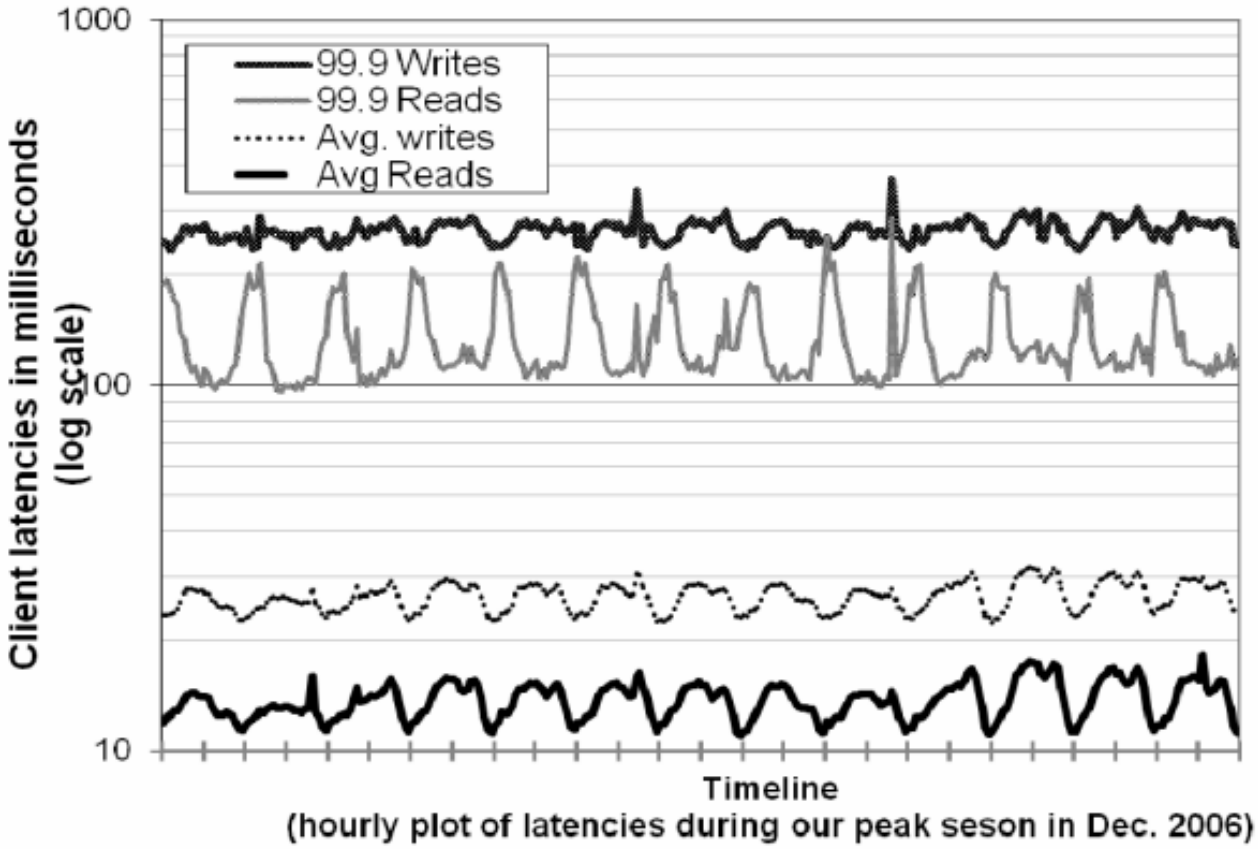


Figure 4: Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages

Dynamo: Amazon's Highly Available Key-value Store . 2007 SOSP

Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. 2022 USENIX ATC

Evaluation

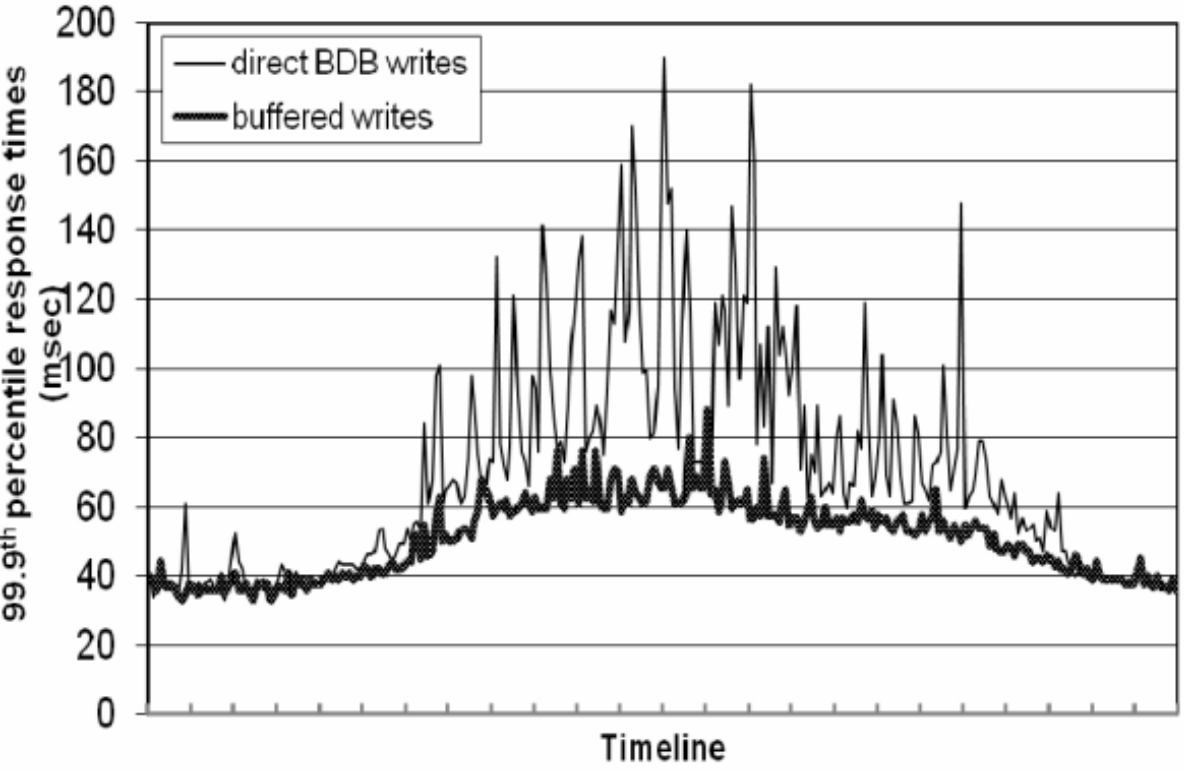


Figure 6: Fraction of nodes that are out-of-balance (i.e., nodes whose request load is above a certain threshold from the average system load) and their corresponding request load. The interval between ticks in x-axis corresponds to a time period of 30 minutes.

Evaluation: versioning due to concurrent writes

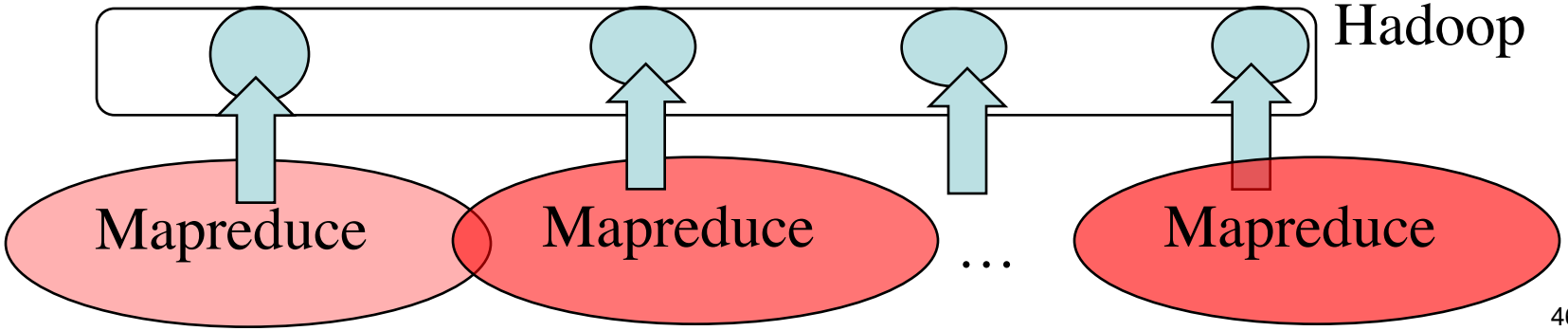
- **Divergence**
  - Number of different versions returned
  - Over 24h period
- **Reason**
  - Node failures, data center failures, network partitions
  - Large number of concurrent writes to an item
- **Occurence**
  - 99.94 % one version
  - 0.00057 % two versions
  - 0.00047 % three versions
  - 0.00009 % four versions

Summary: Key-Value Stores

- **Very large scale storage systems**
  - Distributed hash tables
- **Two operations**
  - put(key, value)
  - value = get(key)
- **Challenges**
  - Fault Tolerance → replication
  - Scalability → serve get()'s in parallel; replicate/cache hot tuples
  - Consistency → quorum consensus to improve put/get performance
- **Amazon's Dynamo key-value store**

Distributed Processing for Indexing and Data Analysis

- Distributed processing driven by need to index and analyze huge amounts of data (i.e., the Web)
- Large numbers of inexpensive servers used rather than larger, more expensive machines
- *MapReduce* is a distributed programming tool
  - Simplify data distribution on a cluster of machines
  - Open source code runs on Hadoop distributed file system
  - Provide fault tolerance
  - But not designed for interactive applications



MapReduce Programming Model

- Data: a set of key-value pairs to model input, intermediate results, and output
  - Initially input data is stored in files
  - stored in Hadoop: distributed file system built on a cluster of machines→ Looks like one machine
- Parallel computation:
  - A set of Map tasks and reduce tasks to access and produce key-value pairs
  - Map Function: (key1, val1) → (key2, val2)
  - Reduce: (key2, [val2 list]) → [val3]

Inspired by LISP Function Programming

- **Lisp *map* function**
  - Input parameters: a function and a set of values
  - This function is applied to each of the values.

