Big Data Processing

M. Tamer Özsu
Patrick Valduriez

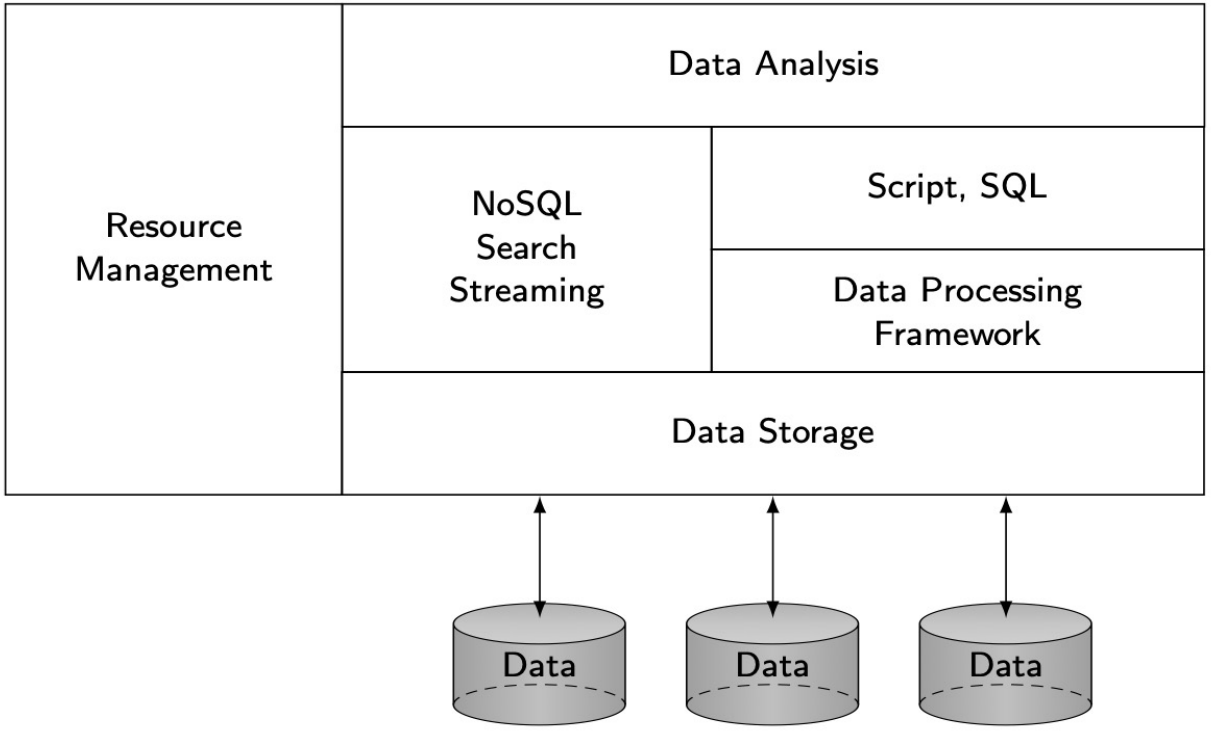
Outline

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

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

Big Data Software Stack



Outline

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

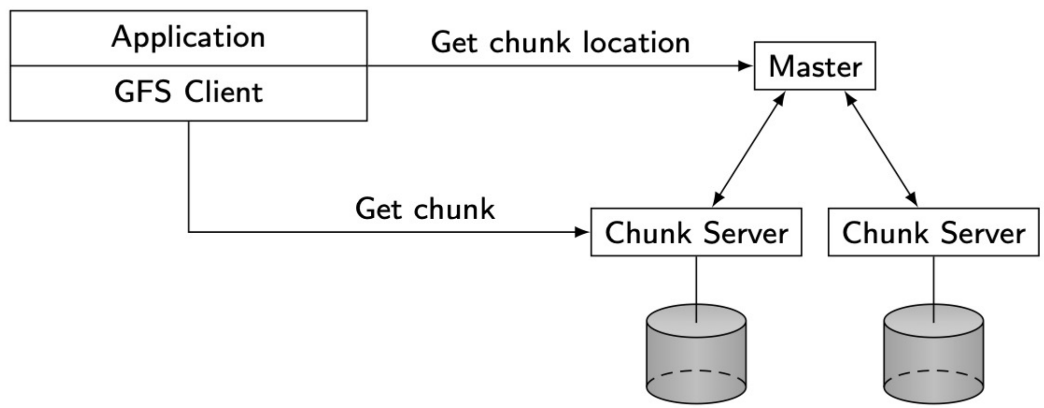
Distributed Storage System

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

