

# NoSQL Concepts & Techniques, HDFS and MapReduce

Valentina Ivanova

IDA, Linköping University

# Outline

- RDBMS → NoSQL → NewSQL
  - NoSQL Concepts and Techniques
    - Horizontal scalability
    - Consistency models
      - CAP theorem: BASE vs ACID
    - Consistent hashing
    - Vector clocks
  - NoSQL Systems - Types and Applications
  - Hadoop Distributed File System - HDFS
  - MapReduce Programming Model
-

# DB rankings – September 2016

2016-11-30

3

Rank			DBMS	Database Model	Score		
Sep 2016	Aug 2016	Sep 2015			Sep 2016	Aug 2016	Sep 2015
1.	1.	1.	Oracle	Relational DBMS	1425.56	-2.16	-37.81
2.	2.	2.	MySQL <span>+</span>	Relational DBMS	1354.03	-3.01	+76.28
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1211.55	+6.51	+113.72
4.	↑ 5.	↑ 5.	PostgreSQL	Relational DBMS	316.35	+1.10	+30.18
5.	↓ 4.	↓ 4.	MongoDB <span>+</span>	Document store	316.00	-2.49	+15.43
6.	6.	6.	DB2	Relational DBMS	181.19	-4.70	-27.95
7.	7.	↑ 8.	Cassandra <span>+</span>	Wide column store	130.49	+0.26	+2.89
8.	8.	↓ 7.	Microsoft Access	Relational DBMS	123.31	-0.74	-22.68
9.	9.	9.	SQLite	Relational DBMS	108.62	-1.24	+0.97
10.	10.	10.	Redis	Key-value store	107.79	+0.47	+7.14
11.	11.	↑ 14.	Elasticsearch <span>+</span>	Search engine	96.48	+3.99	+24.93
12.	12.	↑ 13.	Teradata	Relational DBMS	73.06	-0.57	-1.20
13.	13.	↓ 11.	SAP Adaptive Server	Relational DBMS	69.16	-1.88	-17.36
14.	14.	↓ 12.	Solr	Search engine	66.96	+1.19	-14.98
15.	15.	15.	HBase	Wide column store	57.81	+2.30	-1.22
16.	16.	↑ 17.	FileMaker	Relational DBMS	55.35	+0.34	+4.35
17.	17.	↑ 18.	Splunk	Search engine	51.29	+2.38	+9.06
18.	18.	↓ 16.	Hive	Relational DBMS	48.82	+1.01	-4.71
19.	19.	19.	SAP HANA <span>+</span>	Relational DBMS	43.42	+0.68	+5.22
20.	20.	↑ 25.	MariaDB	Relational DBMS	38.53	+1.65	+14.31
21.	21.	21.	Neo4j <span>+</span>	Graph DBMS	36.37	+0.80	+2.83
22.	↑ 24.	↑ 24.	Couchbase <span>+</span>	Document store	28.54	+1.14	+2.28
23.	23.	↓ 22.	Memcached	Key-value store	28.43	+0.74	-3.99
24.	↓ 22.	↓ 20.	Informix	Relational DBMS	28.19	-0.86	-9.76
25.	25.	↑ 28.	Amazon DynamoDB <span>+</span>	Document store	27.42	+0.82	+7.43

<http://db-engines.com/en/ranking>

RDBMS → NoSQL → NewSQL

# DBMS history (Why NoSQL?)

- 1960 – Navigational databases
- 1970 – Relational databases (RDBMS)
- 1990 – Object-oriented databases and Data Warehouses
- 2000 – XML databases
- Mid 2000 – first NoSQL
- 2011 – NewSQL

# RDBMS

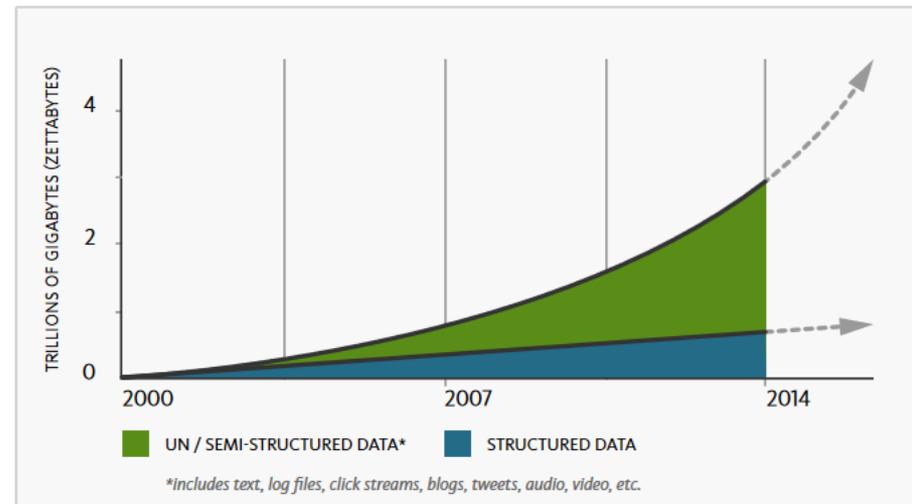
- Established technology
- Transactions support & ACID properties
- Powerful query language - SQL
- Experiences administrators
- Many vendors

Table: Item

item id	name	color	size
45	skirt	white	L
65	dress	red	M

# But ... – One Size Does Not Fit All

- Requirements have changed:
  - Frequent schema changes, management of unstructured and semi-structured data
  - Huge datasets
  - RDBMSs are not designed to be
    - distributed
    - continuously available
  - High read and write scalability
  - Different applications have different requirements<sup>[1]</sup>



[1] “One Size Fits All”: An Idea Whose Time Has Come and Gone [https://cs.brown.edu/~ugur/fits\\_all.pdf](https://cs.brown.edu/~ugur/fits_all.pdf)

Figure from: <http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/NoSQL-Whitepaper.pdf>

# NoSQL (not-only-SQL)

- A broad category of disparate solutions
- Simple and flexible non-relational data models
  - schema-on-read vs schema-on-write
- High availability & relax data consistency requirement (CAP theorem)
  - BASE vs ACID
- Easy to distribute – horizontal scalability
  - data are replicated to multiple nodes
- Cheap & easy (or not) to implement (open source)

## But ...

- No support for SQL → Low level programming → data analysts need to write custom programs
- No ACID
- Huge investments already made in SQL systems and experienced developers
- NoSQL systems do not provide interfaces to existing tools

# NewSQL<sup>[DataMan]</sup>

- First mentioned in 2011
- Supports the relational model
  - with horizontal scalability & fault tolerance
- Query language - SQL
- ACID
- Different data representation internally
- VoltDB, NuoDB, Clustrix, Google Spanner

# NewSQL Applications<sup>[DataMan]</sup>

- RBDMS applicable scenarios
  - transaction and manipulation of more than one object, e.g., financial applications
  - strong consistency requirements, e.g., financial applications
  - schema is known in advance and unlikely to change a lot
- But also Web-based applications<sup>[1]</sup>
  - with different collection of OLTP requirements
    - multi-player games, social networking sites
  - real-time analytics (vs traditional business intelligence requests)

---

[1] <http://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext>

# NoSQL Concepts and Techniques

# NoSQL Databases (not only SQL)

*nosql-database.org*

## **NoSQL Definition:**

Next Generation Databases mostly addressing some of the points: being ***non-relational, distributed, open source*** and ***horizontally scalable***.

The original intention has been modern web-scale databases. ... Often more characteristics apply as: ***schema-free, easy replication support, simple API, eventually consistent/BASE*** (not ACID), a ***huge data amount***, and more.

# NoSQL: Concepts

**Scalability:** system can handle growing amounts of data without losing performance.

- **Vertical Scalability (scale up)**
  - add resources (more CPUs, more memory) to a single node
  - using more threads to handle a local problem
- **Horizontal Scalability (scale out)**
  - add nodes (more computers, servers) to a distributed system
  - gets more and more popular due to low costs for commodity hardware
  - often surpasses scalability of vertical approach

# Distributed (Data Management) Systems

- Number of processing nodes interconnected by a computer network
- Data is stored, replicated, updated and processed across the nodes
- Networks failures are given, not an exception
  - Network is partitioned
  - Communication between nodes is an issue
  - Data consistency vs Availability

# Consistency models<sup>[Vogels]</sup>

- A distributed system through the developers' eyes
  - Storage system as a black box
  - Independent processes that write and read to the storage
- Strong consistency – after the update completes, any subsequent access will return the updated value.
- Weak consistency – the system does not guarantee that subsequent accesses will return the updated value.
  - inconsistency window

# Consistency models<sup>[Vogels]</sup>

- Weak consistency
  - Eventual consistency – if no new updates are made to the object, eventually all accesses will return the last updated value
    - Popular example: DNS

# Consistency models<sup>[Vogels]</sup>

- Server side view of a distributed system – Quorum
  - $N$  – number of nodes that store replicas
  - $R$  – number of nodes for a successful read
  - $W$  – number of nodes for a successful write

# Consistency models<sup>[Vogels]</sup>

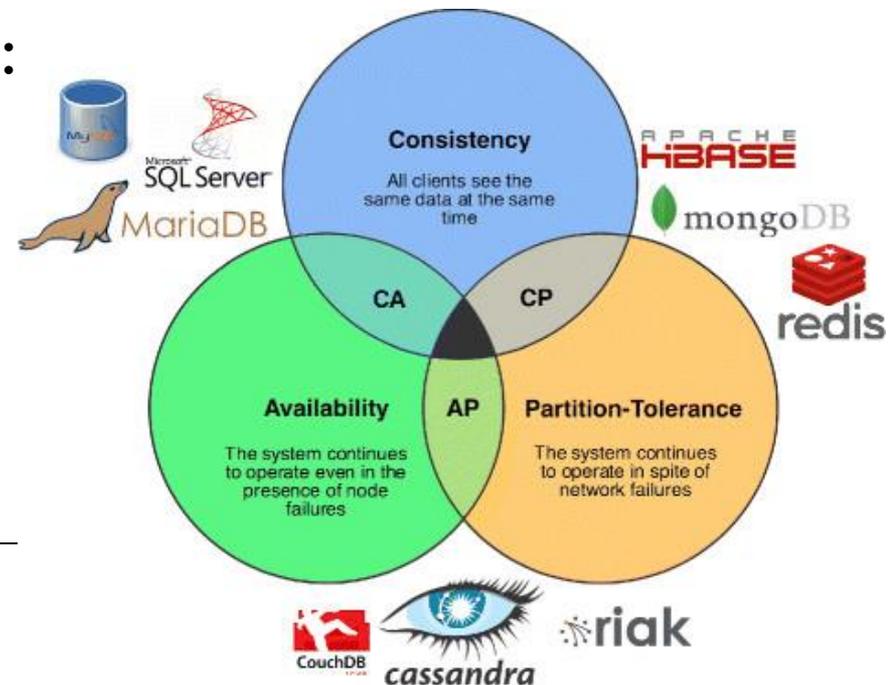
- Server side view of a distributed system – Quorum
  - High read loads – hundreds of N,  $R=1$
  - Fault tolerance/availability (& relaxed consistency)  $W=1$
  - $R + W > N$  strong consistency
    - Consistency (& reduced availability)  $W=N$
  - $R + W \leq N$  eventual consistency
    - Inconsistency window – the period until all replicas have been updated in a lazy manner

# NoSQL: Concepts

## *CAP Theorem: Consistency, Availability, Partition Tolerance* [Brewer]

### Theorem

(Gilbert, Lynch SIGACT'2002):  
only 2 of the 3 guarantees  
can be given in a shared-data  
system.