  Example:
  - (map 'length '(() (a) (ab) (abc)))

  →(length(()) length(a) length(ab) length(abc)). → (0 1 2 3)

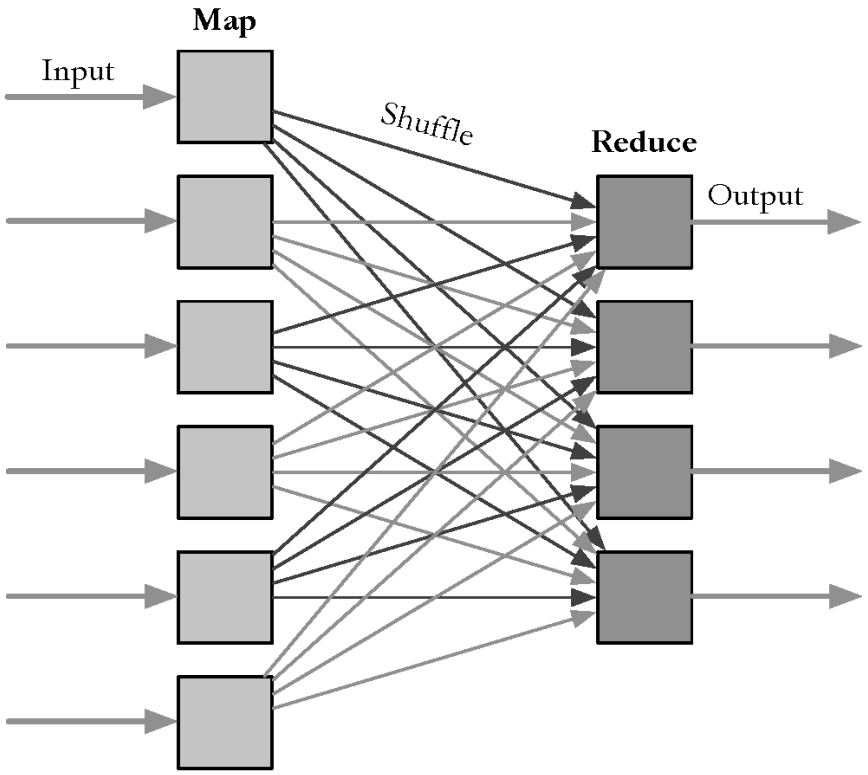
- **Lisp *reduce* function**
  - given a binary function and a set of values.
  - It combines all the values together using the binary function.
  - Example:
    - use the + (add) function to reduce the list (0 1 2 3)
    - (reduce #'+ '(0 1 2 3)) → 6

MapReduce

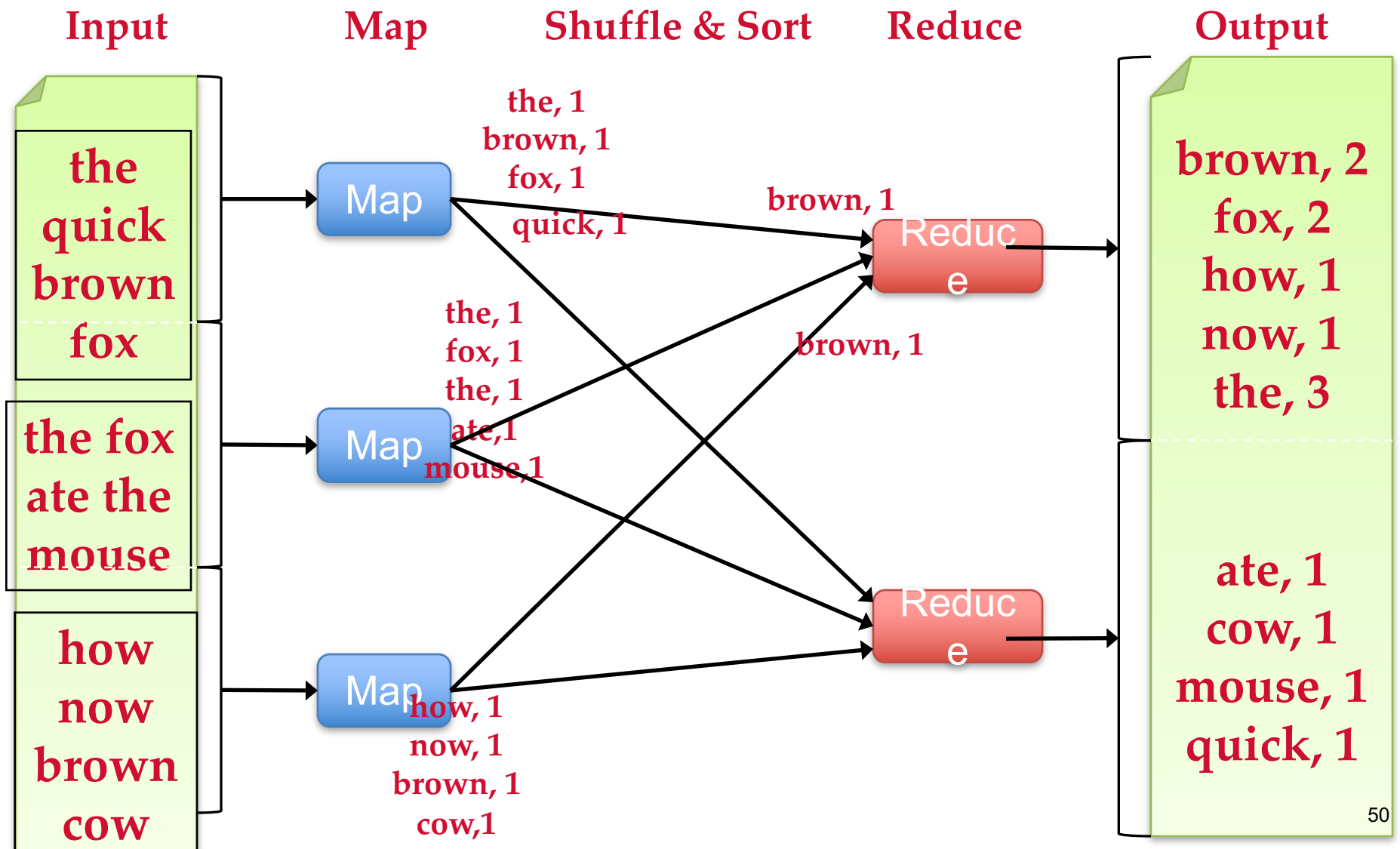
Distributed programming framework that simplifies on data placement and distribution on a cluster of machines

- ***Mapper***
  - Generally, transforms a list of items into another list of items of the same length
- ***Reducer***
  - Transforms a list of items into a single item
  - processes records in batches, where all pairs with the same key are processed at the same time
- ***Shuffle***
  - Uses a hash function so that all pairs with the same key end up the same machine

Suitable for large data mining jobs
Not for interactive jobs



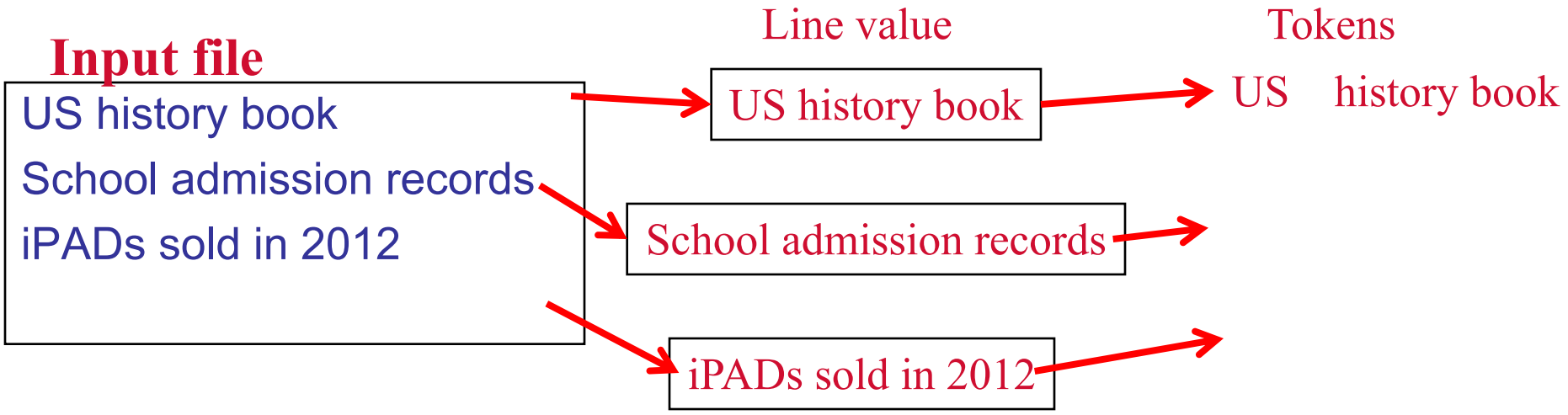
MapReduce to compute document frequency of terms