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`



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

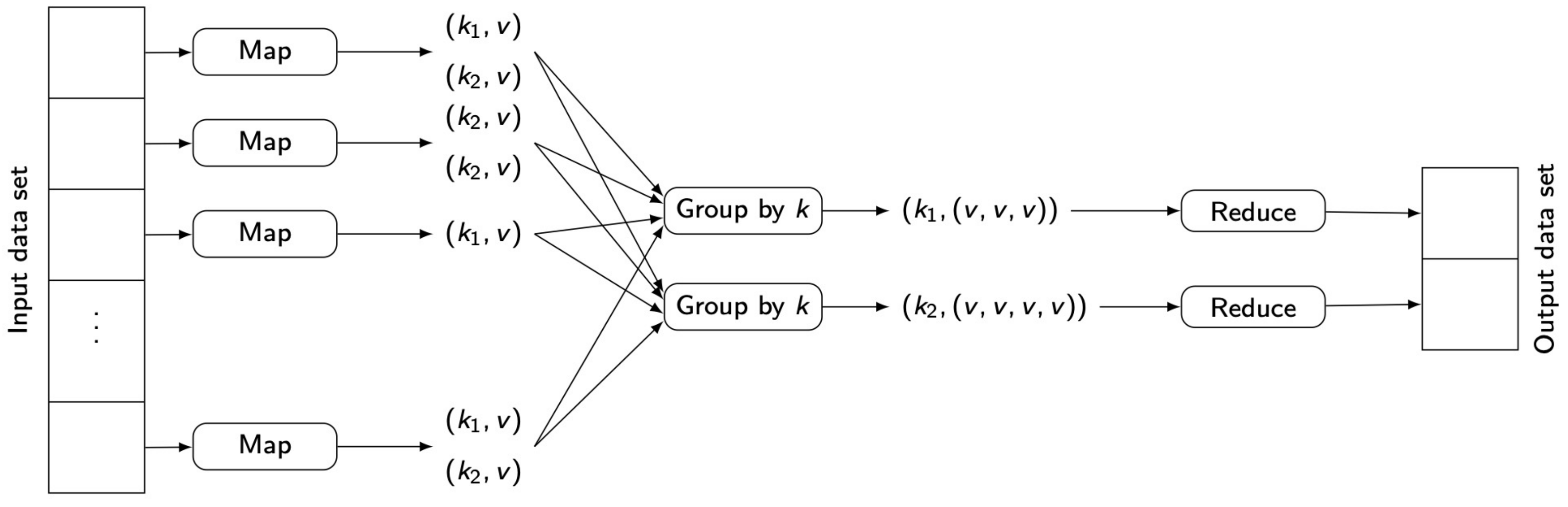
Example

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

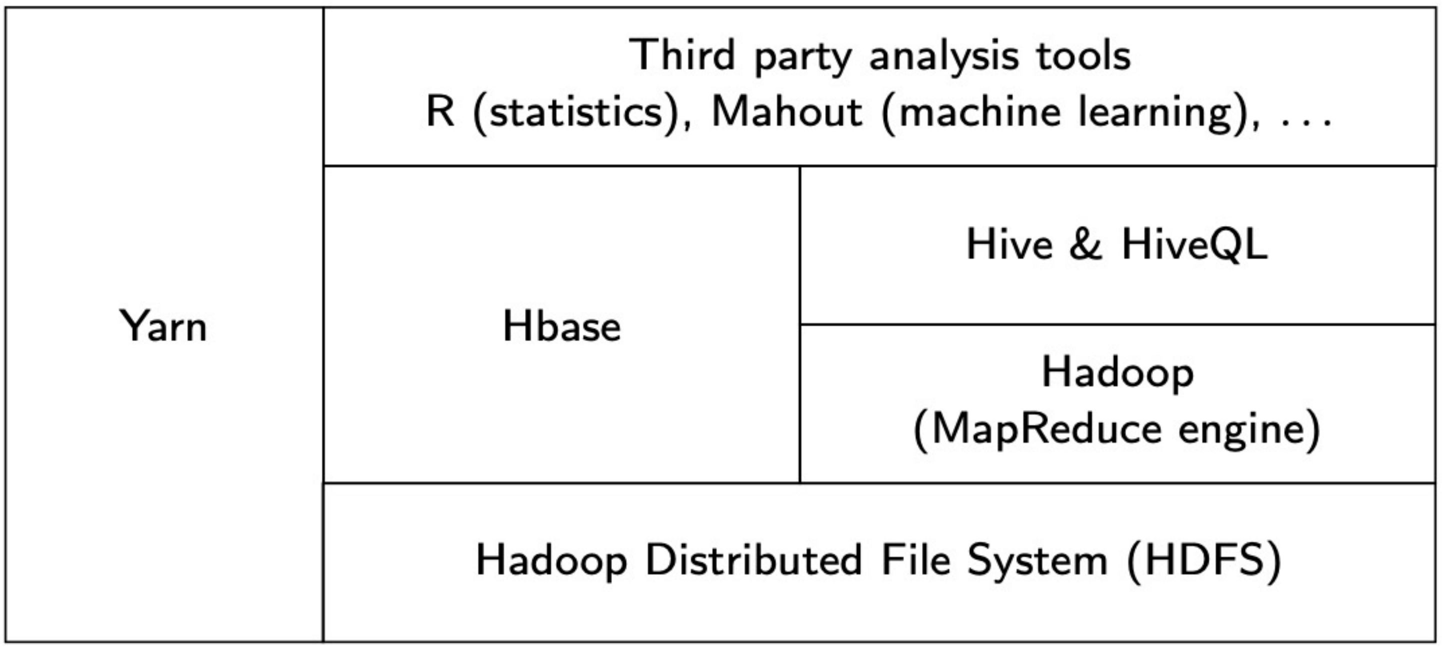
```
SELECT      CITY, COUNT(*)
FROM        EMP
WHERE       ENAME LIKE "%Smith"
GROUP BY    CITY
```

```
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))
```

