Principles of Distributed Database Systems

TS. Phan Thị Hà

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

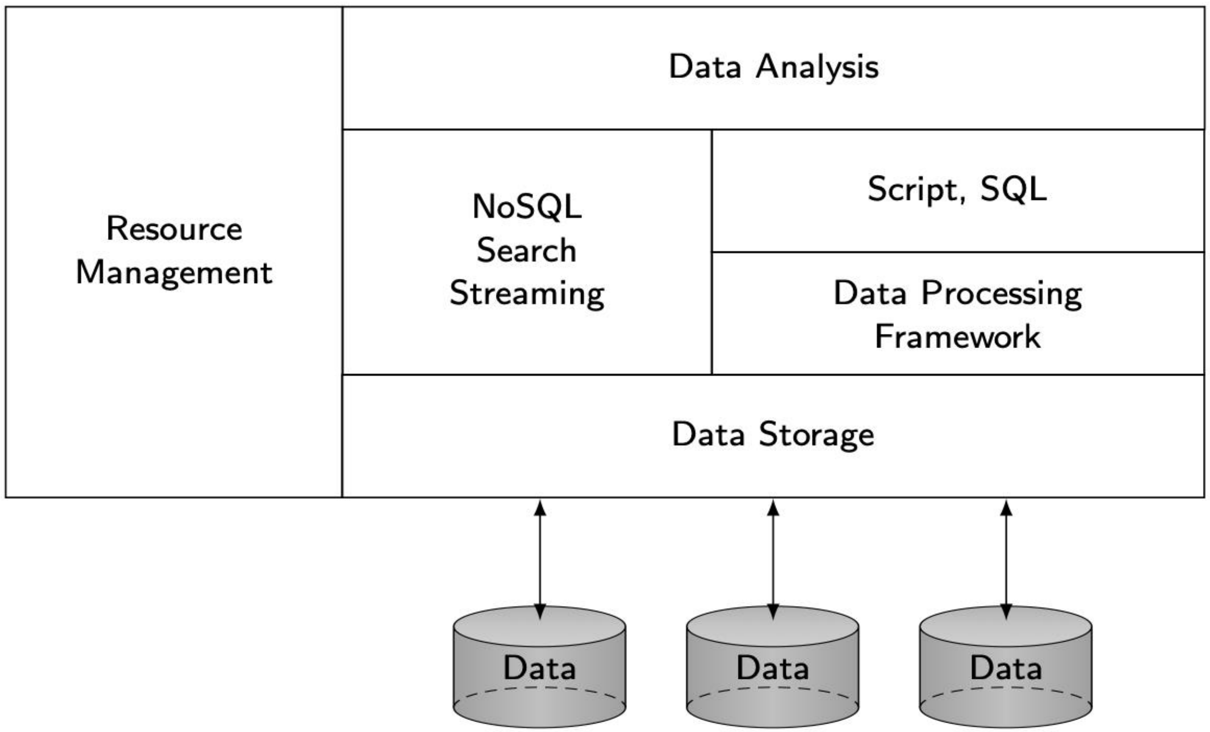
Outline

- 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

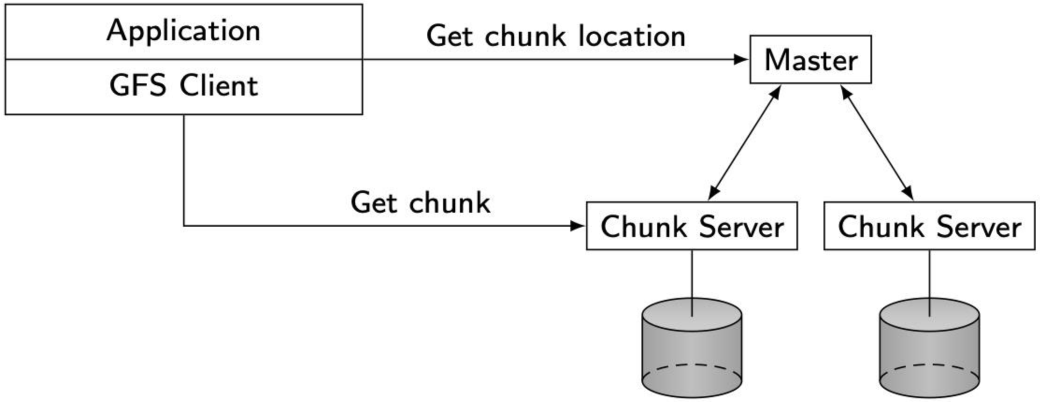
Distributed Storage System

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

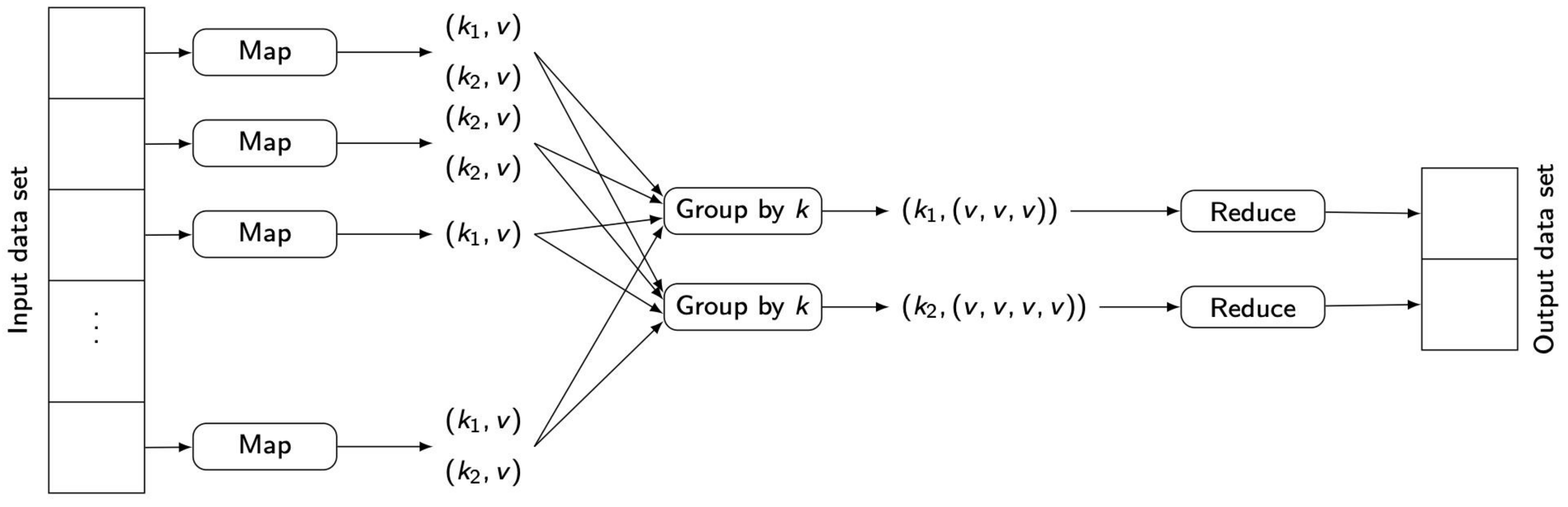
Example

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

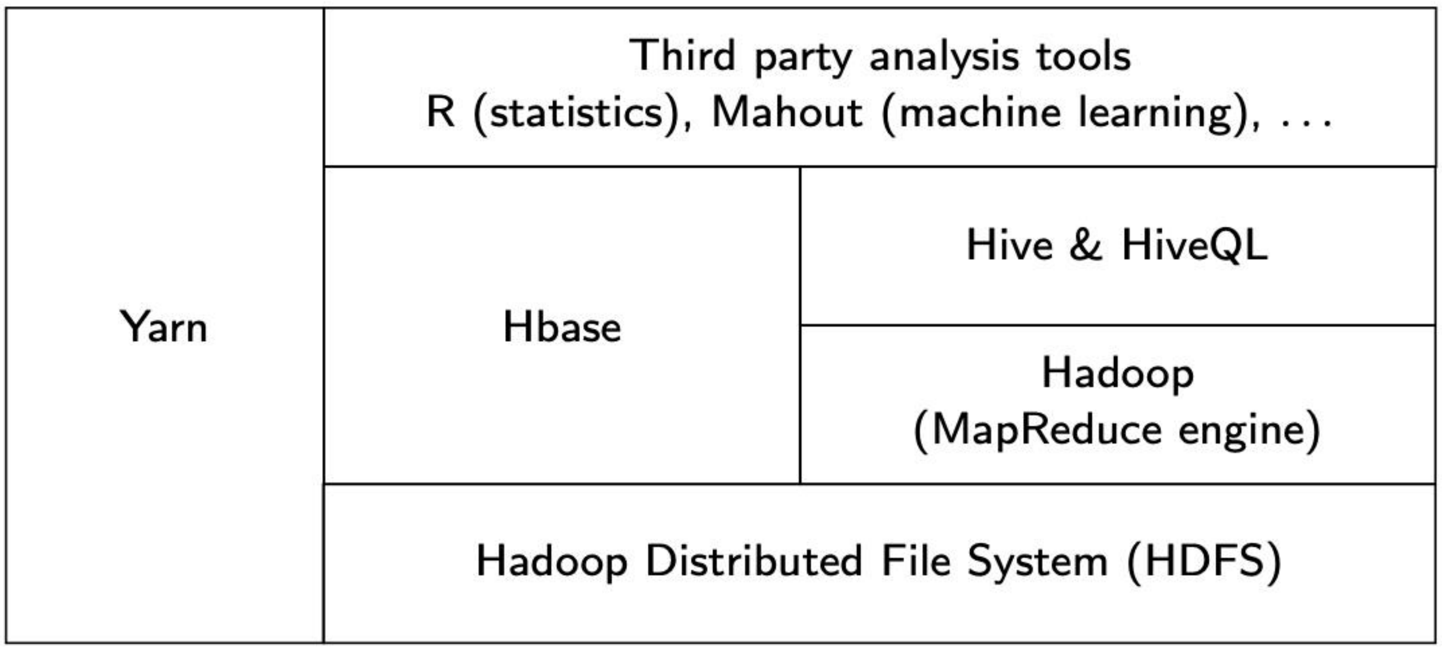
```
SELECT    CITY, COUNT(*)
FROM      EMP
WHERE     ENAME LIKE "%Smith"
GROUP BY  CITY
```

```
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))
```