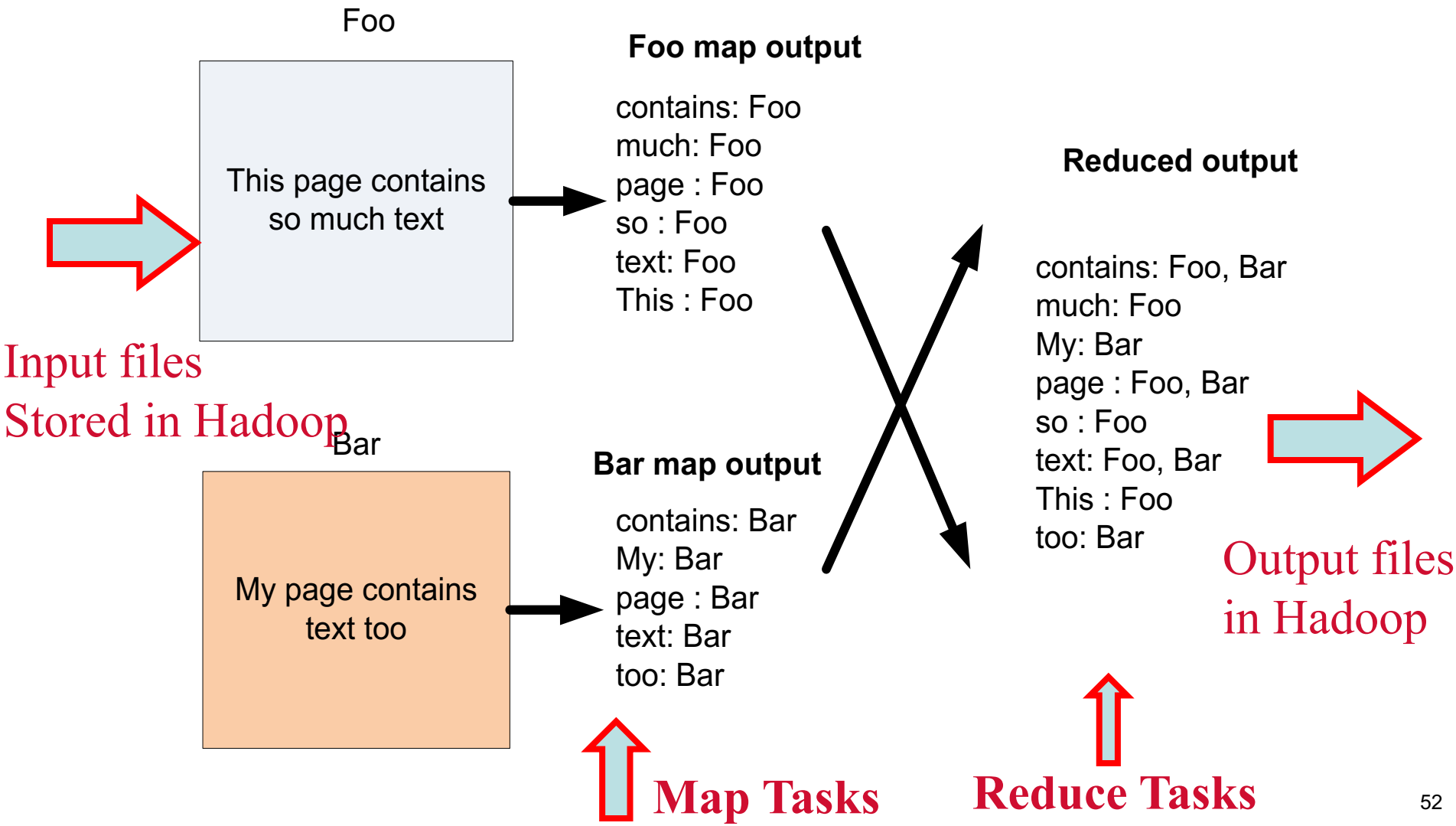
Document Frequency: Input Example

map() gets a key, value

- key - "bytes from the beginning of the line?“
- value - the current line;



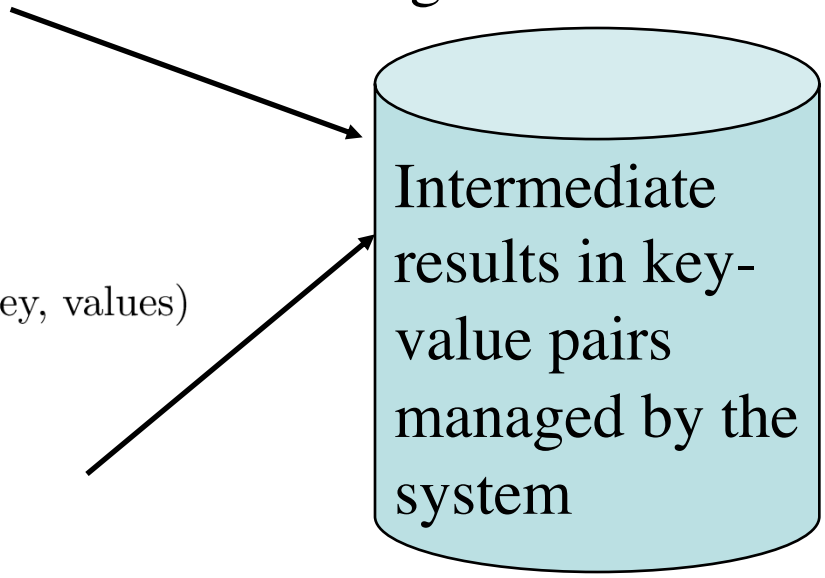
Inverted Indexing with Mapreduce



Pseudo code example for indexing with position information

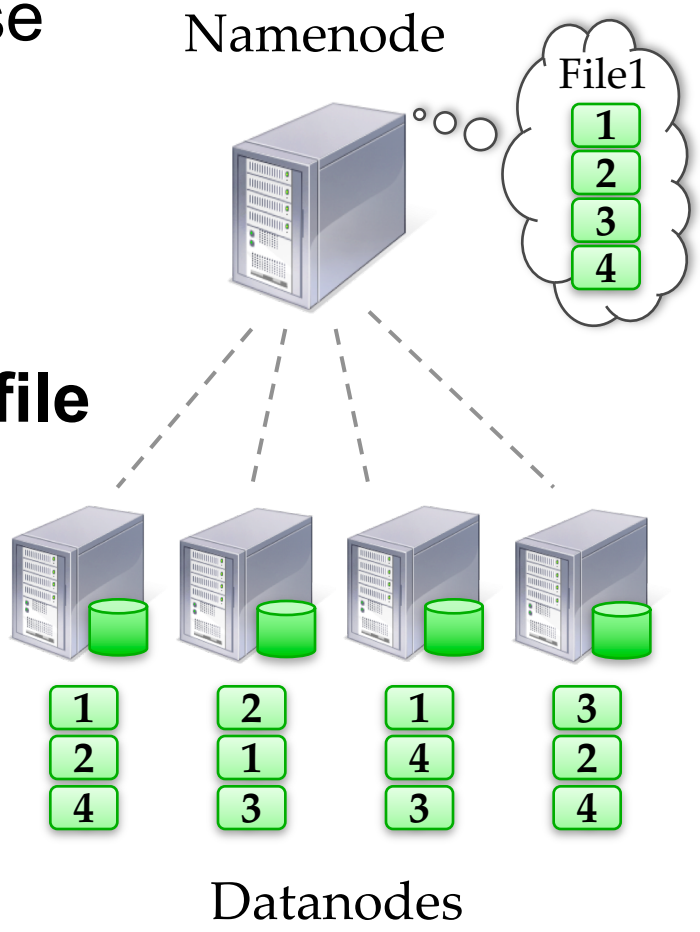
```
procedure MapDocumentsToPostings(input)
    while not input.done() do
        document ← input.next()
        number ← document.number
        position ← 0
        tokens ← Parse(document)
        for each word w in tokens do
            Emit(w, number:position)
            position = position + 1
        end for
    end while
end procedure


  procedure ReducePostingsToLists(key, values)
      word ← key
      WriteWord(word)
      while not input.done() do
          EncodePosting(values.next())
      end while
  end procedure
```

A user writes a small amount of code without worrying about inter-machine management



Hadoop Distributed File System

- **Standard file interface as Linux**
  - Open, seek, read, write, close
- **Files split into 64 MB blocks**
  - Blocks replicated across several datanodes (3)
- **Namenode stores metadata (file names, locations, etc)**
- **Files are append-only. Optimized for large files, sequential reads**
  - Read: use any copy
  - Write: append to 3 replicas



