# TDDD43 HT2014:
# Advanced databases and data models Theme 4: NoSQL, Distributed File System, Map-Reduce

Valentina Ivanova

ADIT, IDA, Linköping University

*Slides based on slides by Fang Wei-Kleiner*

*DFS, Map-Reduce slides based on Material from Chapter 2 in* ***Mining of Massive Datasets*** *Anand Rajaraman, Jeffrey David Ullman*

*http://infolab.stanford.edu/~ullman/mmds.html*

# Outline

- NoSQL – *Not only* SQL
  - Motivation
  - Concepts
  - Techniques
  - Systems
- Distributed File System, Map-Reduce

# NoSQL: Motivation

30, 40 years history of well-established database technology, all in vain? Not at all! But both setups and demands have drastically changed:

- main memory and CPU speed have exploded, compared to the time when System R (the mother of all RDBMS) was developed
- at the same time, huge amounts of data are now handled in real-time
- both data and use cases are getting more and more dynamic
- social networks (relying on graph data) have gained impressive momentum
- full-texts have always been treated shabbily by relational DBMS

# NoSQL: Facebook (Statistics)

*royal.pingdom.com/2010/06/18/the-software-behind-facebook*

- 500 million users
- 570 billion page views per month
- 3 billion photos uploaded per month
- 1.2 million photos served per second
- 25 billion pieces of content (updates, comments) shared every month
- 50 million server-side operations per second
- 2008: 10,000 servers; 2009: 30,000, …

☞ One RDBMS may not be enough to keep this going on!

# NoSQL: Facebook (Architechture)

- Memcached
  - distributed memory caching system
  - caching layer between web and database servers
  - based on a distributed hash table (DHT)
- HipHop for PHP
  - developed by Facebook to improve scalability
  - compiles PHP to C++ code, which can be better optimized
  - PHP runtime system was re-implemented

# NoSQL: Facebook (Architechture)

- Cassandra
  - developed by Facebook for inbox searching
  - data is automatically replicated to multiple nodes
  - no single point of failure (all nodes are equal)
- Hadoop/Hive
  - implementation of Google's MapReduce framework
  - performs data-intensive calculations
  - (initially) used by Facebook for data analysis

# NoSQL: Facebook (Components)

- Varnish
  - HTTP accelerator, speeds up dynamic web sites
- Haystack
  - object store, used for storing and retrieving photos
- BigPipe
  - web page serving system; serves parts of the page (chat, news feed, ...)
- Scribe
  - aggregates log data from different servers

# NoSQL: Facebook

*hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html*

- Architecture: Hadoop Cluster
  - 21 PB in Data Warehouse cluster, spread across 2000 machines:
  - 1200 machines with 8 cores, 800 machines with 16 cores
  - 12 TB disk space per machine, 32 GB RAM per machine
  - 15 map-reduce tasks per machine
- Workload
  - daily: 12 TB of compressed data, and 800 TB of scanned data
  - 25,000 map-reduce jobs and 65 millions files in HDFS

# NoSQL: Facebook

- Conclusion
  - classical database solutions have turned out to be completely insufficient
  - heterogeneous software architecture is needed to match all requirements

# NoSQL: *Not only* SQL

- RDBMS are still a great solution for centralized, tabular data sets
- NoSQL gets interesting if data is heterogeneous and/or too large
- most NoSQL projects are open source and have open communities
- code bases are up-to-date (no 30 years old, closed legacy code)
- they are subject to rapid development and change
- cannot offer general-purpose solutions yet, as claimed by RDBMS

# NoSQL: *Not only* SQL

*www.techrepublic.com/blog/10things/10-things-you-should-know-about-nosql-databases/1772*

## ***10 Things: Five Advantages***

- Elastic Scaling → scaling out: distributing data on commodity clusters instead of buying bigger servers
- Big Data → opens new dimensions that cannot be handled with RDBMS
- Goodbye DBAs (see you later?) → automatic repair, distribution, tuning, ...
- Economics → based on cheap commodity servers and less costs per transaction/second
- Flexible Data Models → non-existing/relaxed data schema, structural changes cause no overhead

# NoSQL: *Not only* SQL

*www.techrepublic.com/blog/10things/10-things-you-should-know-about-nosql-databases/1772*

## ***10 Things: Five Challenges***

- Maturity → still in pre-production phase, key features yet to be implemented
- Support → mostly open source, start-ups, limited resources or credibility
- Administration → require lot of skill to install and effort to maintain
- Analytics and Business Intelligence → focused on web apps scenarios, limited ad-hoc querying
- Expertise → few NoSQL experts available in the market

# NoSQL: Concepts

*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. The movement began early 2009 and is growing rapidly. Often more characteristics apply as: ***schema-free, easy replication support, simple API, eventually consistent/BASE*** (not ACID), a ***huge data amount***, and more.

- Stefan Edlich

# NoSQL: Concepts

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

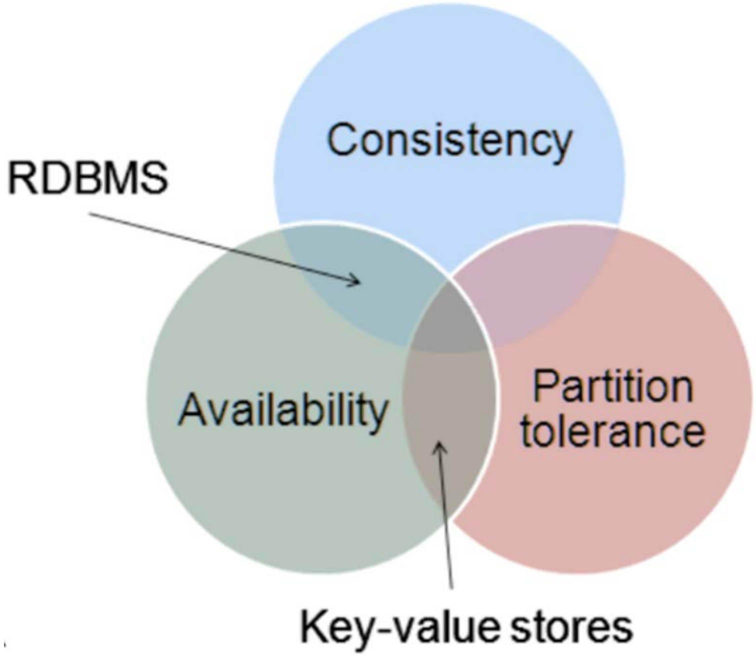
- Vertical Scalability (scale up)
  - add resources (more CPUs, more memory) to a single node
  - using more threads to handle a local problem
- Horizontal Scalability (scale out)
  - add nodes (more computers, servers) to a distributed system
  - gets more and more popular due to low costs for commodity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

# NoSQL: Concepts

***CAP Theorem: Consistency, Availability, Partition Tolerance***

`Brewer [ACM PODC'2000]: Towards Robust Distributed Systems`

**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***

`Brewer [ACM PODC'2000]: Towards Robust Distributed Systems`

- **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***

`Brewer [ACM PODC'2000]: Towards Robust Distributed Systems`

- **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***

`Brewer [ACM PODC'2000]: Towards Robust Distributed Systems`

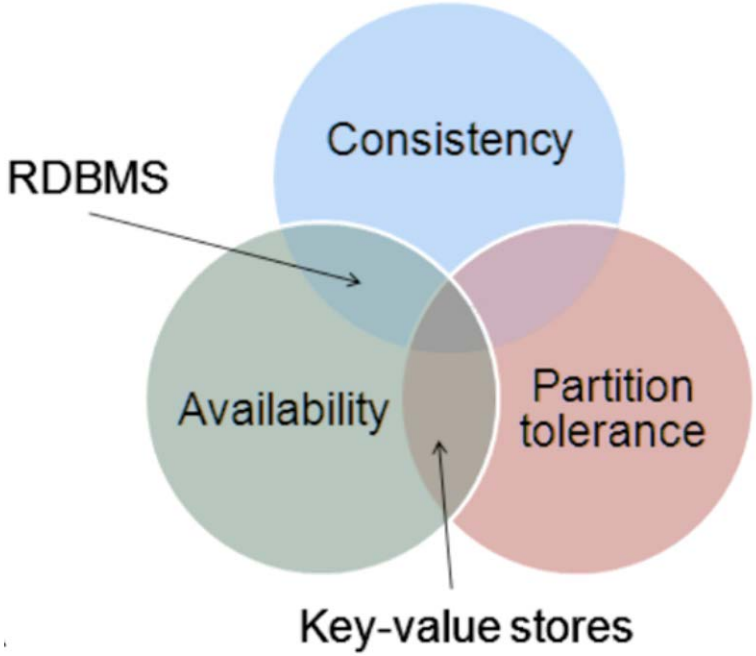
- **Partition Tolerance**
  - system continues to operate, even if two sets of servers get isolated
- Example
  - system gets partitioned if connection between server clusters fails
  - failed connection won't cause troubles if system is tolerant

