

# Massively Parallel Data Analysis with MapReduce

ETH Zurich

Computer Science Department

Fall 2008

# Today

- How would you join two data sources within the MapReduce framework?
  - Map-Reduce-Merge  
*[Yang et al (Yahoo! & UCLA), SIGMOD Conference, June 2007]*
  - Hadoop Join
  - Improvements on Hadoop Join  
*[Rao et al (IBM Almaden Research Center), Bay Area Hadoop User Group Meeting, October 2008]*

# Map-Reduce-Merge

# Basic Database Operations in MapReduce

- Projection
- Selection
- Aggregation
- Binary operations
  - Join, Cartesian product, Set operations
- Only the unary operations can be directly modeled with the original MapReduce framework.
- There is no direct support for operations over multiple, possibly heterogeneous input data sources.
  - Can be done indirectly by chaining extra MapReduce steps.

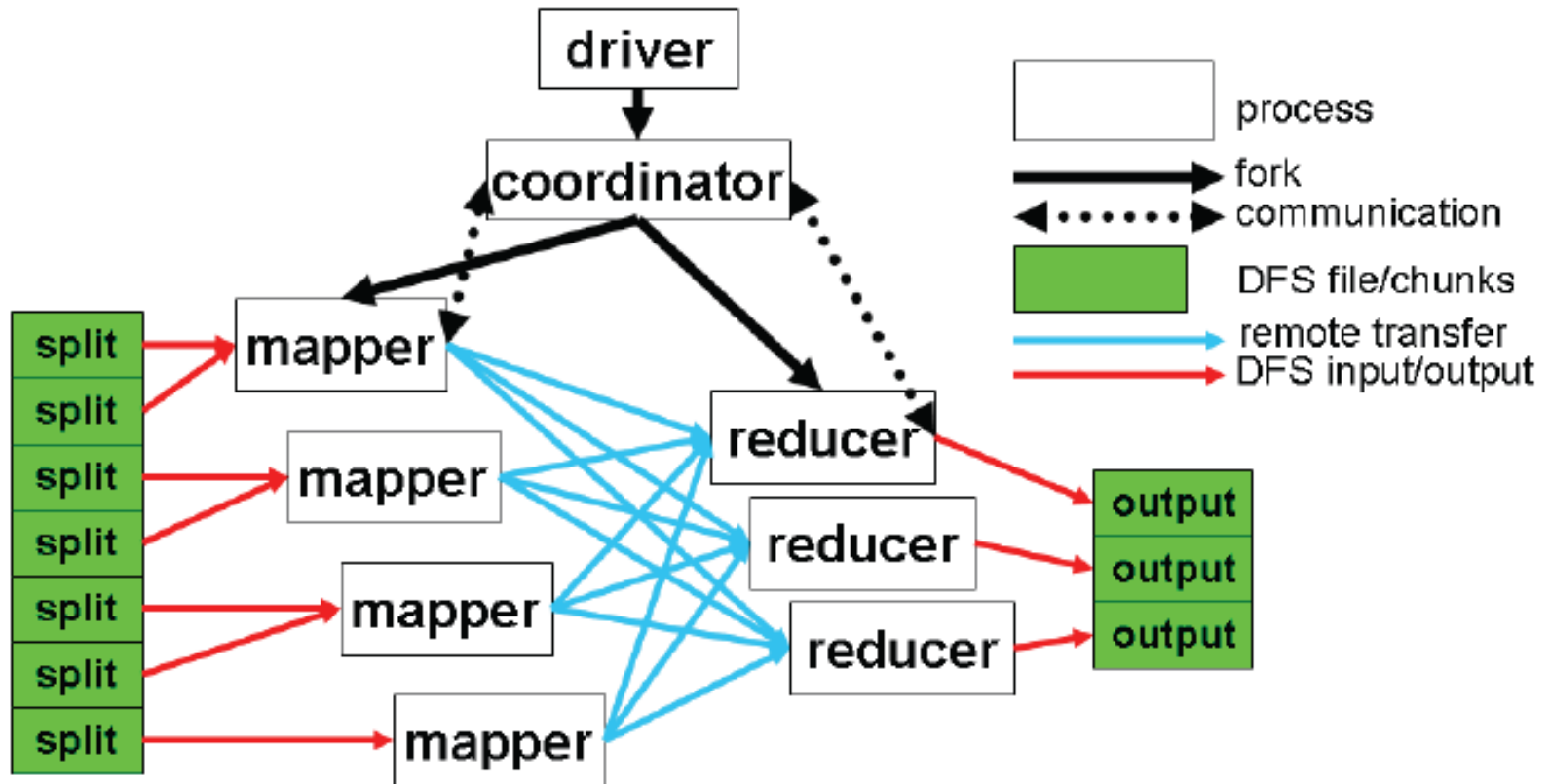
# Search Engine Example

- Search Engines keep data in multiple “databases”.
  - Crawler database (crawled URLs + contents)
  - Index database (inverted indices)
  - Log databases (click or execution logs)
  - Webgraph database (URL linkages + properties)
- Some tasks require access to multiple data sources.
  - Example: Index database is created based on the data in both crawler and webgraph databases.

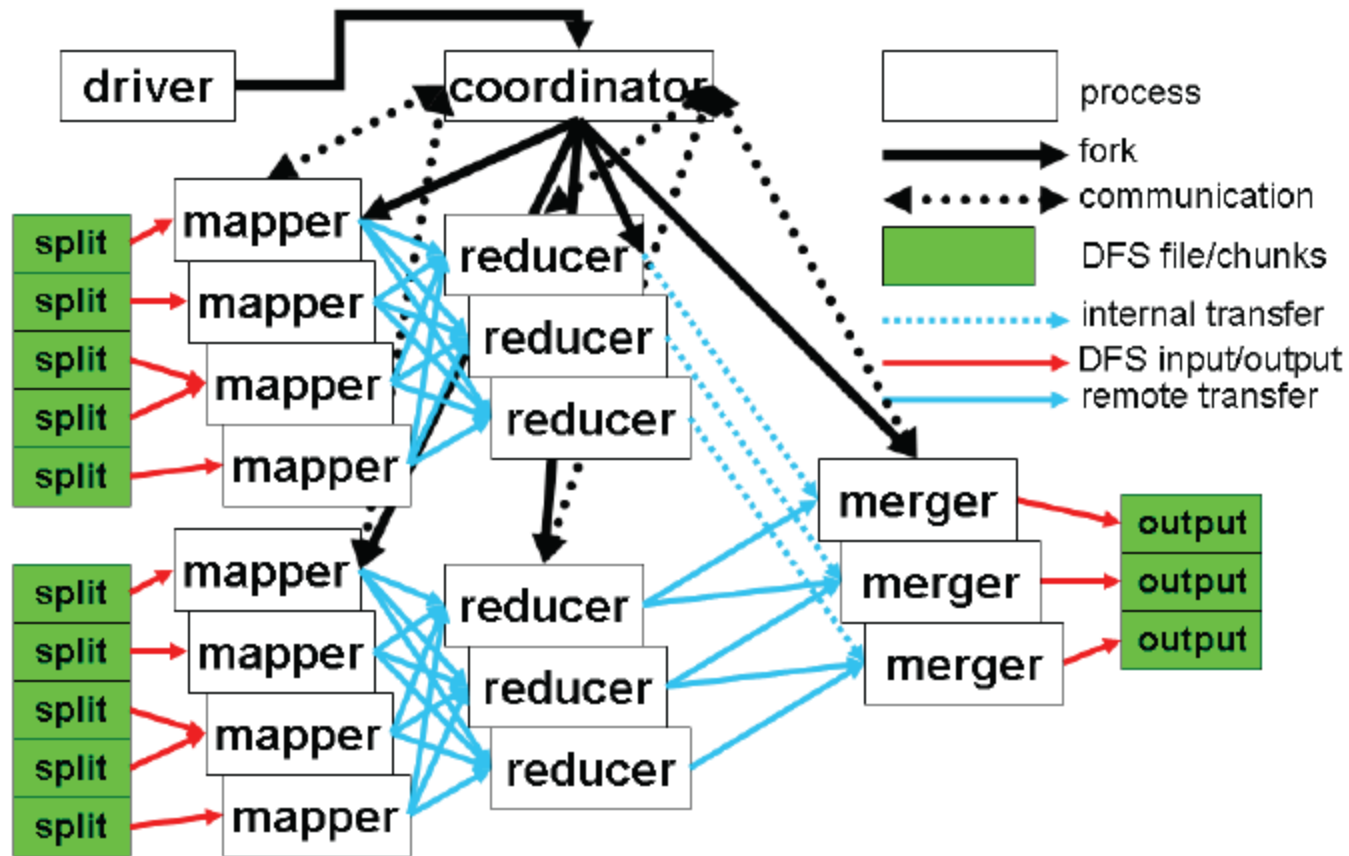
# Map-Reduce-Merge Vision

- Map-Reduce-Merge can form a hierarchical workflow which is similar to, but much more general than a DBMS query execution plan.
  - No query operators, but arbitrary programming logic specified by the developers
  - More general than relational query plans
  - More general than Map-Reduce