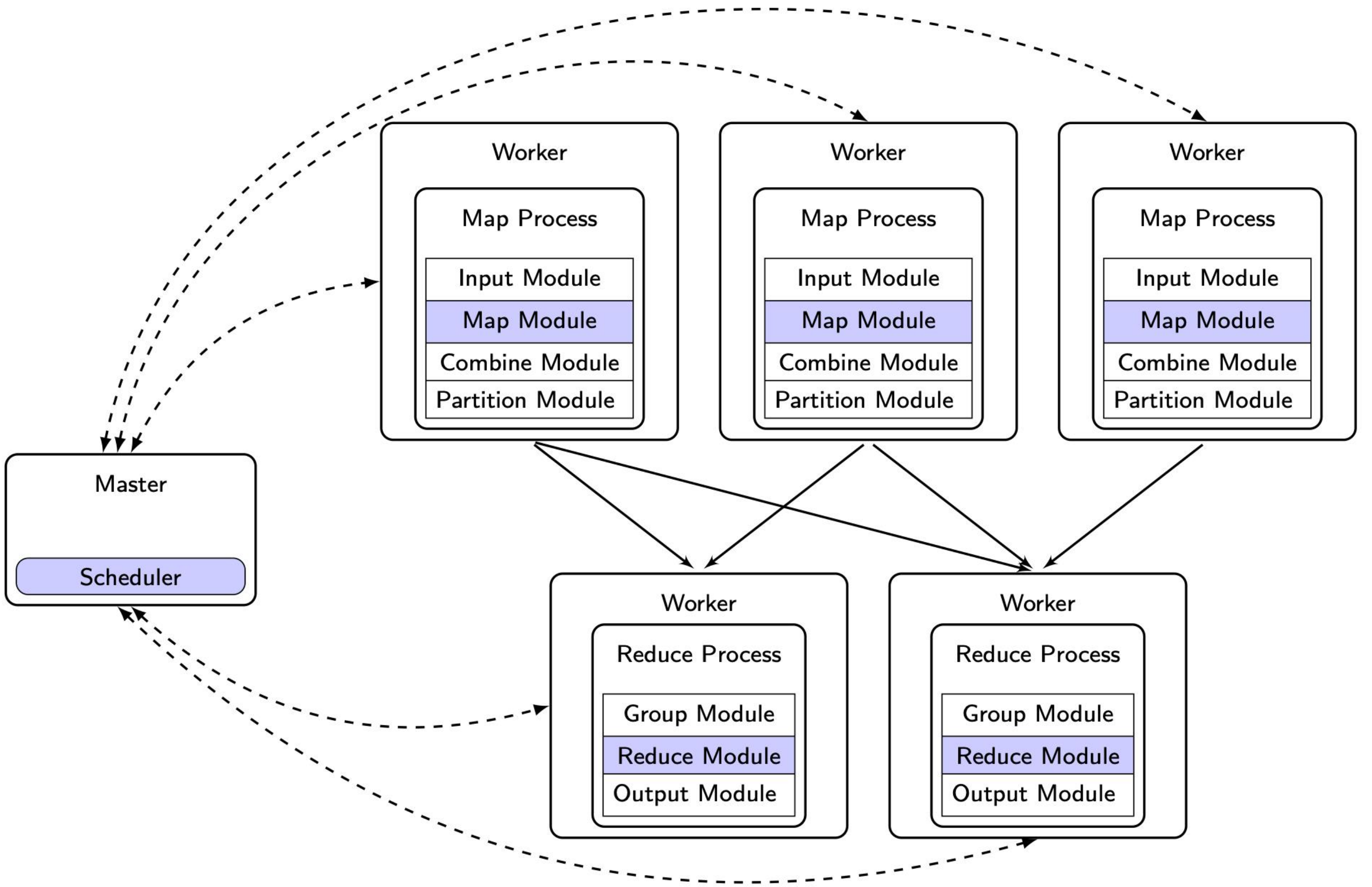
MapReduce Processing



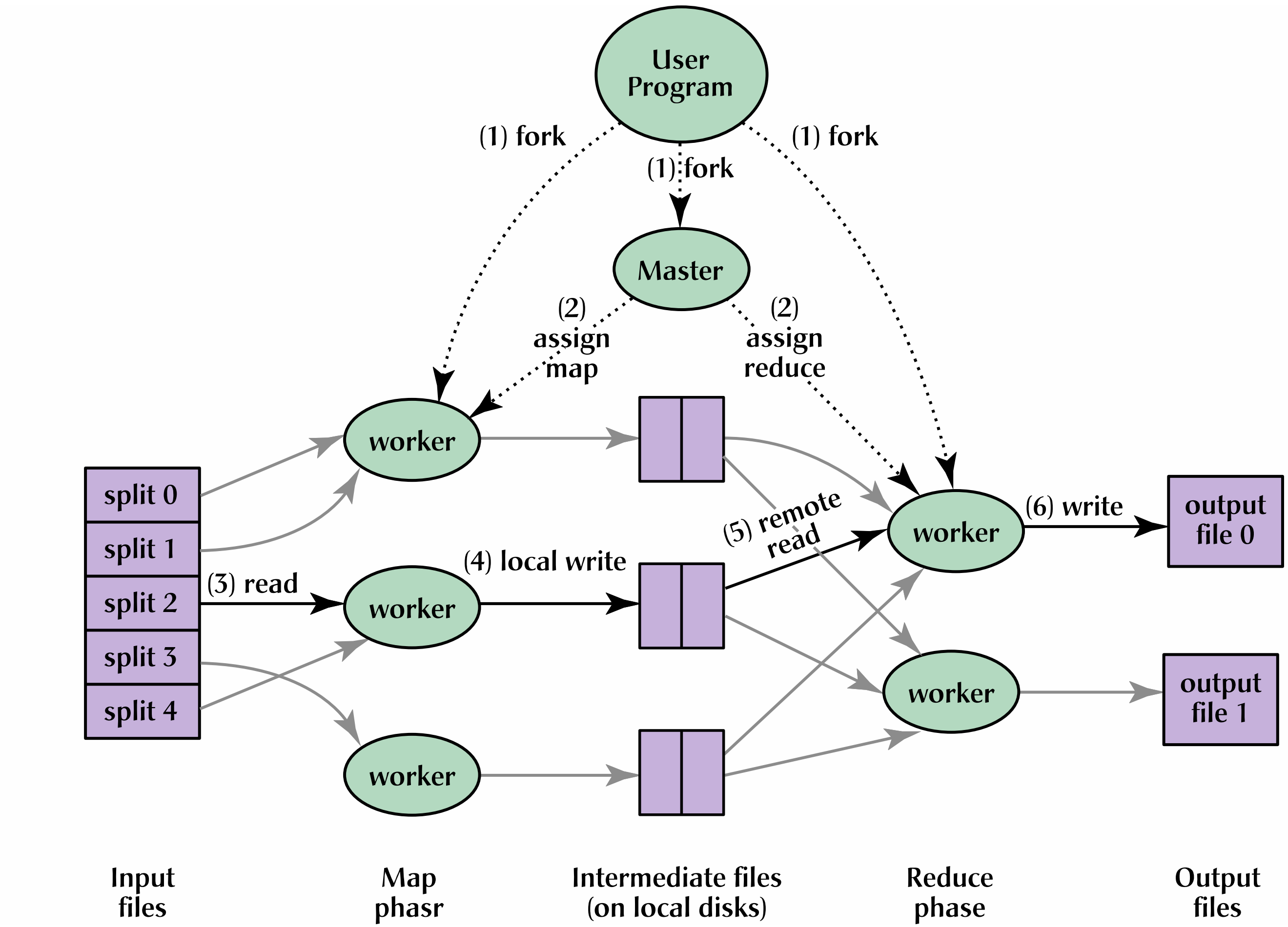
Hadoop Stack



Master-Worker Architecture

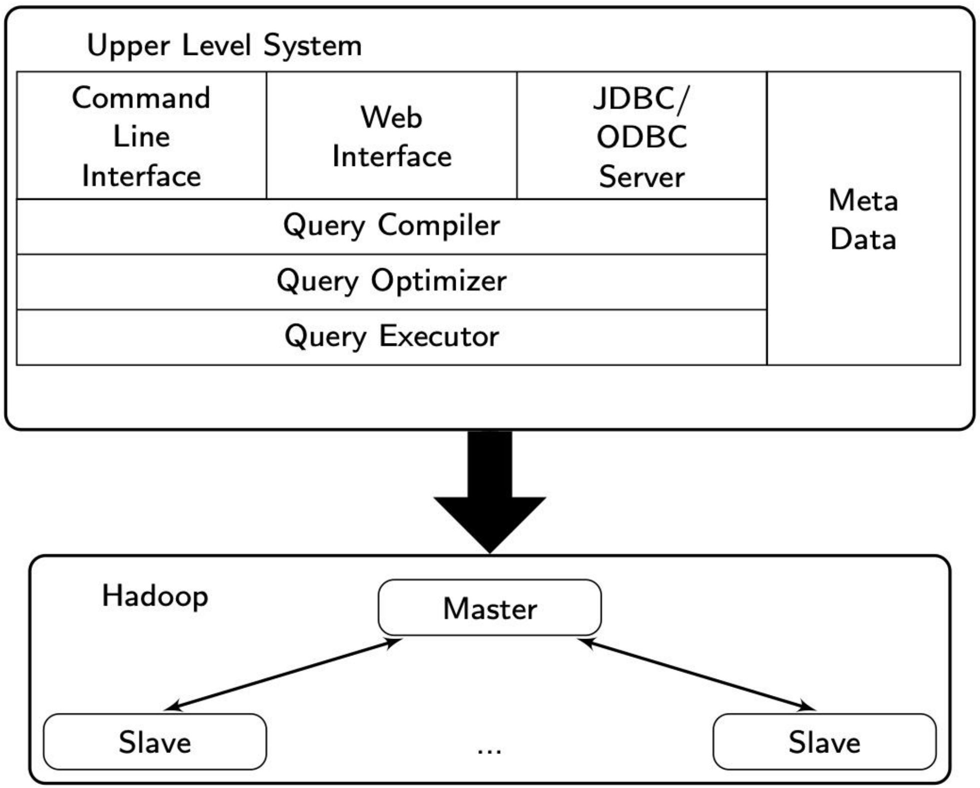


Execution Flow



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

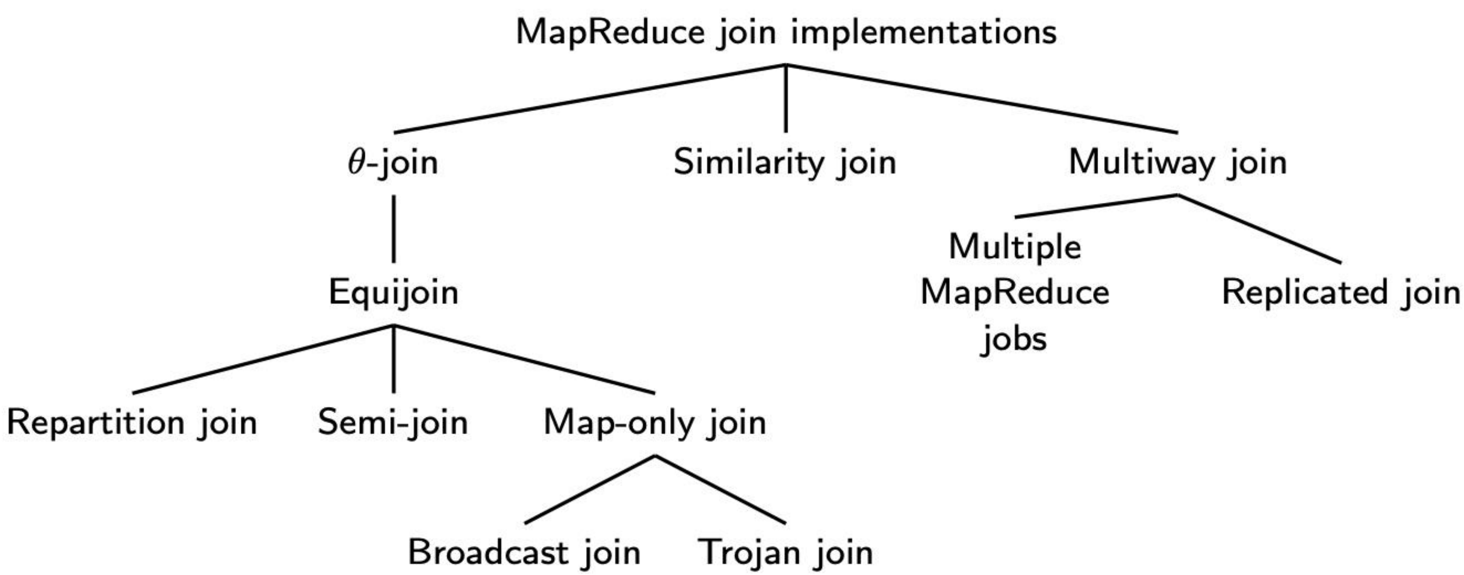
High-Level MapReduce Languages



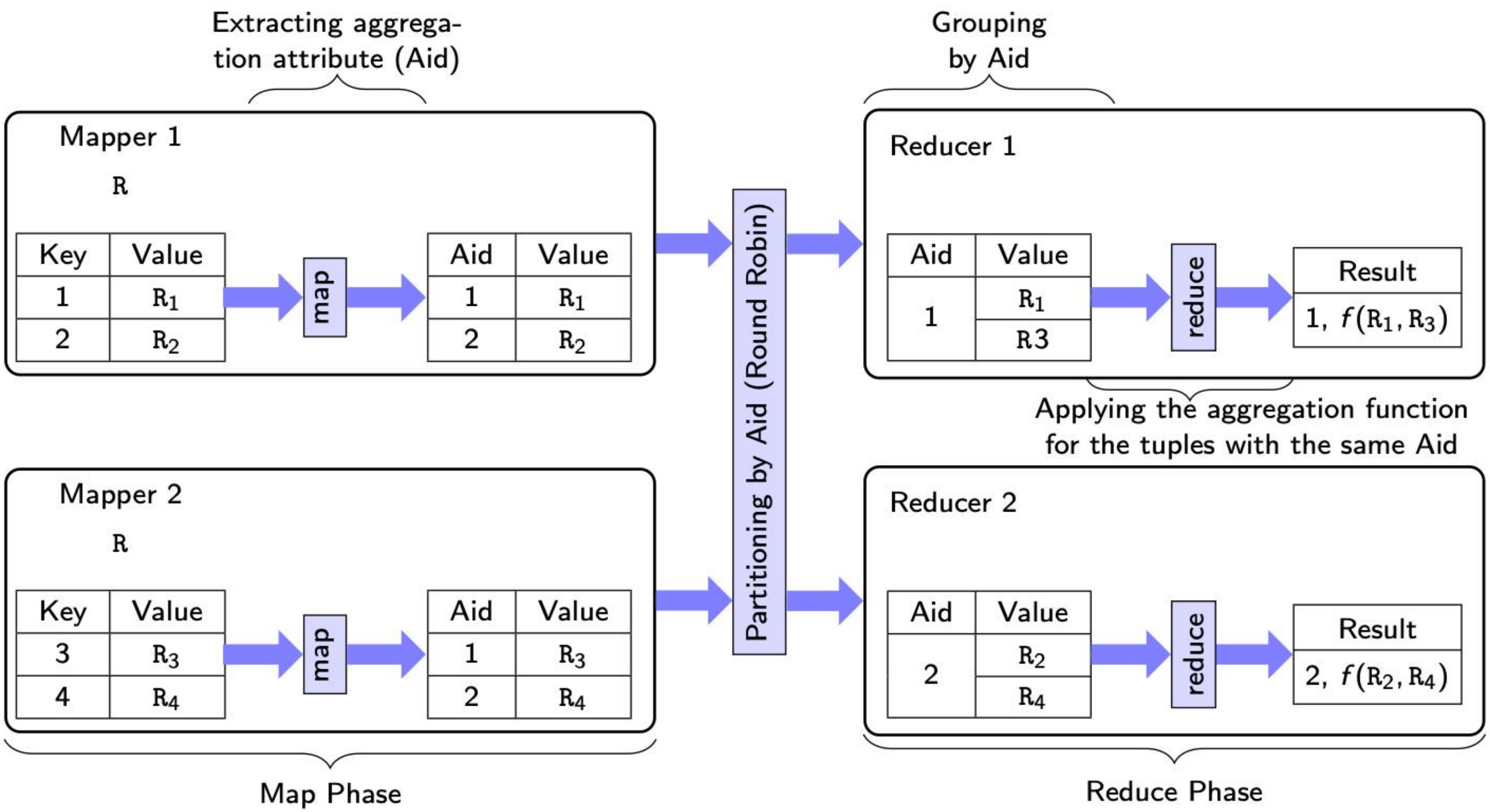
- Declarative
  - HiveQL
  - Tenzing
  - JAQL
- Data flow
  - Pig Latin
- Procedural
  - Sawzall
- Java Library
  - FlumeJava

