Principles of Distributed Database Systems

M. Tamer Özsu
Patrick Valduriez

 1

1

Outline

- Introduction
- Distributed and Parallel Database Design
- Distributed Data Control
- Distributed Query Processing
- Distributed Transaction Processing
- Data Replication
- Database Integration – Multidatabase Systems
- Parallel Database Systems
- Peer-to-Peer Data Management
- Big Data Processing
- NoSQL, NewSQL and Polystores
- Web Data Management

 2

2

Outline

- Big Data Processing
  - Distributed storage systems
  - Processing platforms
  - Stream data management
  - Graph analytics
  - Data lake

 3

3

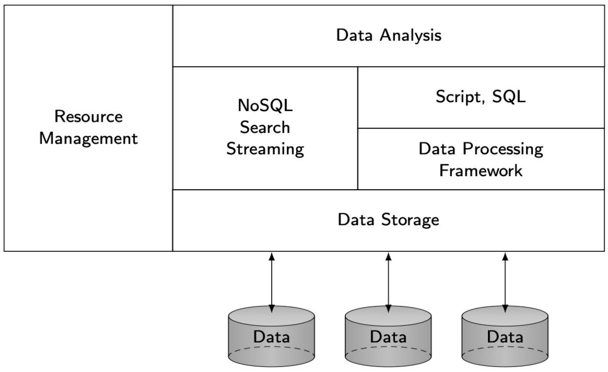
Four Vs

- Volume
  - Increasing data size: petabytes (10^{15}) to zettabytes (10^{21})
- Variety
  - Multimodal data: structured, images, text, audio, video
  - 90% of currently generated data unstructured
- Velocity
  - Streaming data at high speed
  - Real-time processing
- Veracity
  - Data quality

 4

4

Big Data Software Stack



Outline

- Big Data Processing
  - Distributed storage systems
  - Processing platforms
  - Stream data management
  - Graph analytics

Distributed Storage System

Storing and managing data across the nodes of a shared-nothing cluster

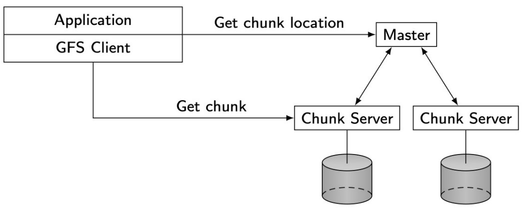
- Object-based
  - Object = ⟨oid, data, metadata⟩
  - Metadata can be different for different object
  - Easy to move
  - Flat object space → billions/trillions of objects
  - Easily accessed through REST-based API (`get`/`put`)
  - Good for high number of small objects (photos, mail attachments)
- File-based
  - Data in files of fixed- or variable-length records
  - Metadata-per-file stored separately from file
  - For large data, a file needs to be partitioned and distributed

 7

7

Google File System (GFS)

- Targets shared-nothing clusters of thousands of machines
- Targets applications with characteristics:
  - Very large files (several gigabytes)
  - Mostly read and append workloads
  - High throughput more important than low latency
- Interface: `create`, `open`, `read`, `write`, `close`, `delete`, `snapshot`, `record append`



 8

8

Outline

- Big Data Processing
  - Distributed storage systems
  - Processing platforms
  - Stream data management
  - Graph analytics

Big Data Processing Platforms

- Applications that do not need full DBMS functionality
  - Data analysis of very large data sets
  - Highly dynamic, irregular, schemaless, …
- "Embarrassingly parallel problems"
- MapReduce/Spark
- Advantages
  - Flexibility
  - Scalability
  - Efficiency
  - Fault-tolerance
- Disadvantage
  - Reduced functionality
  - Increased programming effort

MapReduce Basics

- Simple programming model
  - Data structured as (key, value) pairs
    - E.g. (doc-id, content); (word, count)
  - Functional programming style with two functions
    - `map(k1, v1)` → `list(k2, v2)`
    - `reduce(k2, list(v2))` → `list(v3)`
- Implemented on a distributed file system (e.g. Google File System) on very large clusters