# Original MapReduce



# Map-Reduce-Merge



# Map-Reduce vs. Map-Reduce-Merge

map:  $(k_1, v_1) \rightarrow [(k_2, v_2)]$   
reduce:  $(k_2, [v_2]) \rightarrow [v_3]$

map:  $(k_1, v_1)_\alpha \rightarrow [(k_2, v_2)]_\alpha$   
reduce:  $(k_2, [v_2])_\alpha \rightarrow (k_2, [v_3])_\alpha$   
merge:  $((k_2, [v_3])_\alpha, (k_3, [v_4])_\beta) \rightarrow [(k_4, v_5)]_\gamma$

keep the key

different dataset lineages  
( $\alpha = \beta \Rightarrow$  self-merge)

➤ merge  $\sim$  *two-dimensional list comprehension*  
in functional programming

# Example

- Two relational tables: Department and Employee
- Goal: Compute employee bonuses based on individual rewards and department bonus adjustments.

Emp	Dept	Bonus
1	B	Innovation (\$50)
1	B	Hard-worker (\$100)
2	A	High-performer (\$100)

Map

Emp	Dept	Bonus
1	B	\$50
1	B	\$100
2	A	\$100

Reduce

Emp	Dept	Bonus
2	A	\$100
1	B	\$150

Match keys on Dept

Emp	Bonus
2	\$95
1	\$172.5

Dept	Bonus adjustment
B	1.15
A	0.95

Map

Dept	Bonus adjustment
B	1.15
A	0.95

Reduce

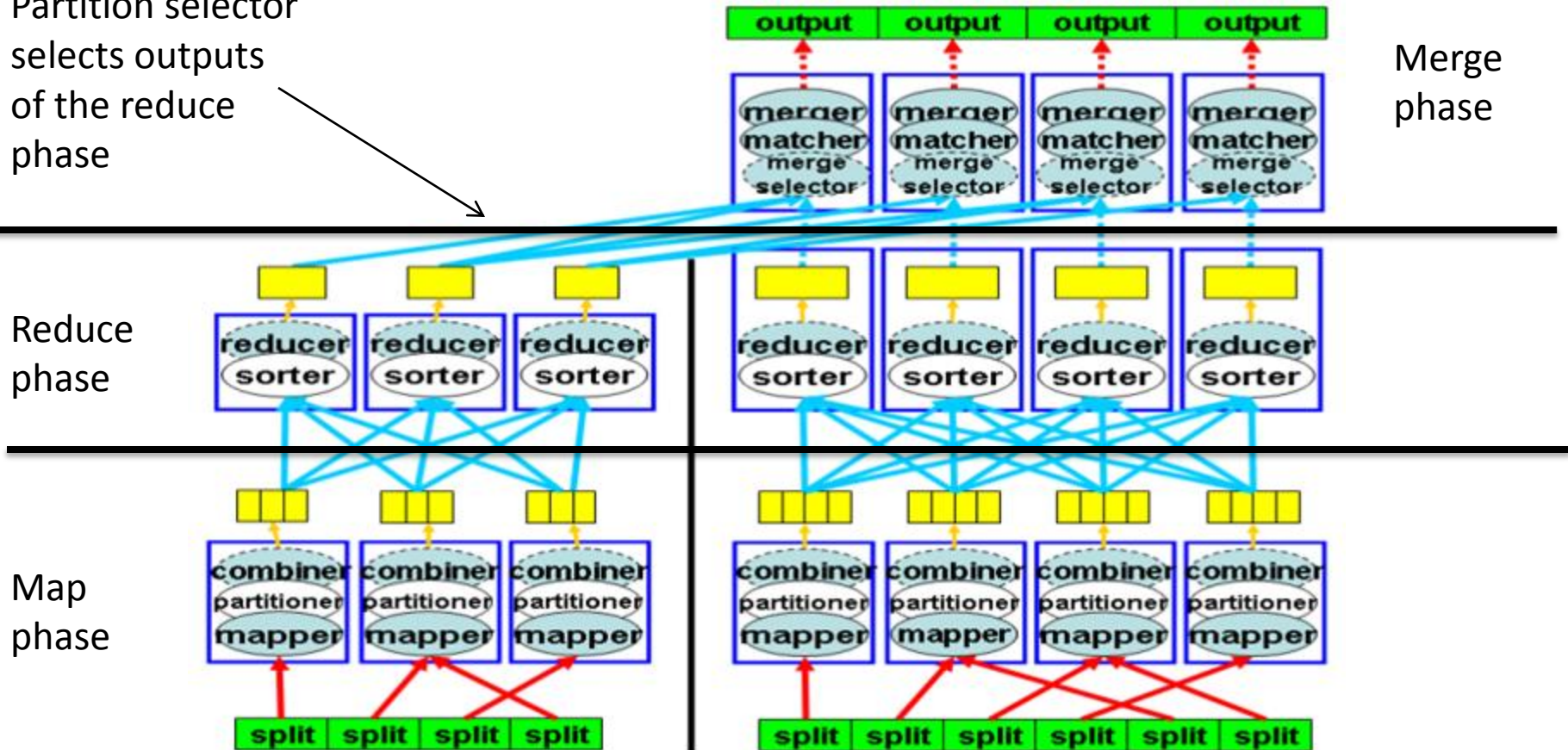
Dept	Bonus adjustment
A	0.95
B	1.15

# Primitive Components of Merge

- **Merge function:** user-defined data processing logic for the merger of two pairs of key/values, each coming from a different source.
- **Processor function:** user-defined function that processes data from one source only.
- **Partition selector:** user-definable module that shows I/O relationship btw reducers and mergers.
- **Configurable iterator:** user-configurable module that shows how to iterate through each input data as the merging is done.

# Implementation Overview

Partition selector selects outputs of the reduce phase



# Sort-Merge Join Algorithm

- **Map:** Partition records into buckets which are mutually exclusive and each key range is assigned to a Reducer.
- **Reduce:** Data in the sets are merged into a sorted set (sort the data).
- **Merge:** The merger joins the sorted data for each key range.

# Hash Join Algorithm

- **Map:** Records are partitioned into hashed buckets.
- **Reduce:** Records from these partitions are grouped and aggregated using a hash table (no sorting).
- **Merge:** Reducer outputs with the same hashing buckets are merged (build & probe).

# Block Nested-Loop Join Algorithm

- **Map and Reduce:** The same as those for the Hash Join.
- **Merge:** Nested-loop join instead of a hash join (i.e., the iteration logic is different).