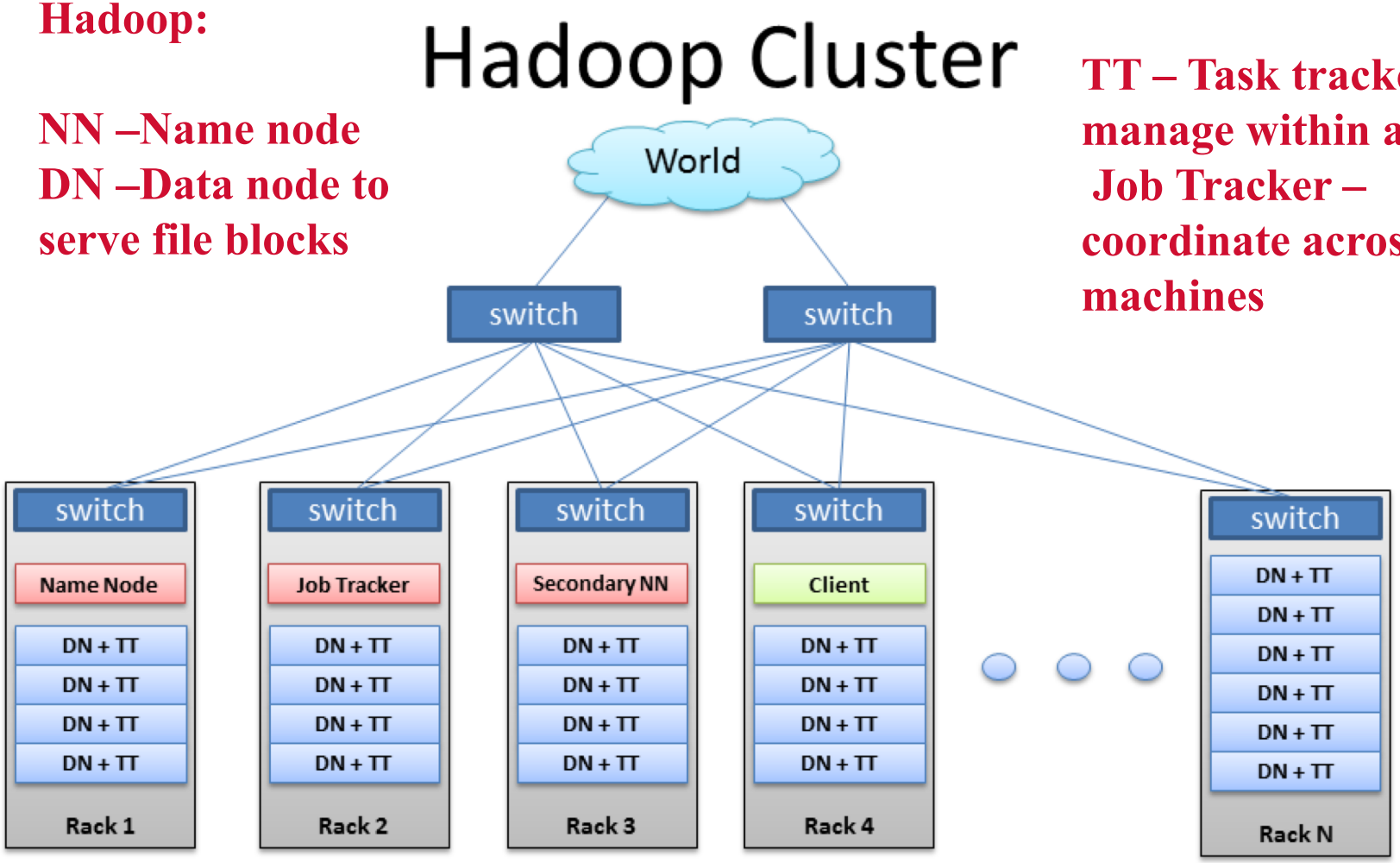
Hadoop Cluster with MapReduce

Daemons for Hadoop:

NN –Name node
DN –Data node to serve file blocks

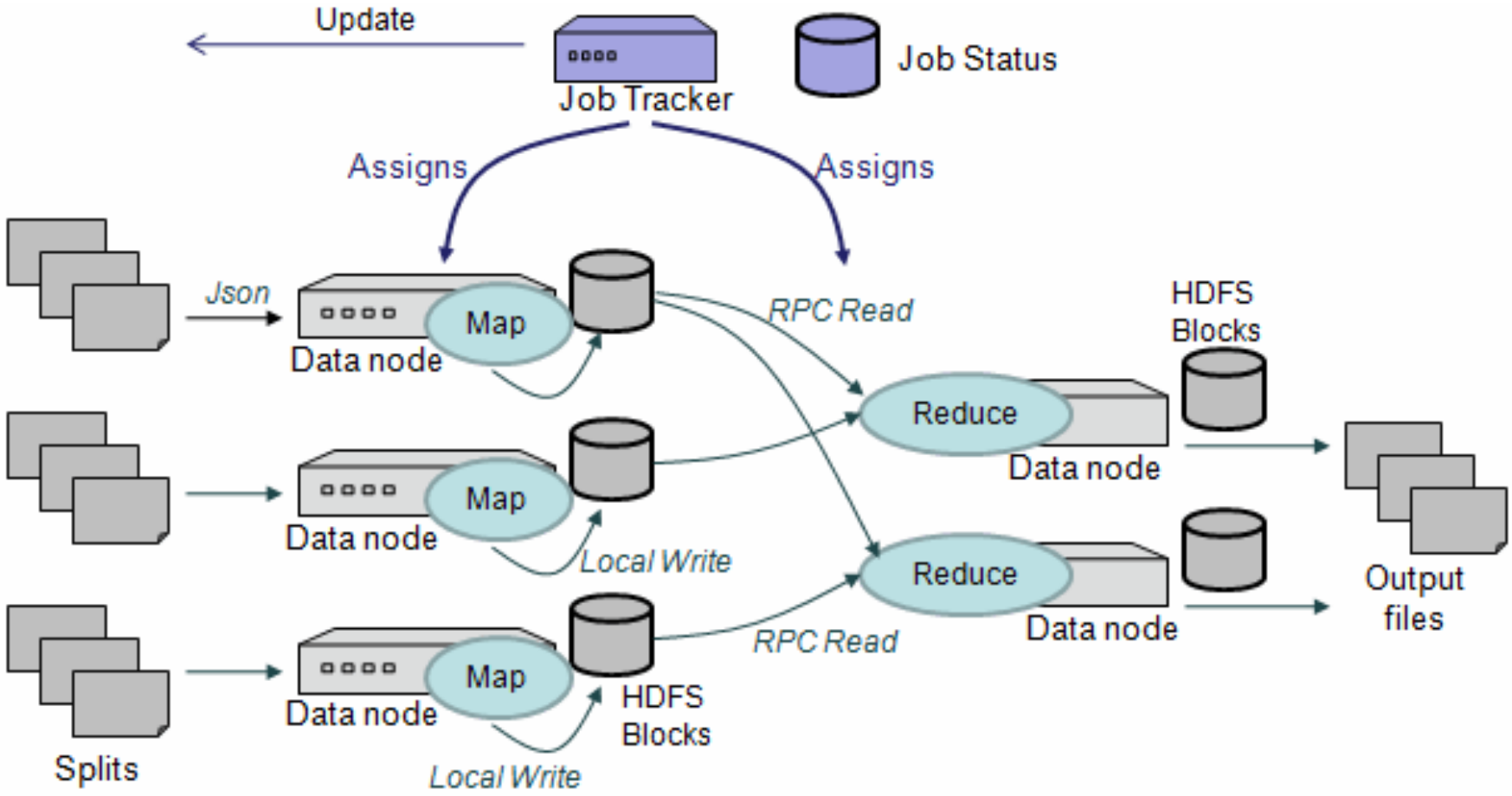
Daemons for MapReduce

TT – Task tracker to manage within a node
Job Tracker – coordinate across machines



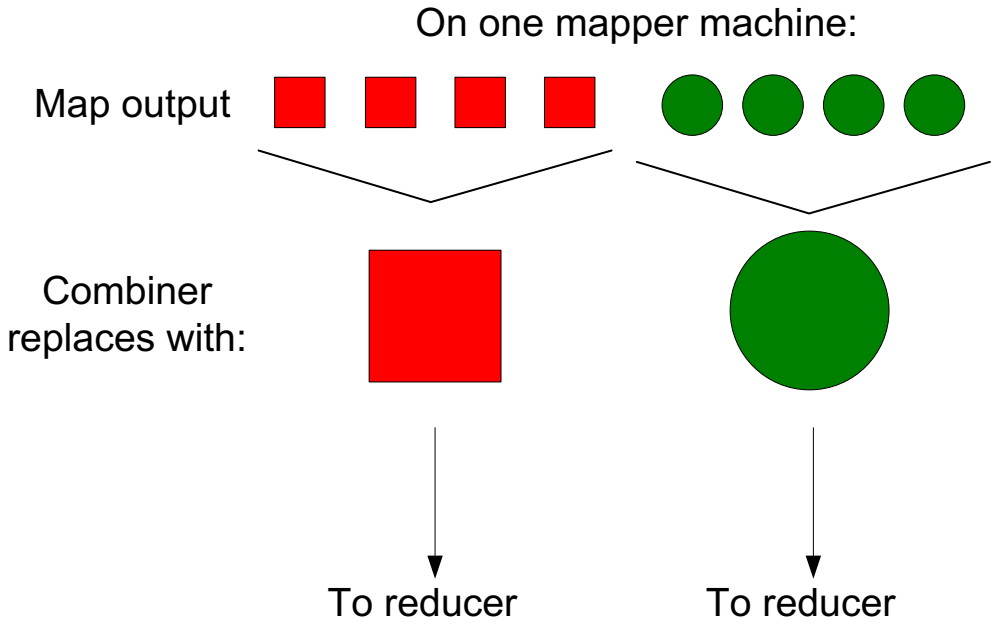
BRAD HEDLUND .com

Execute MapReduce on a cluster of machines with Hadoop DFS



User Code Optimization: Combining Phase

- Run on map machines after map phase
  - "Mini-reduce," only on local map output
  - E.g. job.setCombinerClass(Reduce.class);
- save bandwidth before sending data to full reduce tasks
- Requirement: commutative & associative