# NoSQL: Concepts

## ***CAP Theorem: Consistency, Availability, Partition Tolerance***<sup>[Brewer]</sup>

- **Consistency**

- after an update, all readers in a distributed system see the same data
- all nodes are supposed to contain the same data at all times

- **Example**

- single database instance will always be consistent
- if multiple instances exist, all writes must be duplicated before write operation is completed

# NoSQL: Concepts

## ***CAP Theorem: Consistency, Availability, Partition Tolerance***<sup>[Brewer]</sup>

- **Availability**
  - all requests will be answered, regardless of crashes or downtimes
- **Example**
  - a single instance has an availability of 100% or 0%, two servers may be available 100%, 50%, or 0%

# NoSQL: Concepts

## ***CAP Theorem: Consistency, Availability, Partition Tolerance***<sup>[Brewer]</sup>

- **Partition Tolerance**

- system continues to operate, even if two sets of servers get isolated

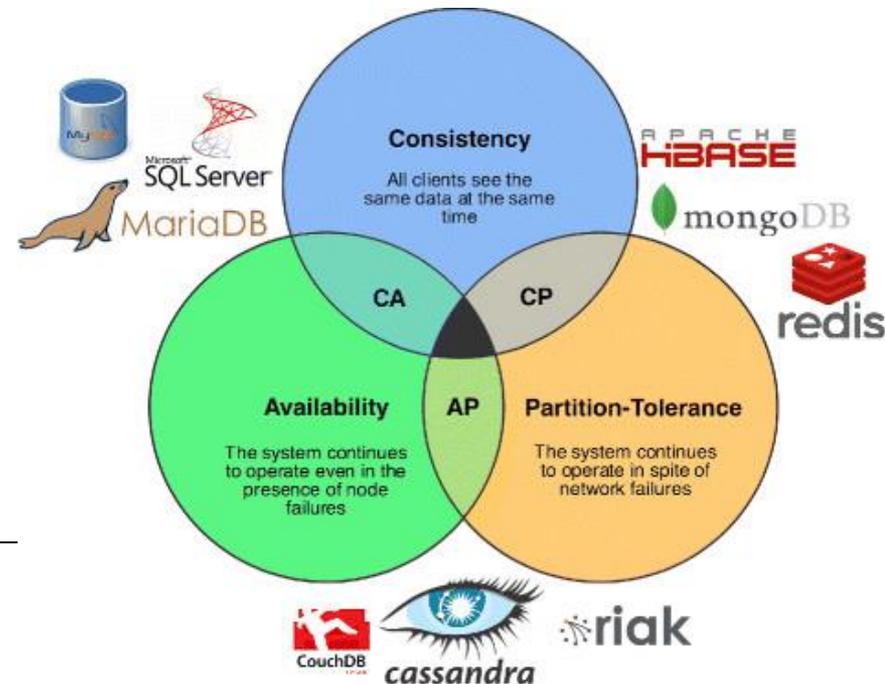
- **Example**

- system gets partitioned if connection between server clusters fails
- failed connection will not cause troubles if system is tolerant

# NoSQL: Concepts

## ***CAP Theorem: Consistency, Availability, Partition Tolerance***<sup>[Brewer]</sup>

- (Positive) consequence: we can concentrate on two challenges
- **ACID** properties needed to guarantee consistency and availability
- **BASE** properties come into play if availability and partition tolerance is favored



# NoSQL: Concepts

## **ACID: Atomicity, Consistency, Isolation, Durability**

- **A**tomicity → all operations in a transaction will complete, or none will
- **C**onsistency → before and after the transaction, the database will be in a consistent state
- **I**solation → operations cannot access data that is currently modified
- **D**urability → data will not be lost upon completion of a transaction

# NoSQL: Concepts

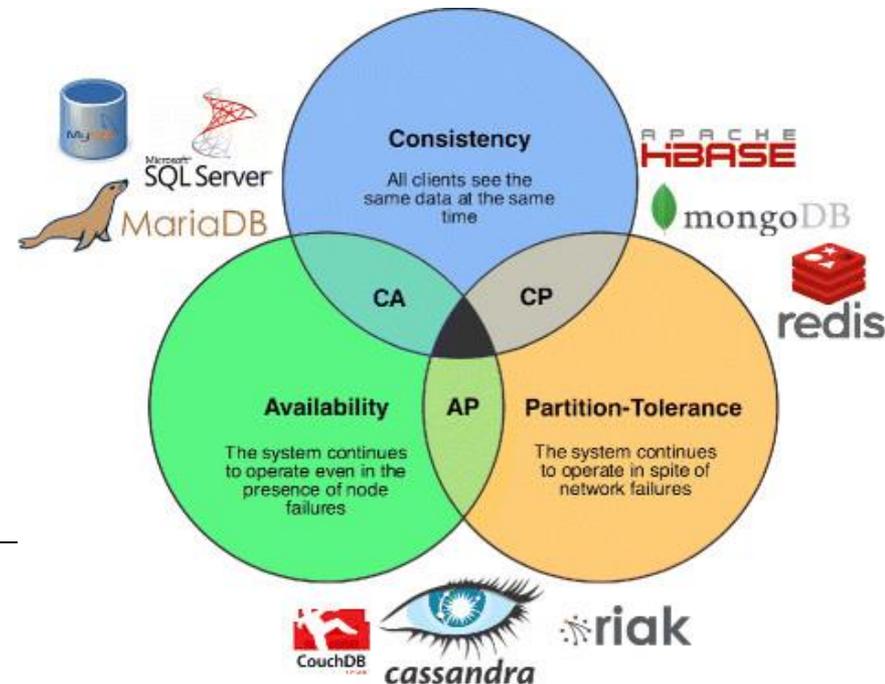
## **BASE: Basically Available, Soft State, Eventual Consistency** [Fox]

- **Basically Available** → an application works basically all the time (despite partial failures)
- **Soft State** → is in flux and non-deterministic (changes all the time)
- **Eventual Consistency** → will be in some consistent state (at some time in future)

# NoSQL: Concepts

## ***CAP Theorem: Consistency, Availability, Partition Tolerance***<sup>[Brewer]</sup>

- (Positive) consequence: we can concentrate on two challenges
- **ACID** properties needed to guarantee consistency and availability
- **BASE** properties come into play if availability and partition tolerance is favored



# NoSQL: Techniques

Basic techniques (widely applied in NoSQL systems)

- distributed data storage, replication (how to distribute the data) → Consistent hashing
- distributed query strategy (horizontal scalability) → MapReduce (later in this lecture)
- recognize order of distributed events and potential conflicts → Vector clock

# NoSQL: Techniques – Consistent Hashing [Karger]

## Task

- find machine that stores data for a specified key  $k$
- trivial hash function to distribute data on  $n$  nodes:  
 $h(k; n) = k \bmod n$
- if number of nodes changes, all data will have to be redistributed!

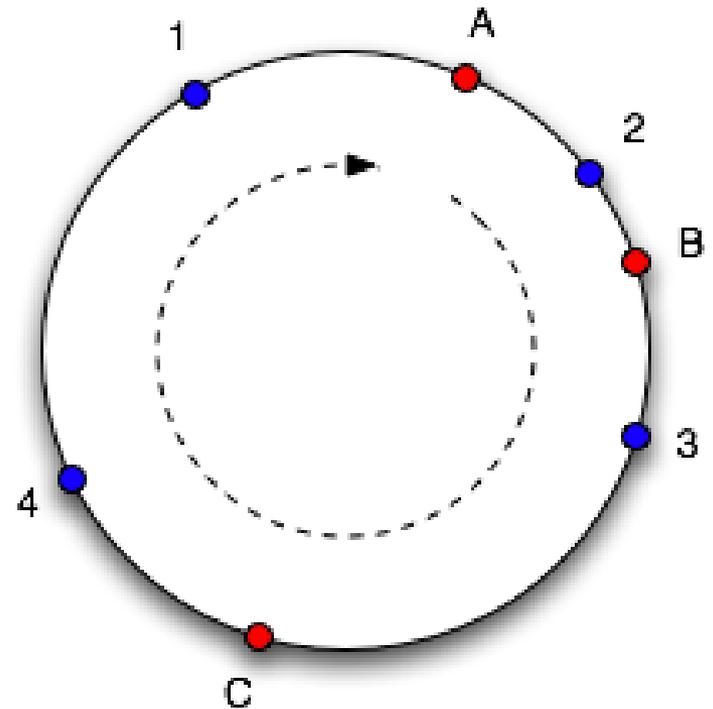
## Challenge

- minimize number of nodes to be updated after a configuration change
- incorporate hardware characteristics into hashing model

# NoSQL: Techniques – Consistent Hashing [Karger]

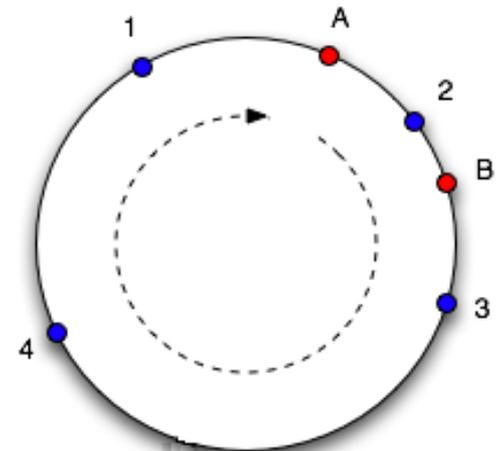
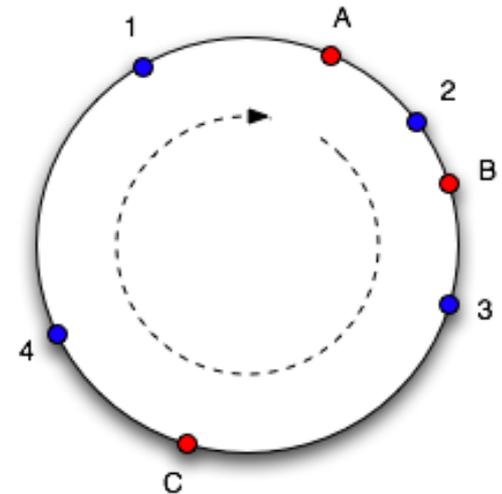
## Basic idea

- arrange the nodes in a ring and each node is in charge of the hash values in the range between its neighbor node
- include hash values of all nodes in hash structure
- calculate hash value of the key to be added/retrieved
- choose node which occurs next clockwise in the ring



# NoSQL: Techniques – Consistent Hashing [Karger]

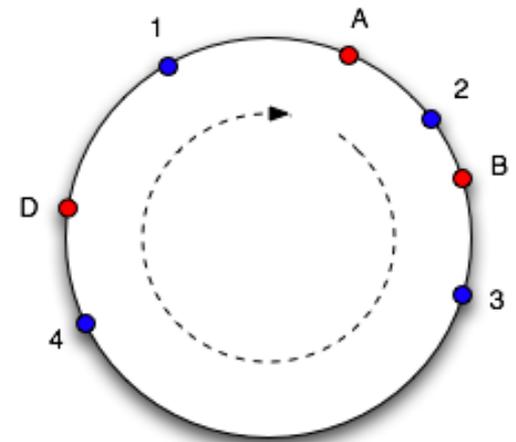
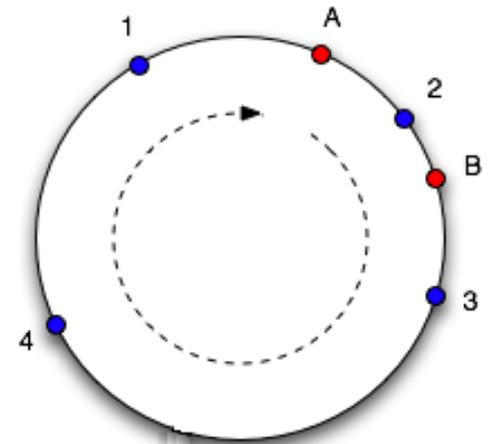
- include hash values of all nodes in hash structure
- calculate hash value of the key to be added/retrieved
- choose node which occurs next clockwise in the ring
- if node is dropped or gets lost, missing data is redistributed to adjacent nodes (replication issue)



# NoSQL: Techniques – Consistent Hashing [Karger]

- if a new node is added, its hash value is added to the hash table
- the hash realm is repartitioned, and hash data will be transferred to new neighbor

→ no need to update remaining nodes!