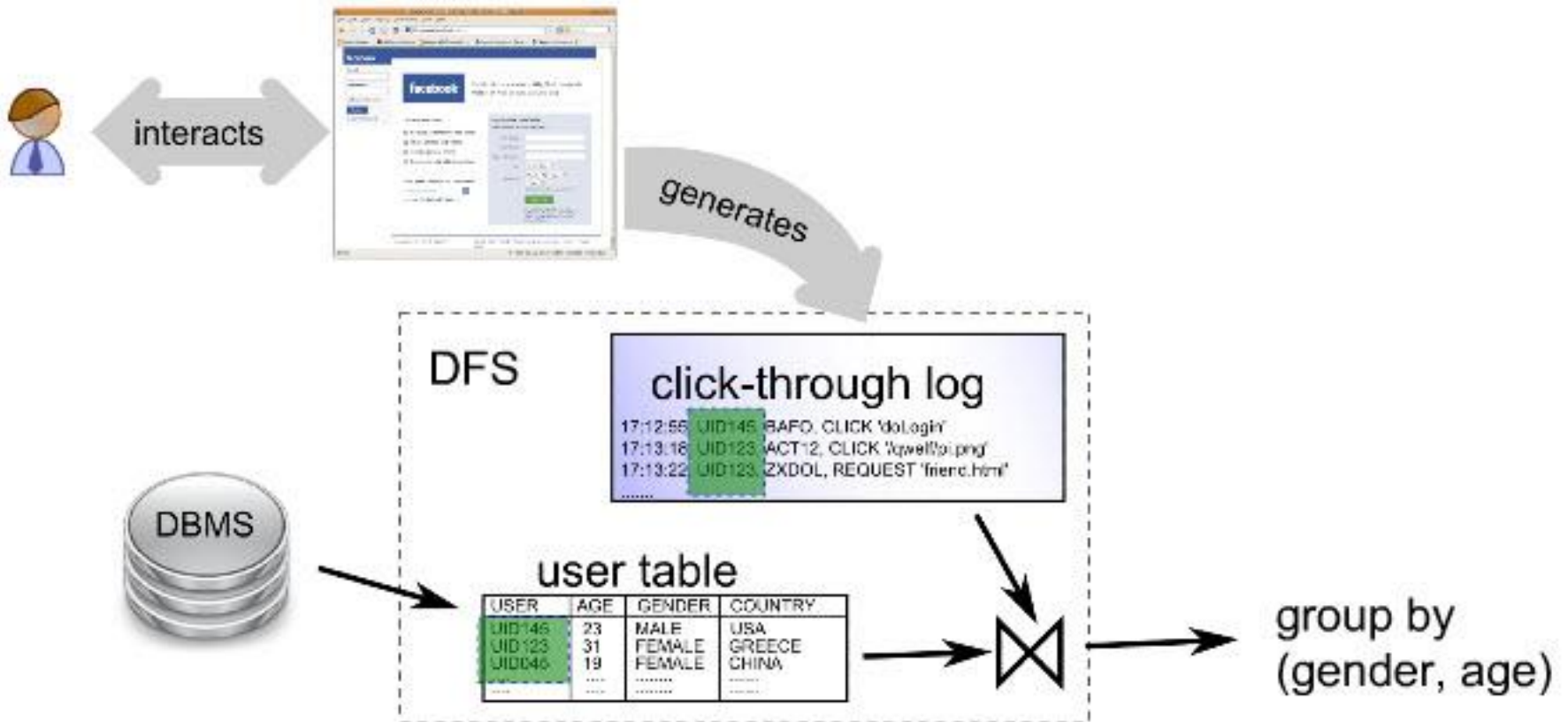
# Optimizations

- MapReduce already optimizes using locality and backup tasks.
- Optimize the **number of network connections** between the outputs of the Reduce phase and the input of the Merge phase (via customizing the partition selector function).
- Reduce **disk I/O** by combining two phases into one (e.g., ReduceMerge)