Types of MapReduce Applications

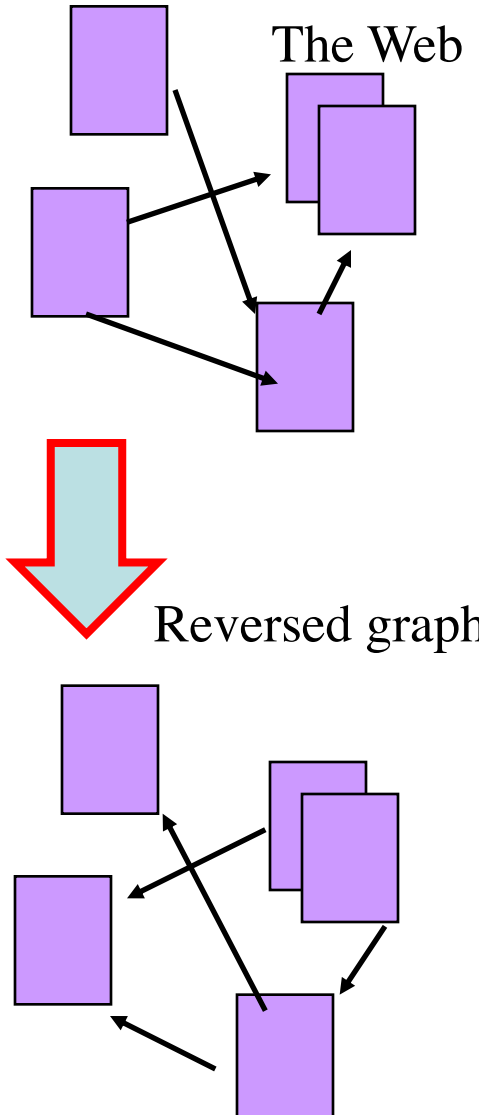
- **Map only parallel processing**
  - Count word usage for each document
- **Map-reduce two-stage processing**
  - Count word usage for the entire document collection
- **Multiple map-reduce stages**
  1. Count word usage in a document set
  2. Identify most frequent words in each document, but exclude those most popular words in the entire document

MapReduce Application: Examples

- **Distributed grep (search for words)**
  - *Map*: emit a line if it matches a given pattern
- **URL access frequency from many web access logs**
  - *Map*: process one log file of web page access; output frequency for each URL
  - *Reduce*: add all values for the same URL

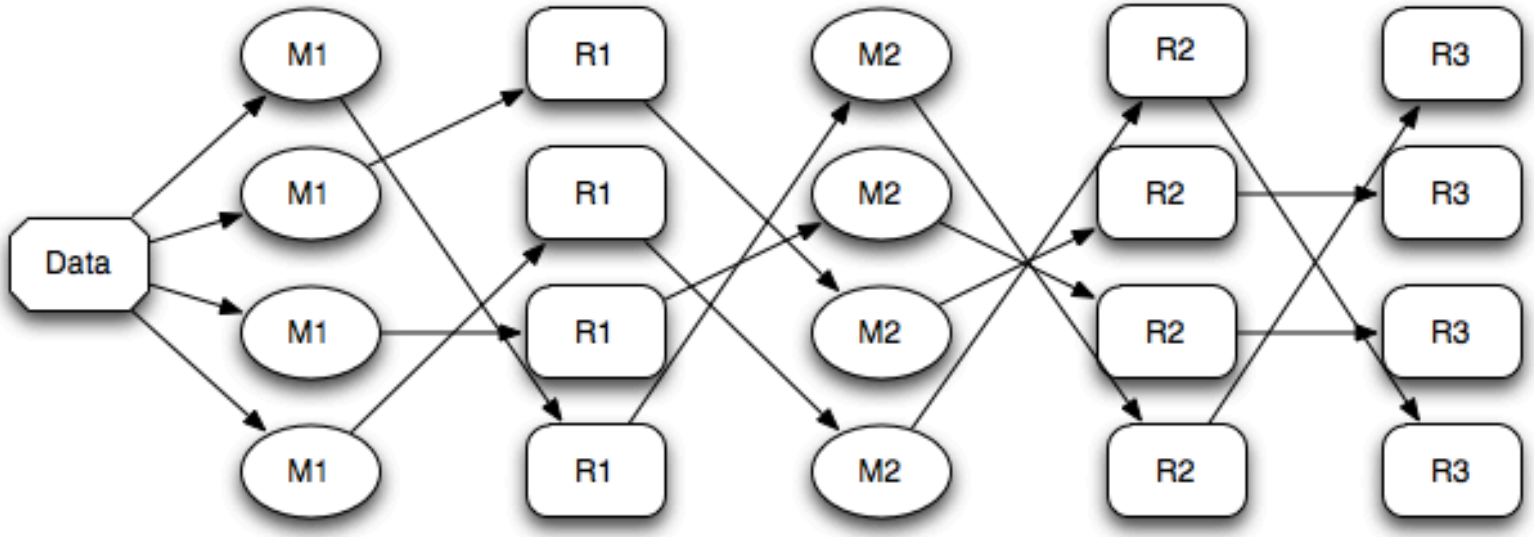
MapReduce Applications: Build a large graph for computing PageRank

- **Input:** a set of web pages and their outgoing links
- **Output:** Reversed web-link graph
  - A set of web page IDs and their incoming links.
- **Parallel code**
  - *Map*: Input is a web page containing outgoing links. Output each link with the target URL as a key.
  - *Reduce*: Concatenate the list of all source pages associated with a target URL



MapReduce Job Chaining

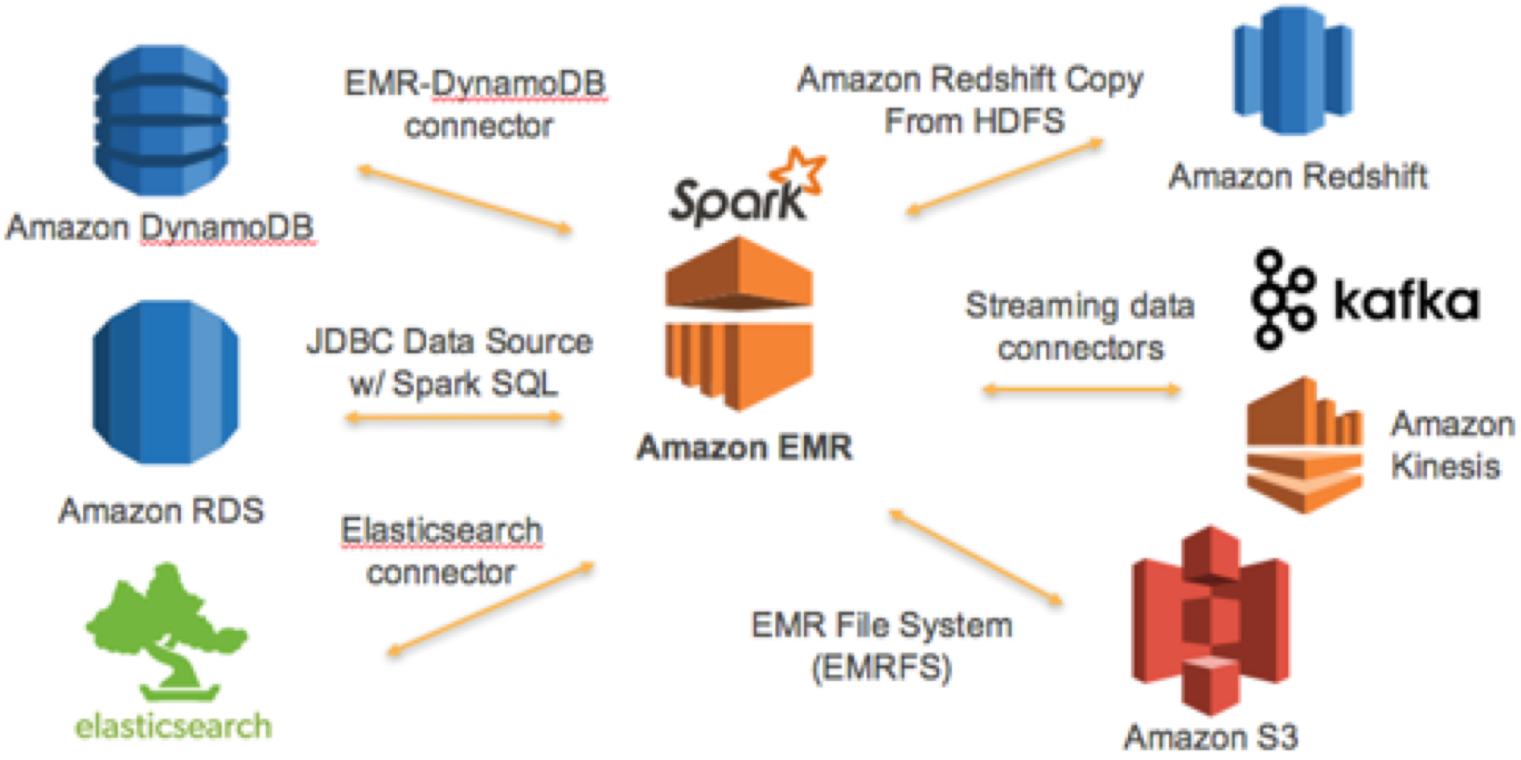
- Run a sequence of map-reduce jobs



Spark and Amazon EMR

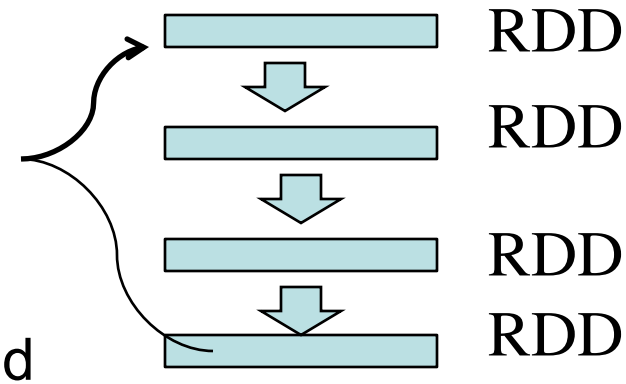
Spark: Berkeley design of Mapreduce programming supported in Python, Scala, & Java