# NoSQL: Techniques – Consistent Hashing [Karger]

- A replication factor  $r$  is introduced: not only the next node but the next  $r$  nodes in clockwise direction become responsible for a key
- Number of added keys can be made dependent on node characteristics (bandwidth, CPU, ...)

DeCandia et al. [ACM SIGOPS'2007], Dynamo:  
Amazon's Highly Available Key-value Store

# NoSQL: Techniques – Logical Time

## Challenge

- recognize order of distributed events and potential conflicts
- most obvious approach: attach timestamp (ts) of system clock to each

event  $e \rightarrow ts(e)$

→ error-prone, as clocks will never be fully synchronized

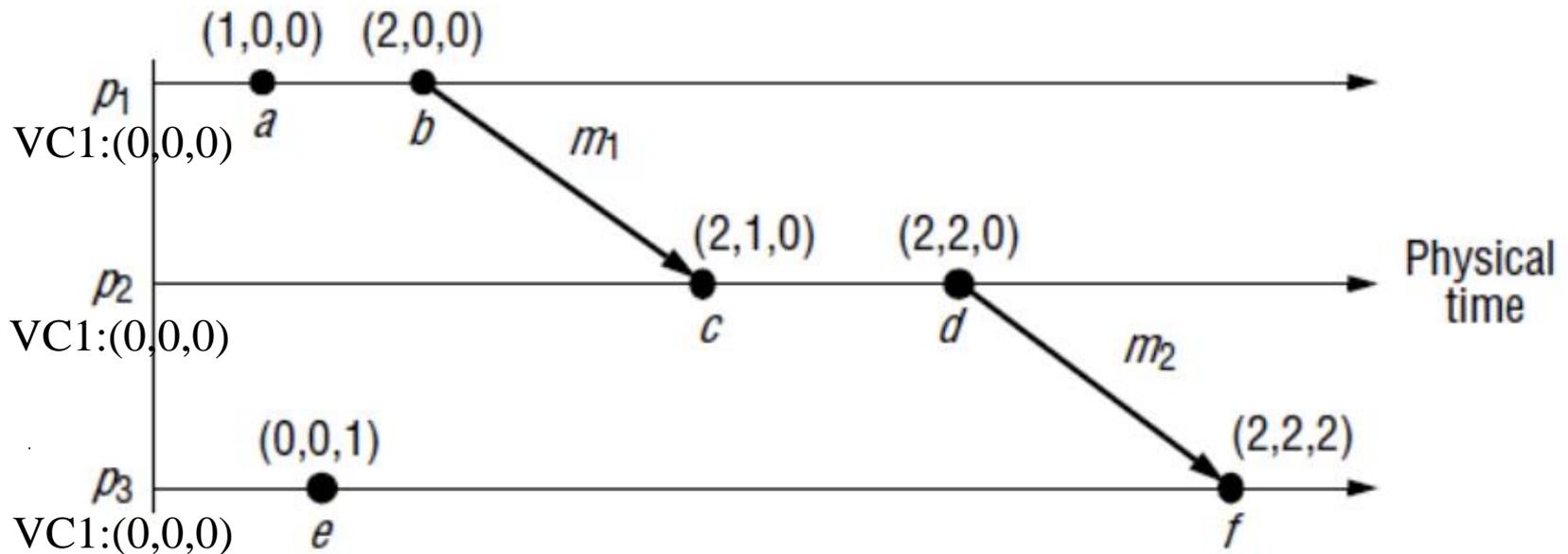
→ insufficient, as we cannot catch causalities (needed to detect conflicts)

# NoSQL: Techniques – Vector Clock<sup>[Coulouris]</sup>

- A vector clock for a system of  $N$  nodes is an array of  $N$  integers.
- Each process keeps its own vector clock,  $V_i$ , which it uses to timestamp local events.
- Processes piggyback vector timestamps on the messages they send to one another, and there are simple rules for updating the clocks:
  - VC1: Initially,  $V_i[j] = 0$ , for  $i, j = 1, 2, \dots, N$
  - VC2: Just before  $p_i$  timestamps an event, it sets  $V_i[i] := V_i[i] + 1$
  - VC3:  $p_i$  includes the value  $t = V_i$  in every message it sends
  - VC4: When  $p_i$  receives a timestamp  $t$  in a message, it sets  $V_i[j] := \max(V_i[j]; t[j])$ , for  $j = 1, 2, \dots, N$

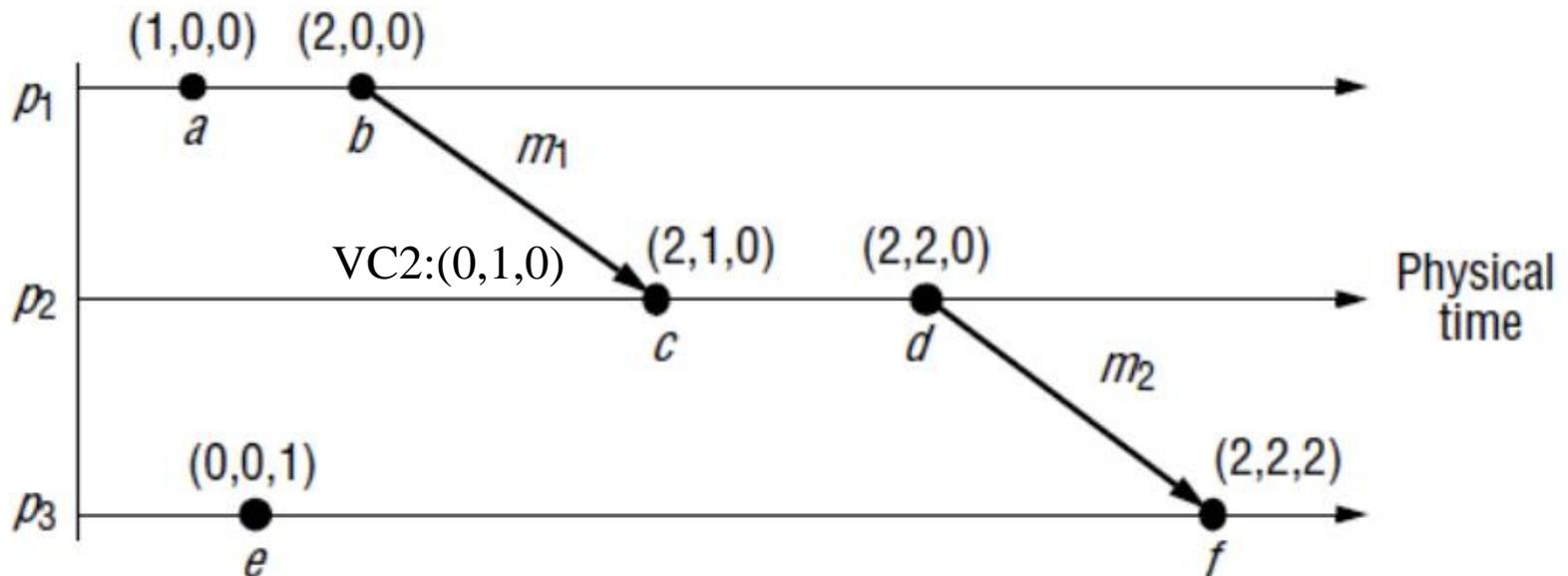
## NoSQL: Techniques – Vector Clock<sup>[Coulouris]</sup>

- VC1: Initially,  $V_i [j] = 0$ , for  $i, j = 1, 2, \dots, N$
- VC2: Just before  $p_i$  timestamps an event, it sets  $V_i [i] := V_i [i] + 1$



## NoSQL: Techniques – Vector Clock<sup>[Coulouris]</sup>

- VC3:  $p_i$  includes the value  $t = V_i$  in every message it sends
- VC4: When  $p_i$  receives a timestamp  $t$  in a message, it sets  $V_i [j] := \max(V_i [j]; t [j])$ , for  $j = 1, 2, \dots, N$



# NoSQL: Techniques – Vector Clock<sup>[Coulouris]</sup>

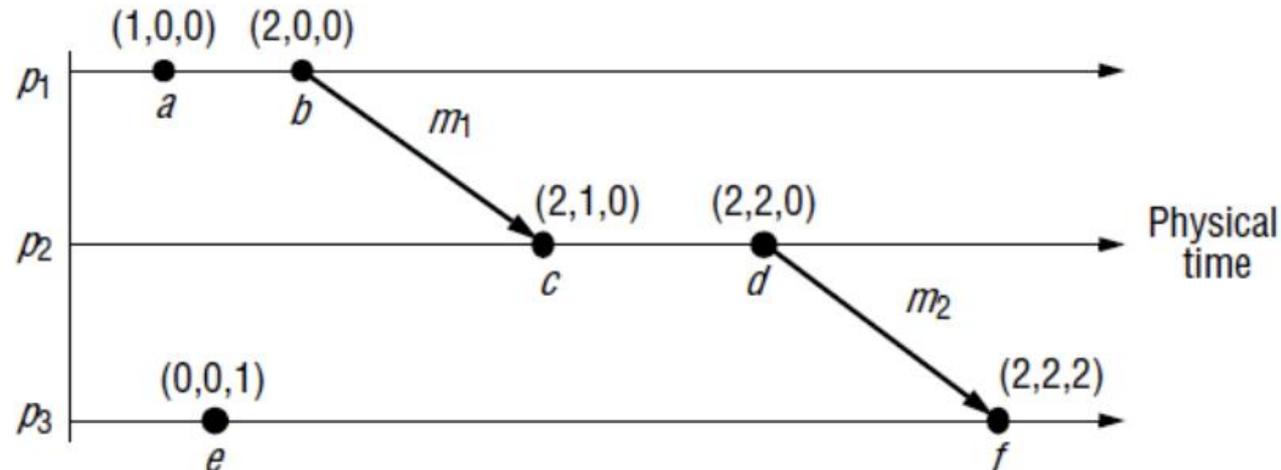
Properties:

- $V = V'$  iff  $V[j] = V'[j]$  for  $j = 1, 2, \dots, N$
- $V \leq V'$  iff  $V[j] \leq V'[j]$  for  $j = 1, 2, \dots, N$
- $V < V'$  iff  $V \leq V'$  and  $V \neq V'$

two events  $e$  and  $e'$ : that  $e \rightarrow e' \leftrightarrow V(e) < V(e')$

→ Conflict detection! ( $c \parallel e$  since neither  $V(c) \leq V(e)$  nor  $V(e) \leq V(c)$ )

$c$  &  $e$  are concurrent



# NoSQL Systems – Types and Applications

# NoSQL Data Models

- Key-Value Stores
- Document Stores
- Column-Family Stores
- Graph Databases
- Impacts application, querying, scalability