# NoSQL: Concepts

***CAP Theorem: Consistency, Availability, Partition Tolerance***

`Brewer [ACM PODC'2000]: Towards Robust Distributed Systems`

- (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**

- **Atomicity** → all operations in the transaction will complete, or none will
- **Consistency** → before and after transactions, database will be in a consistent state
- **Isolation** → operations cannot access data that is currently modified
- **Durability** → data will not be lost upon completion of a transaction

# NoSQL: Concepts

**BASE**: **Basically Available**, **Soft State**, **Eventual Consistency**

`Fox et al. [SOSP'1997]: Cluster-Based Scalable Network Services`

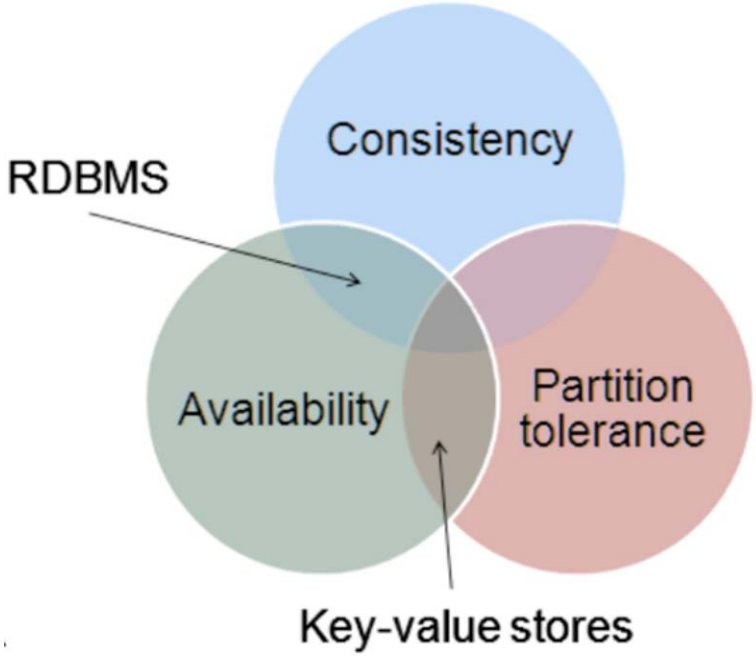
- **Basically Available →** an application works basically all the time (despite partial failures)
- **Soft State →** is in flux and non-deterministic (changes all the time)
- **Eventual Consistency →** will be in some consistent state (at some time in future)

# NoSQL: Concepts

***CAP Theorem: Consistency, Availability, Partition Tolerance***

`Brewer [ACM PODC'2000]: Towards Robust Distributed Systems`

- (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 which is partition tolerant?) → Consistent hashing
- distributed query strategy (horizontal scalability) → MapReduce
- eventual consistency → Vector clock

# NoSQL: Systems

Selected Categories from nosql-databases.org

- Key-Value Stores
- Document Stores
- (Wide) Column Stores
- Graph Databases
- Object Databases

→ no taxonomy exists that all parties agree upon

→ might look completely different some years later

# NoSQL: Systems

- Key-Value Stores
  - simple common baseline: maps or dictionaries, storing keys and values
  - also called: associative arrays, hash tables/maps
  - keys are unique, values may have arbitrary type
  - focus: high scalability (more important than consistency)
  - traditional solution: BerkeleyDB, started in 1986
  - revived by Amazon Dynamo in 2007 (proprietary)
  - recent solutions: Redis, Voldemort, Tokyo Cabinet, Memcached

→ (very) limited query facilities; usually get(key) and put(key, value)

# NoSQL: Systems

- Document Stores
  - basic entities (tuples) are documents
  - schema-less storage
  - document format depends on implementation: XML, JSON, YAML, binary data, …
  - more powerful than key/value stores: offers query and indexing facilities
  - first document store (commercial): LotusDB, developed in 1984
  - recent solutions: CouchDB and MongoDB (free), SimpleDB (commercial)

# NoSQL: Systems

- (Wide) Column Stores
  - tight coupling of column data

    → But single tuples are spread across multiple files/pages
  - efficient for calculating aggregations, accessing single columns
  - space saving for dense or identical column data
  - Column Stores implementations: MonetDB, Sybase, Vertica
  - Wide Column Stores implementations: BigTable, HBase, Cassandra

# NoSQL: Systems

- Graph Databases
  - based on the property graph model
  - stored as directed adjacency lists
  - vertices: entities
  - edges: similar to relations in RDBMSs
  - majority of graph databases are schema free
  - prominent use cases: location-based services (LBS), social networks, shortest paths, …
  - examples: Neo4J, GraphDB, FlockDB, DEX, InfoGrid, OrientDB

# NoSQL: Systems

- Object Databases
  - hot research topic in the 1990s
  - inspired by the success of object-oriented languages
  - requirement: make objects persistent with minimum effort
  - basic entities (tuples) are objects
  - early solutions: GemStone, Objectivity/DB, Versant, Caché
  - new impetus by Open Source movements (most popular: db4o)
  - standardization efforts: SQL:1999, ODMG, Native Queries, LINQ
  - query language: OQL (very similar to SQL)

# NoSQL: *Not only* SQL

## Conclusion

- NoSQL solutions have become essential in distributed environment
- RDBMS are still prevailing (often known as the only alternative)

## Choosing a Database

Before you go for a database system/paradigm, clarify for yourself...

1. Which features are needed? Robust storage vs. real-time results vs. querying
2. What limits are most critical? Memory vs. performance vs. bandwidth
3. How large your data will get? Mega- vs. giga- vs. tera- vs. ...bytes

# References


- http://www.inf.uni-konstanz.de/dbis/teaching/ss11/advanced-database-technologies/
- Brewer [ACM PODC'2000]: Towards Robust Distributed Systems - https://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf
- CAP Twelve Years Later: How the "Rules" Have Changed - http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed
- Lynch, Seth [ACM SIGACT'2002]: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services
- Fox et al. [SOSP'1997]: Cluster-Based Scalable Network Services
- Dean, Ghemawat [OSDI'2004]: MapReduce: Simplified Data Processing on Large Clusters
- Karger et al. [ACM STOC'1997], Consistent Hashing and Random Trees
- G. Coulouris et al. Distributed Systems: Concepts and Design, 5th Edition
- DeCandia et al. [ACM SIGOPS'2007], Dynamo: Amazon's Highly Available Key-value Store
- Chang et al. [OSDI'2006], Bigtable: A Distributed Storage System for Structured Data
- Ghemawat et al. [SOSP'2003], The Google File System