# How is Hadoop doing it?

# Example Join Application

## Log ⋈ User

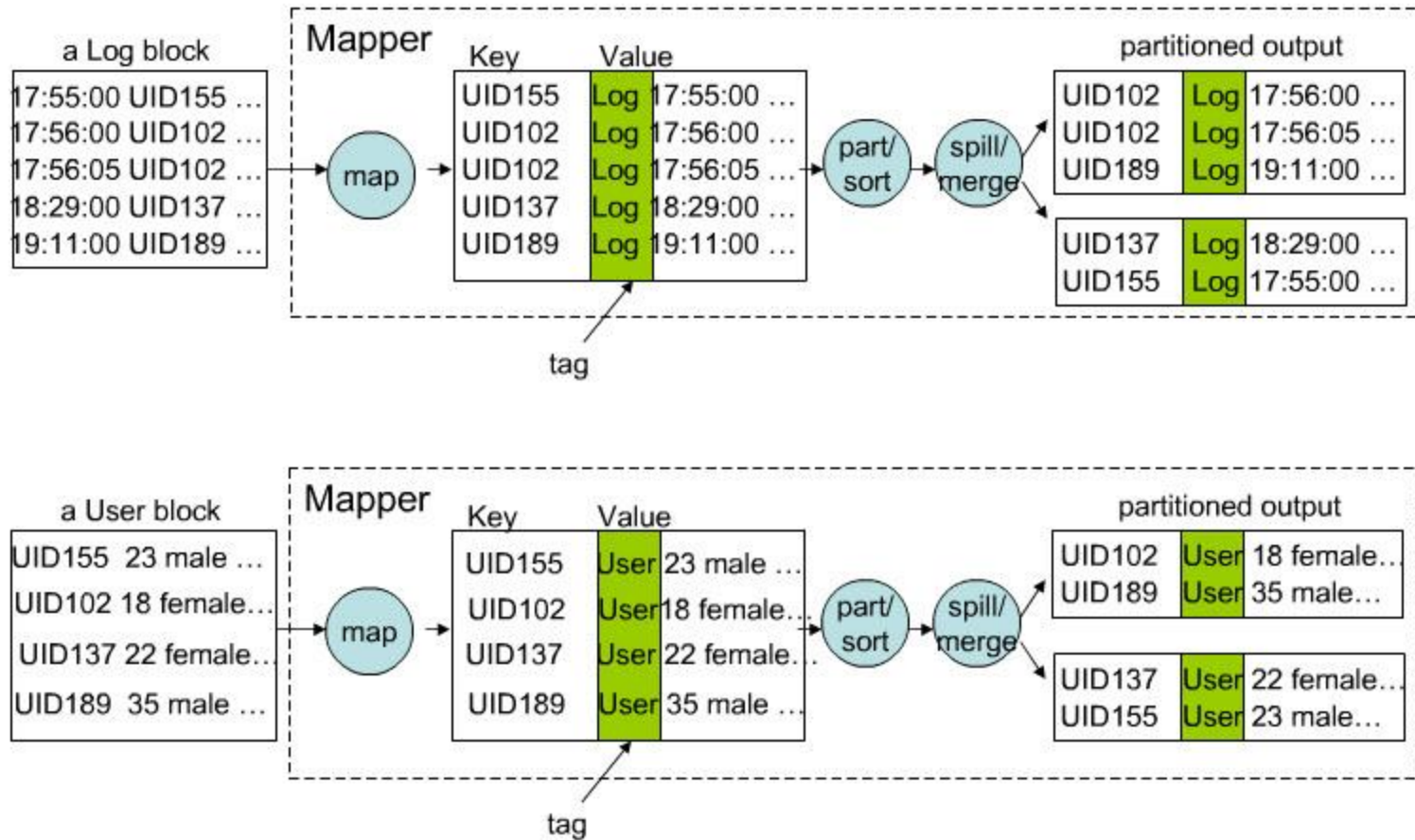


# Default Join in Hadoop

- `org.apache.hadoop.contrib.utils.join`
- Like **Repartitioned Sort-Merge Join** in databases
- Single MapReduce job (`DataJoinJob`)
- Mapper (`DataJoinMapperBase`):
  - Tag each input record with data source label
  - Extract join key
- Reducer, for each key (`DataJoinReducerBase`):
  - Separate records from different data sources
  - Generate cross product

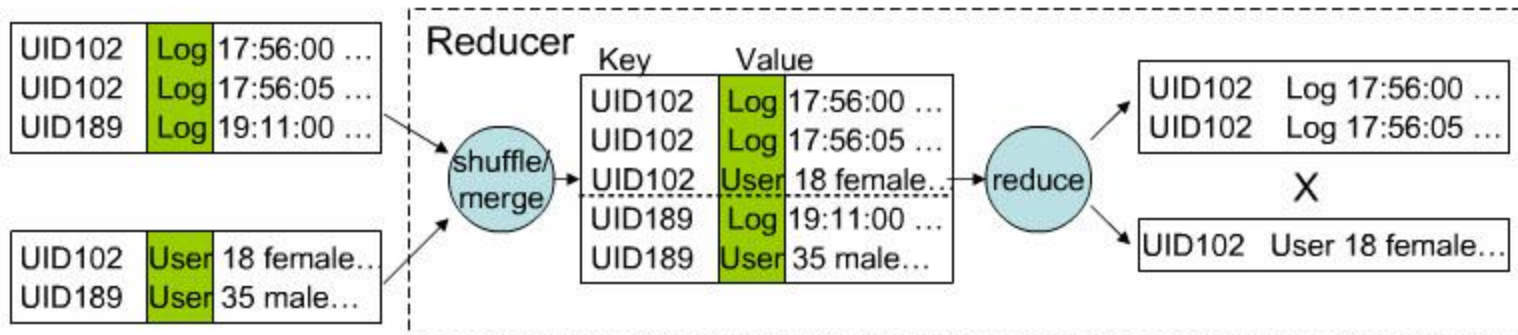
# Default Join in Hadoop

## Log $\bowtie$ User (Mapper)



# Default Join in Hadoop

## Log $\bowtie$ User (Reducer)



# Problems with Repartitioned Sort-Merge Join

- Major problems
  - Has to sort Log
  - Has to move Log across the network
    - Due to shuffling, must send the whole Log data.
- Minor problems
  - Popular key problem (skew)
    - All records for a particular key are sent to the same reducer.
  - Tagging overhead