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



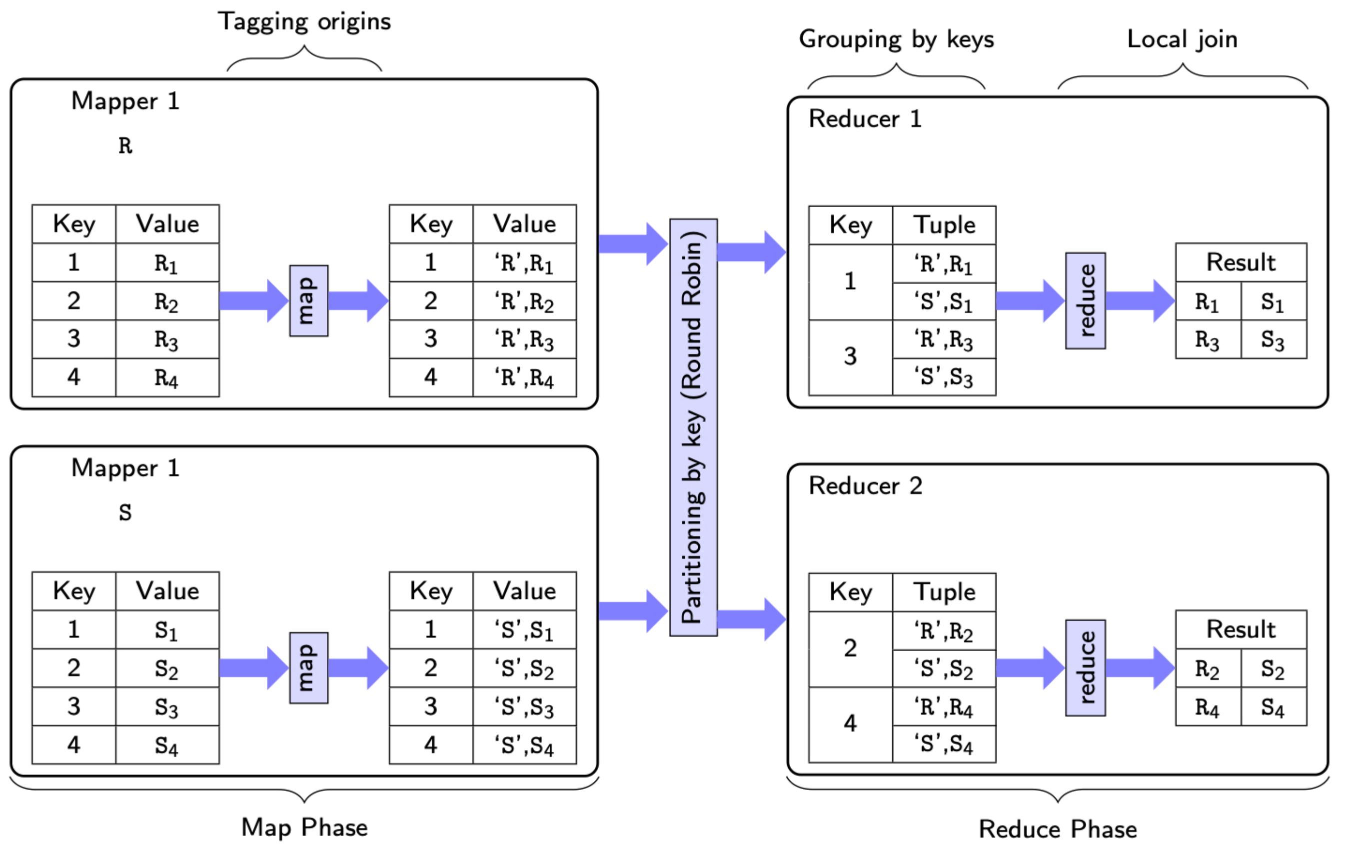
θ-Join

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

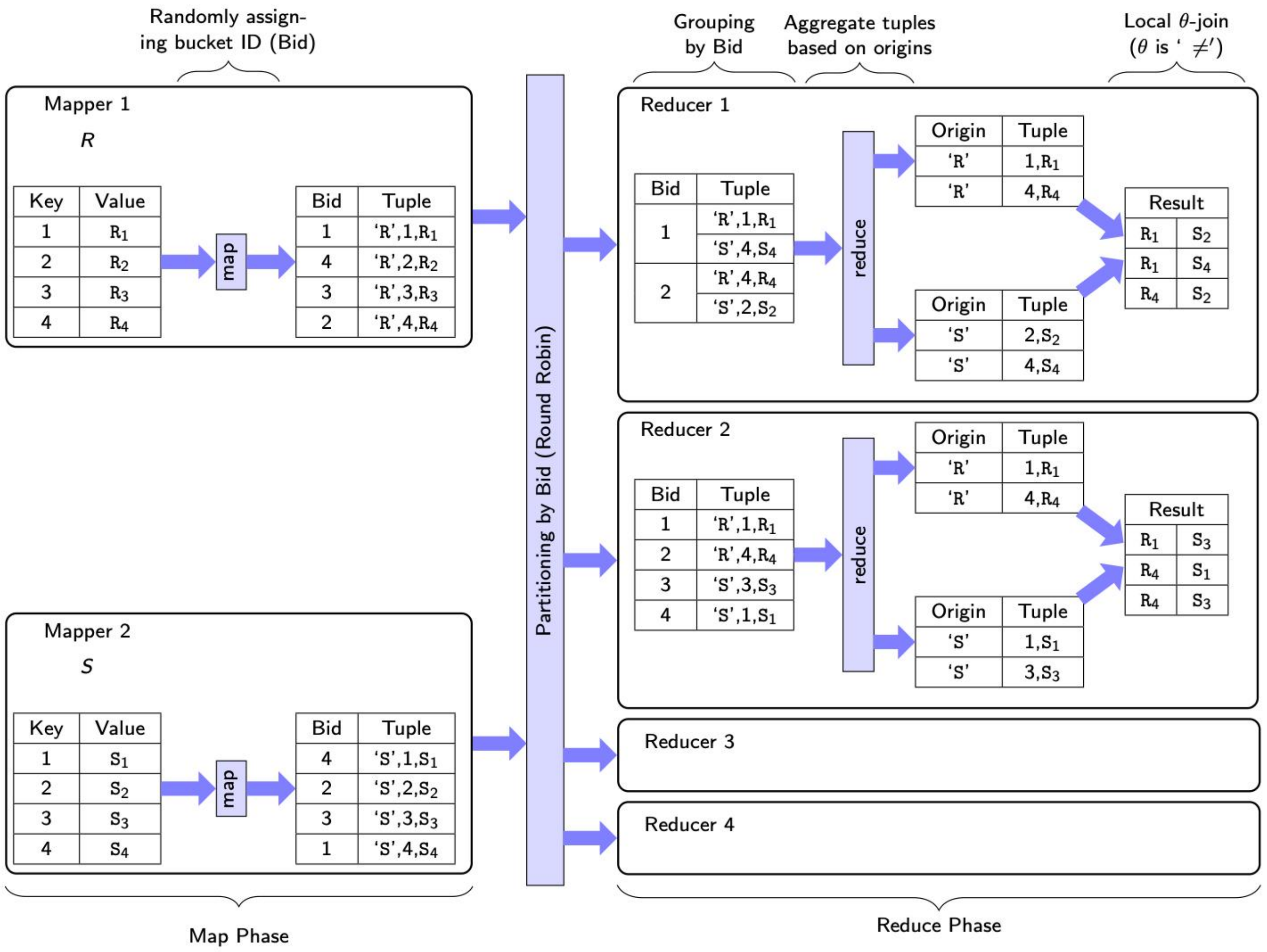
1) Partition R and assign each partition to mappers
2) Each mapper takes $\langle a,b \rangle$ tuples and converts them to a list of key-value pairs of the form $(b, \langle a,R \rangle)$
3) Each reducer pulls the pairs with the same key
4) Each reducer joins tuples of R with tuples of S

θ-Join (θ is =)

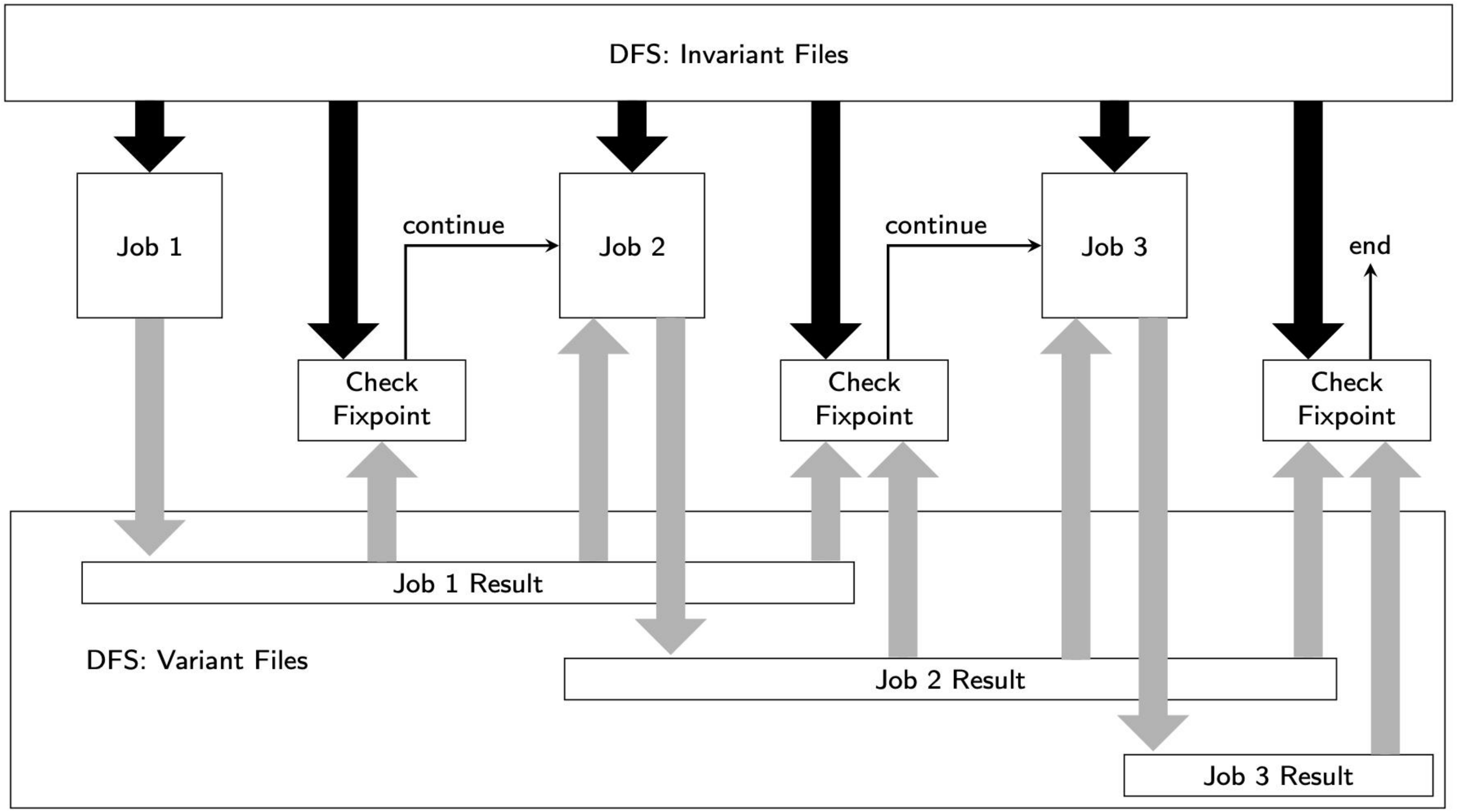
- Repartition join



θ-Join (θ is ≠)



MapReduce Iterative Computation



Problems with Iteration

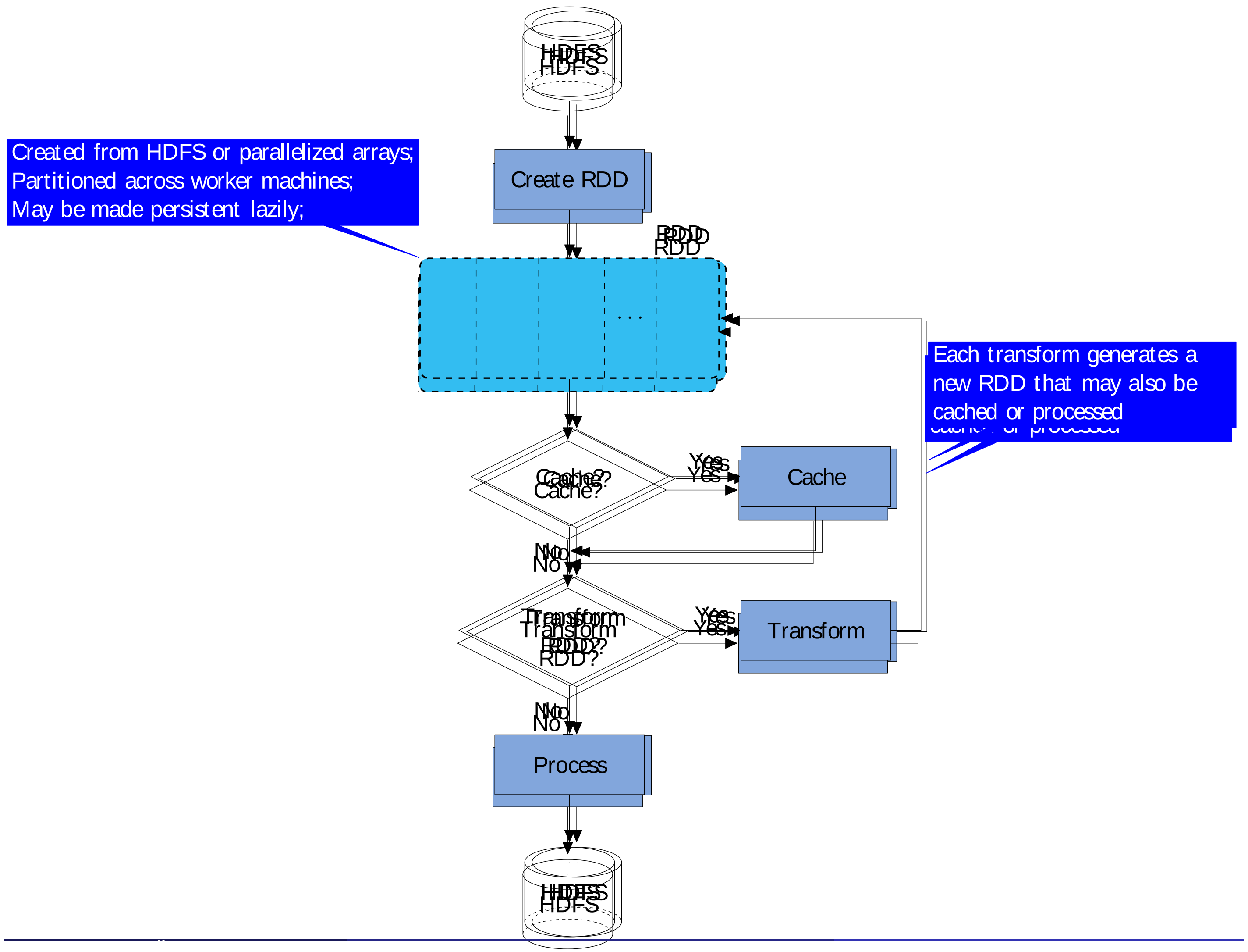
- MapReduce workflow model is acyclic
  - Iteration: Intermediate results have to be written to HDFS after each iteration and read again
- At each iteration, no guarantee that the same job is assigned to the same compute node
  - Invariant files cannot be locally cached
- Check for fixpoint
  - At the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