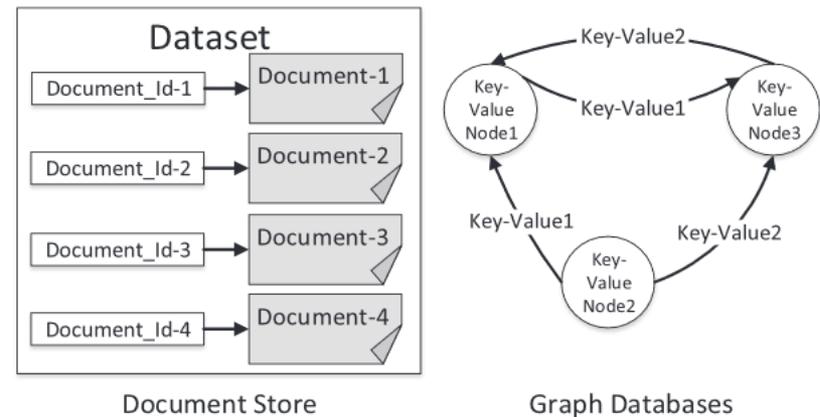
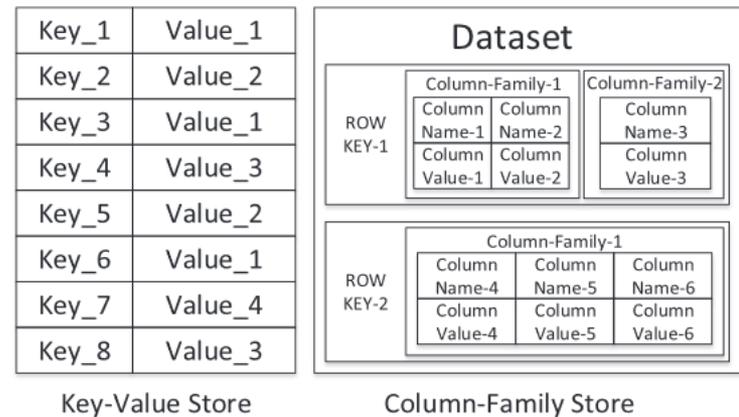


figure from [DataMan]

# DBs not referred as NoSQL

- Object DBs
- XML DBs
- Special purpose DBs
  - Stream processing

# Key-Value Stores<sup>[DataMan]</sup>

- Schema-free
  - Keys are unique
  - Values of arbitrary types
- Efficient in storing distributed data
- (very) Limited query facilities and indexing
  - get(key), put(key, value)
  - Value → opaque to the data store → no data level querying and indexing

Key_1	Value_1
Key_2	Value_2
Key_3	Value_1
Key_4	Value_3
Key_5	Value_2
Key_6	Value_1
Key_7	Value_4
Key_8	Value_3

Key-Value Store

# Key-Value Stores<sup>[DataMan]</sup>

- Types
  - In-memory stores – Memcached, Redis
  - Persistent stores – BerkeleyDB, Voldemort, RiakDB

Key_1	Value_1
Key_2	Value_2
Key_3	Value_1
Key_4	Value_3
Key_5	Value_2
Key_6	Value_1
Key_7	Value_4
Key_8	Value_3

Key-Value Store

- Not suitable for
  - structures and relations
  - accessing multiple items (since the access is by key and often no transactional capabilities)

# Key-Value Stores<sup>[DataMan]</sup>

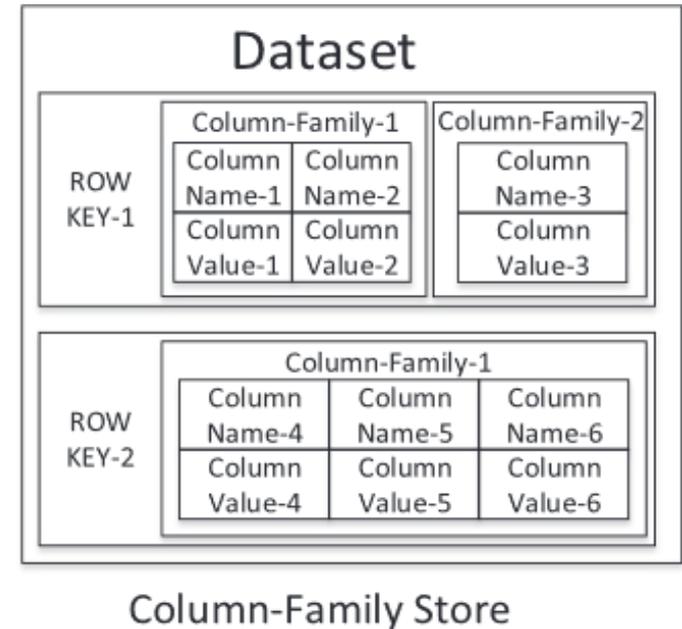
- Applications:
  - Storing web session information
  - User profiles and configuration
  - Shopping cart data
  - Using them as a caching layer to store results of expensive operations (create a user-tailored web page)

Key_1	Value_1
Key_2	Value_2
Key_3	Value_1
Key_4	Value_3
Key_5	Value_2
Key_6	Value_1
Key_7	Value_4
Key_8	Value_3

Key-Value Store

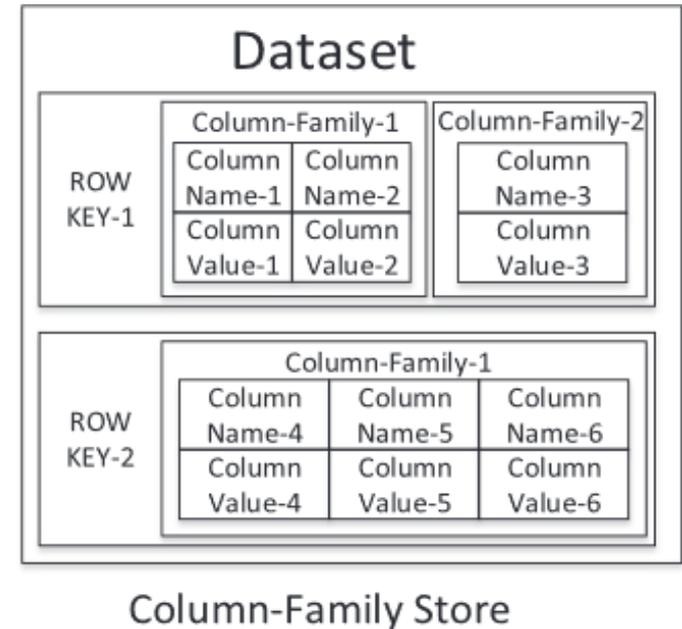
# Column-Family Stores<sup>[DataMan]</sup>

- Schema-free
  - Rows have unique keys
  - Values are varying column families and act as keys for the columns they hold
  - Columns consist of key-value pairs
- Better than key-value stores for querying and indexing



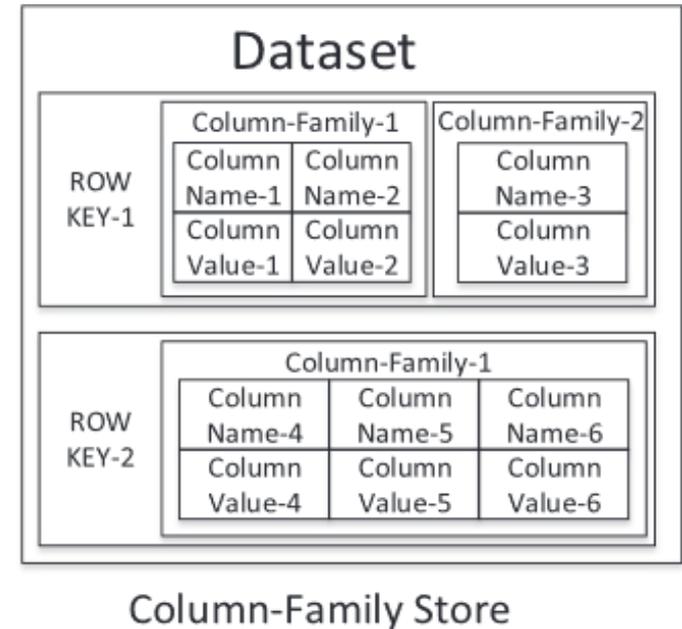
# Column-Family Stores<sup>[DataMan]</sup>

- Types
  - Googles BigTable, Hadoop HBase
  - No column families – Amazon SimpleDB, DynamoDB
  - Supercolumns - Cassandra
- Not suitable for
  - structures and relations



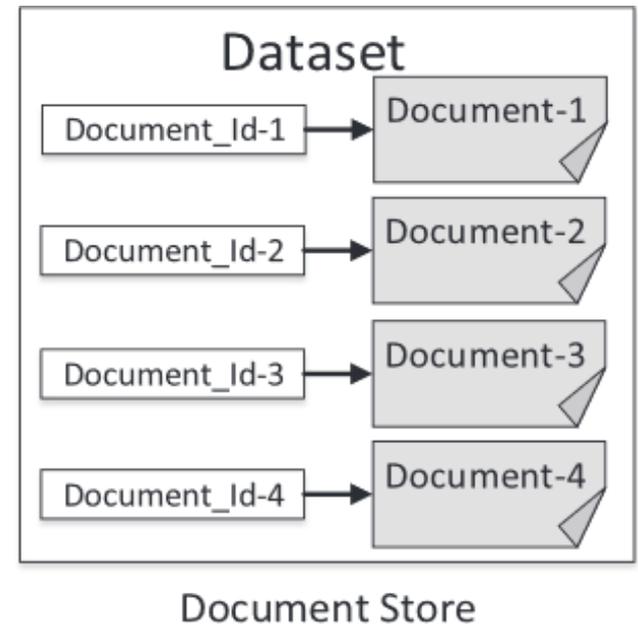
# Column-Family Stores<sup>[DataMan]</sup>

- Applications:
  - Document stores applications
  - Analytics scenarios – HBase and Cassandra
    - Web analytics
    - Personalized search
    - Inbox search



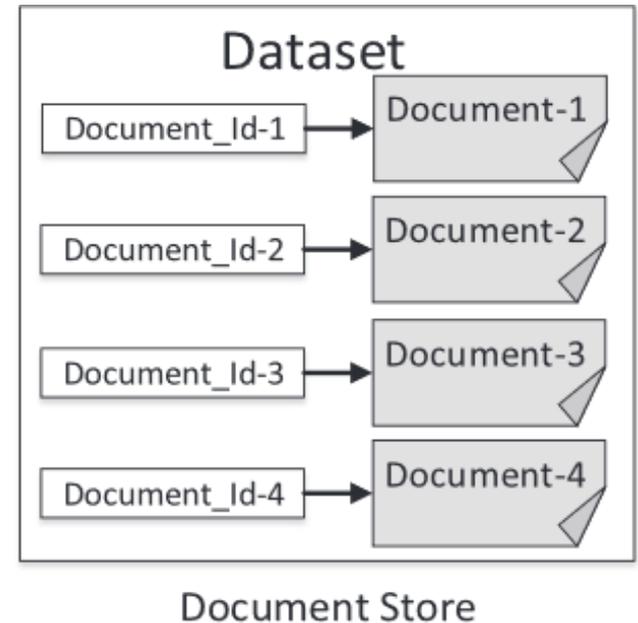
# Document Stores<sup>[DataMan]</sup>

- Schema-free
  - Keys are unique
  - Values are documents – complex (nested) data structures in JSON, XML, binary (BSON), etc.
- Indexing and querying based on primary key and content
- The content needs to be representable as a document
- MongoDB, CouchDB, Couchbase



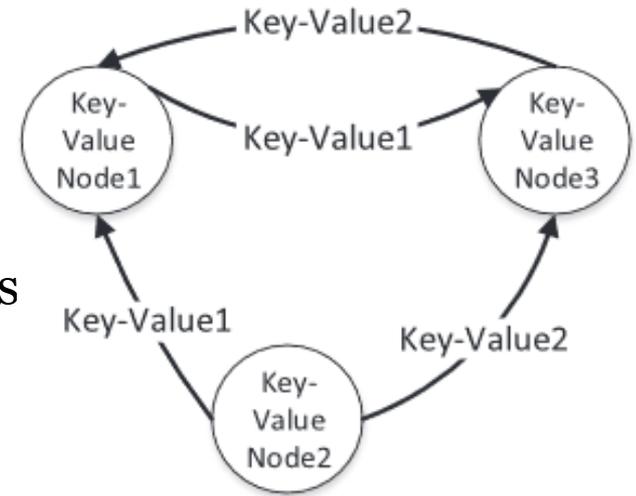
# Document Stores<sup>[DataMan]</sup>

- Applications:
  - Items with similar nature but different structure
  - Blogging platforms
  - Content management systems
  - Event logging
  - Fast application development