# Outline

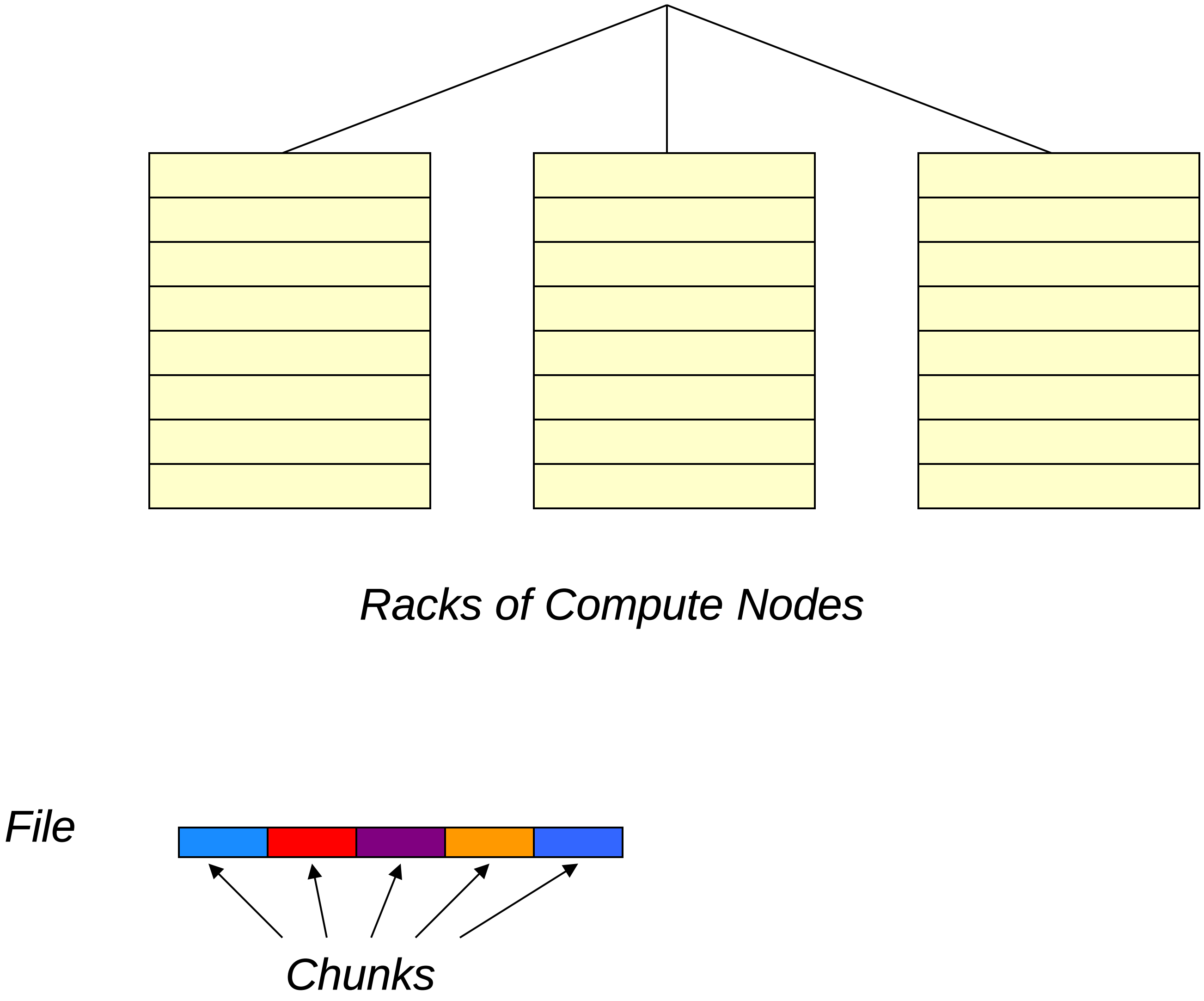
- NoSQL – *Not only* SQL
  - Motivation
  - Concepts
  - Techniques
  - Systems
- Distributed File System, Map-Reduce

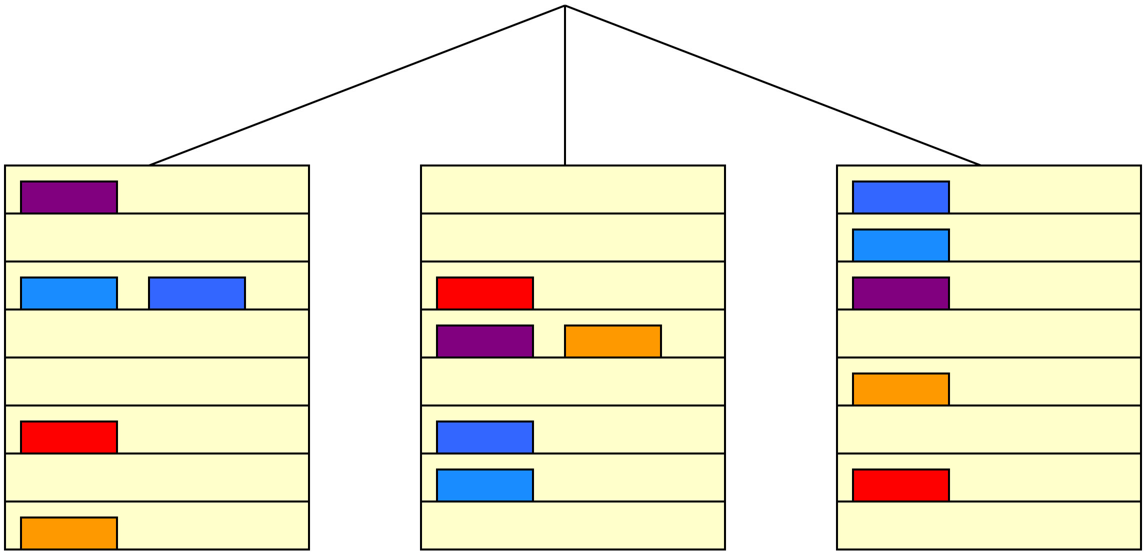
# Compute Nodes

- 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.

# Distributed File System

- Files are very large, read/append.
- They are divided into *chunks*.
  - Typically 64MB to a chunk.
- Chunks are replicated at several *compute-nodes*.
- A *master* (possibly replicated) keeps track of all locations of all chunks.


Racks of Compute Nodes
File
Chunks



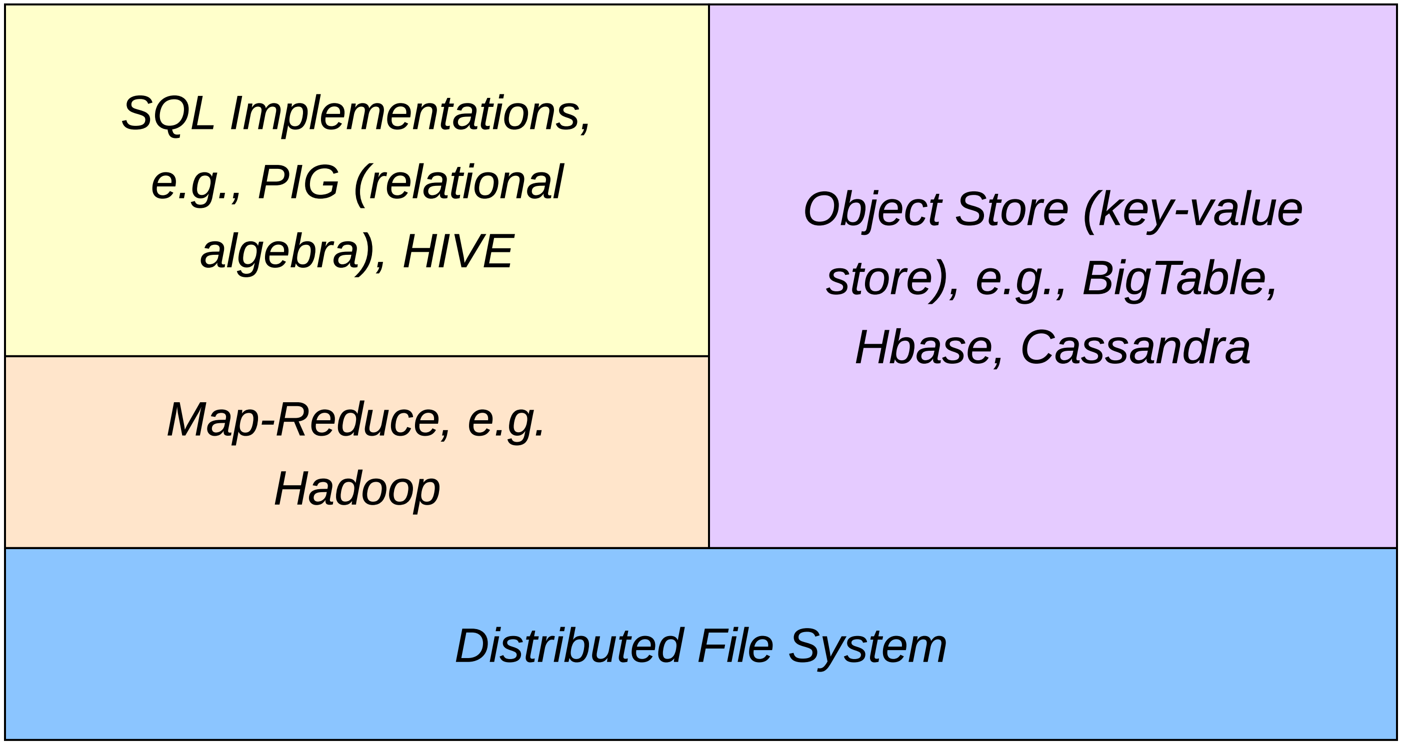
*3-way replication of files, with copies on different racks.*

*Source: J. D. Ullman invited talk EDBT 2011*

# Implementations

- *GFS* (Google File System – proprietary).
- *HDFS* (Hadoop Distributed File System – open source).
- *CloudStore* (Kosmix File System – open source).