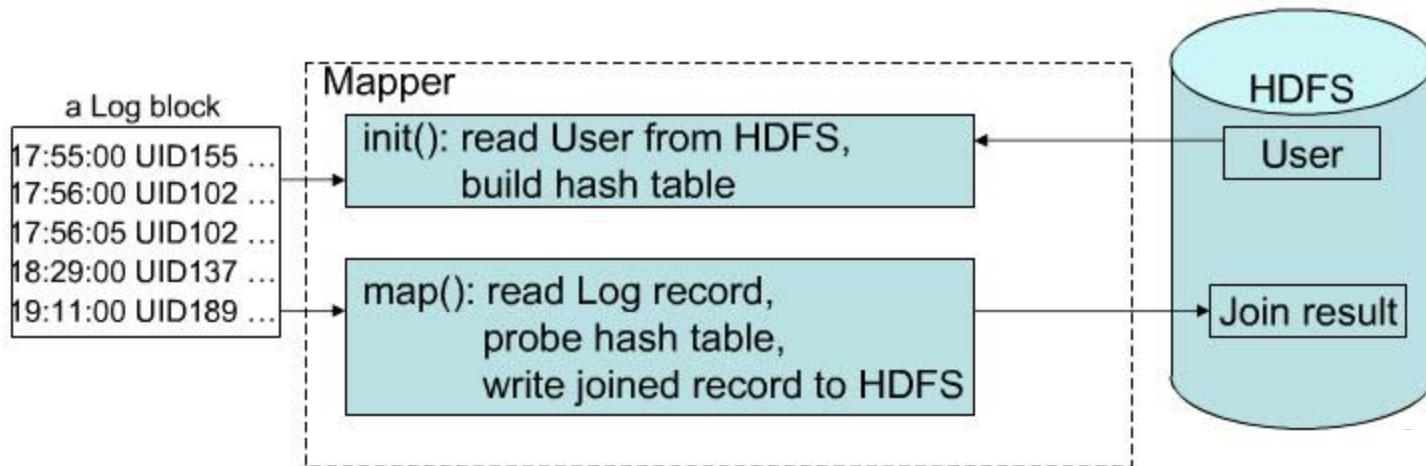
Can we do better?

# Improvements on Hadoop Join

- DB community has studied distributed joins for a long time
  - Strategies for avoiding sort
  - Strategies for reducing network overhead
- Apply database techniques to MapReduce
- Does it fit into the Hadoop framework?

# Replicated Join Strategy

- Observation: large to small join is common
  - Example: Log is orders of magnitude larger than User.
- Strategy:
  - Mapper-only job to avoid sort
  - Schedule Mapper on large input source to reduce data movement across the network.
- So-called, “map-side join strategy”



# Replicated Join Strategy

- Pluses:
  - No sort
  - Log data not moved over the network
- Minuses:
  - If User data is also large:
    - Full User data is copied to every Mapper
    - Full User data is used to build hash table in every Mapper

# Improvement on Replicated Join

- Observation: User data may be large, but Log may reference a small fraction of all users.
- Strategy:
  - Shrink User data size through semi-join (by pre-processing).

# Semi-join Strategy

- Use 3 separate MapReduce jobs.
- Phase 1: Extract
  - Extract unique user IDs referenced in Log.
- Phase 2: Filter
  - Filter User data with referenced user IDs.
- Phase 3: Join
  - Join Log with filtered User data.

# Semi-join: Phase 1

- Extract unique user IDs referenced in Log
- A MapReduce job:
  - Mapper: Extract user IDs from Log records
  - Reducer: Accumulate all unique user IDs
- Special map() code to reduce sort overhead