# Graph Databases<sup>[DataMan]</sup>

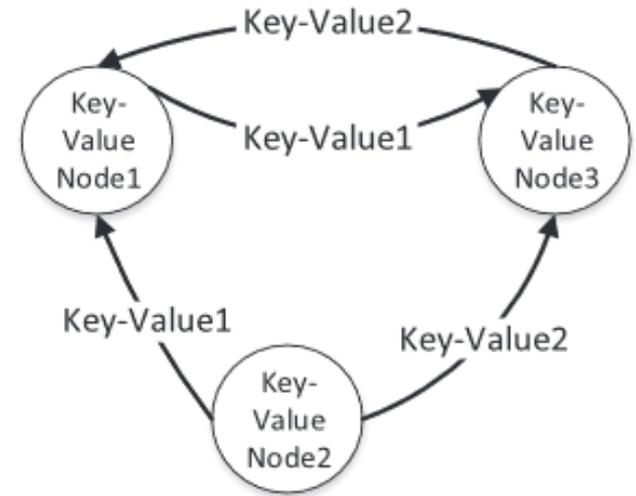
- Graph model
  - Nodes/vertices and links/edges
  - Properties consisting of key-value pairs
- Suitable for very interconnected data since they are efficient in traversing relationships
- Not as efficient
  - as other NoSQL solutions for non-graph applications
  - horizontal scaling
- Neo4J, HyperGraphDB



Graph Databases

# Graph Databases[DataMan]

- Applications:
  - location-based services
  - recommendation engines
  - complex network-based applications
    - social, information, technological, and biological network
  - memory leak detection



Graph Databases

# Visual Guide to NoSQL Systems

**Availability:**  
Each client can  
always read  
and write.

**A**

**Data Models**

Relational (comparison)  
Key-Value  
Column-Oriented/Tabular  
Document-Oriented

**CA**

RDBMSs  
(MySQL,  
Postgres,  
etc)

Aster Data  
Greenplum  
Vertica

**AP**

Dynamo  
Voldemort  
Tokyo Cabinet  
KAI

Cassandra  
SimpleDB  
CouchDB  
Riak

**Pick Two**

**C**

**Consistency:**  
All clients always  
have the same view  
of the data.

**CP**

BigTable  
Hypertable  
Hbase

MongoDB  
Terrastore  
Scalaris

Berkeley DB  
MemcacheDB  
Redis

**P**

**Partition Tolerance:**  
The system works  
well despite physical  
network partitions.

# HDFS[Hadoop][HDFS][HDFSpaper] Hadoop Distributed File System



# Compute Nodes<sup>[Massive]</sup>

- Compute node – processor, main memory, cache and local disk
- Organized into racks
- Intra-rack connection typically gigabit speed
- Inter-rack connection slower by a small factor



# HDFS (Hadoop Distributed File System)

- Runs on top of the native file system
  - Files are very large divided into 128 MB chunks/blocks
    - To minimize the cost of seeks
  - Caching blocks is possible
  - Single writer, multiple readers
  - Exposes the locations of file blocks via API
  - Fault tolerance and availability to address disk/node failures
    - Usually replicated three times on different compute nodes
- Based on GFS (Google File System - proprietary)

# HDFS is Good for ...

- Store very large files – GBs and TBs
- Streaming access
  - Write-once, read many times
  - Time to read the entire dataset is more important than the latency in reading the first record.
- Commodity hardware
  - Clusters are built from commonly available hardware
  - Designed to continue working without a noticeable interruption in case of failure

# HDFS is currently Not Good for ...

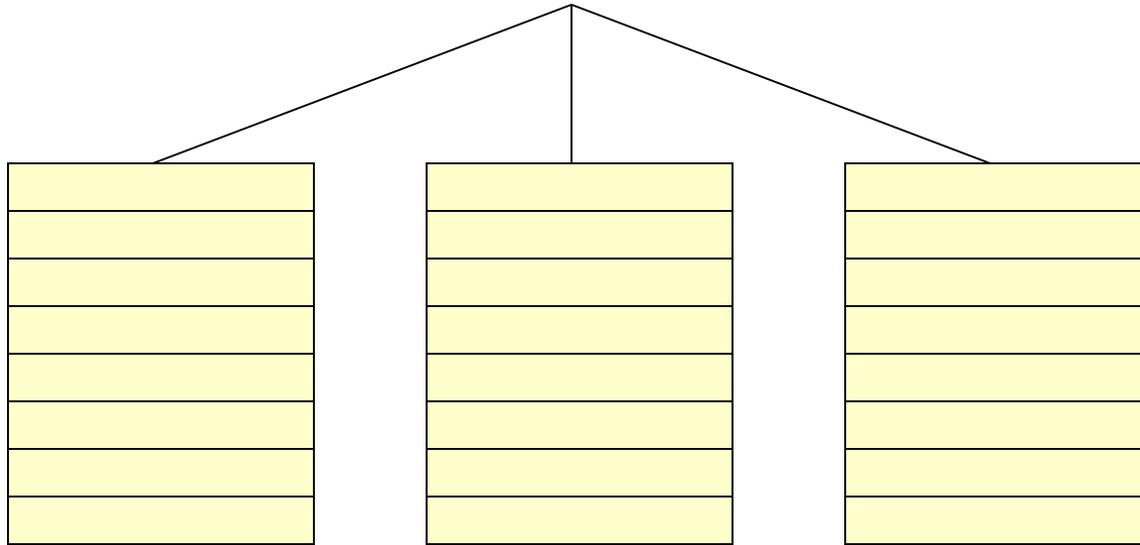
- Low-latency data access
  - HDFS is optimized for delivering high throughput of data
- Lots of small files
  - the amount of files is limited by the memory of the namenode; replica location is stored in memory
- Multiple writers and arbitrary file modifications
  - HDFS files are append only – write always at the end of the file

# HDFS Organization

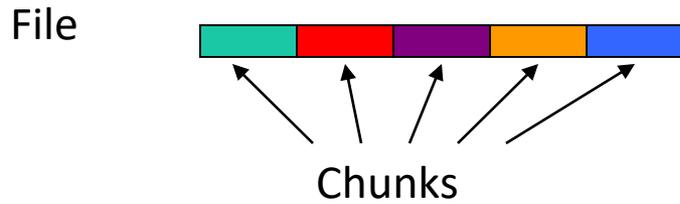
- Namenode (master)
    - Manages the filesystem namespace and metadata
    - Stores in memory the location of all blocks for a given file
  - Datanodes (workers)
    - Store and retrieve blocks
    - Send heartbeat to the namenode
  - Secondary namenode
    - Periodically merges the namespace image with the edit log
    - **Not** a backup for a namenode, only a checkpoint
  - High availability - pair of namenodes in stand-by
-

# Block Placement and Replication

- Aim – improve data reliability, availability and network bandwidth utilization
- Default replica placement policy
  - No Datanode contains more than one replica
  - No rack contains more than two replicas of the same block
- Namenode ensures the number of replicas is reached
- Balancer tool – balances the disk space usage
- Block scanner – periodically verifies checksums



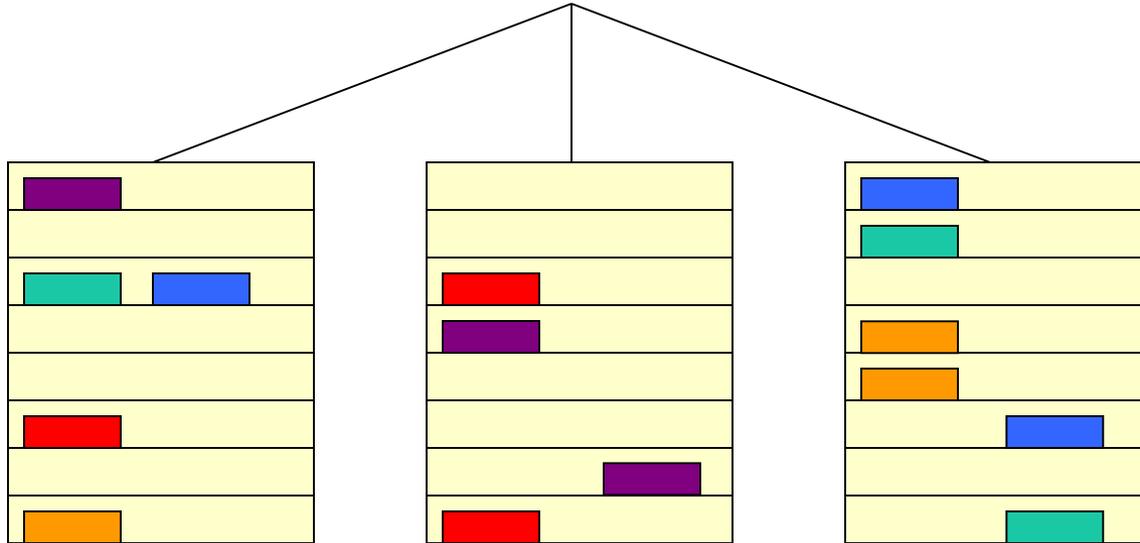
Racks of Compute Nodes



Source: J. D. Ullman invited talk EDBT 2011

# Default HDFS Block Placement Policy

61



- 1<sup>st</sup> replica located on the writer node
- 2<sup>nd</sup> and 3<sup>rd</sup> replicas on two different nodes in a different rack
- The other replicas are located on random nodes

# HDFS commands

- List all options for the hdfs dfs
  - `hdfs dfs -help`
  - `dfs` – run a filesystem command
- Create a new folder
  - `hdfs dfs -mkdir /BigDataAnalytics`
- Upload a file from the local file system to the HDFS
  - `hdfs dfs -put bigdata /BigDataAnalytics`

# HDFS commands

- **List the files in a folder**

  - `hdfs dfs -ls /BigDataAnalytics`

- **Determine the size of a file**

  - `hdfs dfs -du -h /BigDataAnalytics/bigdata`

- **Print the first 5 lines from a file**

  - `hdfs dfs -cat /BigDataAnalytics/bigdata | head -n 5`

- **Copy a file to another folder**

  - `hdfs dfs -cp /BigDataAnalytics/bigdata /BigDataAnalytics/AnotherFolder`

# HDFS commands

- **Copy a file to a local filesystem and rename it**
  - `hdfs dfs -get /BigDataAnalytics/bigdata  
bigdata_localcopy`
- **Scan the entire HDFS for problems**
  - `hdfs fsck /`
- **Delete a file from HDFS**
  - `hdfs dfs -rm /BigDataAnalytics/bigdata`
- **Delete a folder from HDFS**
  - `hdfs dfs -rm -r /BigDataAnalytics`