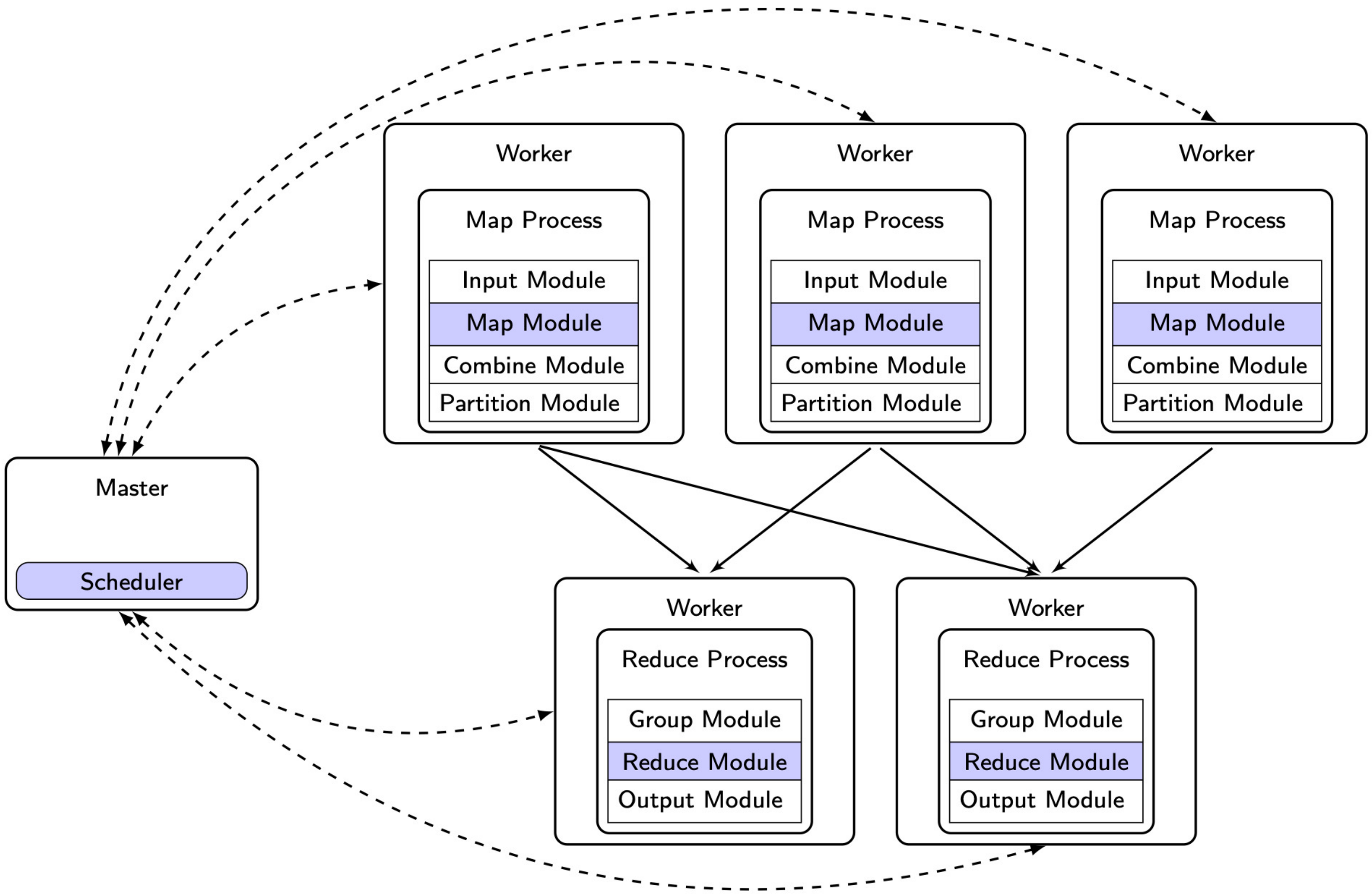
MapReduce Processing



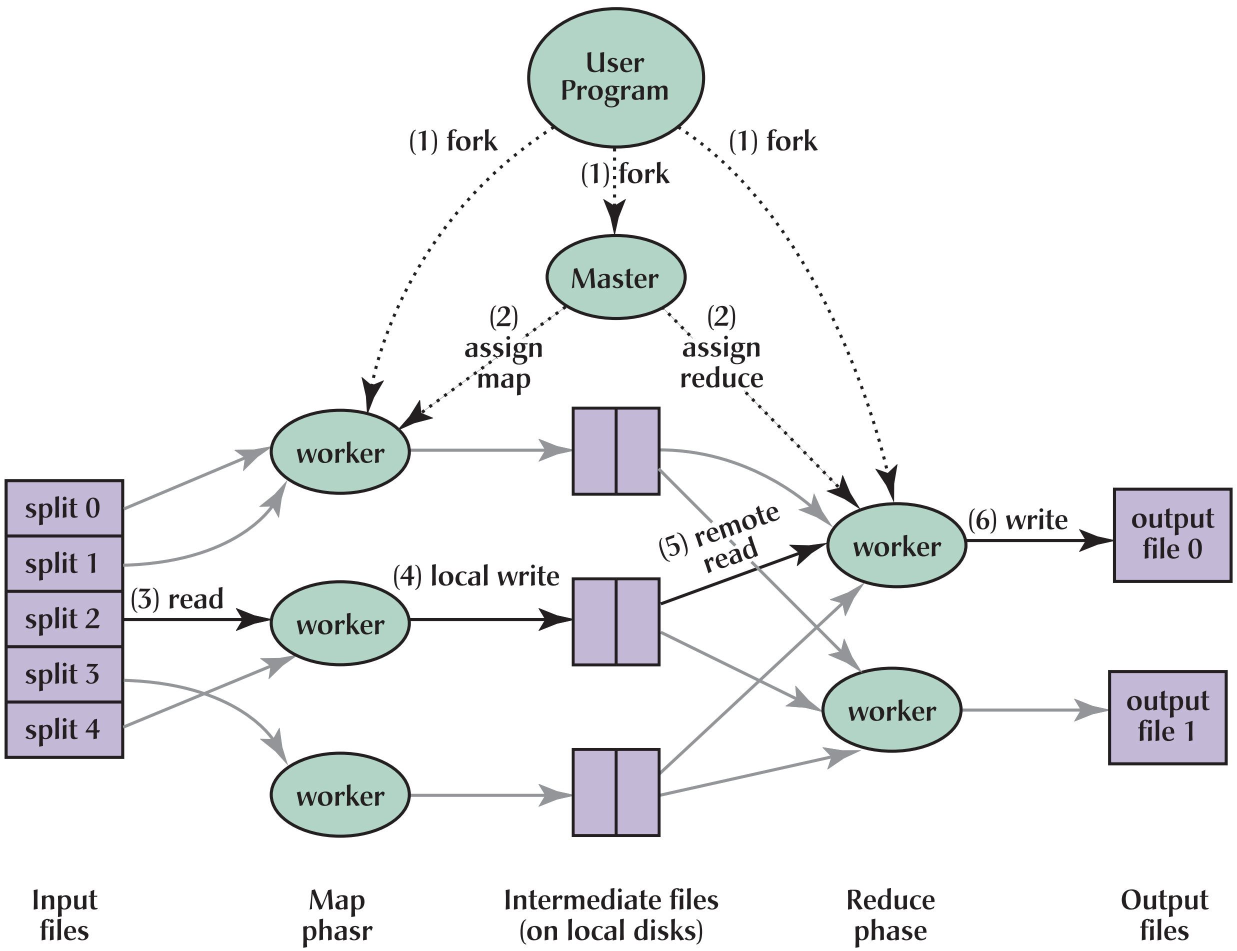
Hadoop Stack



Master-Worker Architecture

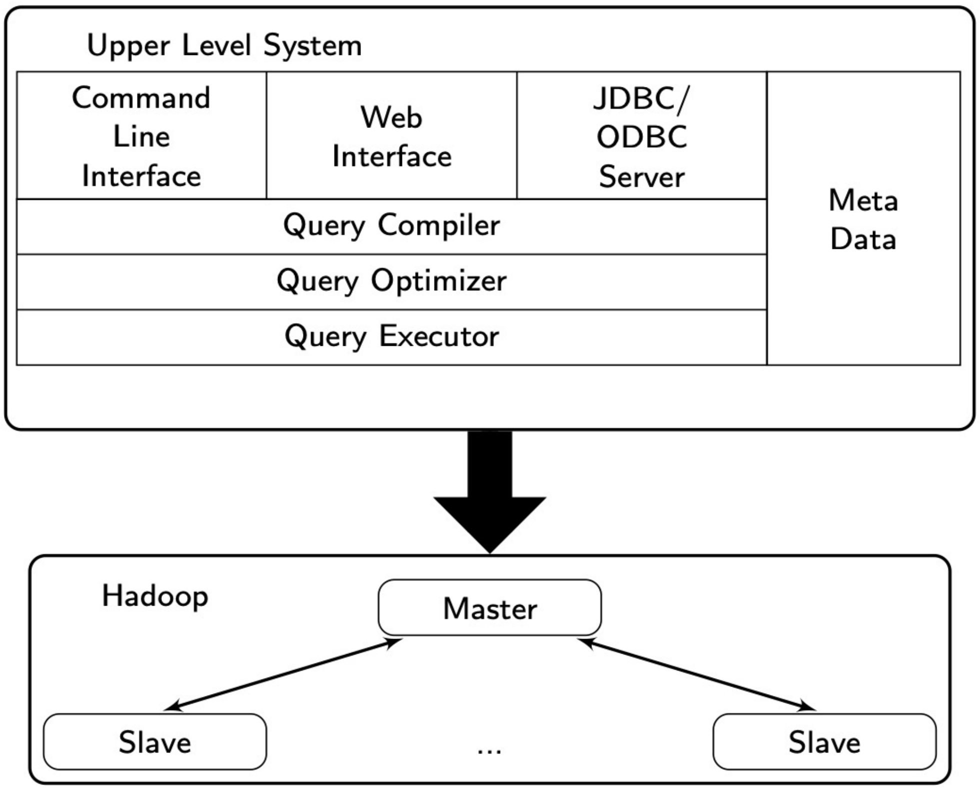


Execution Flow



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