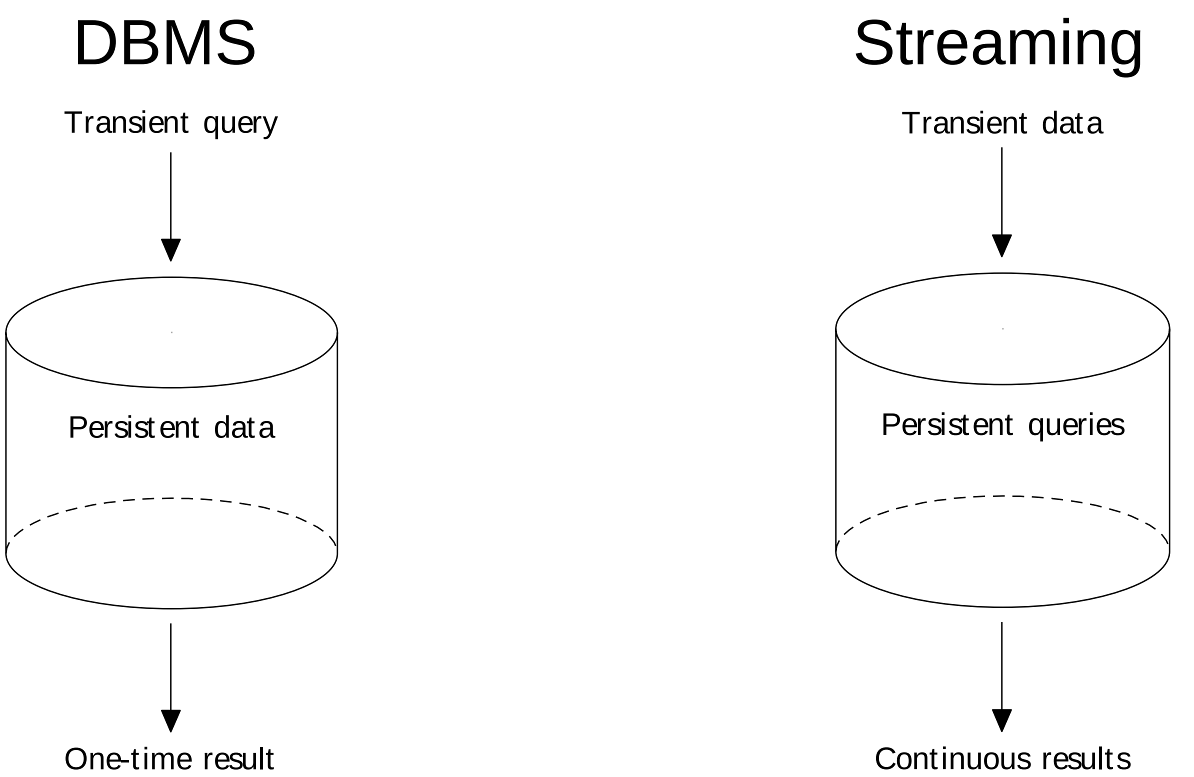
Spark Program Flow



Outline

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

Traditional DBMS vs Streaming

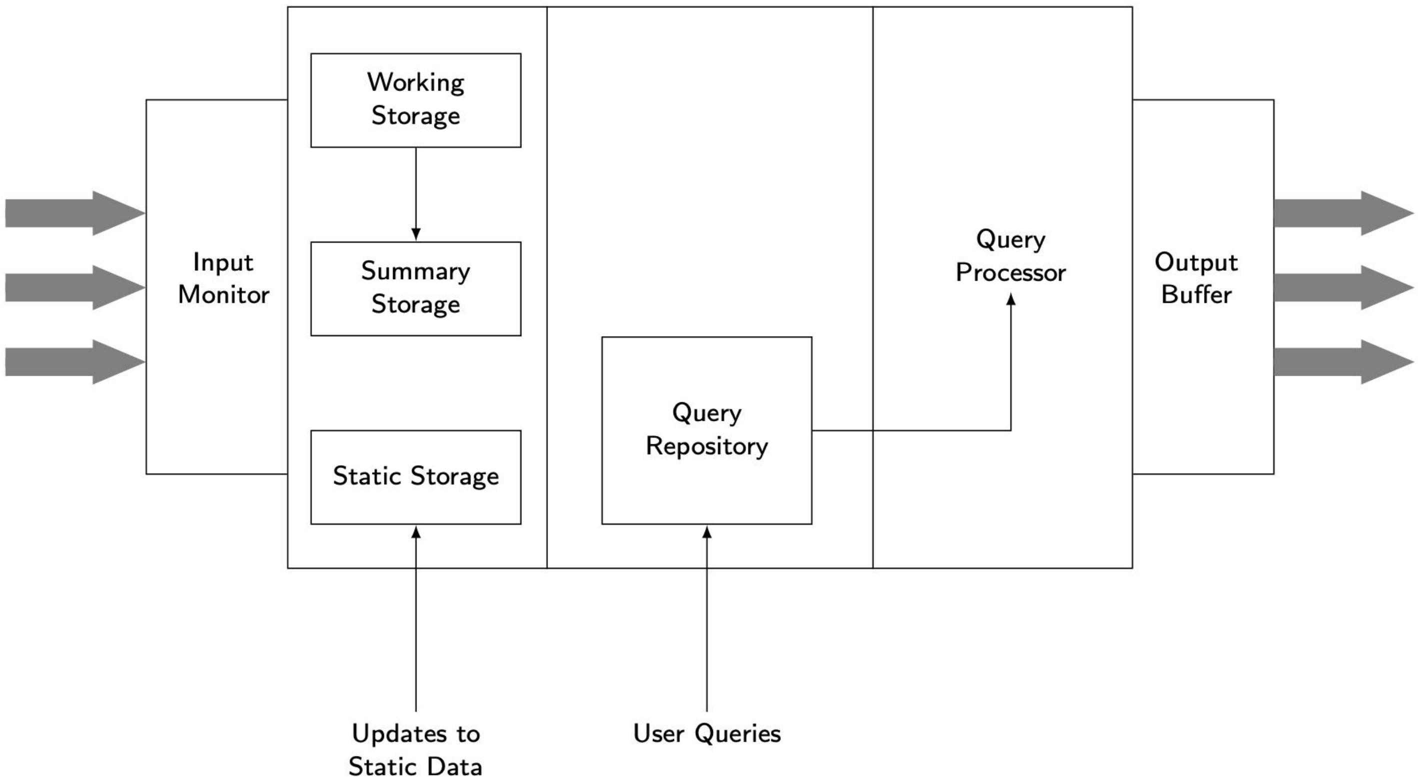


- Other differences
  - Push-based (data-driven)
  - Persistent queries
  - Unbounded stream
  - System conditions may not be stable

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

Window Definition

- According to the direction of endpoint movement
  - Fixed window: both endpoints are fixed
  - Sliding window: both endpoints can slide (backward or forward)
  - Landmark window: one endpoint fixed, the other sliding
- According to definition of window size
  - Logical window (time-based) – window length measured in time units
  - Physical window (count-based) – window length measured in number of data items
  - Partitioned window: split a window into multiple count-based windows
  - Predicate window: arbitrary predicate defines the contents of the window

Stream Query Models

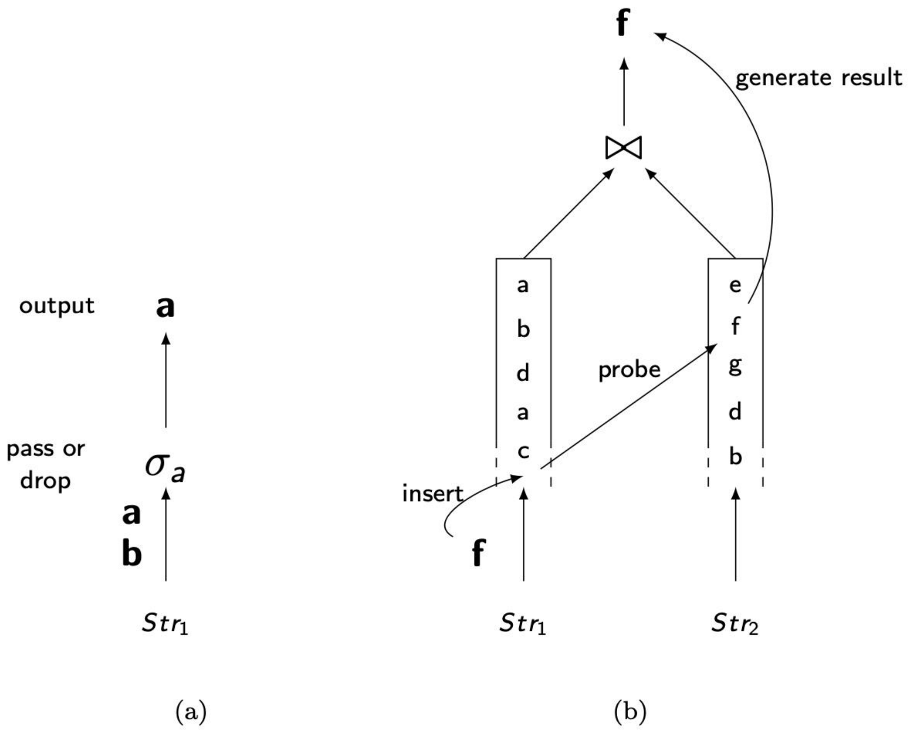
- Queries are typically persistent
- They may be monotonic or non-monotonic
- Monotonic: result set always grows
  - Results can be updated incrementally
  - Answer is continuous, append-only stream of results
  - Results may be removed from the answer only by explicit deletions (if allowed)
- Non-monotonic: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

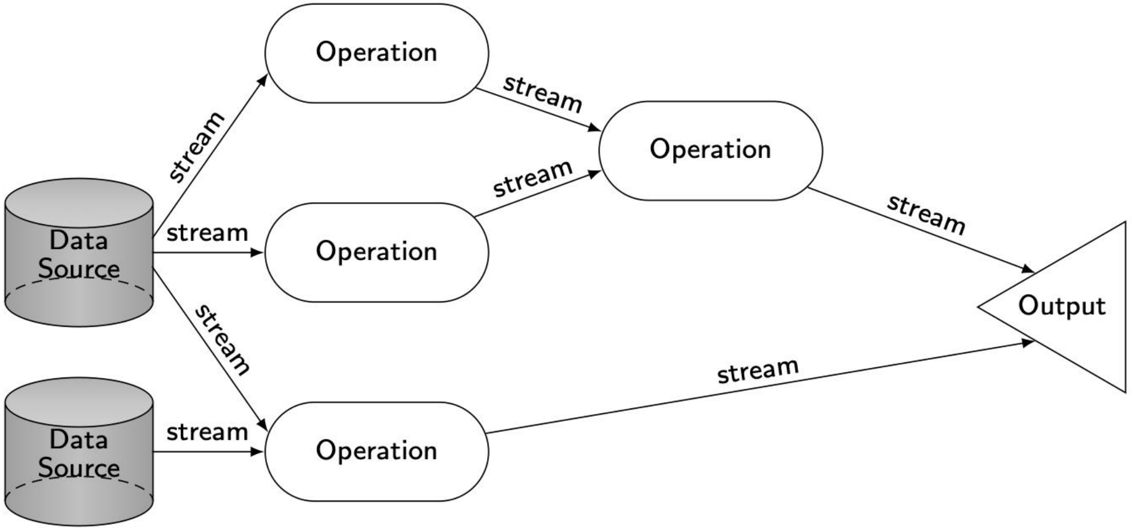
Streaming Operators

- Stateless operators are no problem: e.g., selection, projection
- Stateful operators (e.g., nested loop join) are blocking
  - You need to see the entire inner operand
- For some blocking operators, non-blocking versions exist (symmetric hash join)
- Otherwise: windowed execution



Query Processing over Streams

- Similar to relational, except
  - persistent queries: registered to the system and continuously running
  - data pushed through the query plan, not pulled
- Stream query plan



Query Processing Issues

- Continuous execution
  - Each new arrival is processed as soon as the system gets it
  - E.g. Apache Storm, Heron
- Windowed execution
  - Arrivals are batched and processed as a batch
  - E.g. Aurora, STREAM, Spark Streaming
- More opportunities for multi-query optimization
  - E.g. Easier to determine shared subplans

Windowed Query Execution

- Two events need to be managed
  - Arrivals
  - Expirations
- System actions depend on operators
  - E.g. Join generates new result, negation removes previous result
- Window movement also affects results
  - As window moves, some items in the window move out
  - What to do to results
  - If monotonic, nothing; if non-monotonic, two options
    - Direct approach
    - Negative tuple approach