Amazon EMR is a managed service for Hadoop and Spark to run large analytic jobs on an Amazon cluster



Mapreduce programming with SPAK: key concept

Write programs in terms of **operations** on implicitly distributed **datasets** (RDD)



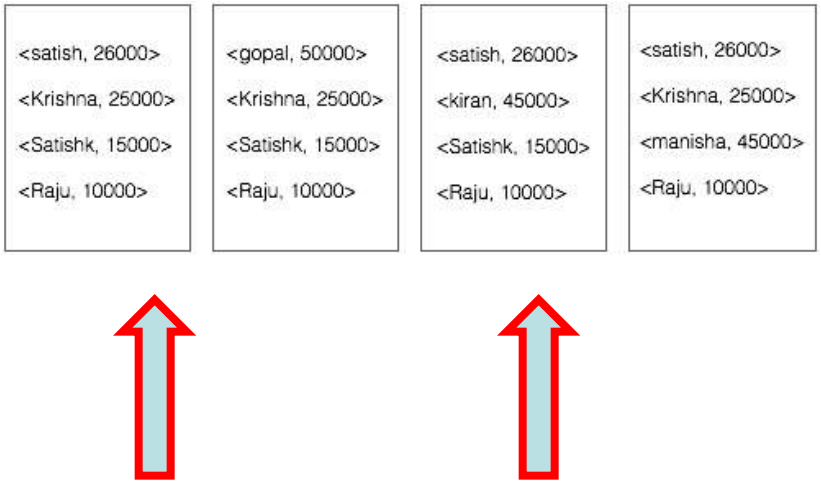
RDD: Resilient Distributed Datasets

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

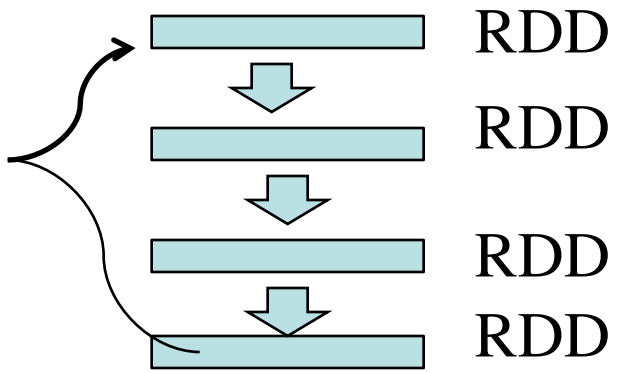
Operations

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

MapReduce vs Spark



Map and reduce tasks operate on key-value pairs

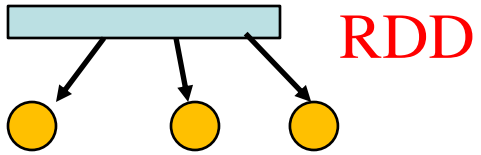


Spark operates on **RDD** with aggressive memory caching

Spark Context and Creating RDDs

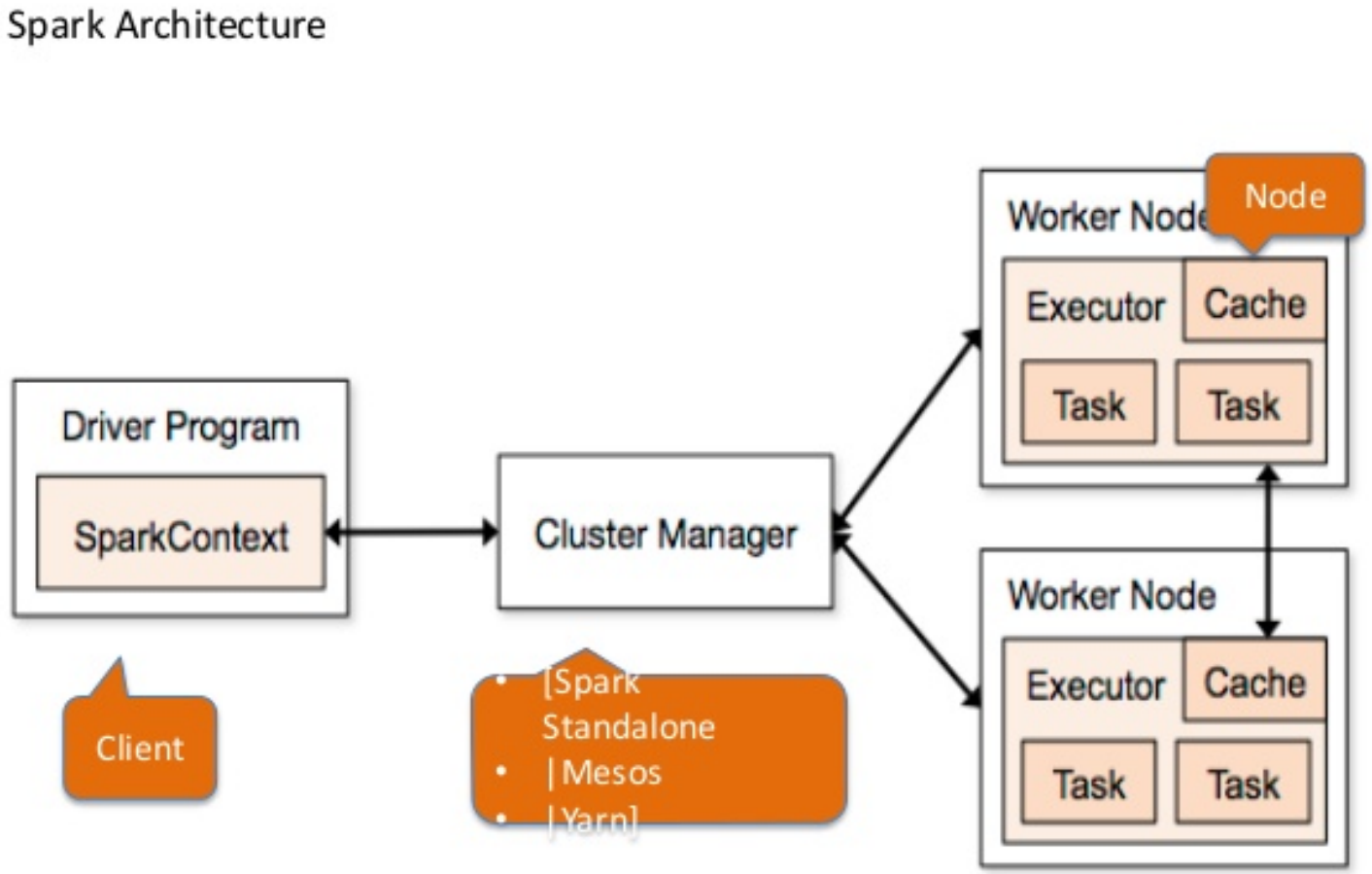
```
#Start with sc – SparkContext as
```
Main entry point to Spark functionality

```
#Turn a Python collection into an RDD
>sc.parallelize([1, 2, 3])
```