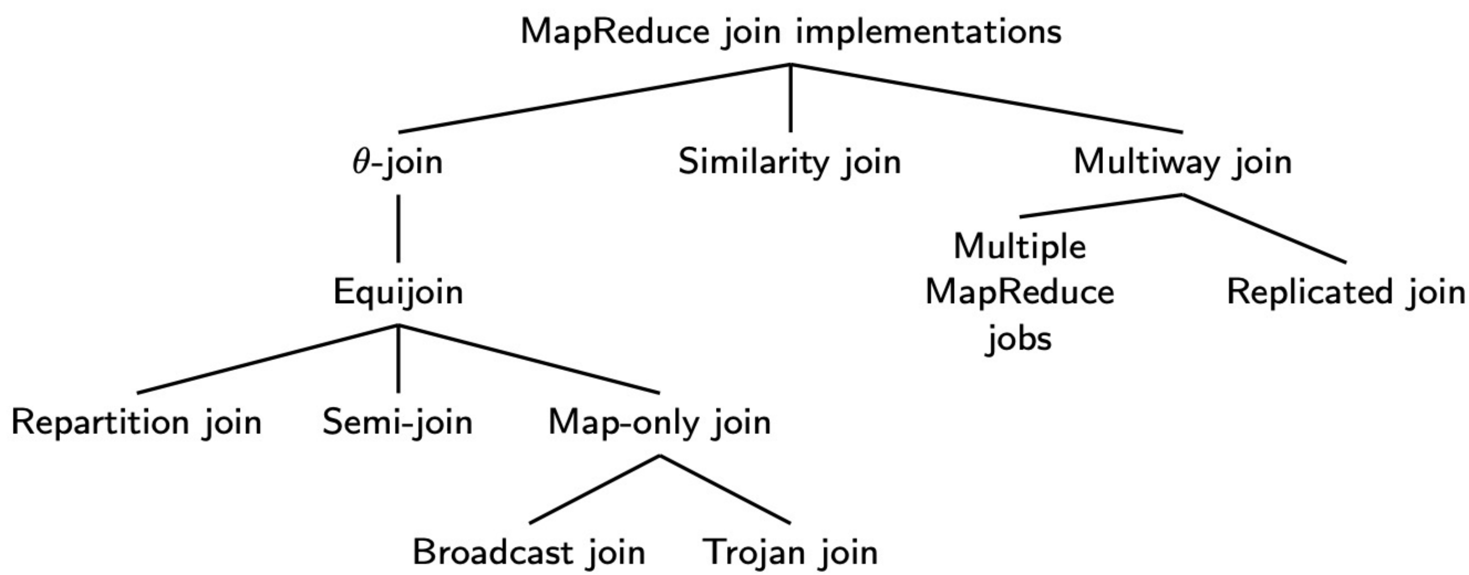
High-Level MapReduce Languages



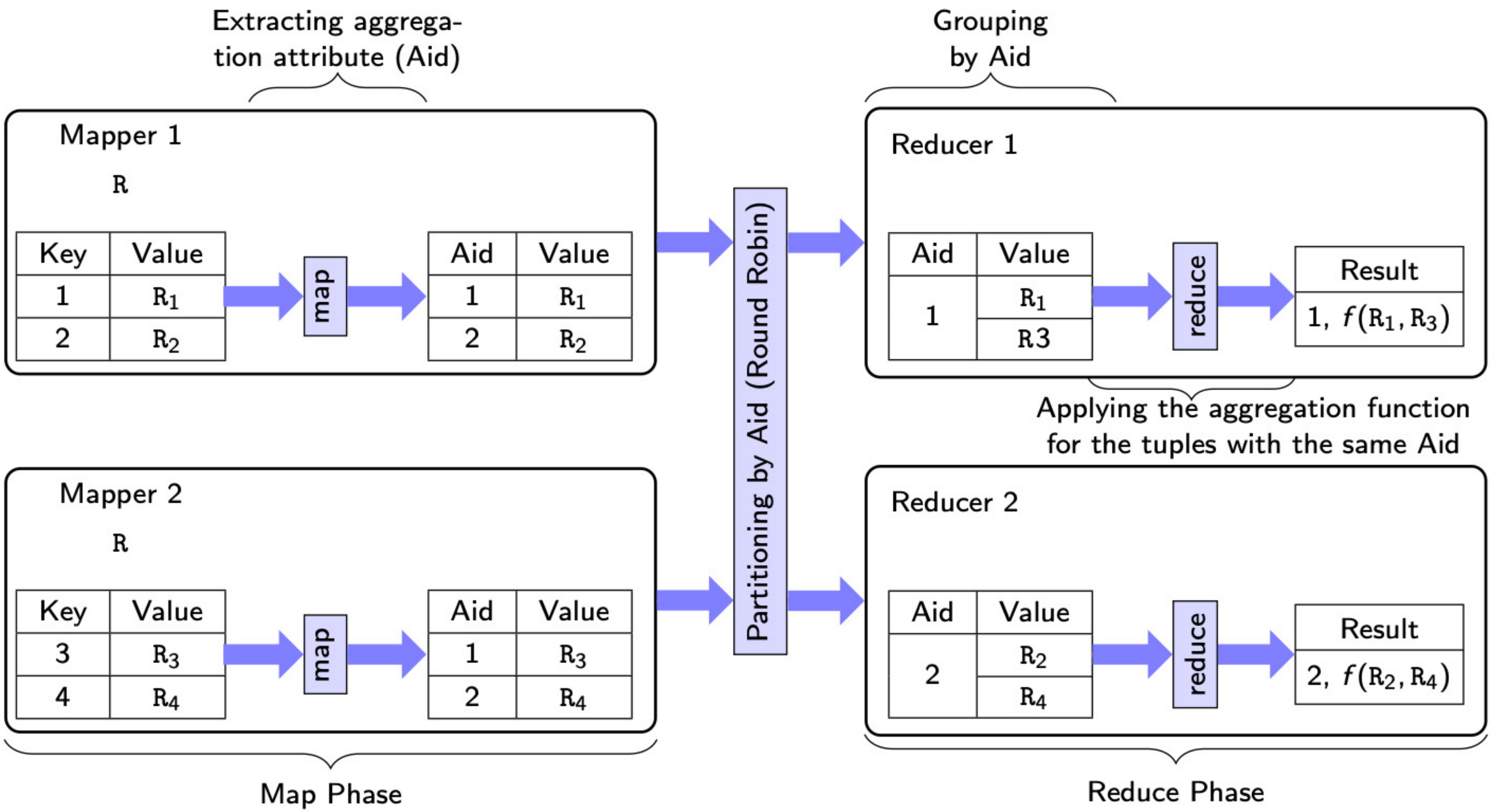
- Declarative
  - HiveQL
  - Tenzing
  - JAQL
- Data flow
  - Pig Latin
- Procedural
  - Sawzall
- Java Library