`map` Function

- User-defined function
  - Processes input (key, value) pairs
  - Produces a set of intermediate (key, value) pairs
  - Executes on multiple machines (called mapper)
- `map` function I/O
  - **Input**: read a chunk from distributed file system (DFS)
  - **Output**: Write to intermediate file on local disk
- MapReduce library
  - Execute `map` function
  - Groups together all intermediate values with same key
  - Passes these lists to `reduce` function
- Effect of `map` function
  - Processes and partitions input data
  - Builds a distributed map (transparent to user)
  - Similar to "group by" operator in SQL

reduce Function

- User-defined function
  - Accepts one intermediate key and a set of values for that key (i.e. a list)
  - Merges these values together to form a (possibly) smaller set
  - Computes the reduce function generating, typically, zero or one output per invocation
  - Executes on multiple machines (called reducer)
- reduce function I/O
  - **Input**: read from intermediate files using remote reads on local files of corresponding mappers
  - **Output**: Write result back to DFS
- Effect of map function
  - Similar to aggregation function in SQL

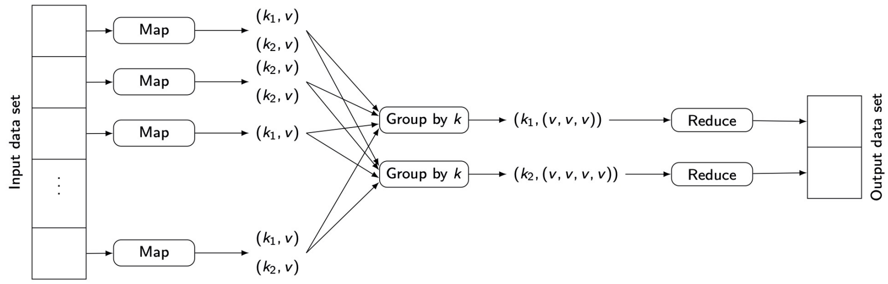
Example

Consider `EMP(ENO,ENAME,TITLE,CITY)`

```
SELECT     CITY, COUNT(*)
FROM       EMP
WHERE      ENAME LIKE "%Smith"
GROUP BY   CITY
```

```
map (Input: (TID,EMP), Output: (CITY, 1)
   if EMP.ENAME like ``\%Smith'' return (CITY, 1)
reduce (Input: (CITY, list(1)), Output: (CITY,
SUM(list)))
   return (CITY, SUM(1))
```