# The New Stack




*Source: J. D. Ullman invited talk EDBT 2011*

# MapReduce Overview

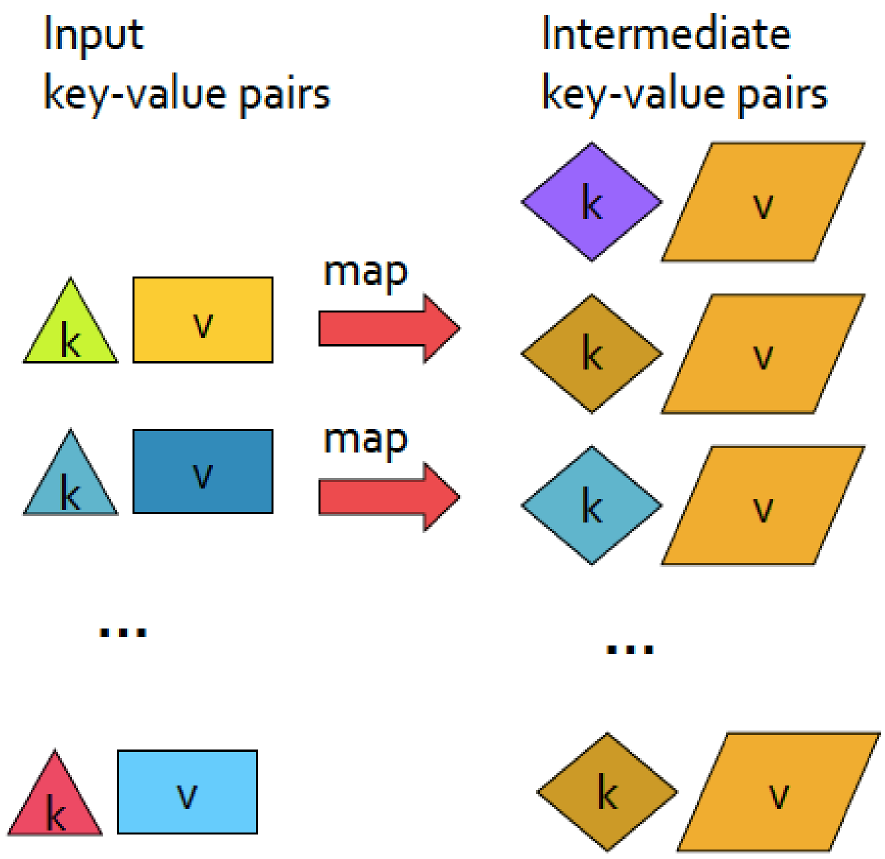
- Sequentially read a lot of data
- **Map:** Extract something you care about

- **Group by key:** Sort and Shuffle

- **Reduce:** Aggregate, summarize, filter or transform
- Write the result

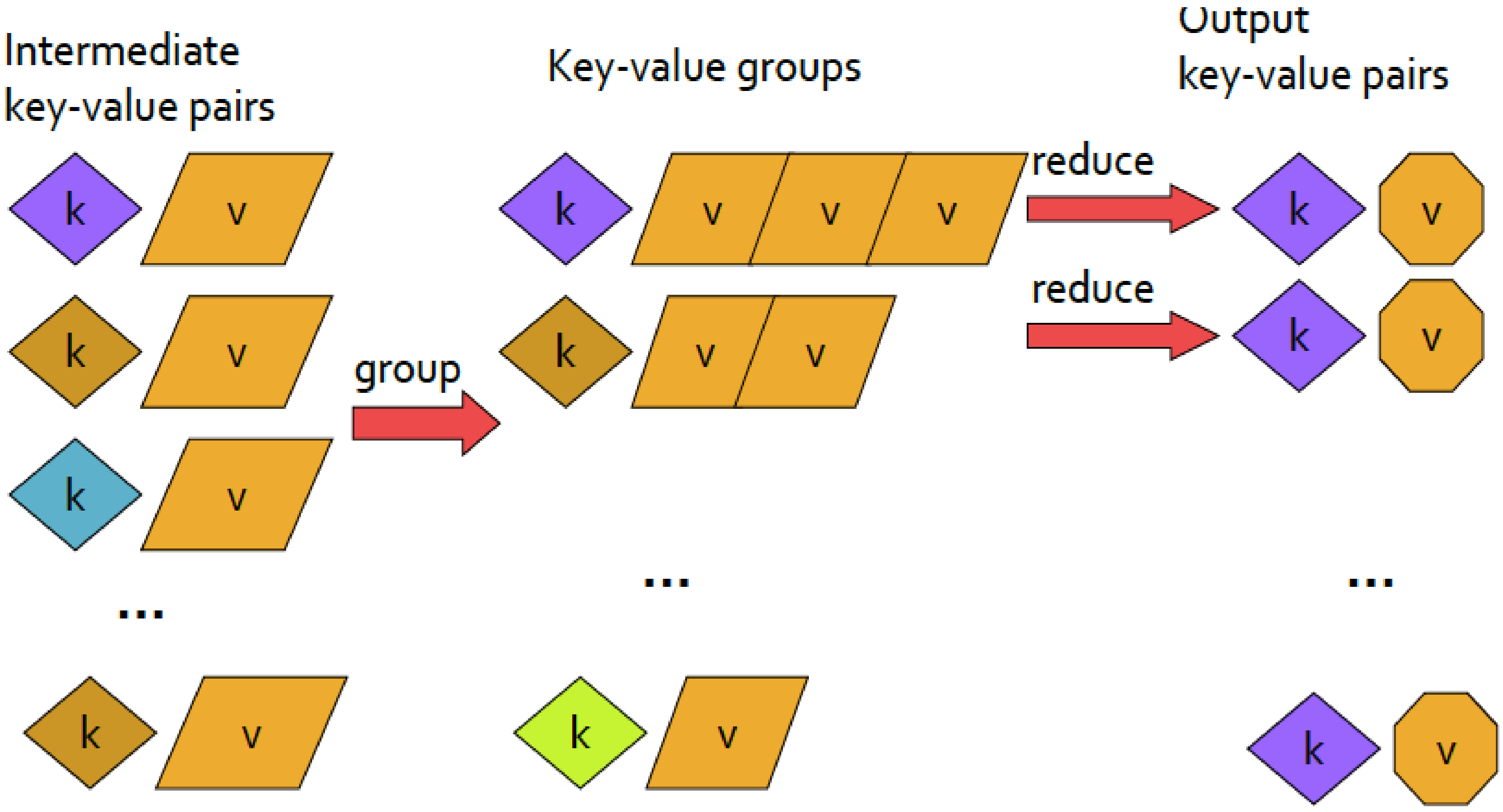
  - ✓ Outline stays the same, **map** and **reduce** change to fit the problem

# Map step



Input
key-value pairs
Intermediate
key-value pairs
k
v
map
k
v
map
k
v
k
v
k
v
k
v
...
...
k
v
k
v

# Reduce step



Intermediate key-value pairs
Key-value groups
Output key-value pairs
k
v
group
reduce
reduce
...