  - FlumeJava

MapReduce Implementations of DB Ops

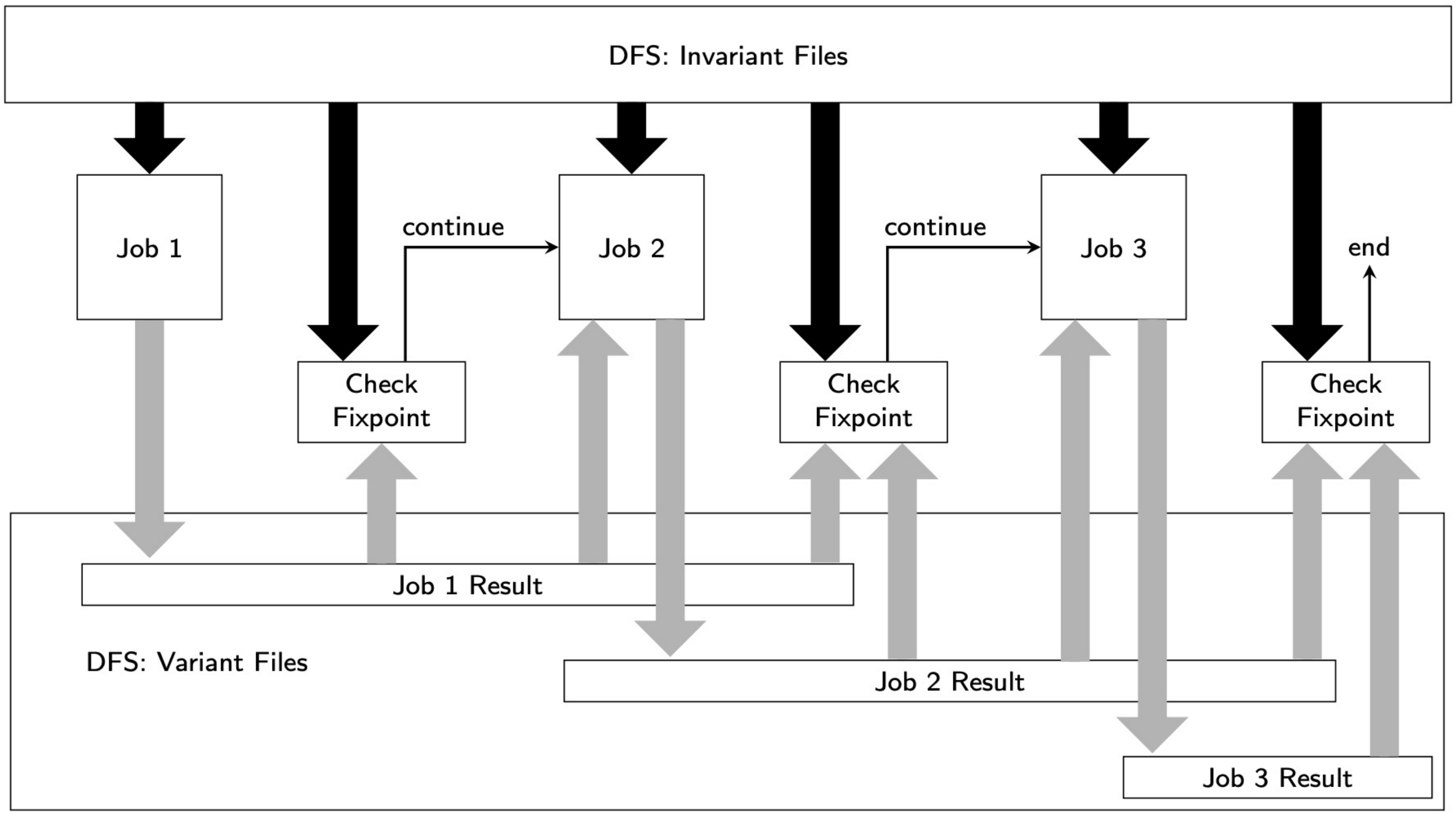
- `Select` and `Project` can be easily implemented in the map function
- `Aggregation` is not difficult (see next slide)
- `Join` requires more work