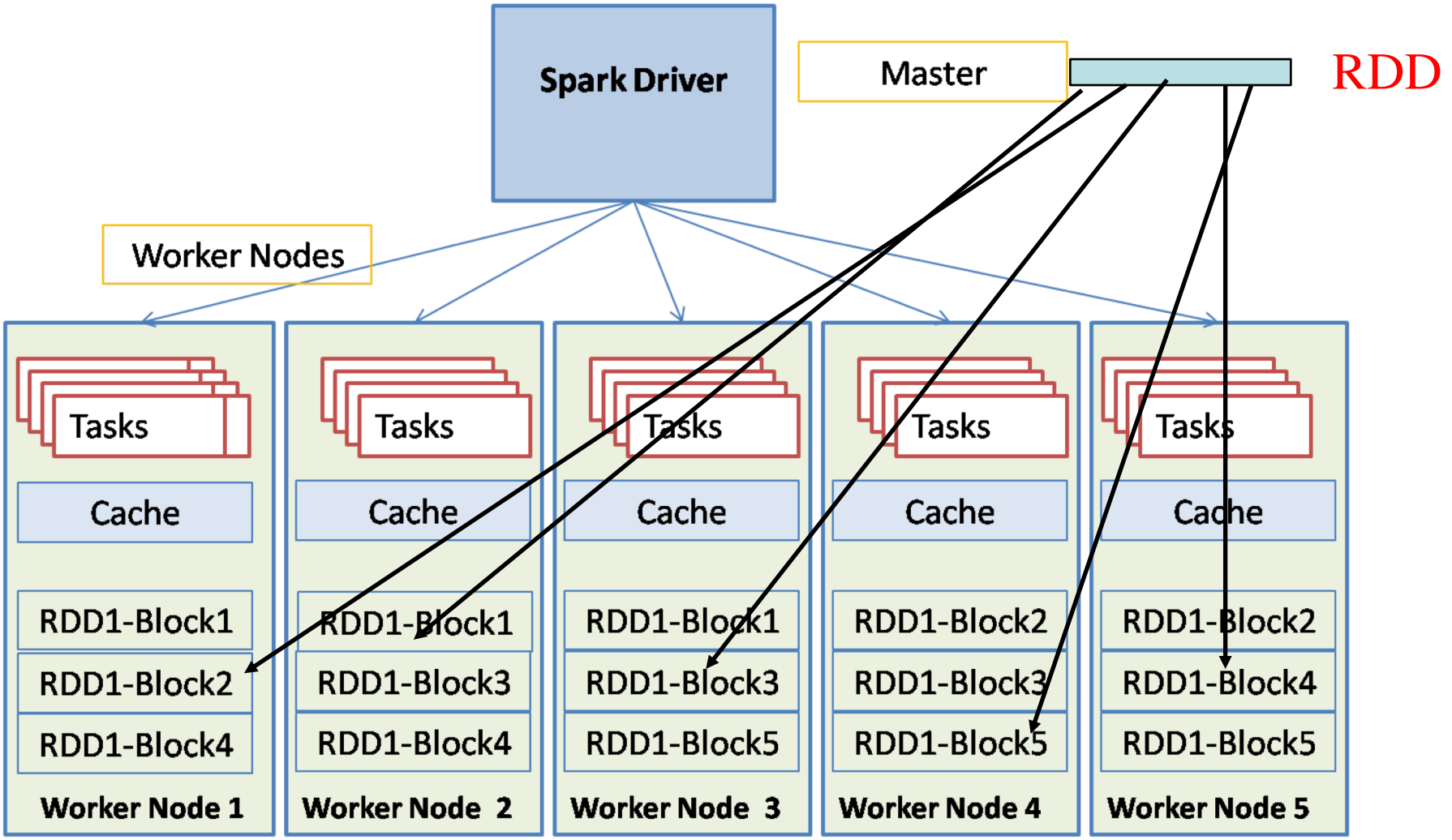



```
# Load text file from local FS, HDFS, or S3
>sc.textFile("file.txt")
>sc.textFile("directory/*.txt")
>sc.textFile("hdfs://namenode:9000/path/file")
```

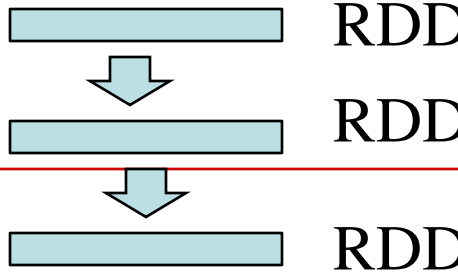
Spark Architecture



Spark Components



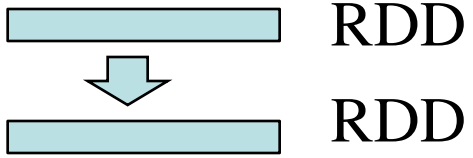
Basic Transformations




```
#read a text file and count number of lines containing error

lines = sc.textFile("file.log")
lines.filter(lambda s: "ERROR" in s).count()
```

```
> nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
> squares = nums.map(lambda x: x*x)   // {1, 4, 9}

# Keep elements passing a predicate
> even = squares.filter(lambda x: x % 2 == 0) // {4}
```

Basic Actions




```
> nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
> nums.collect() # => [1, 2, 3]

# Return first K elements
> nums.take(2)   # => [1, 2]

# Count number of elements
> nums.count()   # => 3

# Merge elements with an associative function
> nums.reduce(lambda x, y: x + y)  # => 6

# Write elements to a text file
> nums.saveAsTextFile("hdfs://file.txt")
```

Working with Key-Value Pairs

Spark's "distributed reduce" transformations operate on RDDs of key-value pairs

Python:

```
pair = (a, b)
       pair[0] # => a
                  pair[1] # => b
```

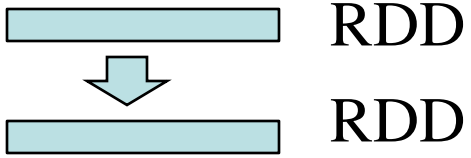
Scala:

```
val pair = (a, b)
                  pair._1 // => a
                  pair._2 // => b
```

Java:

```
Tuple2 pair = new Tuple2(a, b);
                  pair._1 // => a
                  pair._2 // => b
```

Some Key-Value Operations




```
> pets = sc.parallelize(
    [("cat", 1), ("dog", 1), ("cat", 2)])
> pets.reduceByKey(lambda x, y: x + y)
                        # => {(cat, 3), (dog, 1)}
> pets.groupByKey() # => {(cat, [1, 2]), (dog, [1])}
> pets.sortByKey()  # => {(cat, 1), (cat, 2), (dog, 1)}
```

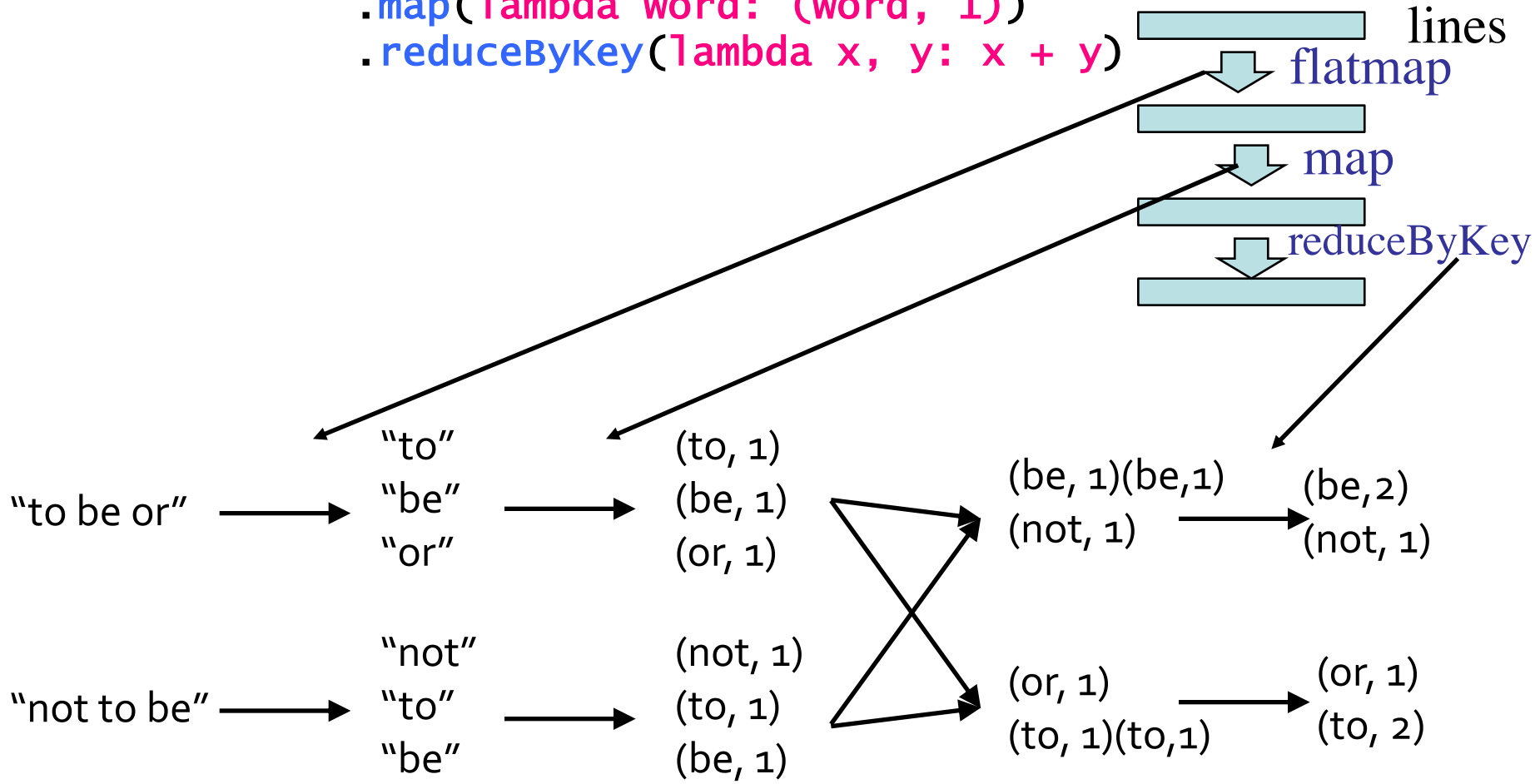
`reduceByKey()` also automatically implements combiners on the map side

Other Key-Value Operations

```
> visits = sc.parallelize([ (“index.html”, “1.2.3.4”),
                            (“about.html”, “3.4.5.6”),
                            (“index.html”, “1.3.3.1”) ])

> pageNames = sc.parallelize([ (“index.html”, “Home”),
                               (“about.html”, “About”) ])

> visits.join(pageNames)
  # (“index.html”, (“1.2.3.4”, “Home”))
  # (“index.html”, (“1.3.3.1”, “Home”))
  # (“about.html”, (“3.4.5.6”, “About”))

> visits.cogroup(pageNames)
  # (“index.html”, ([“1.2.3.4”, “1.3.3.1”], [“Home”]))
  # (“about.html”, ([“3.4.5.6”], [“About”]))
```