# Hadoop<sup>[Massive]</sup>[MapReduce]



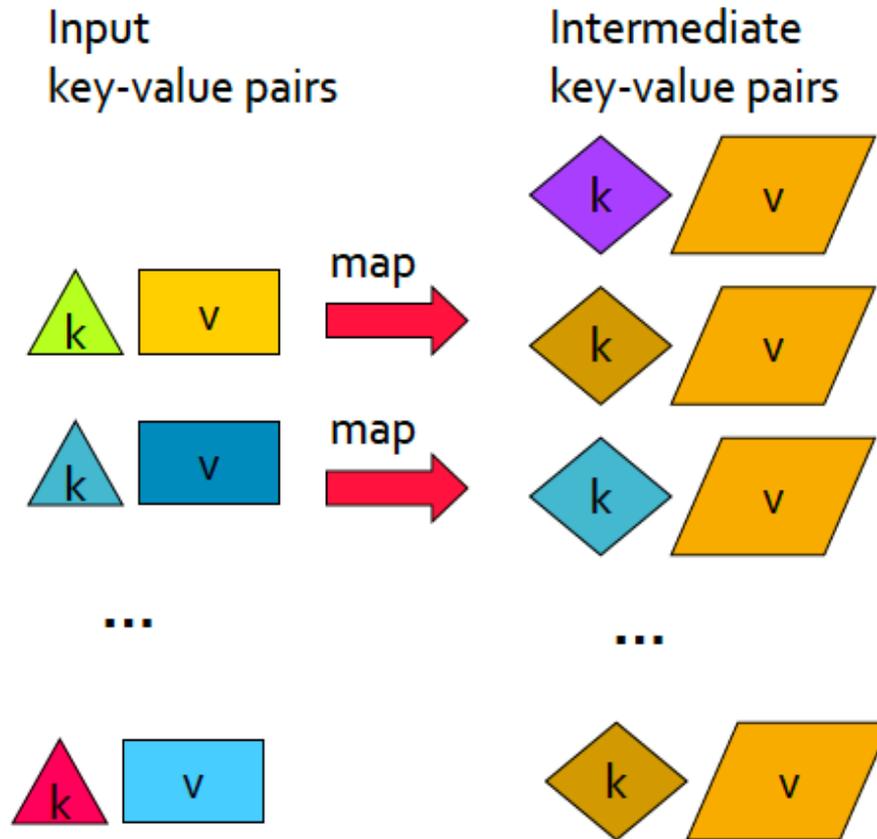
# Challenge

- How to perform computations in such setting?
  - Data are huge and stored on many machines
  - Machines fail
- How to write easy distributed programs?
- MapReduce programming model
  - High-level parallel programming construct
  - Implemented in Hadoop and others, ie. Spark

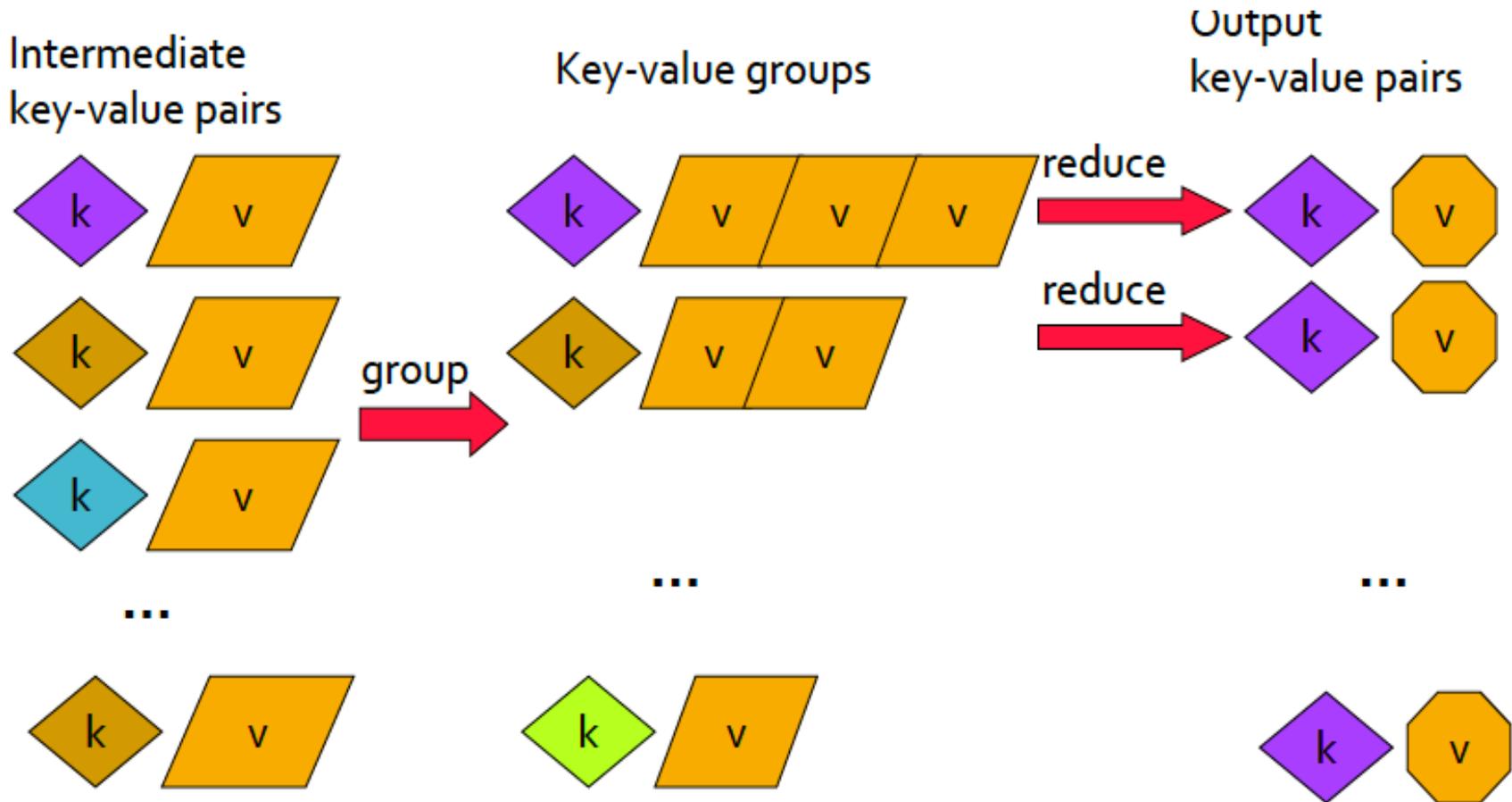
# MapReduce Overview

- Sequentially read a lot of data
  - **Map:** Extract something you care about
  - **Group by key:** Sort and Shuffle
  - **Reduce:** Aggregate, summarize, filter or transform
  - Write the result
- ✓ Outline stays the same, **map** and **reduce** change to fit the problem

# Map step



# Reduce step



# More specifically

- **Input:** a set of key/value pairs
- Programmer specifies two methods:
  - **Map(k, v) → <k',v'>\***
    - Takes a key value pair and outputs a set of key value pairs (input: e.g., key is the filename, value is the text of the document;)
    - There is one Map call for every (k,v) pair
  - **Reduce(k', <v'>\*) → <k', v''>\***
    - All values v' with same key k' are reduced together and processed in v' order
    - There is one Reduce function call per unique key k'

# Word Count

- We have a huge text document
- Count the number of times each distinct word appears in the file
- **Sample application:** Analyze web server logs to find popular URLs

# MapReduce: Word Counting

**Provided by the programmer**

**MAP:**  
Read input and produces a set of key-value pairs

**Group by key:**  
Collect all pairs with same key

**Provided by the programmer**

**Reduce:**  
Collect all values belonging to the key and output

The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre bot is the first step in a long-term space-based man/mache partnership. "The work we're doing now -- the robotics we're doing - is what we're going to need .....

**Big document**

(The, 1)  
(crew, 1)  
(of, 1)  
(the, 1)  
(space, 1)  
(shuttle, 1)  
(Endeavor, 1)  
(recently, 1)  
....

**(key, value)**



(crew, 1)  
(crew, 1)  
(space, 1)  
(the, 1)  
(the, 1)  
(the, 1)  
(shuttle, 1)  
(recently, 1)  
...

**(key, value)**

(crew, 2)  
(space, 1)  
(the, 3)  
(shuttle, 1)  
(recently, 1)  
...

**(key, value)**

# Word Count using MapReduce

## **map(key, value) :**

```
// key: document name; value: text of the document
  for each word w in value:
    emit(w, 1)
```

## **reduce(key, values) :**

```
// key: a word; value: an iterator over counts
  result = 0
  for each count v in values:
    result += v
  emit(key, result)
```

# MapReduce environment

## MapReduce environment takes care of:

- **Partitioning** the input data
- **Scheduling** the program's execution across a set of machines
- Performing the **group by key** step
- Handling machine **failures**
- Managing required inter-machine **communication**

The user program forks a Master controller process and some number of Worker processes at different compute nodes.

Normally, a Worker handles either Map tasks (a Map worker) or Reduce tasks (a Reduce worker), but not both.

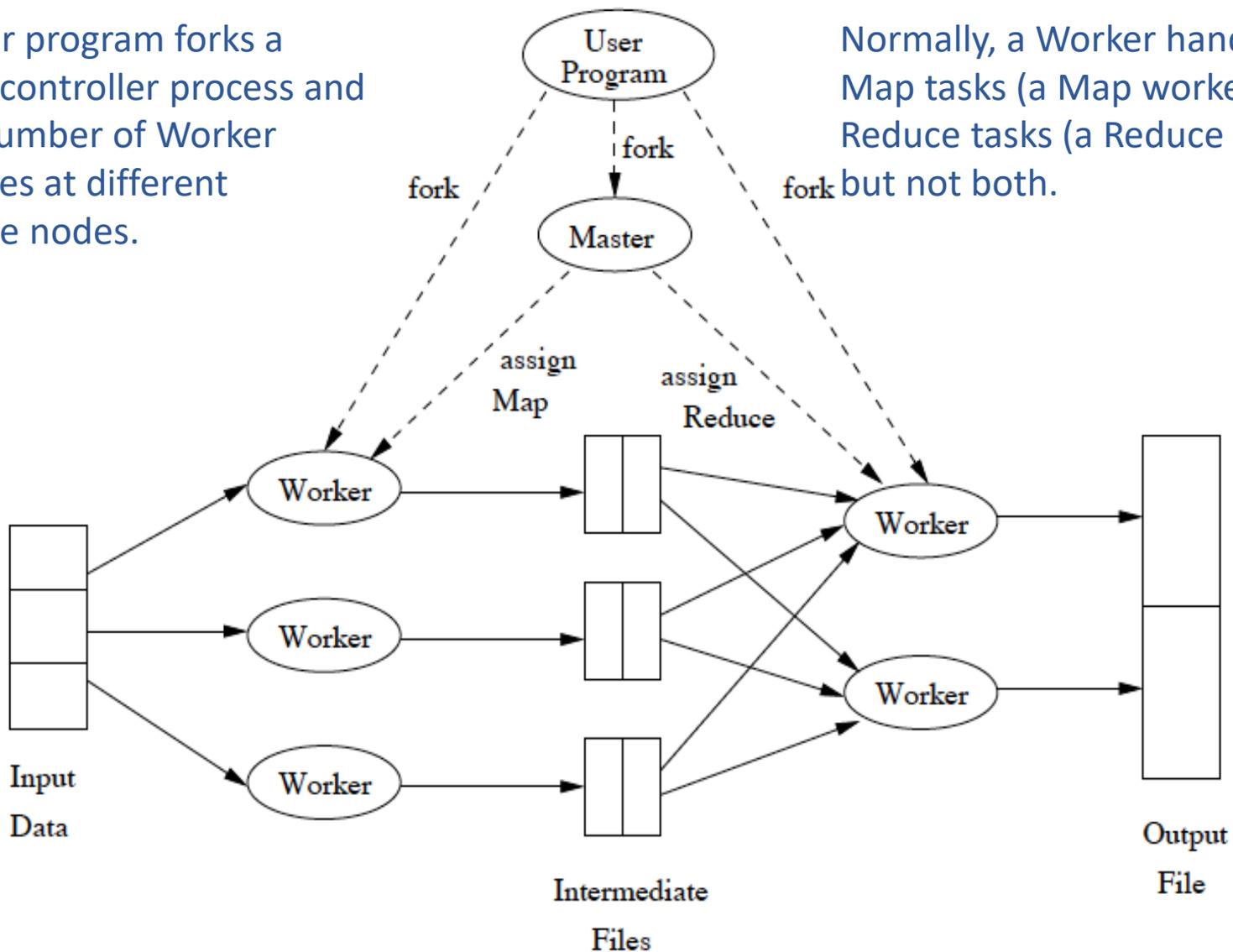


Figure 2.3: Overview of the execution of a map-reduce program

The Master creates some number of Map tasks and some number of Reduce tasks.