# 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

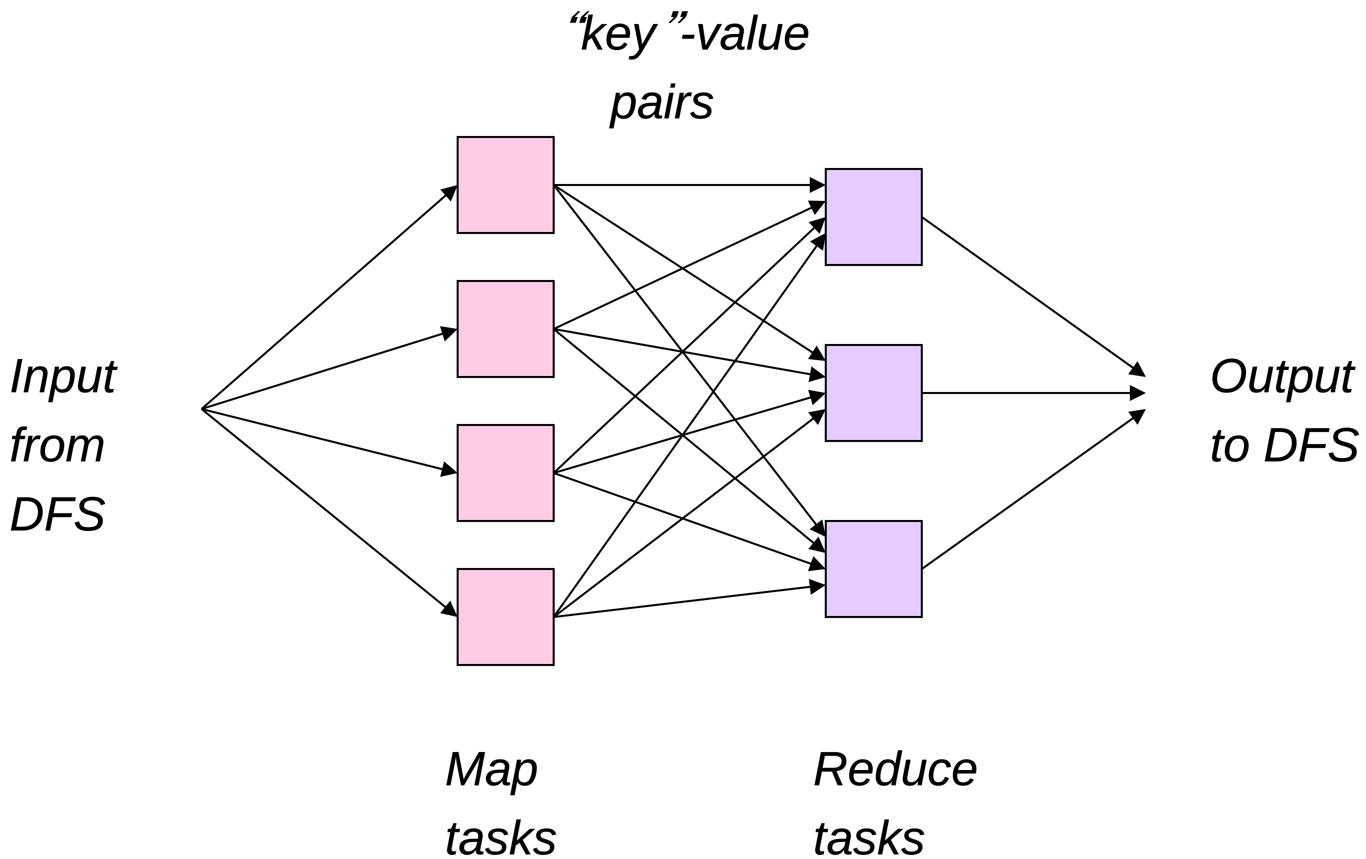
- **Sample application:** Analyze web server logs to find popular URLs

# Word Count using MR

```
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)
```

Linköping University

# Map-Reduce Pattern



# Map-Reduce environment

**Map-Reduce 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**

# MapReduce Implementation Details

- 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.
- 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.

# MapReduce Implementation Details

- 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.

# MapReduce Implementation Details

- 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.
- 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 and writes its output to a file that is part of the surrounding distributed file system.