Load Management

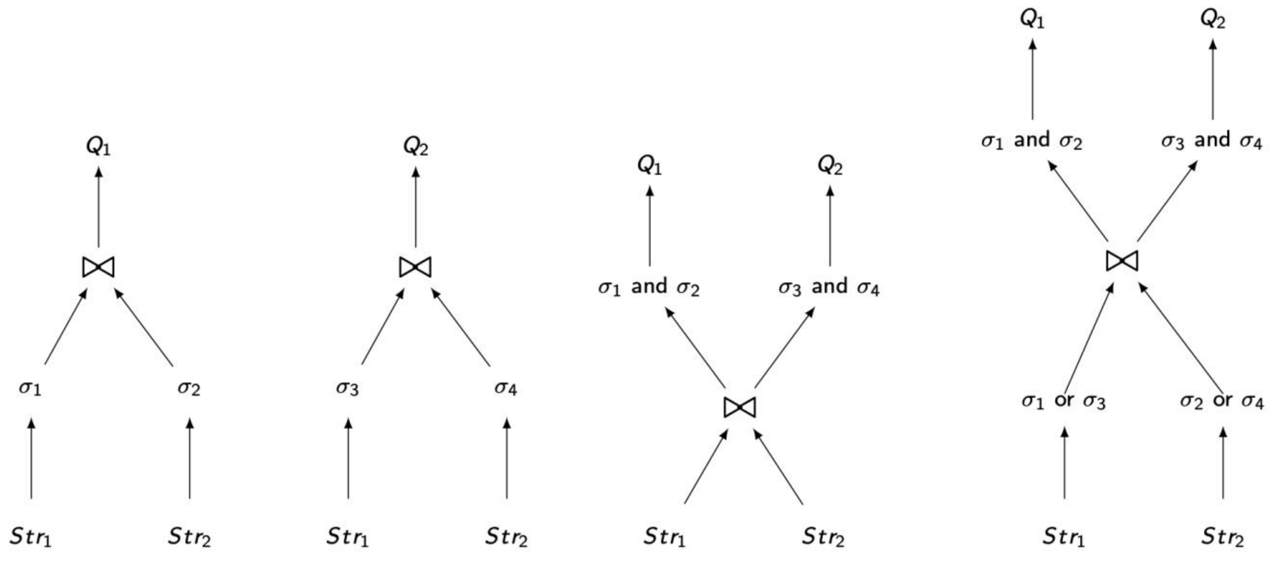
- Stream arrival rate > processing capability
- Load shedding
  - Random
  - Semantic
- Early drop
  - All of the downstream operators will benefit
  - Accuracy may be negatively affected
- Late drop
  - May not reduce the system load much
  - Allows the shared subplans to be evaluated

Out-of-Order Processing

- Assumption: arrivals are in timestamp order
- May not hold
  - Arrival order may not match generation order
  - Late arrivals → no more or just late?
  - Multiple sources
- Approaches
  - Built-in slack
  - Punctuations

Multiquery Optimization

- More opportunity since the persistent queries are known beforehand
  - Aggregate queries over different window lengths or with different slide intervals
  - State and computation may be shared (usual)

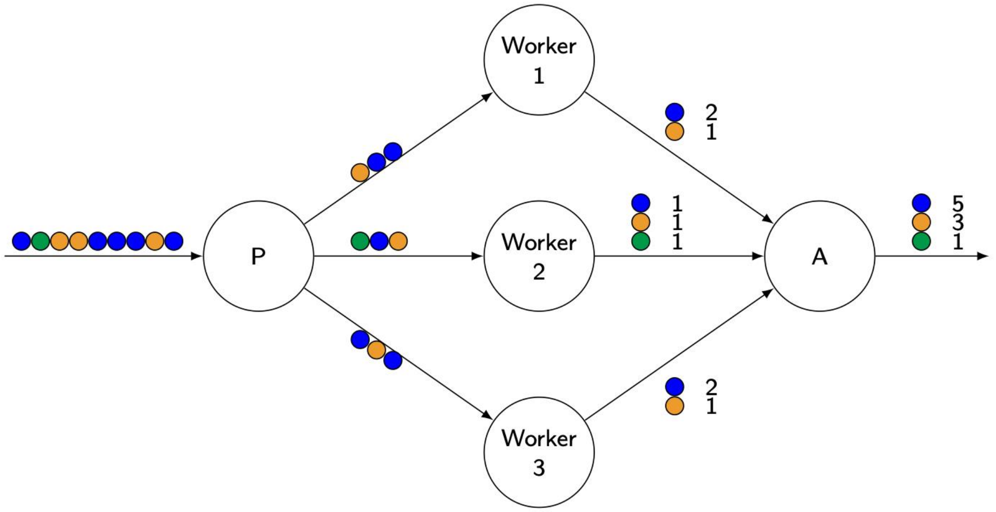


Parallel Data Stream Processing

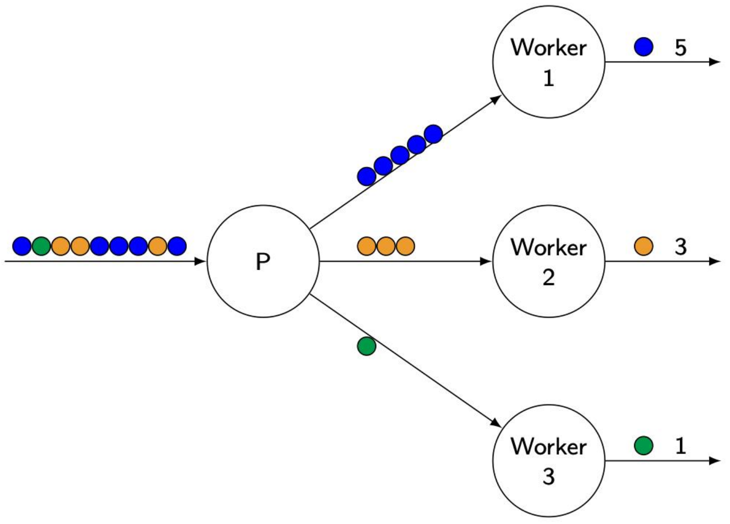
1) Partitioning the incoming stream
2) Execution of the operation on the partition
3) (Optionally) aggregation of the results from multiple machines

Stream Partitioning

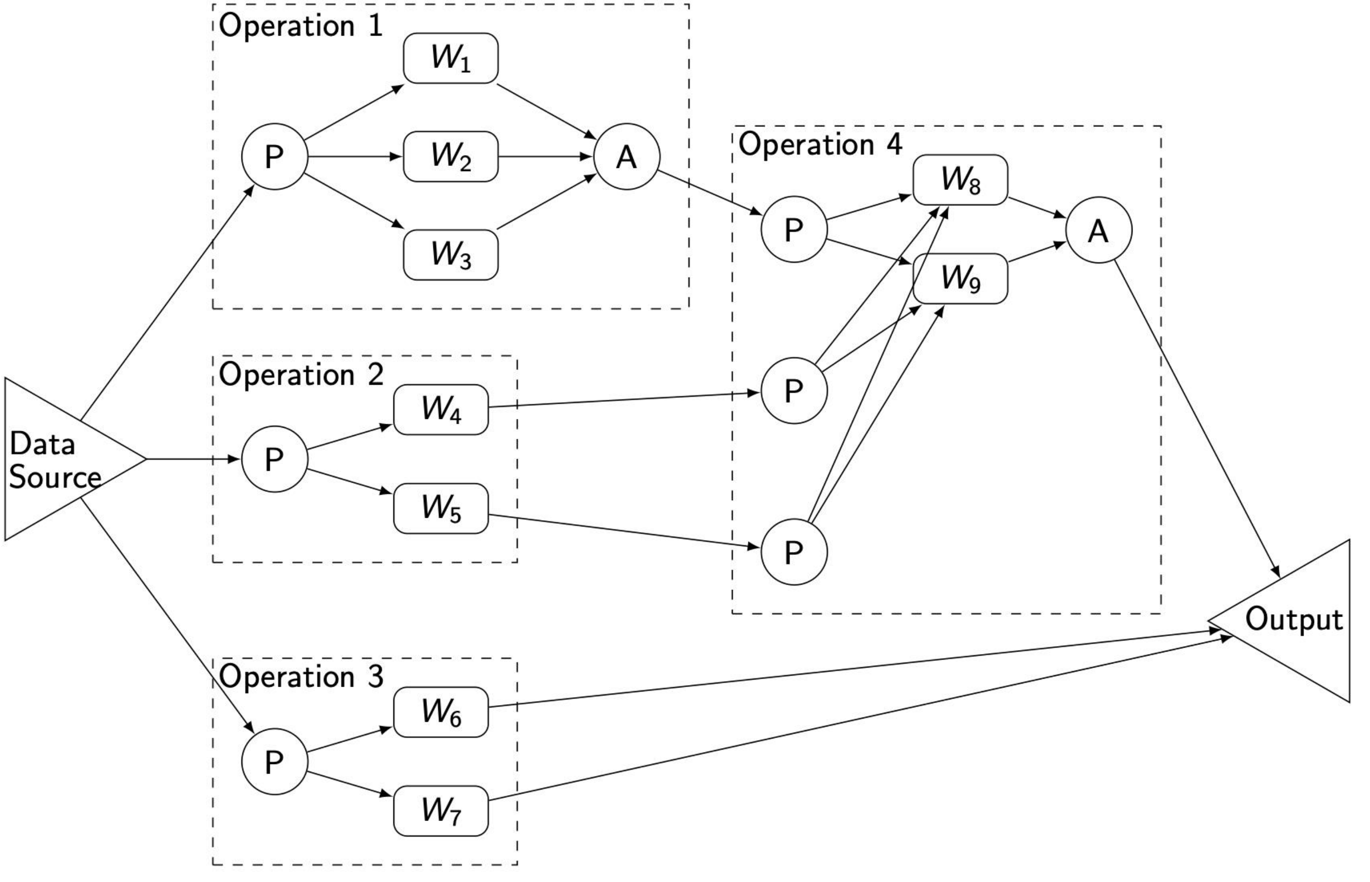
- Shuffle (round-robin) partitioning



- Hash partitioning



Parallel Stream Query Plan



Outline

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

Property Graph

- Graph $G=(V, E, D_V, D_E)$ where V is set of vertices, E is set of edges, D_V is set of vertex properties, D_E is set of edge properties
- Vertices represent entities, edges relationships among them.
- Multigraph: multiple edges between a pair of vertices
- Weighted graph: edges have weights
- Directed vs undirected

Graph Workloads

Analytical

- Multiple iterations
- Process each vertex at each iteration
- Examples
  - PageRank
  - Clustering
  - Connected components
  - Machine learning tasks

Online

- No iteration
- Usually access portion of the graph
- Examples
  - Reachability
  - Single-source shortest path
  - Subgraph matching

PageRank as Analytical Example

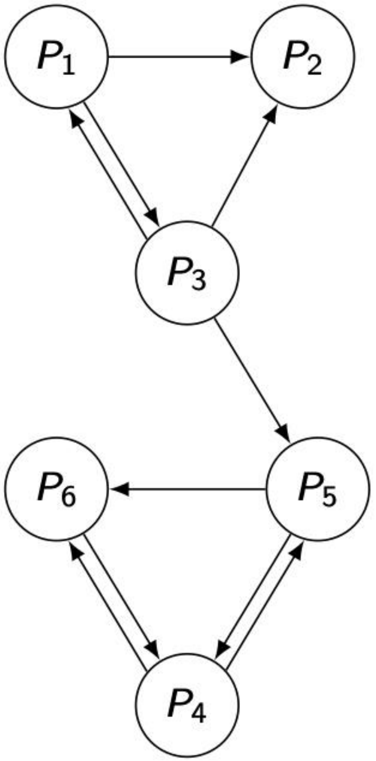
A web page is important if it is pointed at by other important web pages.

$$PR(P_i) = (1 - d) + d \sum_{P_j \in B_{P_i}} \frac{PR(P_j)}{|F_{P_j}|}$$

B_{P_i}: in−neighbors of P_i

F_{P_i}: out−neighbors of P_i

(let $d = 0.85$)



$$PR(P_2) = 0.15 + 0.85\left(\frac{PR(P_1)}{2} + \frac{PR(P_3)}{3}\right)$$

Recursive!...