Aggregation



MapReduce Iterative Computation



Problems with Iteration

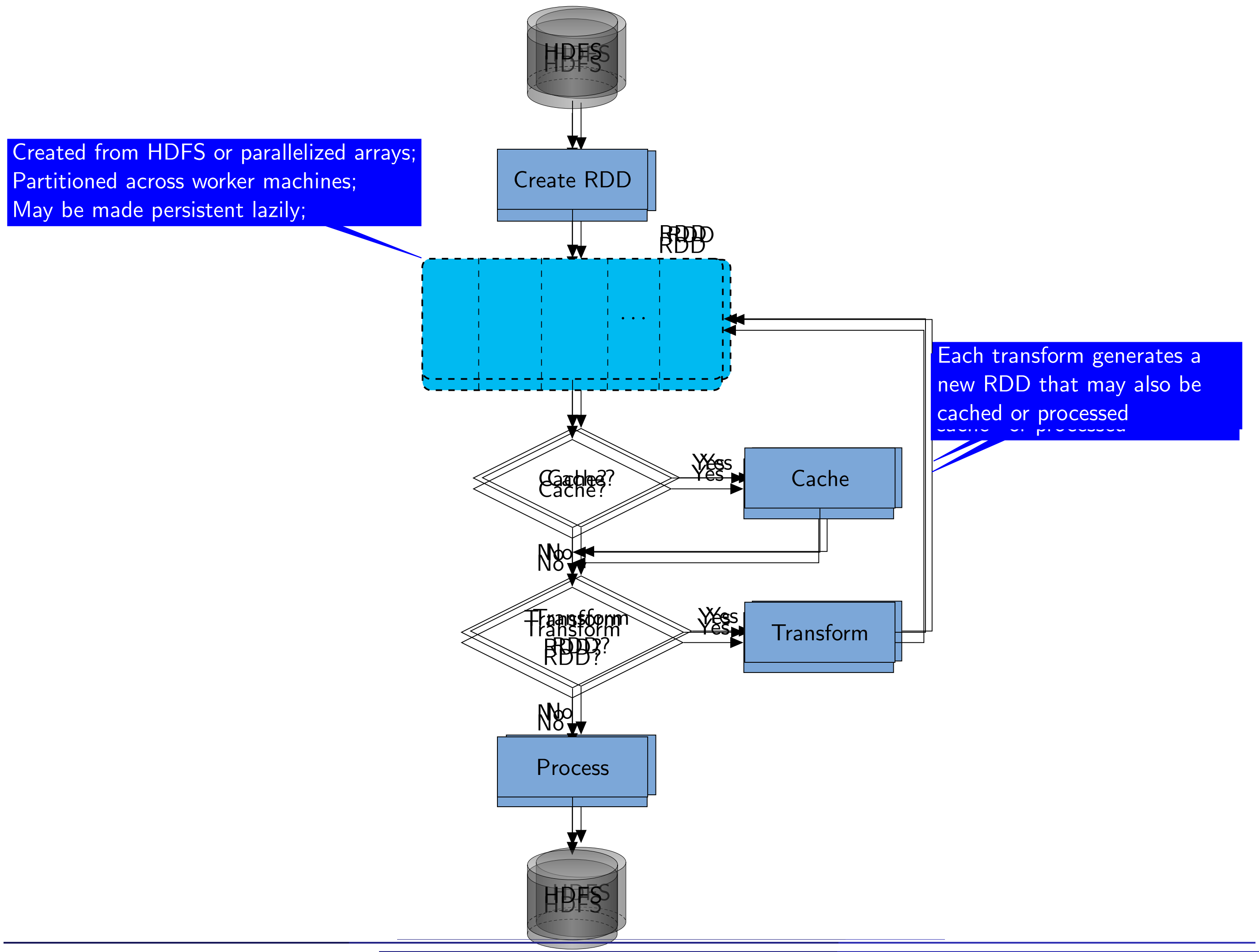
- MapReduce workflow model is acyclic
  - Iteration: Intermediate results have to be written to HDFS after each iteration and read again
- At each iteration, no guarantee that the same job is assigned to the same compute node
  - Invariant files cannot be locally cached
- Check for fixpoint
  - At the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