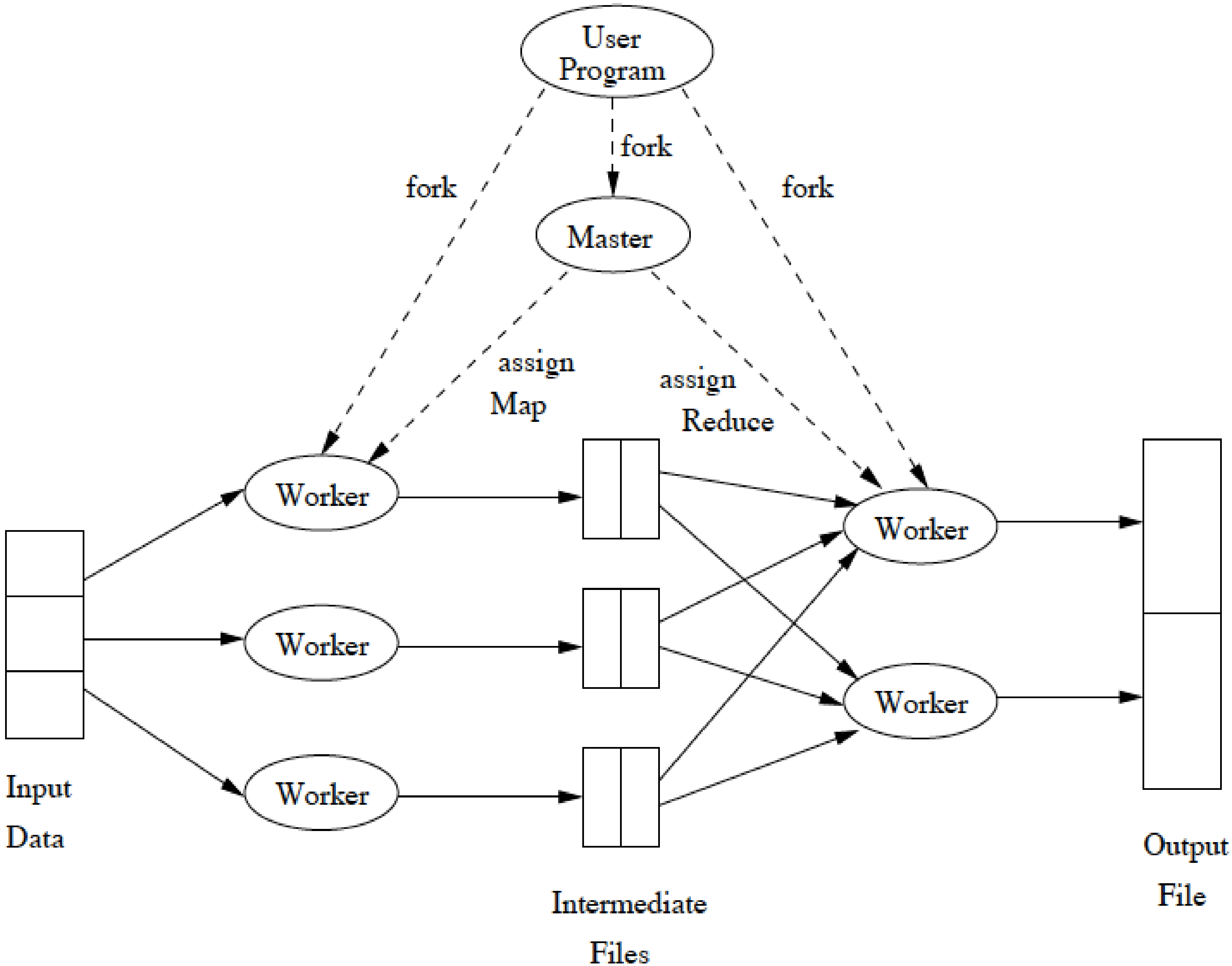



Figure 2.3: Overview of the execution of a map-reduce program

*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.*

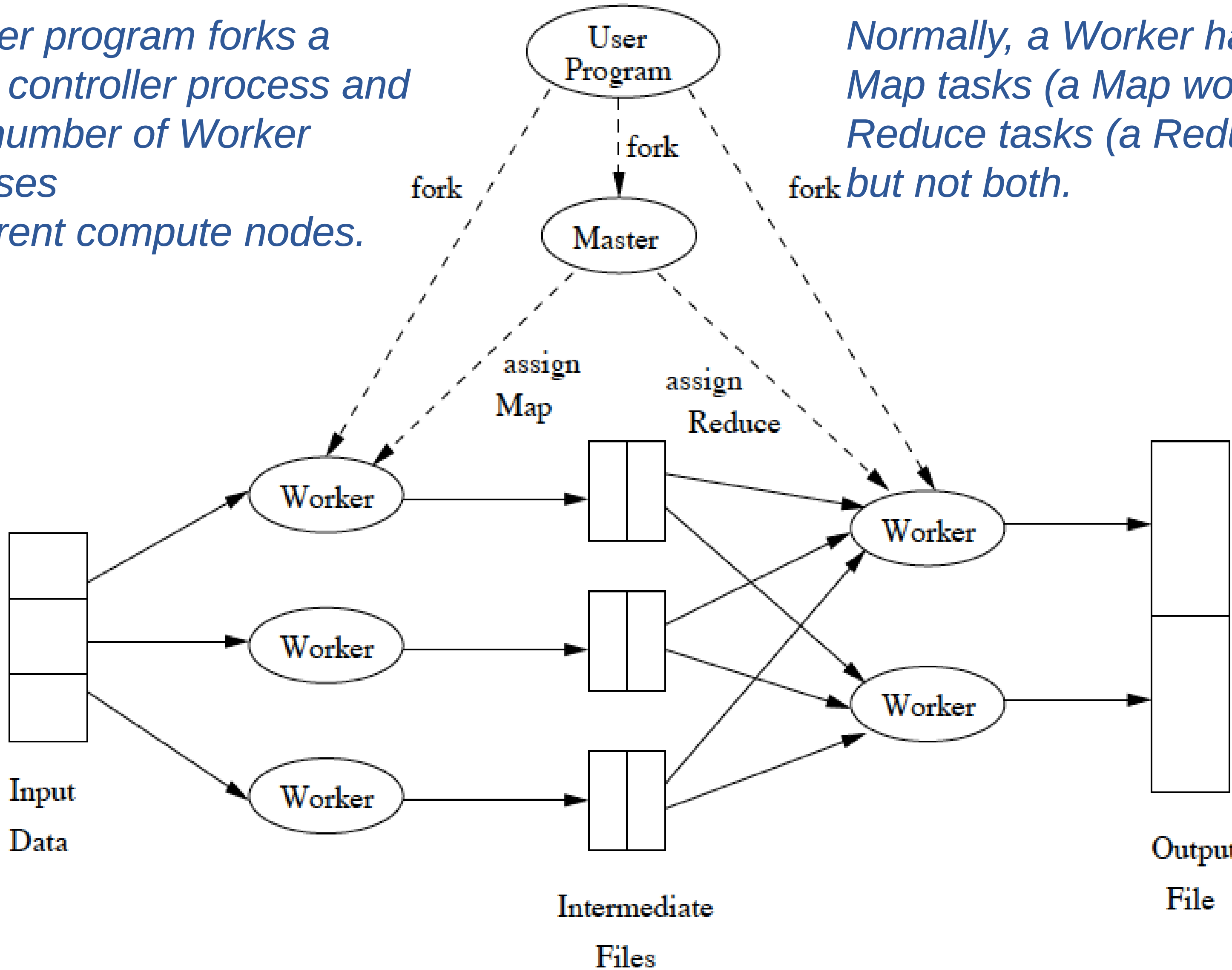



Figure 2.3: Overview of the execution of a map-reduce program

*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.*

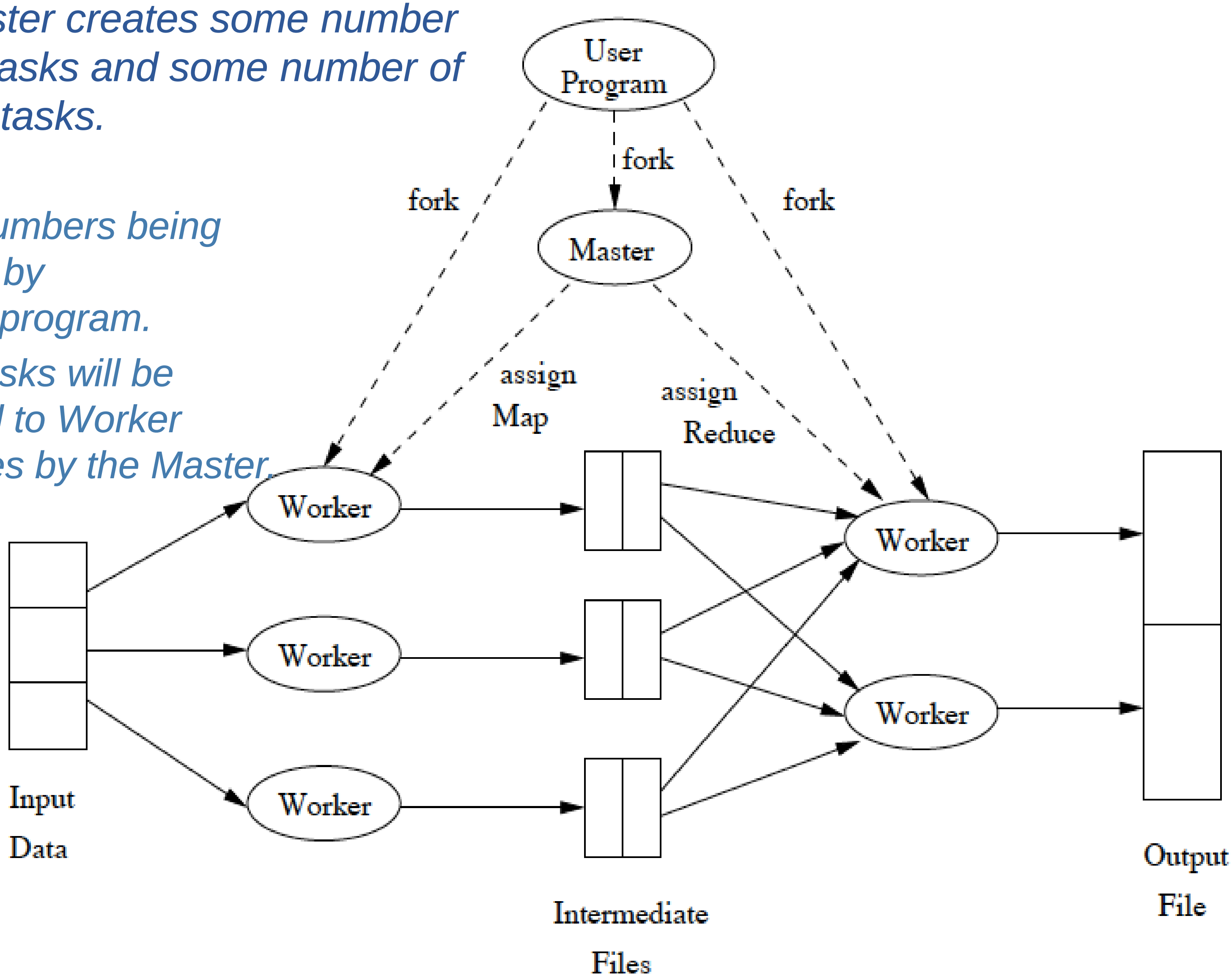



Figure 2.3: Overview of the execution of a map-reduce program