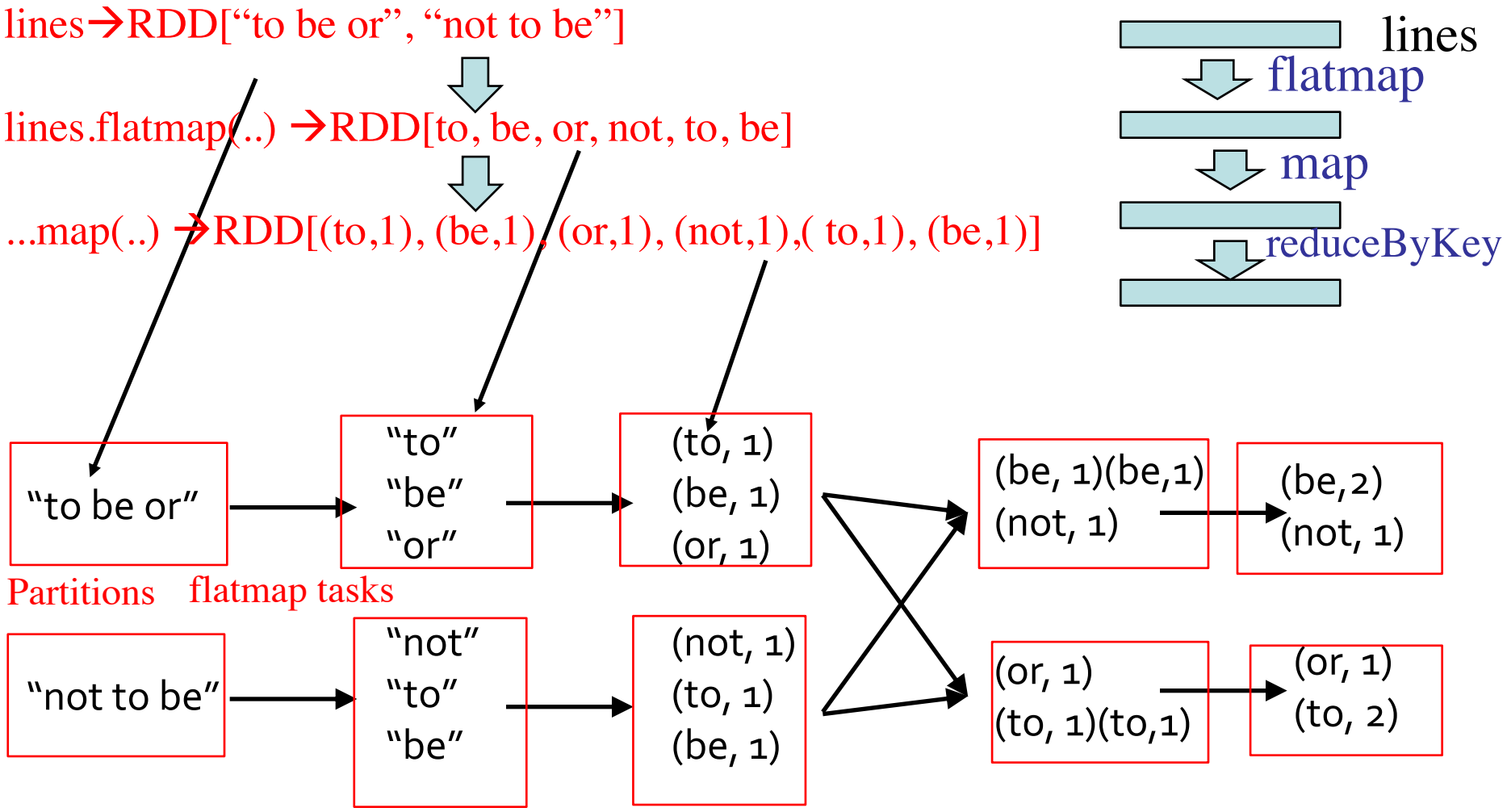
Example: Word Count

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



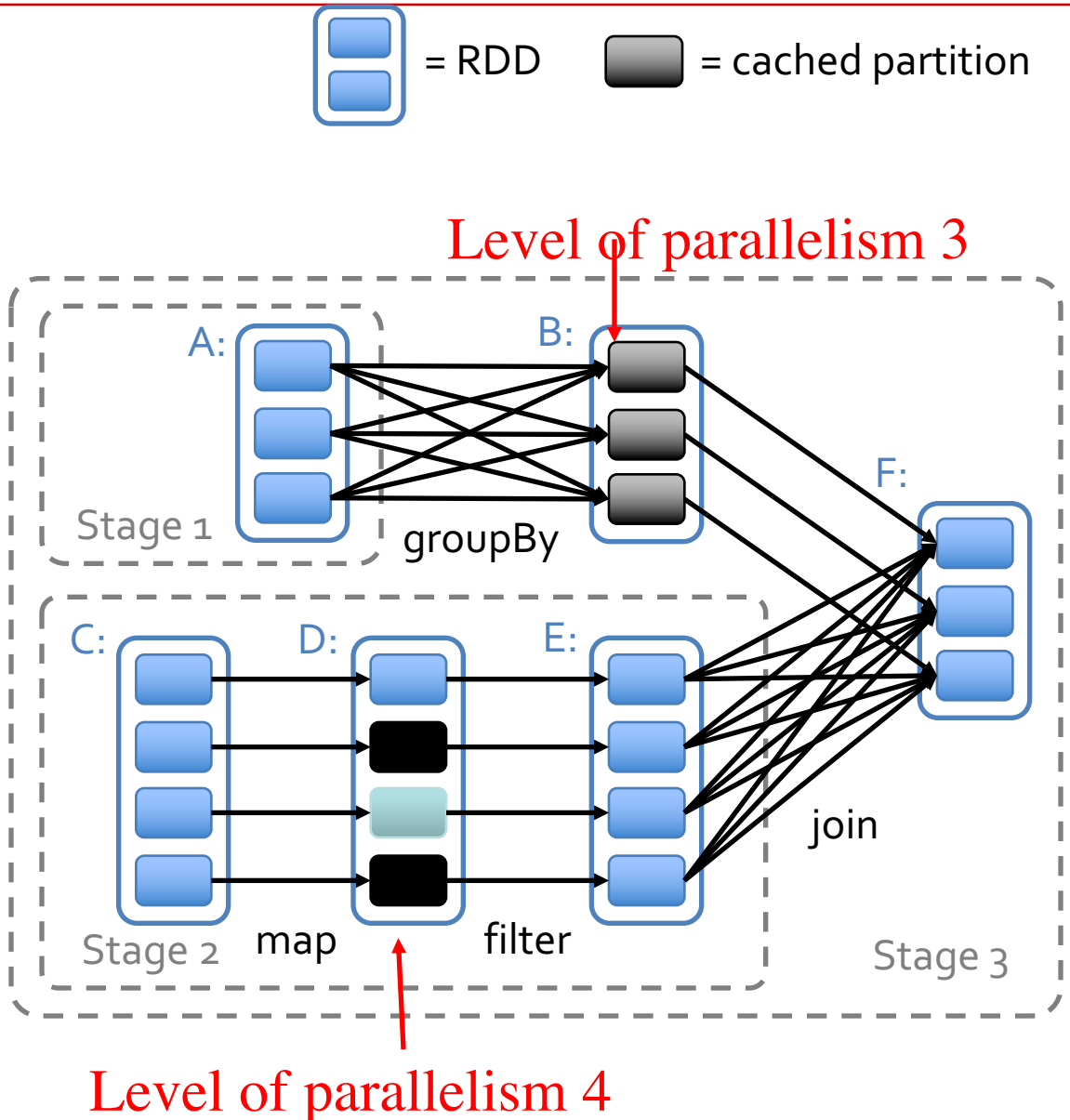
Partitioning and Parallel Tasks in Spark

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



Mapping and Scheduling of a Spark Task Graph

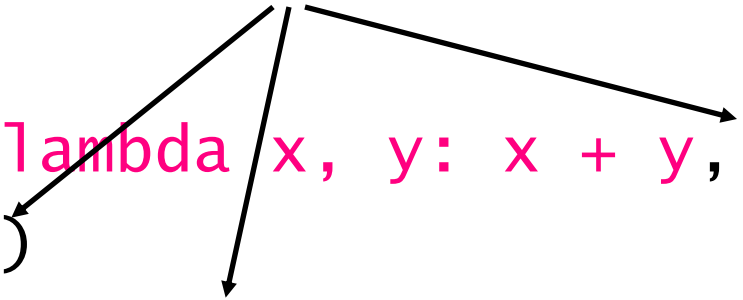
- RDD is partitioned and distributed among threads/machines
- Task computation is partitioning aware to avoid/minimize data shuffles
- Acyclic task graph structure
- Data flows through dependence pipelines
- Data is cached in memory as much as possible.
- Computation scheduling is data locality aware



Setting the Level of Parallelism

All the pair RDD operations take an optional second parameter for number of tasks

```
> words.reduceByKey(lambda x, y: x + y, 5)
> words.groupByKey(5)
> visits.join(pageViews, 5)
```



More RDD Operators

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

sample

take

first

partitionBy

mapWith

pipe

save ...

Summary

- **Offline incremental data processing**
  - All kinds of text mining and data transformation
    - Indexing, duplicate removal, content classification, spam analysis
  - Combine information from different sources
    - Web pages, entity/knowledge graph, link data, click data, database tables
- **Offline architectures and infrastructure**
  - Flow control for large system components
    - Pipeline, incremental update, 24x7 support
  - Examples of system software for parallel/distributed processing
    - Key-value stores, Map-Reduce, Spark