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

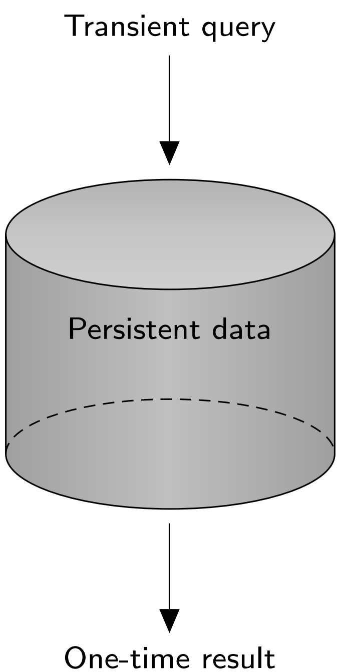


Outline

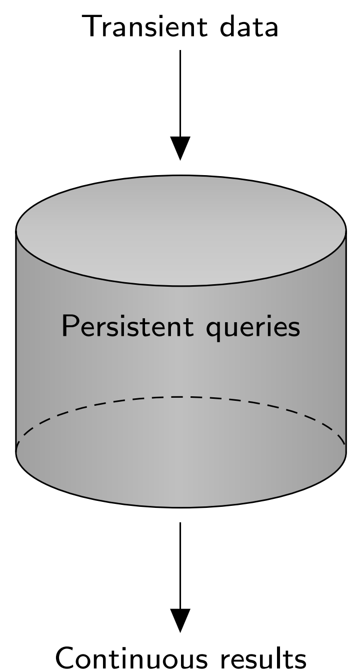
- Big Data Processing
  - Distributed storage systems
  - Processing platforms
  - Stream data management
  - Graph analytics

Traditional DBMS vs Streaming

DBMS



Streaming

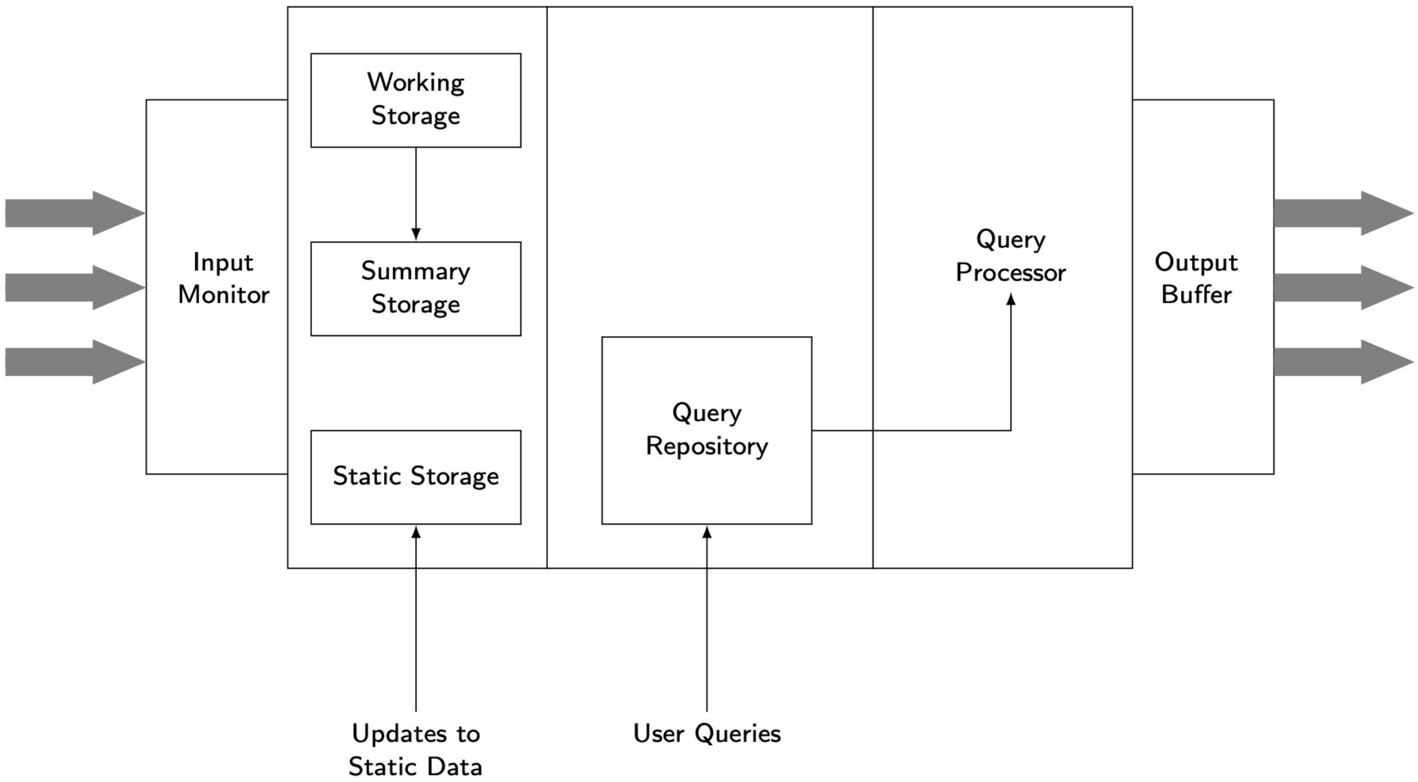


- Other differences
  - Push-based (data-driven)
  - Persistent queries
  - Unbounded stream
  - System conditions may not be stable

History

- Data Stream Management System (DSMS)
  - Typical DBMS functionality, primarily query language
  - Earlier systems: STREAM, Gigascope, TelegraphCQ, Aurora, Borealis
  - Mostly single machine (except Borealis)
- Data Stream Processing System (DSPS)
  - Do not embody DBMS functionality
  - Later systems: Apache Storm, Heron, Spark Streaming, Flink, MillWheel, TimeStream
  - Almost all are distributed/parallel systems
- Use Data Stream System (DSS) when the distinction is not important

DSMS Architecture



Stream Data Model

- Standard def: An append-only sequence of timestamped items that arrive in some order
- Relaxations
  - Revision tuples
  - Sequence of events that are reported continually (publish/subscribe systems)
  - Sequence of sets of elements (bursty arrivals)
- Typical arrival:

$$\langle \text{timestamp, payload} \rangle$$

  - Payload changes based on system
    - Relational: tuple
    - Graph: edge
    - …