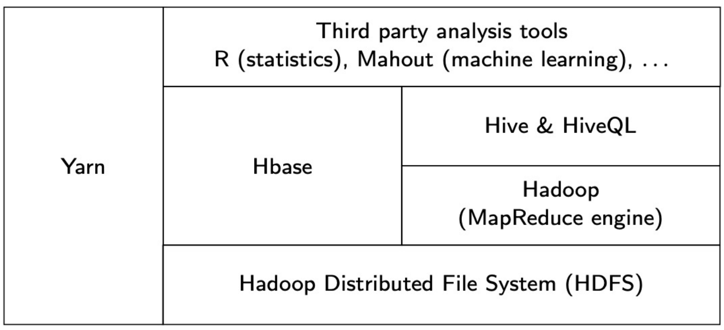

MapReduce Processing



 15

15

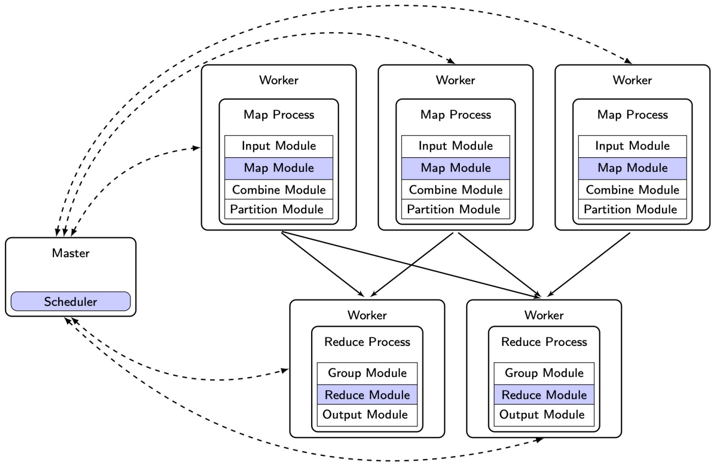
Hadoop Stack



 16

16

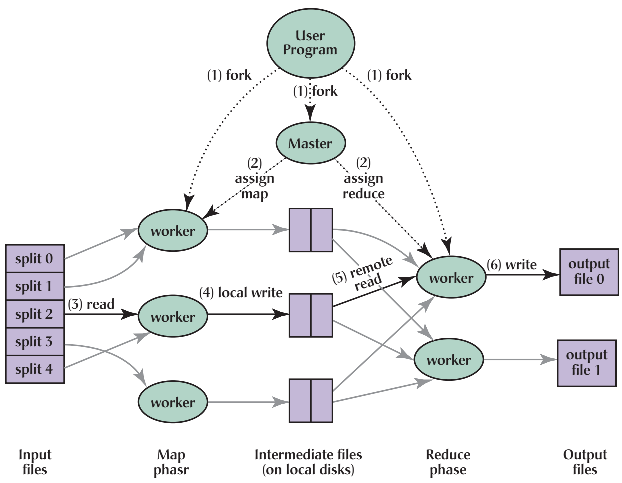
Master-Worker Architecture



 17

17

Execution Flow

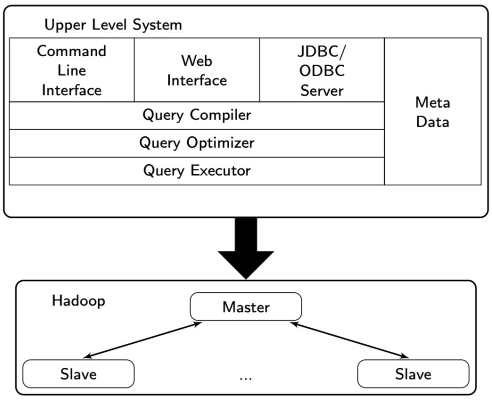


From: J. Dean and S.Ghemawat. MapReduce: Simplified data processing on large clusters, *Comm. ACM*, 51(1), 2008.

 18

18

High-Level MapReduce Languages

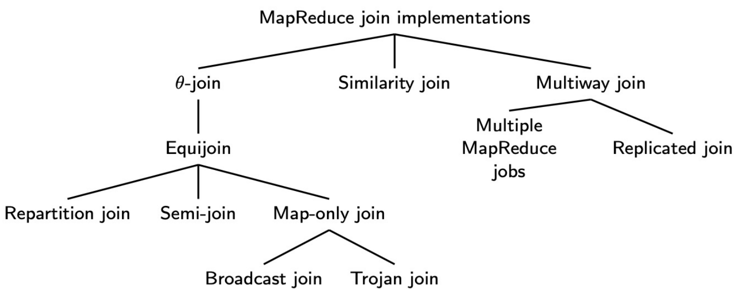


- Declarative
  - HiveQL
  - Tenzing
  - JAQL
- Data flow
  - Pig Latin
- Procedural
  - Sawzall
- Java Library
  - FlumeJava

19

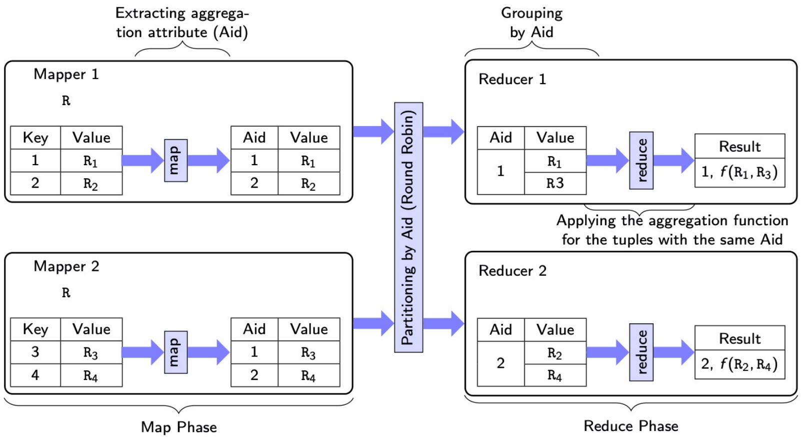
MapReduce Implementations of DB Ops

- `Select` and `Project` can be easily implemented in the map function
- `Aggregation` is not difficult (see next slide)
- `Join` requires more work