Graph Partitioning

- Graph partitioning is more difficult than relational partitioning because of edges
- Two approaches
  - Edge-cut or vertex-disjoint
    - Each vertex assigned to one partition, edges may be replicated
  - Vertex-cut or edge-disjoint
    - Each edge is assigned to one partition, vertices may be replicated
- Objectives
  - Allocate each vertex/edge to partitions such that partitions are mutually exclusive
  - Partitions are balanced
  - Minimize edge-/vertex-cuts to minimize communication

Graph Partitioning Formalization

$$\text{minimize } C(P)$$

Total communication cost due to partitioning

$$\text{subject to:}$$

$$w(P_i) \leq \beta * \frac{\sum_{j=1}^{k} w(P_j)}{k}, \forall i \in \{1, \ldots, k\}$$

Abstract cost of processing partition P_i

Slackness for unbalanced partitioning

- $C(P)$ and $w(P_i)$ differ for different partitionings

Vertex-Disjoint (Edge-Cut)

- Objective is to minimize the number of edge cuts
- Objective function

$$C(P) = \frac{\sum_{i=1}^{k} |e(P_i, V \setminus P_i)|}{|E|} \quad \text{where } \left|e\left(P_i, P_j\right)\right| = \#\text{edges between } P_i \text{ and } P_j$$

- $w(P_i)$ defined in terms of the number of vertices-per-partition

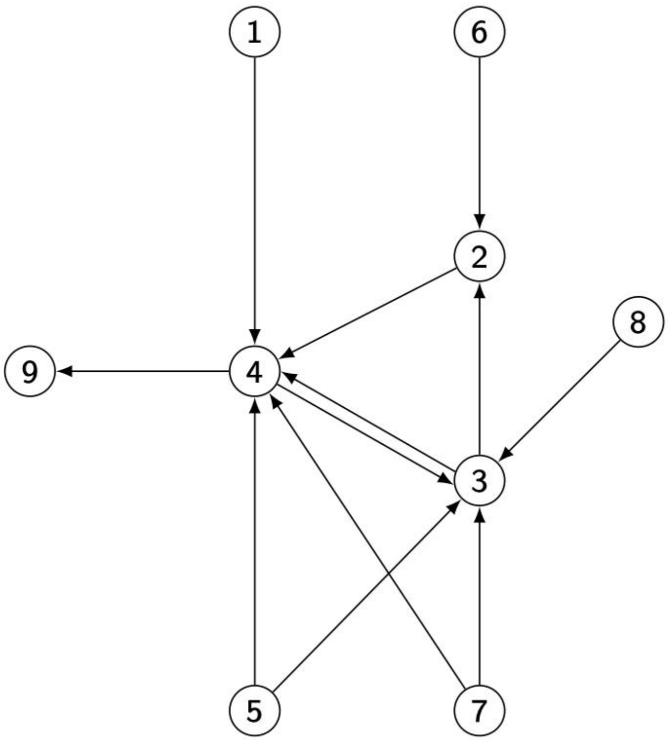
METIS Vertex-Disjoint Partitioning

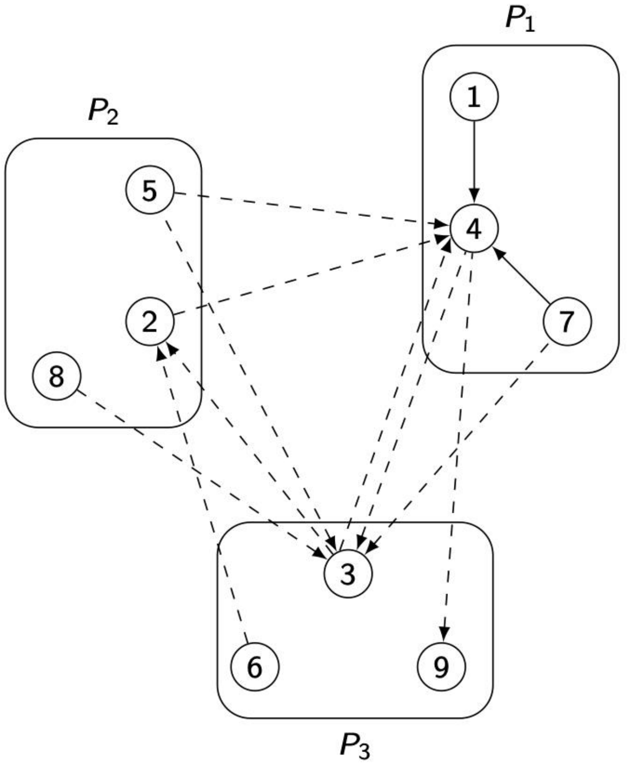
- METIS is a family of algorithms
- Usually the gold standard for comparison

Given an initial graph $G_0 = (V, E)$:

1) Produce a hierarchy of successively coarsened graphs $G_1, \ldots, G_n$ such that $|V(G_i)| > |V(G_j)|$ for $i < j$
2) Partition G_n using some partitioning algorithm
   - Small enough that it won't matter what algorithm is used
3) Iteratively coarsen G_n to G_0, and at each step
   a) Project the partitioning solution on graph G_j to graph G_{j-1}
   b) Improve the partitioning of G_0

Vertex-Disjoint Partitioning Example





Edge-Disjoint Partitioning

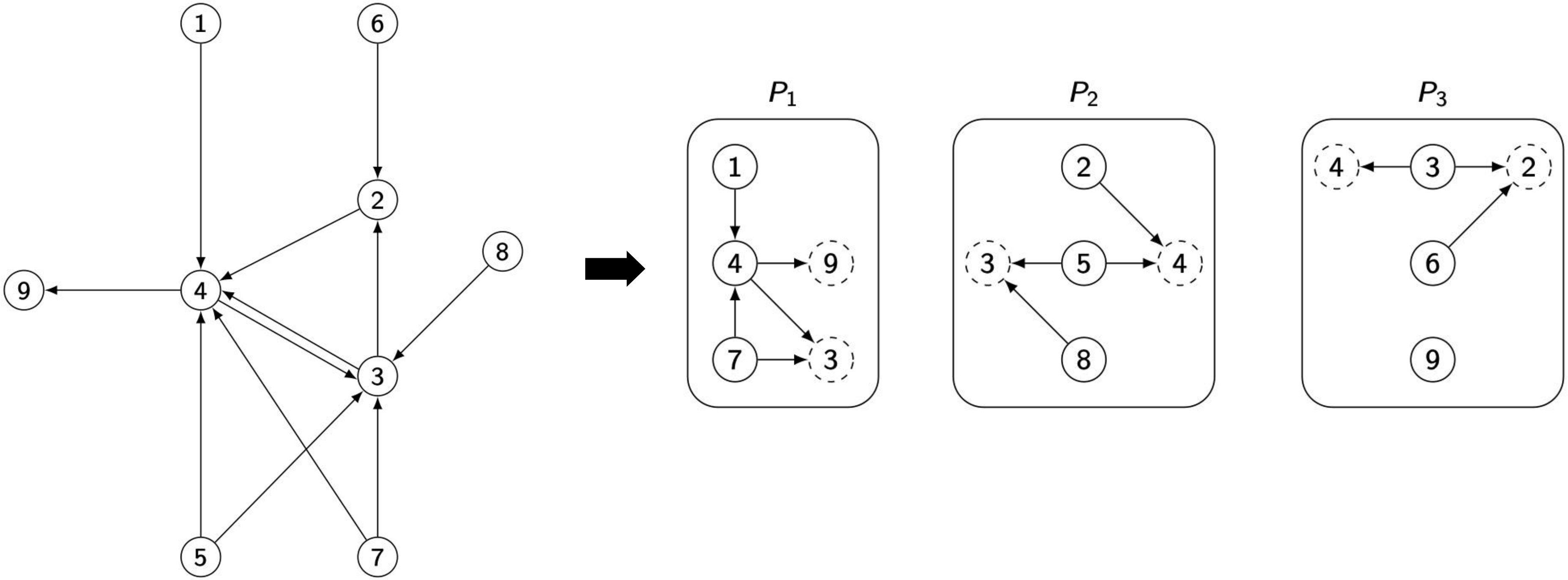
- Vertex-disjoint perform
  - well for graphs with low-degree vertices
  - poorly on power-law graphs causing many edge-cuts
- Edge-disjoint (vertex-cut) better for these
  - Put each edge in one partition
  - Vertices may need to be replicated – minimize these
- Objective function

  $C(P) = \frac{\sum_{v \in V} |A(v)|}{|V|}$ where $A(v) \subseteq \{P_1, \ldots, P_k\}$ is set of partitions in which v exists
- $w(P_i)$ is the number of edges in partition P_i

Edge-Disjoint Alternatives

- Hashing (on the ids of the two vertices incident on edge)
  - Fast and highly parallelizable
  - Gives good balance
  - But may lead to high vertex replication
- Heuristics cognizant of graph characteristics
  - Greedy: decide on allocation edge *i*+1 based on the allocation of the previous *i* edges to minimize vertex replication

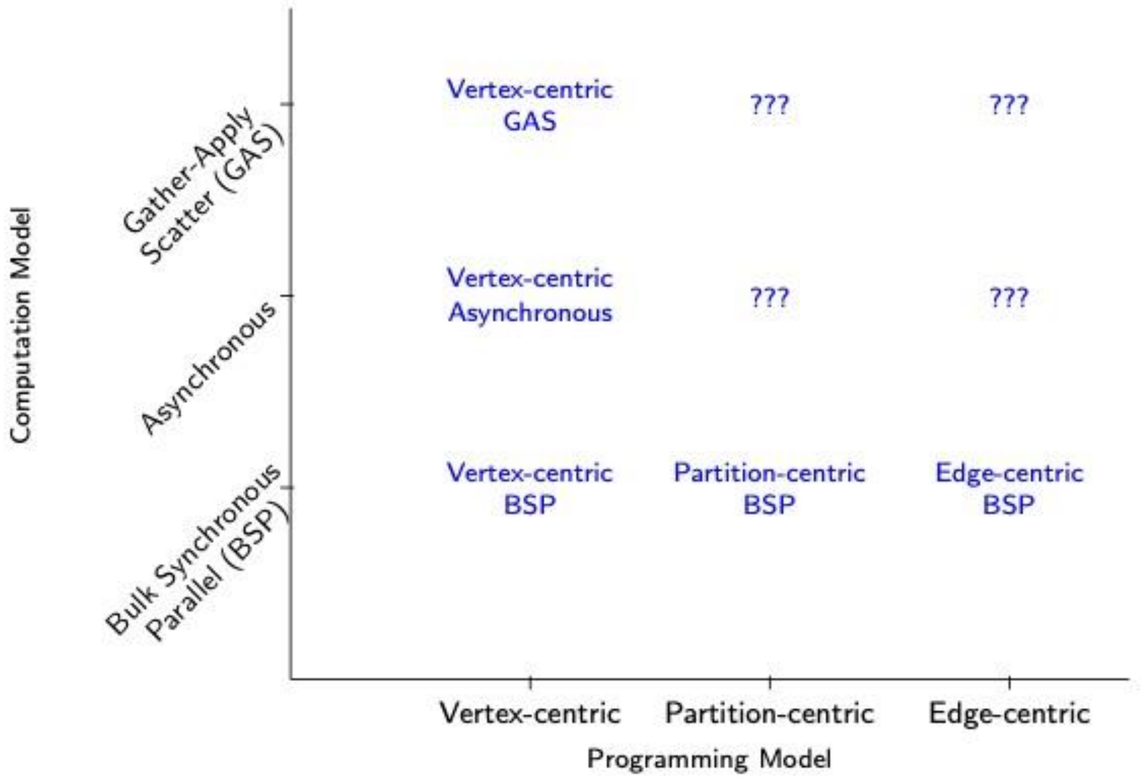
Edge-Disjoint Partitioning Example



Can MapReduce be used for Graph Analytics?

- `map` and `reduce` functions can be written for gaph analytics
  - There are works that have done this
- Graph analytics tasks are iterative
  - Recall: MapReduce is not good for iterative tasks
- Spark improves MapReduce (e.g., Hadoop) for iterative tasks
  - GraphX on top of Spark
  - Edge-disjoint partitioning
  - Vertex table & edge table as RDDs on each worker
  - Join vertex & edge tables
  - Perform an aggregation

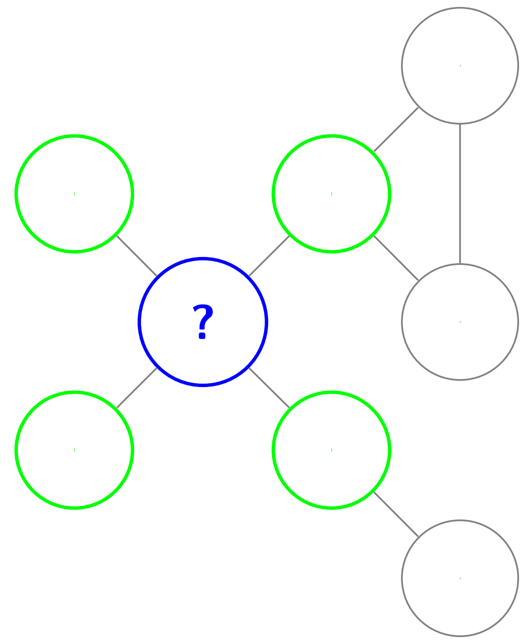
Special-Purpose Graph Analytics Systems



???: Systems do not exist

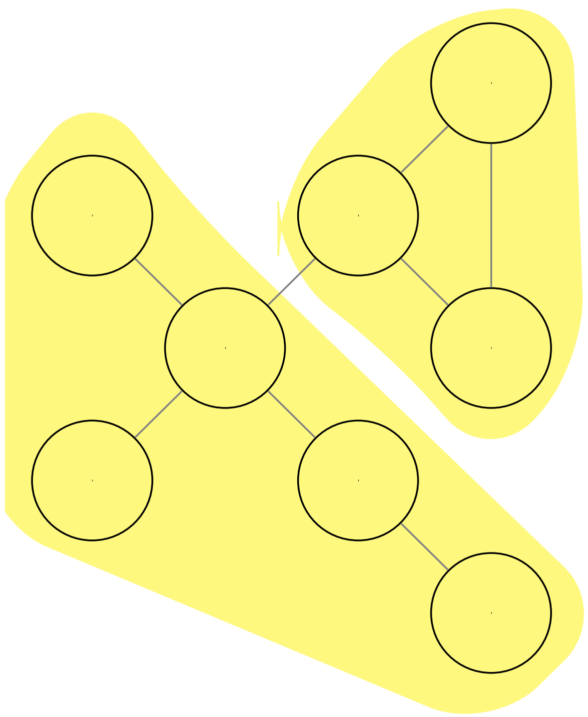
Vertex-Centric Model

- Computation on a vertex is the focus
- “Think like a vertex”
- Vertex computation depends on its own state + states of its neighbors
- `Compute(vertex v)`
- `GetValue()`, `WriteValue()`



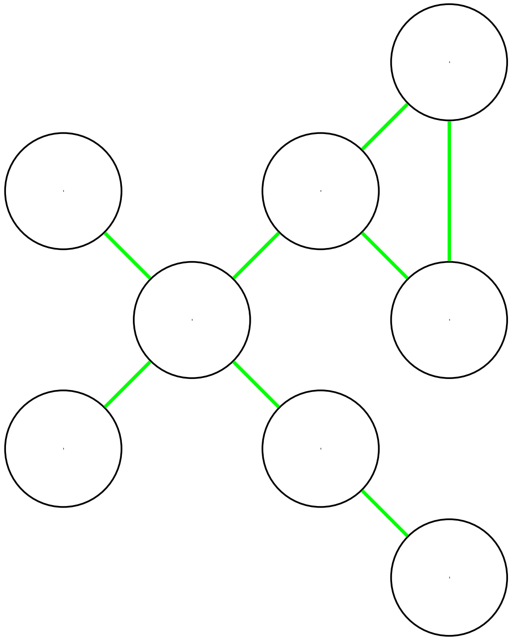
Partition-centric (Block-centric) Model

- Computation on an entire partition is specified
- "Think like a block" or "Think like a graph"
- Aim is to reduce the communication cost among vertices