Processing Models

- Continuous
  - Each new arrival is processed as soon as it arrives in the system.
  - Examples: Apache Storm, Heron
- Windowed
  - Arrivals are batched in windows and executed as a batch.
  - For user, recently arrived data may be more interesting and useful.
  - Examples: Aurora, STREAM, Spark Streaming

Stream Query Models

- Queries are typically persistent
- They may be monotonic or non-monotonic
- Monotonic: result set always grows
  - Results can be updated incrementally
  - Answer is continuous, append-only stream of results
  - Results may be removed from the answer only by explicit deletions (if allowed)
- Non-monotonic: some answers in the result set become invalid with new arrivals
  - Recomputation may be necessary

Stream Query Languages

- Declarative
  - SQL-like syntax, stream-specific semantics
  - Examples: CQL, GSQL, StreaQuel
- Procedural
  - Construct queries by defining an acyclic graph of operators
  - Example: Aurora
- Windowed languages
  - `size`: window length
  - `slide`: how frequently the window moves
  - E.g.: `size=10min, slide=5sec`
- Monotonic vs non-monotonic

Outline

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

Data Lake

- Collection of raw data in native format
  - Each element has a unique identifier and metadata
  - For each business question, you can find the relevant data set to analyze it
- Originally based on Hadoop
  - Enterprise Hadoop

Advantages of a Data Lake

- Schema on read
  - Write the data as they are, read them according to a diagram (e.g. code of the Map function)
  - More flexibility, multiple views of the same data
- Multi-workload data processing
  - Different types of processing on the same data
  - Interactive, batch, real time
- Cost-effective data architecture
  - Excellent cost/performance and ROI ratio with SN cluster and open source technologies

Principles of a Data Lake

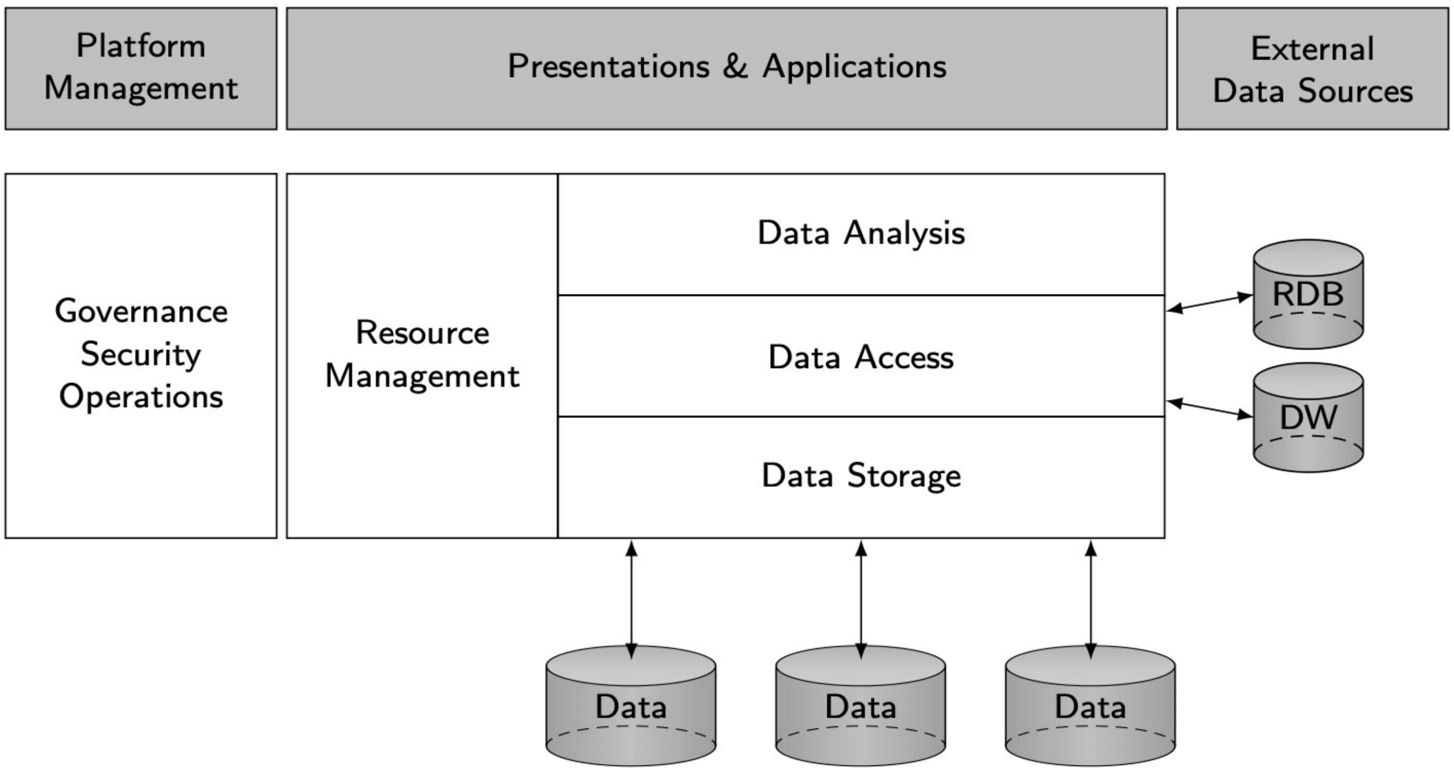
- Collect all useful data
  - Raw data, transformed data
- Dive from anywhere
  - Users from different business units can explore and enrich the data
- Flexible access
  - Different access paths to shared infrastructure
    - Batch, interactive (OLAP and BI), real-time, search,.....

Main Functions

- Data management, to store and process large amounts of data
- Data access: interactive, batch, real time, streaming
- Governance: load data easily, and manage it according to a policy implemented by the *data steward*
- Security: authentication, access control, data protection
- Platform management: provision, monitoring and scheduling of tasks (in a cluster)

Data Lake Architecture

A collection of multi-modal data stored in their raw formats



Data Lake vs Data Warehouse

Data Lake

- Shorter development process
- Schema-on-read
- Multiworkload processing
- Cost-effective architecture

Data Warehouse

- Long development process
- Schema-on-write
- OLAP workloads
- Complex development with ETL