20

Aggregation



 21

21

θ-Join

Baseline implementation of $R(A,B) \bowtie S(B,C)$

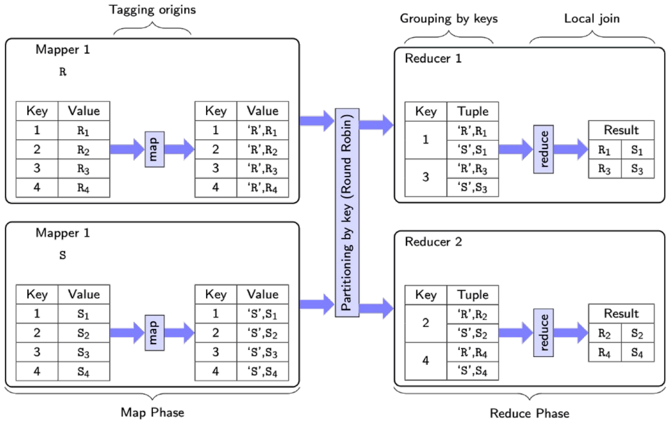
1) Partition R and assign each partition to mappers
2) Each mapper takes $\langle a,b \rangle$ tuples and converts them to a list of key-value pairs of the form $(b, \langle a,R \rangle)$
3) Each reducer pulls the pairs with the same key
4) Each reducer joins tuples of R with tuples of S

 22

22

θ-Join (θ is =)

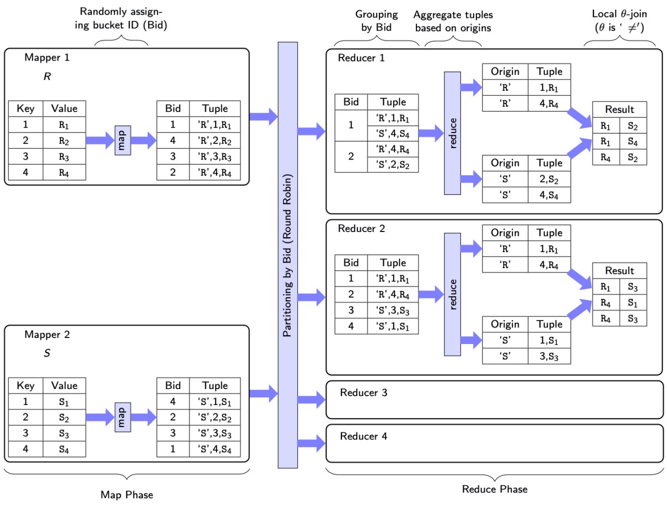
- Repartition join



 23

23

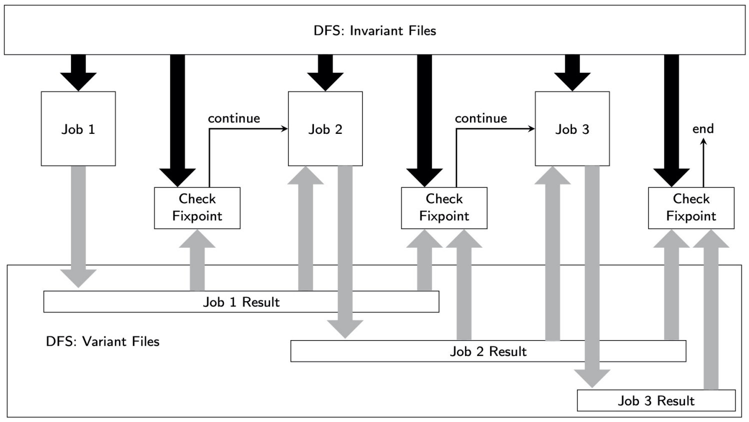
θ-Join (θ is ≠)



 24

24

MapReduce Iterative Computation



 25

25

Problems with Iteration

- MapReduce workflow model is acyclic
  - Iteration: Intermediate results have to be written to HDFS after each iteration and read again
- At each iteration, no guarantee that the same job is assigned to the same compute node
  - Invariant files cannot be locally cached
- Check for fixpoint
  - At the end of each iteration, another job is needed

 26

26

Spark

- Addresses MapReduce shortcomings
- Data sharing abstraction: Resilient Distributed Dataset (RDD)

1) Cache working set (i.e. RDDs) so no writing-to/reading-from HDFS
2) Assign partitions to the same machine across iterations
3) Maintain lineage for fault-tolerance

Spark Program Flow