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

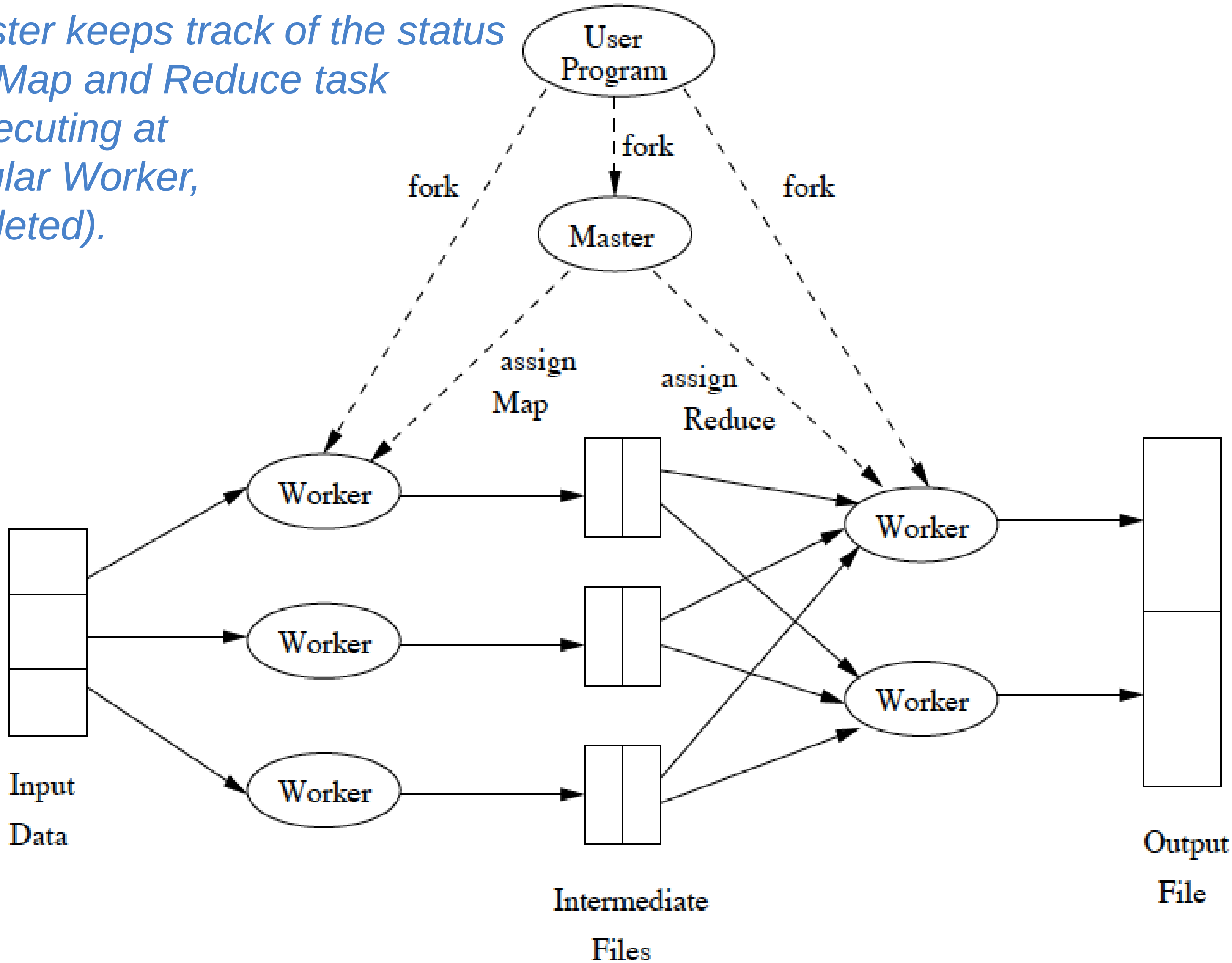



Figure 2.3: Overview of the execution of a map-reduce program

*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.*

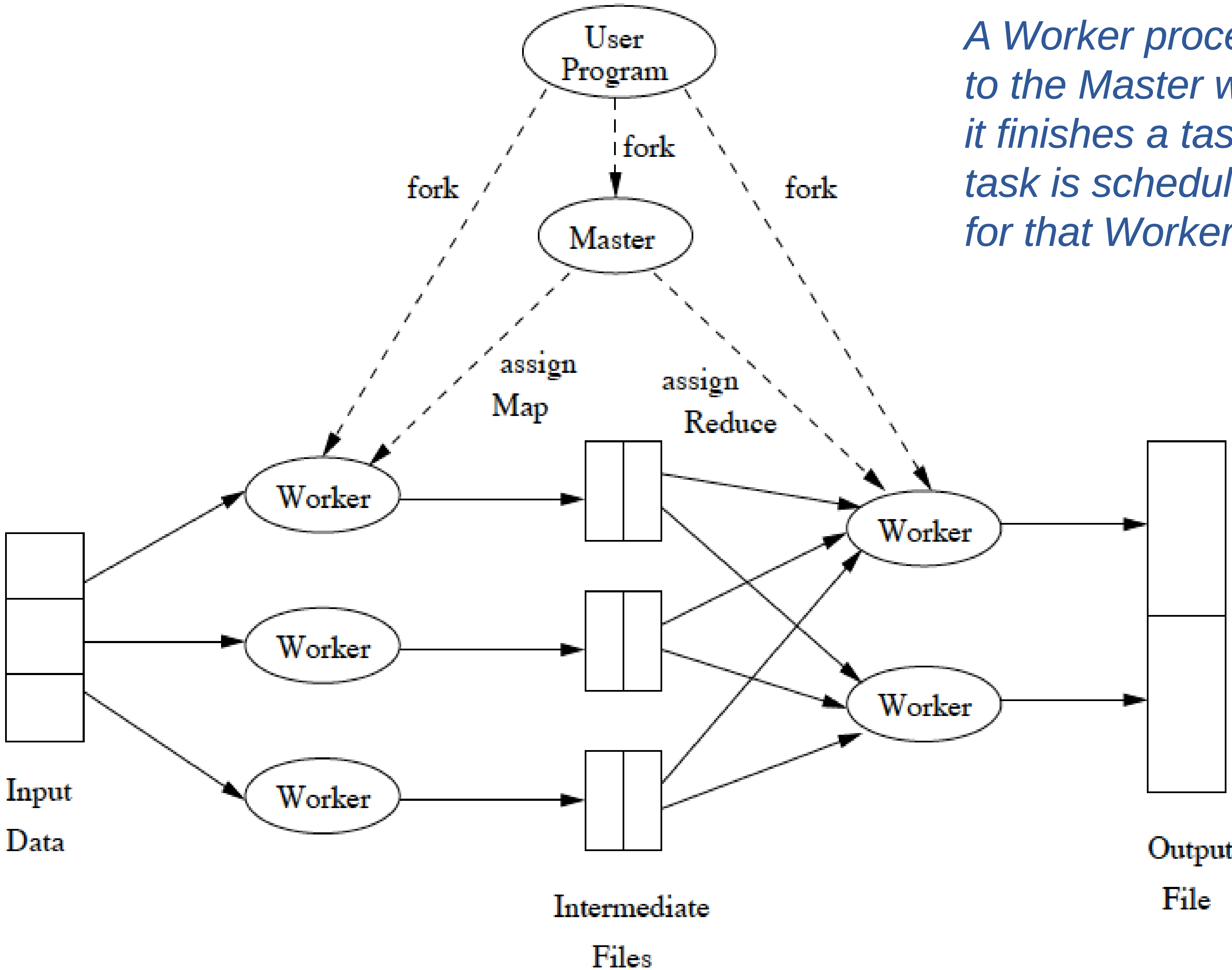



Figure 2.3: Overview of the execution of a map-reduce program

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 execute the Map task.