These numbers being selected by the user program. These tasks will be assigned to Worker processes by the Master.

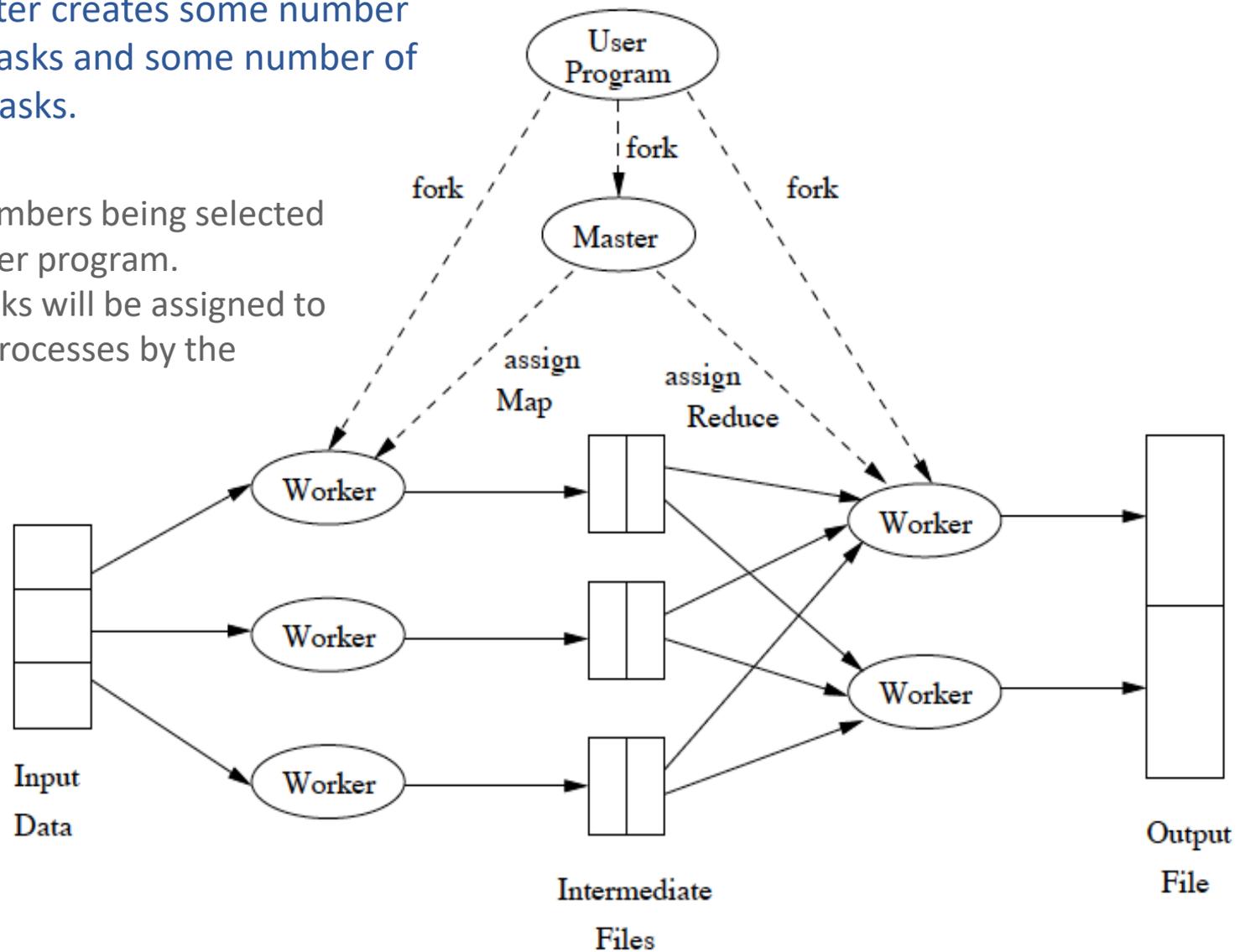


Figure 2.3: Overview of the execution of a map-reduce program

The Master keeps track of the status of each Map and Reduce task (idle, executing at a particular Worker, or completed).

A Worker process reports to the Master when it finishes a task, and a new task is scheduled by the Master for that Worker process.

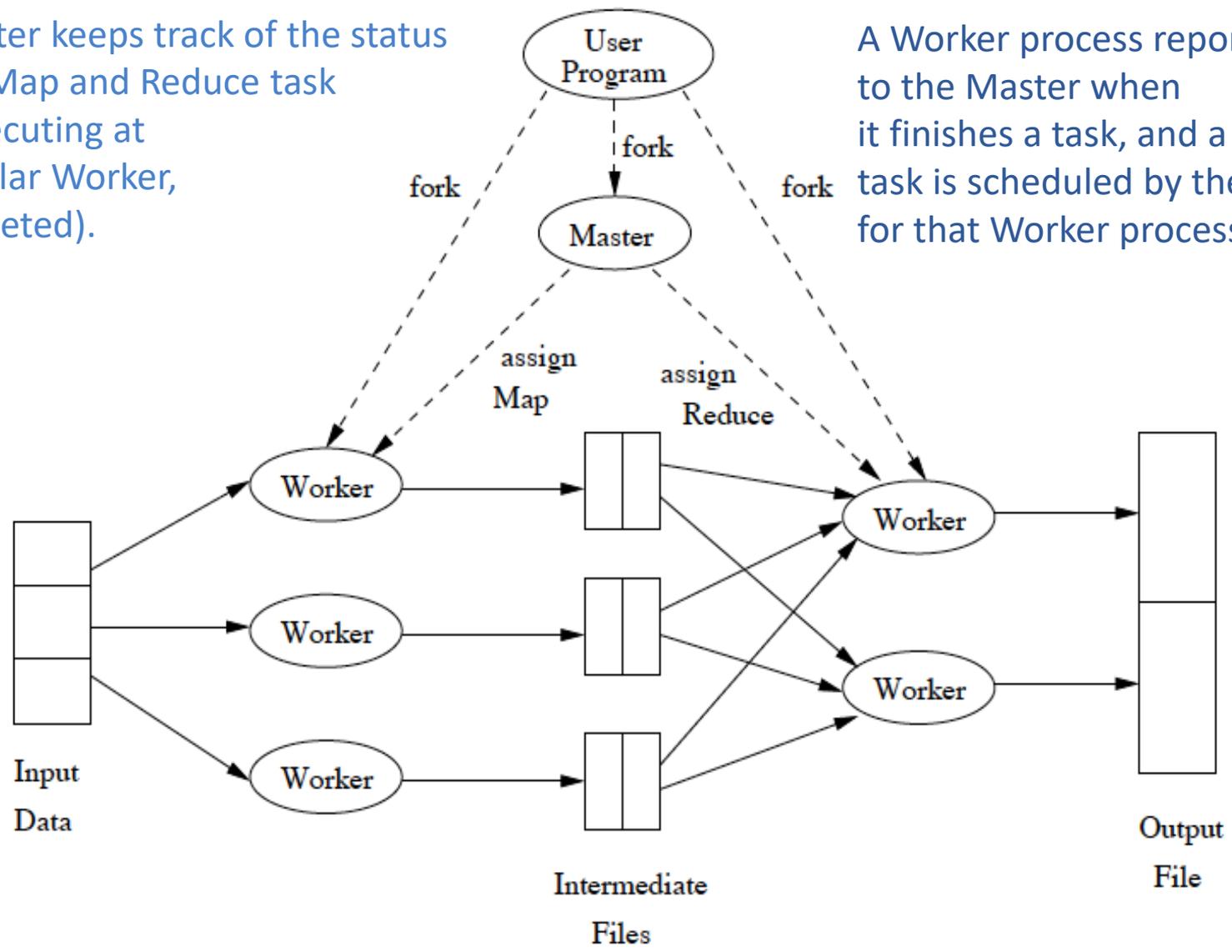


Figure 2.3: Overview of the execution of a map-reduce program

Each Map task is assigned one or more chunks of the input file(s) and executes on it the code written by the user.

The Map task creates a file for each Reduce task on the local disk of the Worker that executes the Map task.

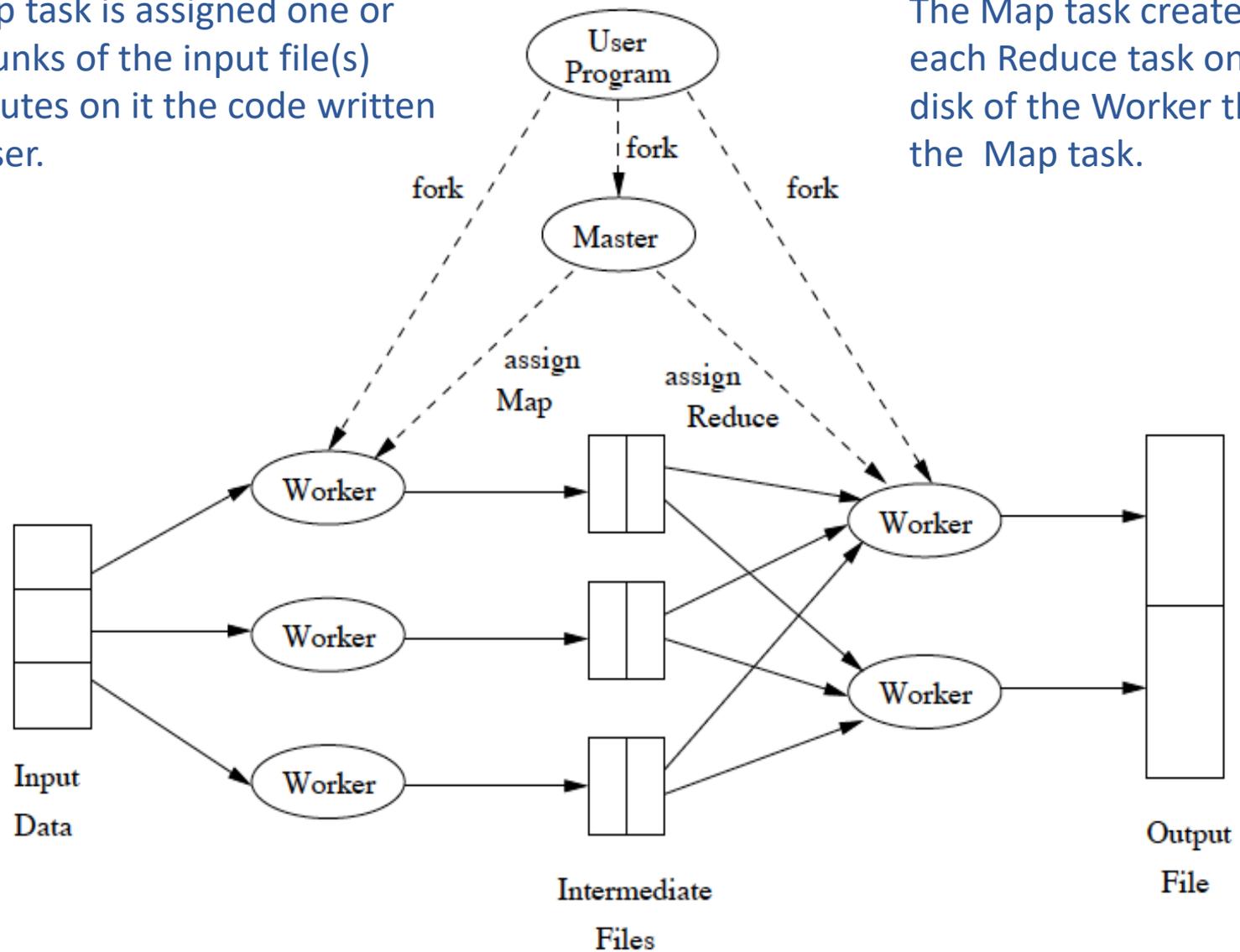


Figure 2.3: Overview of the execution of a map-reduce program

The Master is informed of the location and sizes of each of these files, and the Reduce task for which each is destined.