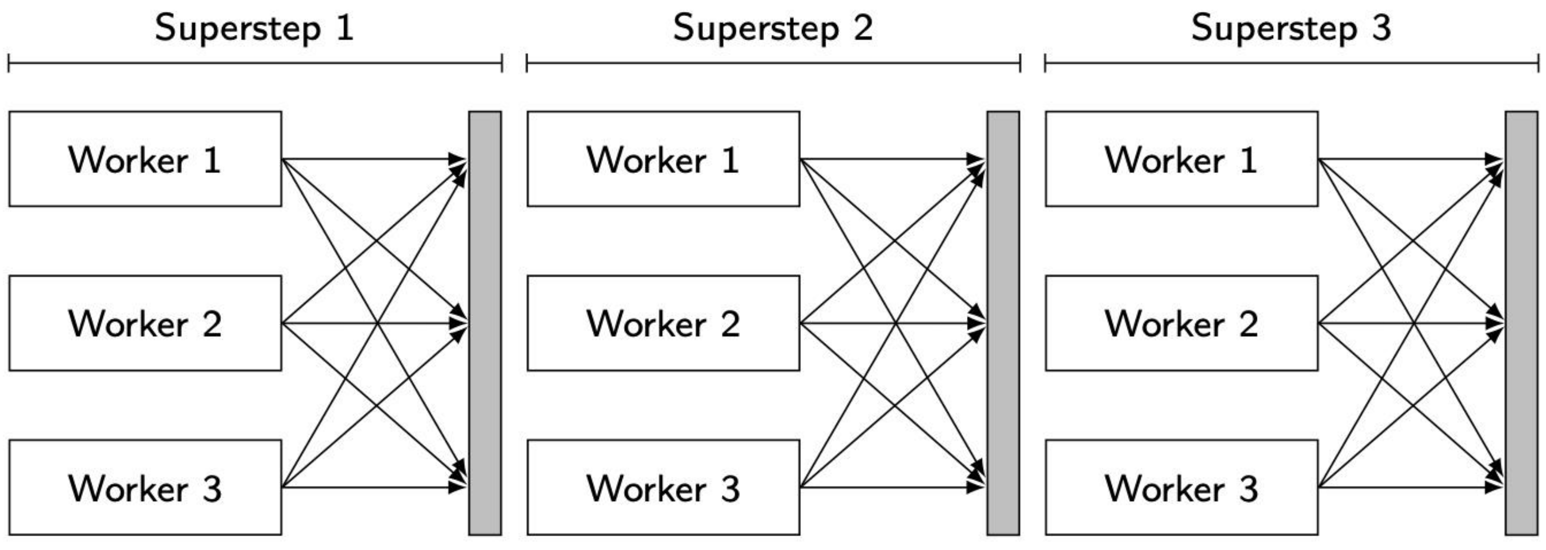


Edge-centric Model

- Computation is specified on each edge rather than on each vertex or bloc
- `Compute(edge e)`



Bulk Synchronous Parallel (BSP) Model

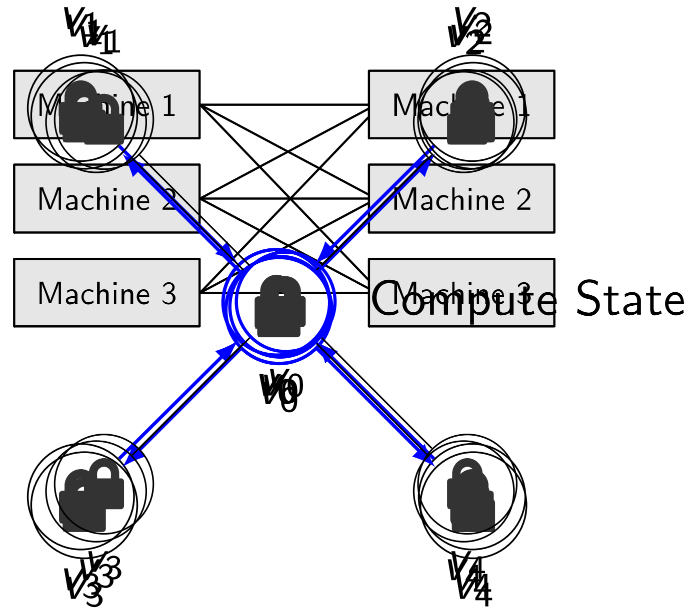


Each machine performs computation on its graph partition

At the end of each superstep results are pushed to other workers

Asynchronous Parallel (AP) Model

- Supersteps, but no communication barriers
- Uses the most recent values
- Computation in step *k* may be based on neighbor states of step *k*-1 (of received late) or of state *k*
- Consistency issues → requires distributed locking



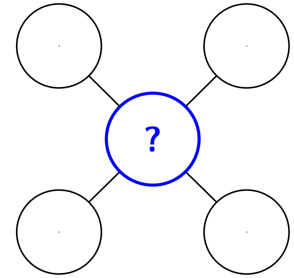
Consider vertex-centric

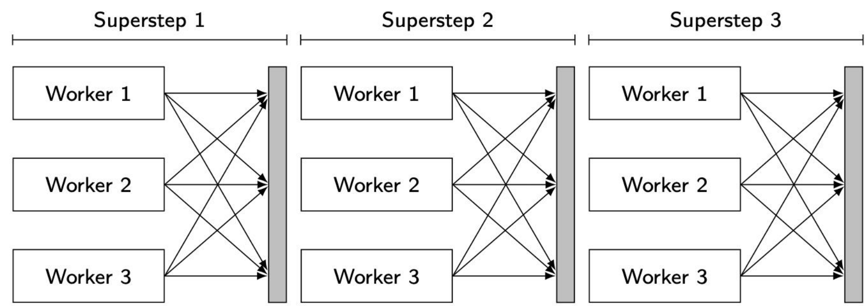
Gather-Apply-Scatter (GAS) Model

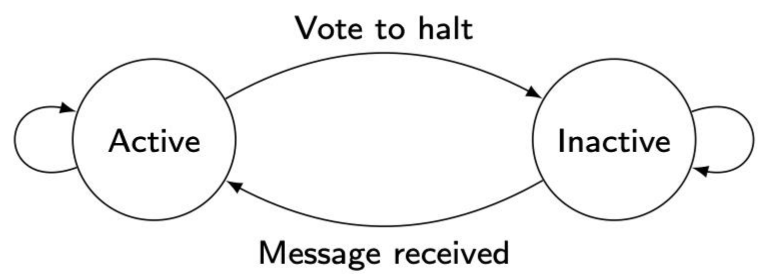
- Similar to BSP, but pull-based
- Gather: pull state
- Apply: Compute function
- Scatter: Update state
- Updates of states separated from scheduling

Vertex-Centric BSP Systems

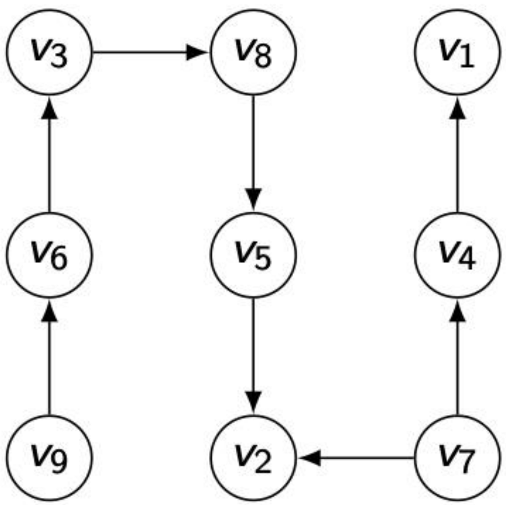
- "Think like a vertex"
- `Compute(vertex v)`
- BSP Computation – push state to neighbor vertices at the end of each superstep
- Continue until all vertices are inactive
- Vertex state machine

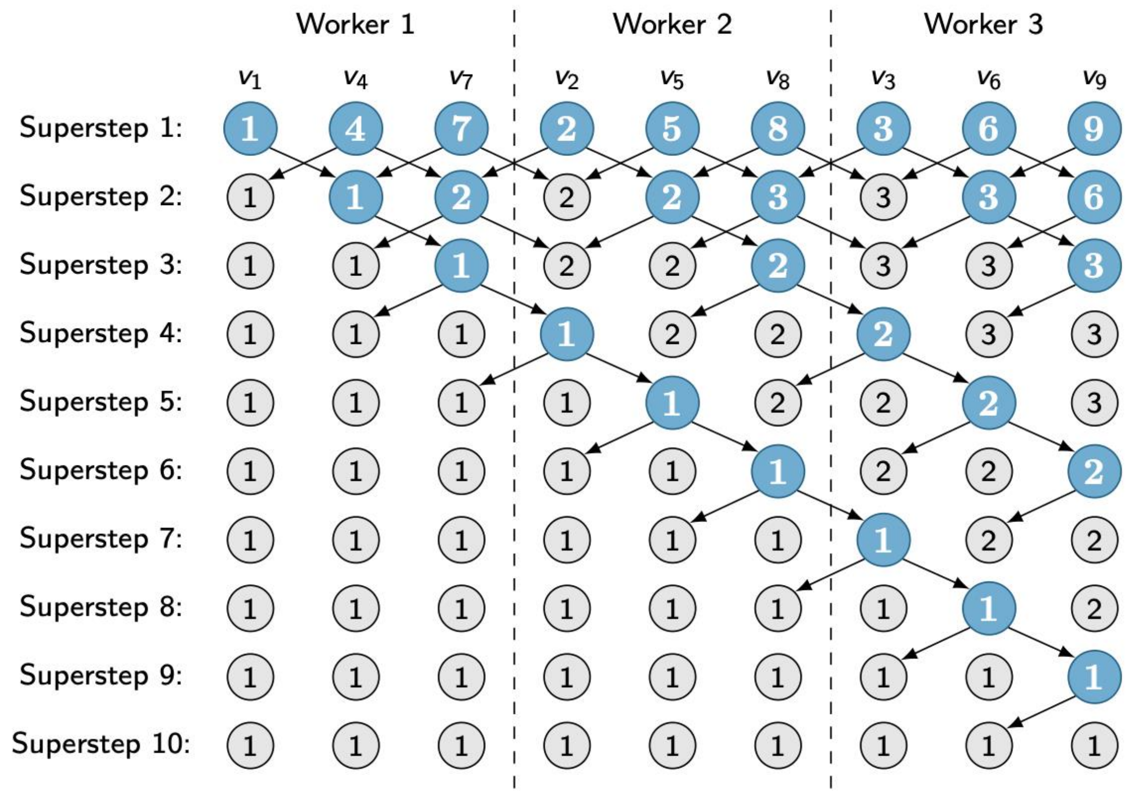






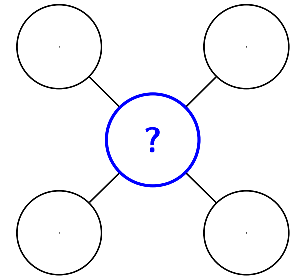
Connected Components: Vertex-Centric BSP

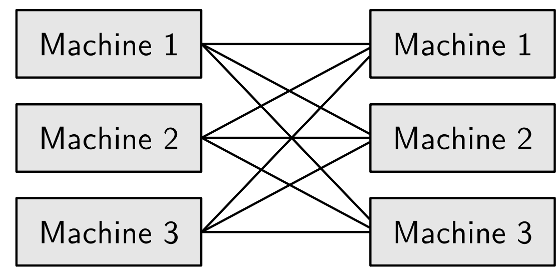


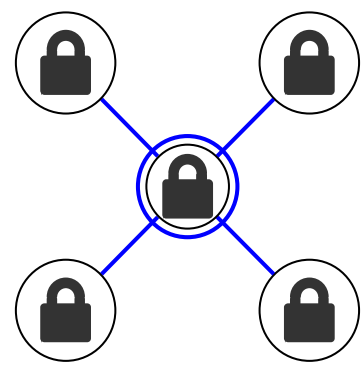


Vertex-Centric AP Systems

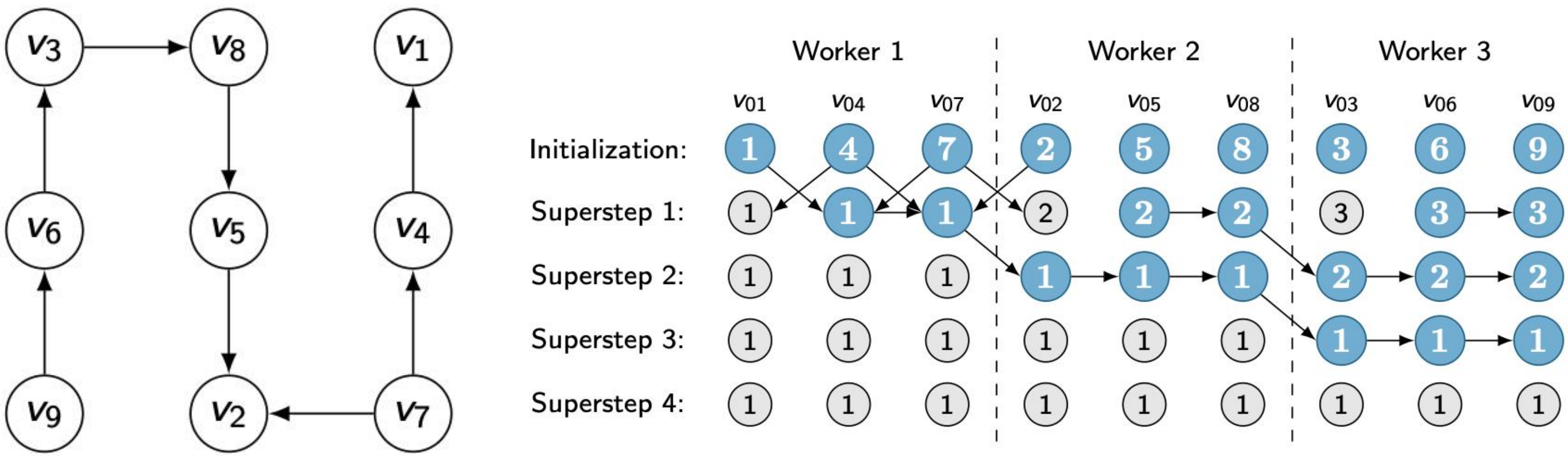
- "Think like a vertex"
- `Compute(vertex v)`
- Supersteps exist along with synchronization barriers, but …
- `Compute(vertex v)` can see msgs sent in the same superstep or previous one
- Consistency of vertex states: distributed locking
- Consistency issues: no guarantee about input to `Compute()`