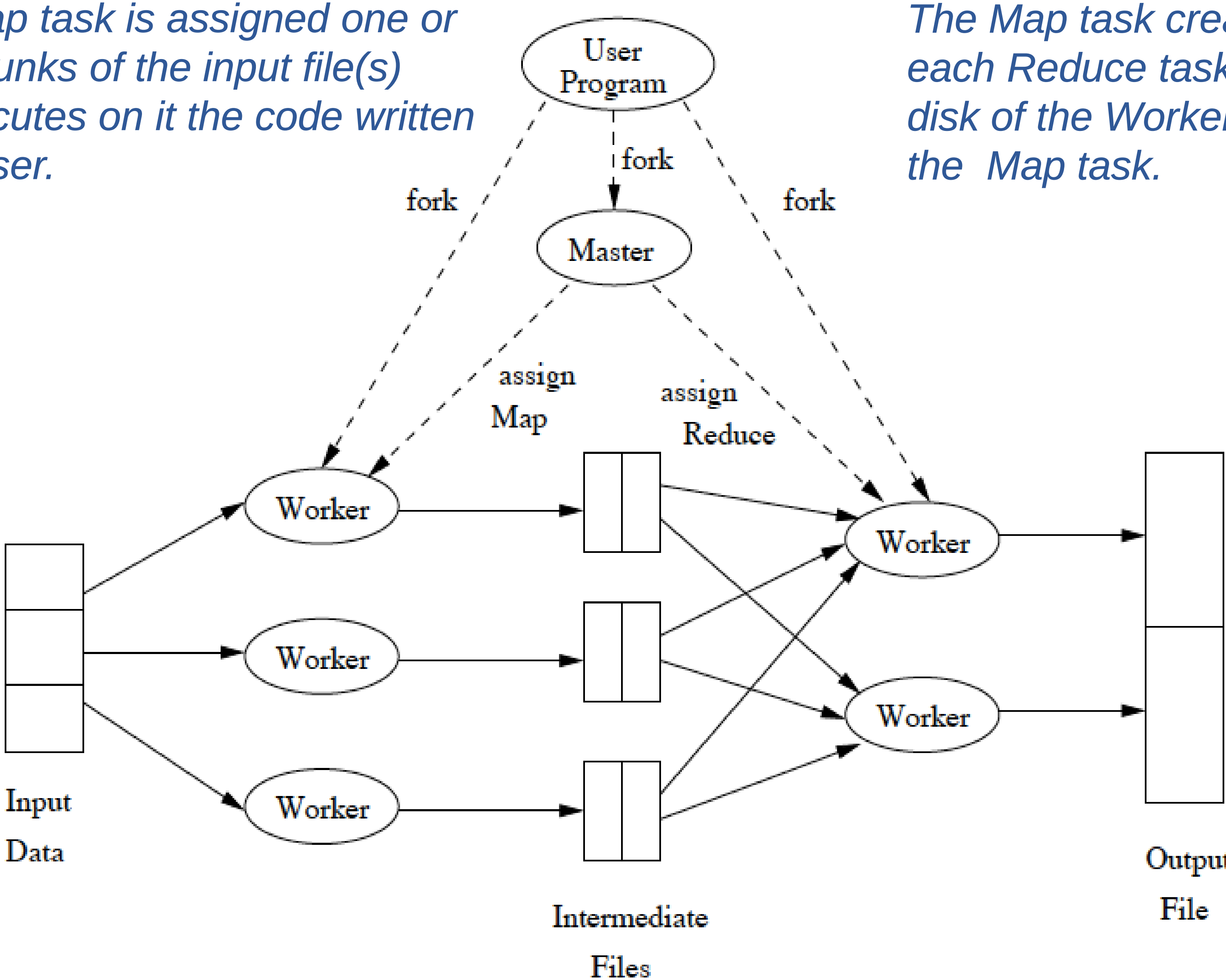



Figure 2.3: Overview of the execution of a map-reduce program

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 and writes its output to a file that is part of the surrounding distributed file system.

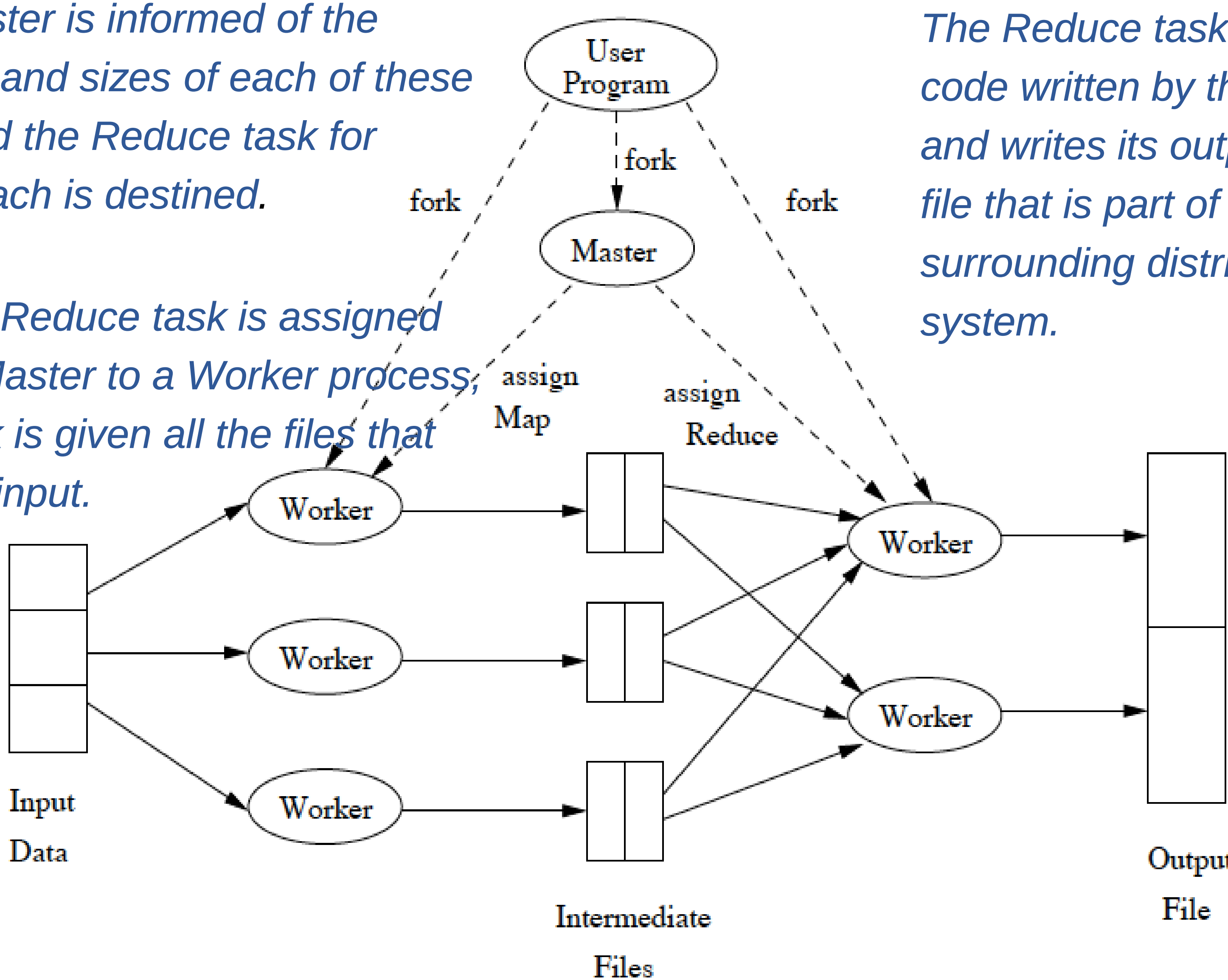



Figure 2.3: Overview of the execution of a map-reduce program

# 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

# Things Map-Reduce is Good At

- Matrix-Matrix and Matrix-vector multiplication.
  - One step of the PageRank iteration was the original application.
- Relational algebra operations.
- Many other  parallel operations.

# Matrix-Vector Multiplication

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

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

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

# Matrix-Vector Multiplication

- The matrix $M$ and the vector **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 **v** will be discoverable in the analogous way.

# Matrix-Vector Multiplication

- The Map Function:
  - Each Map task will take the entire vector **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 tasks don't really do anything except pass their input tuples as key-value pairs to the Reduce tasks.
  - The latter need only eliminate duplicates.
- 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, 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 **t** , do the following.
    - If the associated value list is [**R**], then produce (**t, t**).
    - If the associated value list is anything else, which could only be [**R, S**],[**S,R**], or [**S**], produce (**t,** 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**), . . .]),
  - 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 [$b_1$, $b_2$, . . . , $b_n$] of b-values associated with key a. The output is the pair (a, x), where $x = b_1 + b_2 + . . . + b_n$.

Thank you!