When a Reduce task is assigned by the Master to a Worker process, that task is given all the files that form its input.

The Reduce task executes code written by the user<sup>79</sup> and writes its output to a file that is part of the surrounding distributed file system.

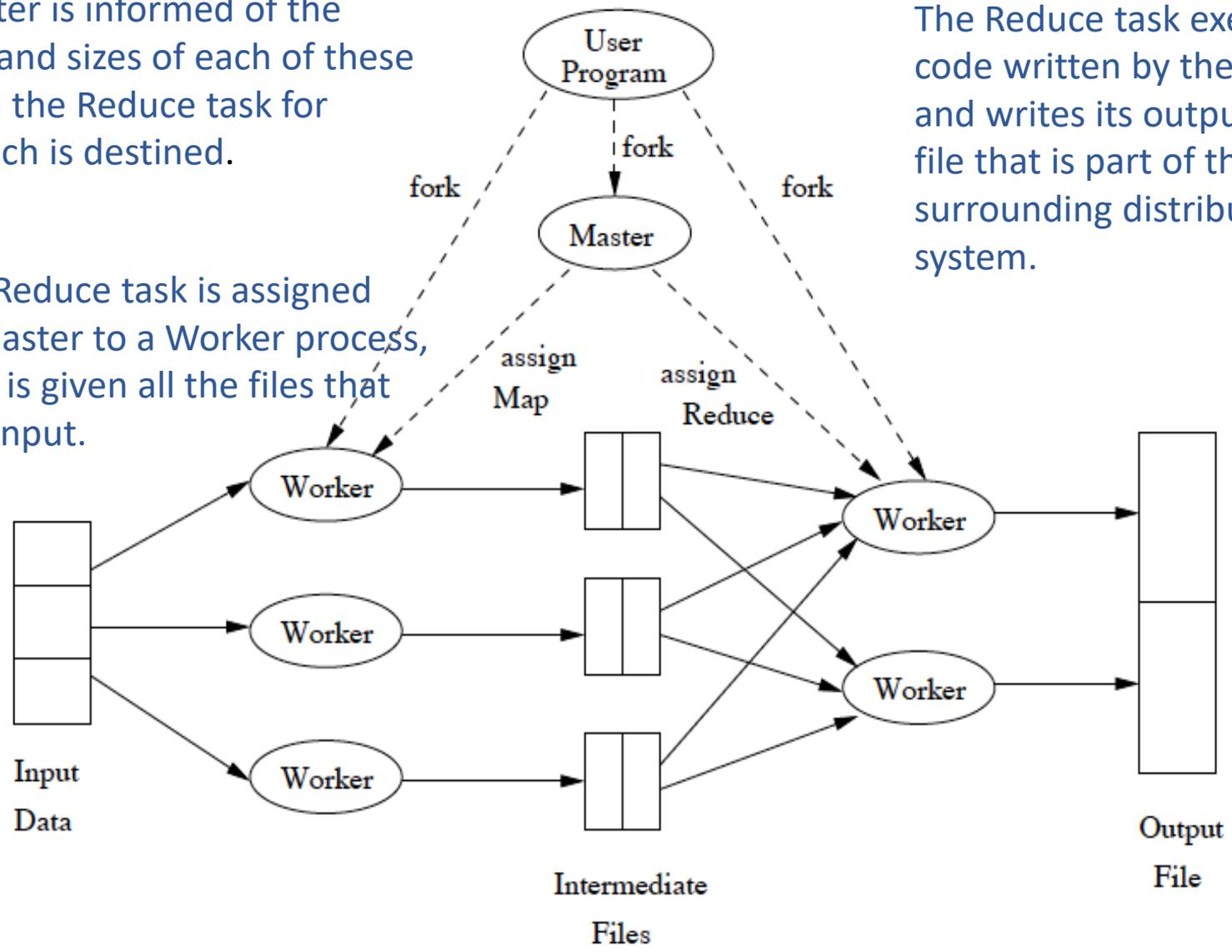


Figure 2.3: Overview of the execution of a map-reduce program

# Coping With Failures

- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
- **Master failure**
  - MapReduce task is aborted and client is notified

# Applications

- Counting URL frequencies
- Indexing web documents
- Link analysis and graph processing
- Machine Learning algorithms
- Matrix-Matrix and Matrix-Vector multiplication
  - the original application - one step of the PageRank iteration
- Relational algebra operations
- Statistical machine translation
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents

# Applications

*Slideshare.net - Hadoop Distributed File System by Dhruba Borthaku, 2009*

- Search
  - Yahoo, Amazon, Zvents
- Log processing
  - Facebook, Yahoo, ContextWeb, Joost, Last.fm
- Recommendation systems
  - Facebook
- Data warehouse
  - Facebook, AOL
- Video and image analysis
  - New York Times, Eyealike

# Matrix-Vector Multiplication

- ❑ Suppose we have an  $n \times n$  matrix  $\mathbf{M}$ , whose element in row  $\mathbf{i}$  and column  $\mathbf{j}$  will be denoted  $m_{ij}$ .
- ❑ Suppose we also have a vector  $\mathbf{v}$  of length  $n$ , whose  $\mathbf{j}$ th element is  $v_j$ .
- ❑ Then the matrix-vector product is the vector  $\mathbf{x}$  of length  $n$ , whose  $\mathbf{i}$ th element  $x_i$  is given by

$$x_i = \sum_{j=1}^n m_{ij} v_j$$

A handwritten example of matrix-vector multiplication on a chalkboard. The matrix is a 3x3 grid with elements A, B, C in the first row; D, E, F in the second row; and G, H, I in the third row. The vector is a column with elements P, Q, R. The result is a column vector with elements AP + BQ + CR, DP + EQ + FR, and GP + HQ + IR. The entire equation is enclosed in large red brackets.

$$\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \begin{bmatrix} P \\ Q \\ R \end{bmatrix} = \begin{bmatrix} AP + BQ + CR \\ DP + EQ + FR \\ GP + HQ + IR \end{bmatrix}$$

# Matrix-Vector Multiplication

- The matrix  $M$  and the vector  $\mathbf{v}$  each will be stored in a file of the DFS. We assume that the row-column coordinates of each matrix element will be discoverable, either from its position in the file, or because it is stored with explicit coordinates, as a triple  $(i, j, m_{ij})$ .
- We also assume the position of element  $v_j$  in the vector  $\mathbf{v}$  will be discoverable in the analogous way.

# Matrix-Vector Multiplication

- The Map Function:
  - Each Map task will take the entire vector  $\mathbf{v}$  and a chunk of the matrix  $M$ .
  - From each matrix element  $m_{ij}$  it produces the key-value pair  $(i, m_{ij}v_j)$ . Thus, all terms of the sum that make up the component  $x_i$  of the matrix-vector product will get the same key.
- The Reduce Function:
  - A Reduce task has simply to sum all the values associated with a given key  $i$ . The result will be a pair  $(i, x_i)$ .

# Relational Algebra

- Selection
  - Projection
  - Union, Intersection, Difference
  - Natural join
  - Grouping and Aggregation
- ✓ A relation can be stored as a file in a distributed file system. The elements of this file are the tuples of the relation.

# Union

- Suppose relations  $R$  and  $S$  have the same schema.
- The input for the Map tasks are chunks from either  $R$  or  $S$ .
  
- The Map Function:
  - Turn each input tuple  $t$  into a key-value pair  $(t, t)$ .
- The Reduce Function:
  - Associated with each key  $t$  there will be either one or two values. Produce output  $(t, t)$  in either case.

# Intersection

- Suppose relations  $R$  and  $S$  have the same schema.
- The input for the Map tasks are chunks from either  $R$  or  $S$ .
- The Map Function:
  - Turn each input tuple  $t$  into a key-value pair  $(t, t)$ .
- The Reduce Function:
  - If key  $t$  has value list  $[t, t]$ , then produce  $(t, t)$ . Otherwise, produce  $(t, \text{NULL})$ .

# Difference

- Suppose relations  $R$  and  $S$  have the same schema.
- The input for the Map tasks are chunks from either  $R$  or  $S$ .
- The Map Function:
  - For a tuple  $t$  in  $R$ , produce key-value pair  $(t, R)$ , and for a tuple  $t$  in  $S$ , produce key-value pair  $(t, S)$ . Note that the intent is that the value is the name of  $R$  or  $S$ , not the entire relation.
- The Reduce Function:
  - For each key  $\mathbf{t}$ , do the following.
    - If the associated value list is  $[\mathbf{R}]$ , then produce  $(\mathbf{t}, \mathbf{t})$ .
    - If the associated value list is anything else, which could only be  $[\mathbf{R}, \mathbf{S}]$ ,  $[\mathbf{S}, \mathbf{R}]$ , or  $[\mathbf{S}]$ , produce  $(\mathbf{t}, \text{NULL})$ .

# Natural Join

- Joining  $R(A,B)$  with  $S(B,C)$ .
- We must find tuples that agree on their  $B$  components.
- The Map Function:
  - For each tuple  $(a, b)$  of  $R$ , produce the key-value pair  $(b, (R, a))$ .
  - For each tuple  $(b, c)$  of  $S$ , produce the key-value pair  $(b, (S, c))$ .
- The Reduce Function:
  - Each key value  $b$  will be associated with a list of pairs that are either of the form  $(R, a)$  or  $(S, c)$ .
  - Construct all pairs consisting of one with first component  $R$  and the other with first component  $S$ , say  $(R, a)$  and  $(S, c)$ . The output for key  $b$  is  $(b, [(a_1, b, c_1), (a_2, b, c_2), \dots])$ ,
  - that is,  $b$  associated with the list of tuples that can be formed from an  $R$ -tuple and an  $S$ -tuple with a common  $b$  value.

# Grouping and Aggregation

- $R(A,B,C)$
- Select  $SUM(B)$  From  $R$  Group by  $A$
- The Map Function:
  - For each tuple  $(a, b, c)$  produce the key-value pair  $(a, b)$ .
- The Reduce Function:
  - Each key  $a$  represents a group. Apply  $SUM$  to the list  $[b_1, b_2, \dots, b_n]$  of  $b$ -values associated with key  $a$ . The output is the pair  $(a, x)$ , where  $x = b_1 + b_2 + \dots + b_n$ .

# References

- A comparison between several NoSQL databases with comments and notes by Bogdan George Tudorica, Cristian Bucur
  - nosql-databases.org
  - Scalable SQL and NoSQL data stores by Rick Cattell
  - [Brewer] Towards Robust Distributed Systems @ACM PODC'2000
  - [12 years later] CAP Twelve Years Later: How the "Rules" Have Changed, Eric A. Brewer, @Computer Magazine 2012. <https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed>
  - [Fox et al.] Cluster-Based Scalable Network Services @SOSP'1997
  - [Karger et al.] Consistent Hashing and Random Trees @ACM STOC'1997
  - [Coulouris et al.] Distributed Systems: Concepts and Design, Chapter: Time & Global States, 5th Edition
  - [DataMan] Data Management in cloud environments: NoSQL and NewSQL data stores.
-

# References

- NoSQL Databases - Christof Strauch – University of Stuttgart
- The Beckman Report on Database Research
- [Vogels] Eventually Consistent by Werner Vogels, doi:10.1145/1435417.1435432
- [Hadoop] Hadoop The Definitive Guide, Tom White, 2011
- [Massive] Mining of Massive Datasets, chapter 2, mmds.org
- [MapReduce] MapReduce: Simplified Data Processing on Large Clusters, OSDI 2004
- [HDFS] The Hadoop Distributed File System
- [HDFSpaper] The Hadoop Distributed File System @MSST2010

Thank you!