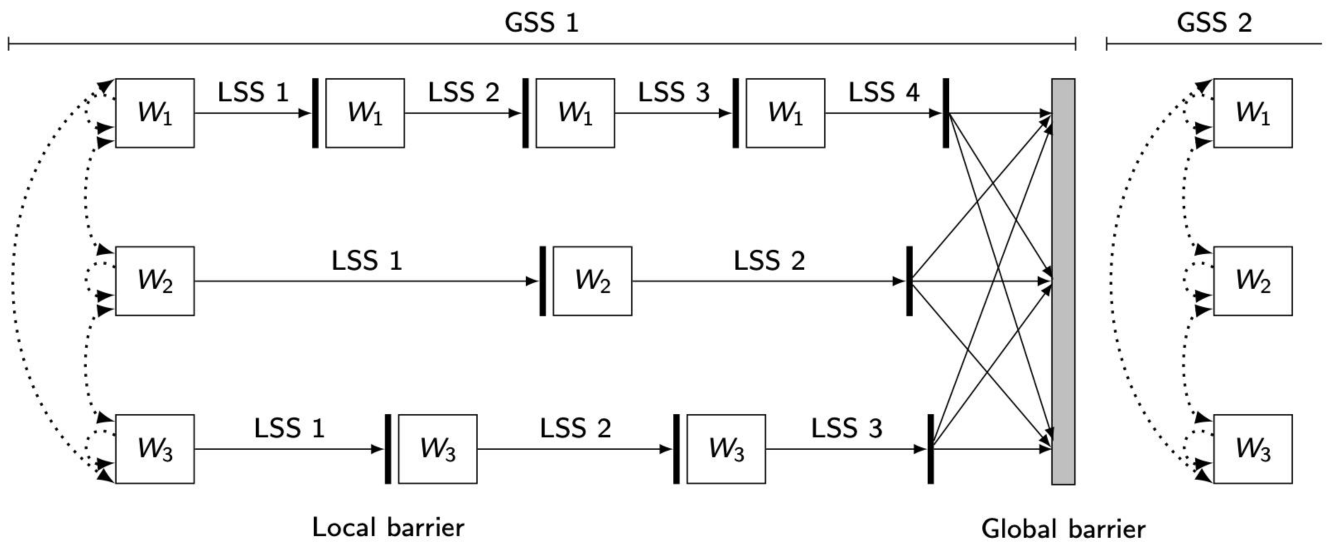


Connected Components: Vertex-Centric AP



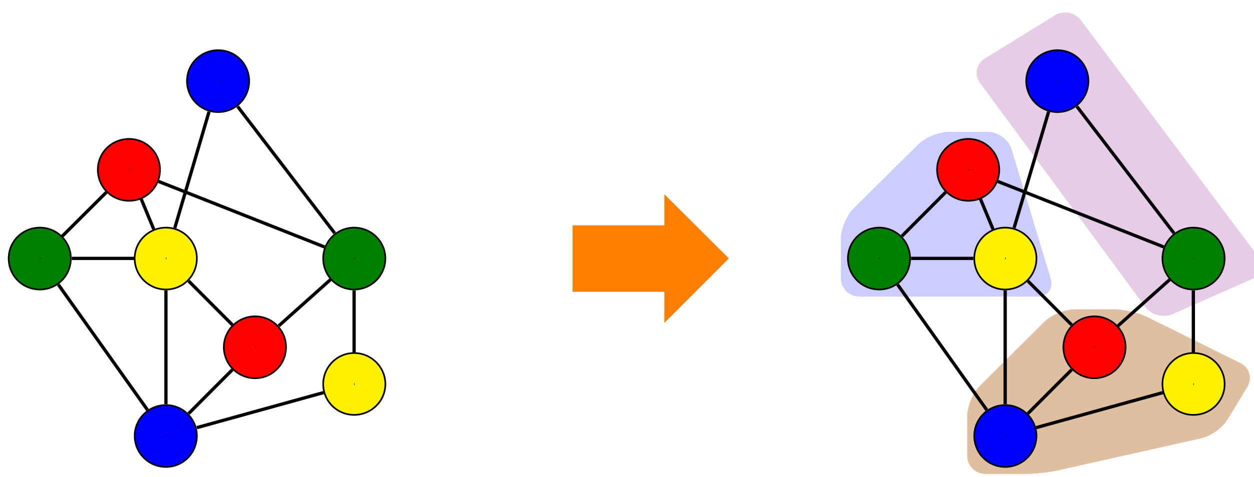
Barrierless Asynchronous Parallel (BAP)

- Divides each global superstep into logical supersteps separated by very lightweight local barriers
- `Compute()` can be executed multiple times in each superstep (once-per-logical superstep)
- Synchronizes at global barriers as in AP

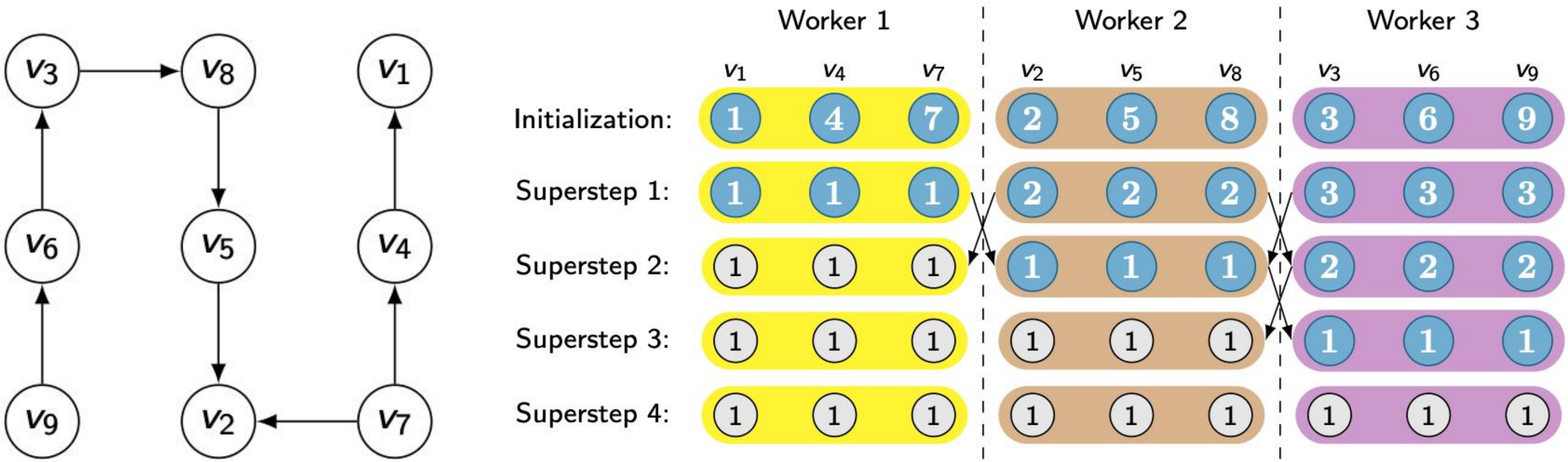


Partition- (Block-)Centric BSP Systems

- "Think like a block"; also "think like a graph"
- Reduces communication
- Exploit the partitioning of the graph
  - Message exchanges only among blocks; BSP in this case
  - Within a block, run a serial in-memory algorithm

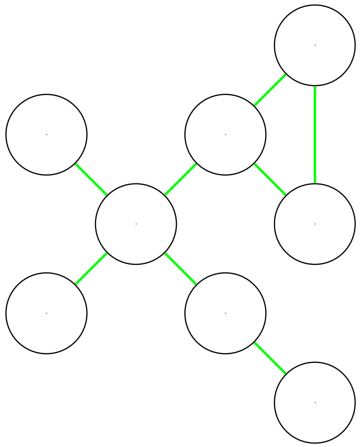


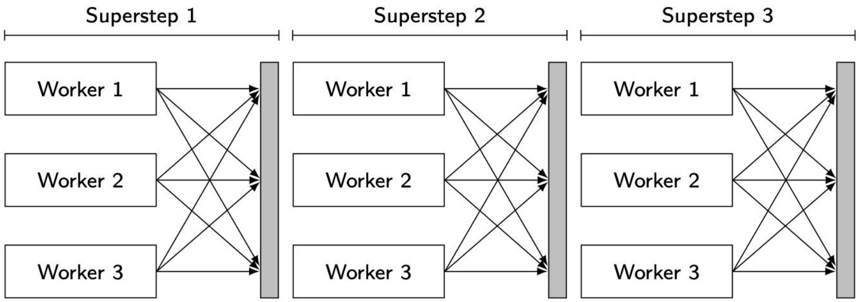
Connected Components: Block-Centric BSP



Edge-Centric BSP Systems

- Dual of vertex-centric BSP
- "Think like an edge"
- `Compute(edge e)`
- BSP Computation – push state to neighbor vertices at the end of each superstep
- Continue until all vertices are inactive
- Number of edges ≫ number of vertices
  - Fewer msgs, more computation
  - No random reads





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

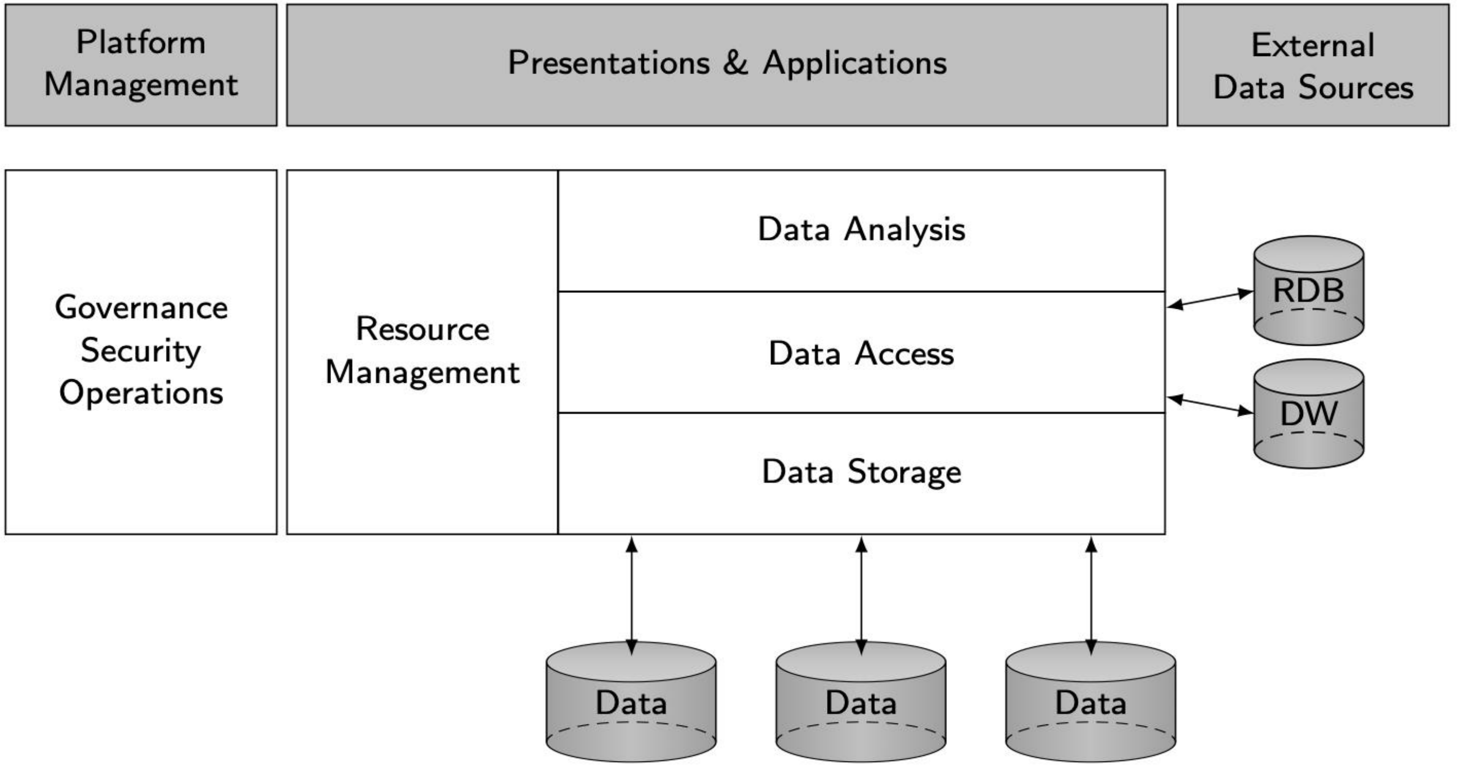
- Collect all useful data
  - Raw data, transformed data
- Dive from anywhere
  - Users from different business units can explore and enrich the data
- Flexible access
  - Different access paths to shared infrastructure
    - Batch, interactive (OLAP and BI), real-time, search,.....

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