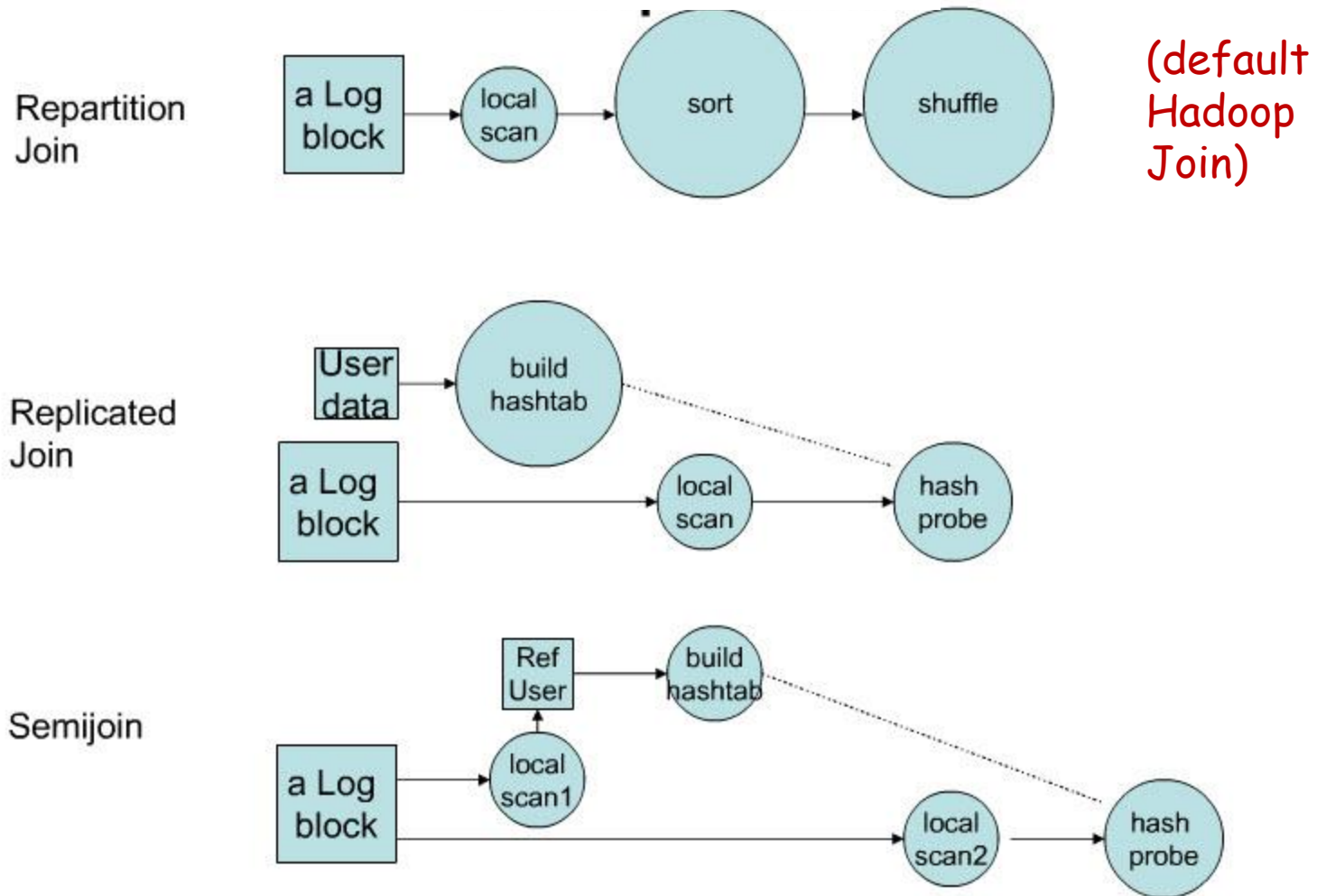
```
void map(key, value, collector) {  
    if (!hashset.contains(key)) {  
        collector.output(key)  
        hashset.add(key)  
    }  
}
```

- Number of records to sort = uniquely referenced user IDs, not number of Log records.

# Semi-join: Phases 2 and 3

- Phase 2: Filter
  - Full User data  $\bowtie$  referenced unique user IDs
  - Apply replicated join
    - Replicate referenced unique user IDs
  - Output: user ID + needed user attributes
- Phase 3: Join
  - Log data  $\bowtie$  filtered User data from Phase 2
  - Apply replicated join
    - Replicate filtered User data

# Comparison of Strategies



# A Further Improvement

## Customizing the Split Size

- Overhead per Mapper in replicated join
  - Initialize/destroy JVM
  - Pull User data from HDFS
  - Build hash table
- Strategy: Fewer Mappers by using larger splits
  - Assign multiple (non-consecutive) blocks per split
  - Preserve locality
- Caveat:
  - Losing load balancing
  - More stress on the network

# Conclusions

- Joins in MapReduce is still research in progress.
  - Applying DB techniques looks helpful
- Hadoop provides a default mechanism.
- You can experiment with the presented research ideas (+ your own ideas) in your projects and compare against the default.

# Next Week

- Project proposal presentations

Portions of this presentation are based on the content provided at <http://developer.yahoo.net/blogs/hadoop/>