MapReduce II

Big Data Management

Knowledge objectives

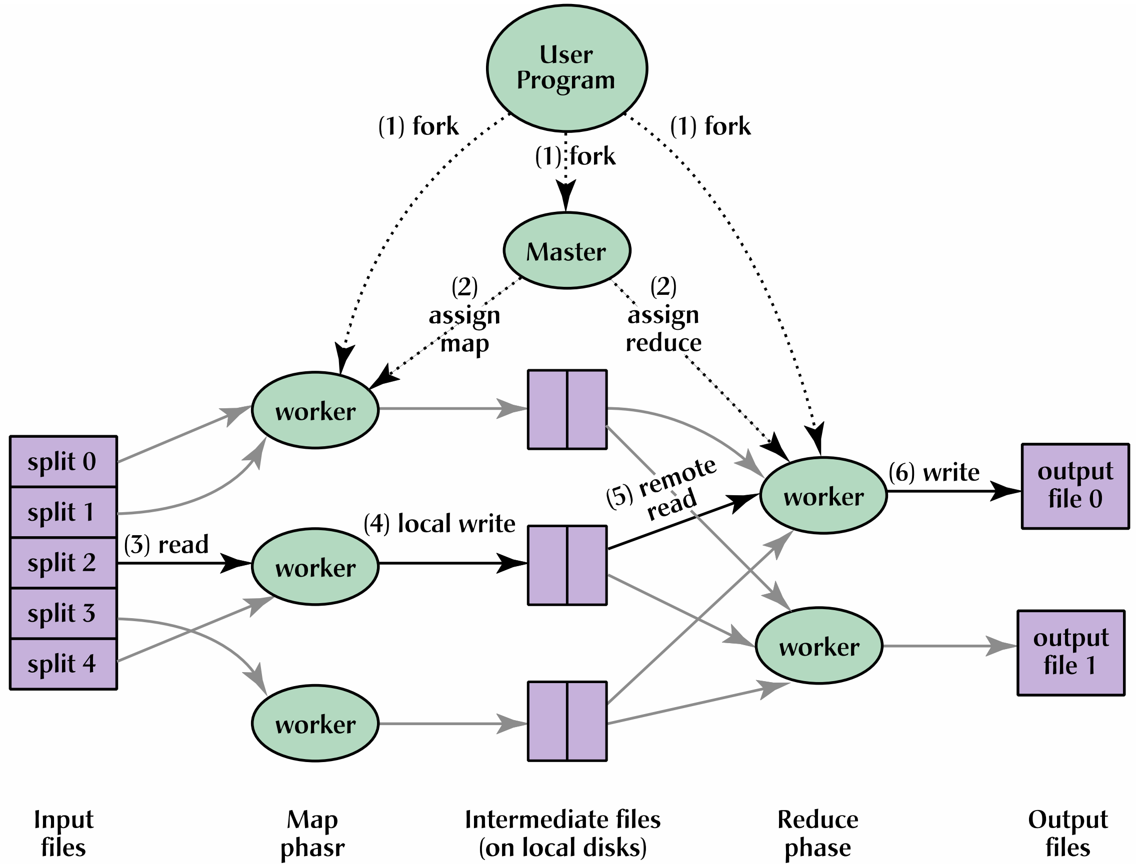
1. Enumerate the different kind of processes in Hadoop MapReduce
2. Draw the hierarchy of Hadoop MapReduce objects
3. Explain the information kept in the Hadoop MapReduce master node
4. Explain how to decide the number of mappers and reducers
5. Explain the fault tolerance mechanisms in Hadoop MapReduce in case of
   a) Worker failure
   b) Master failure
6. Identify query shipping and data shipping in MapReduce
7. Explain the effect of using the combine function in MapReduce
8. Identify the synchronization barriers of MapReduce
9. Explain the main problems and limitations of Hadoop MapReduce

Understanding Objectives

1. Apply the different steps of a MapReduce execution at the implementation level
2. Decide on the use of the combine function

Architecture

Processes



J. Dean et al.

Architectural decisions

- Users submit jobs to a master scheduling system
  - There is one master and many workers
  - Jobs are decomposed into a set of tasks
  - Tasks are assigned to available workers within the cluster/cloud by the master
    - O(M + R) scheduling decisions
    - Try to benefit from locality
    - As computation comes close to completion, master assigns the same task to multiple workers
- The master keeps all relevant information
  a) Map and Reduce tasks
    - Worker state (i.e., idle, in-progress, or completed)
    - Identity of the worker machine
  b) Intermediate file regions
    - Location and size of each intermediate file region produced by each map task
      - Stores O(M * R) states in memory

Design decisions

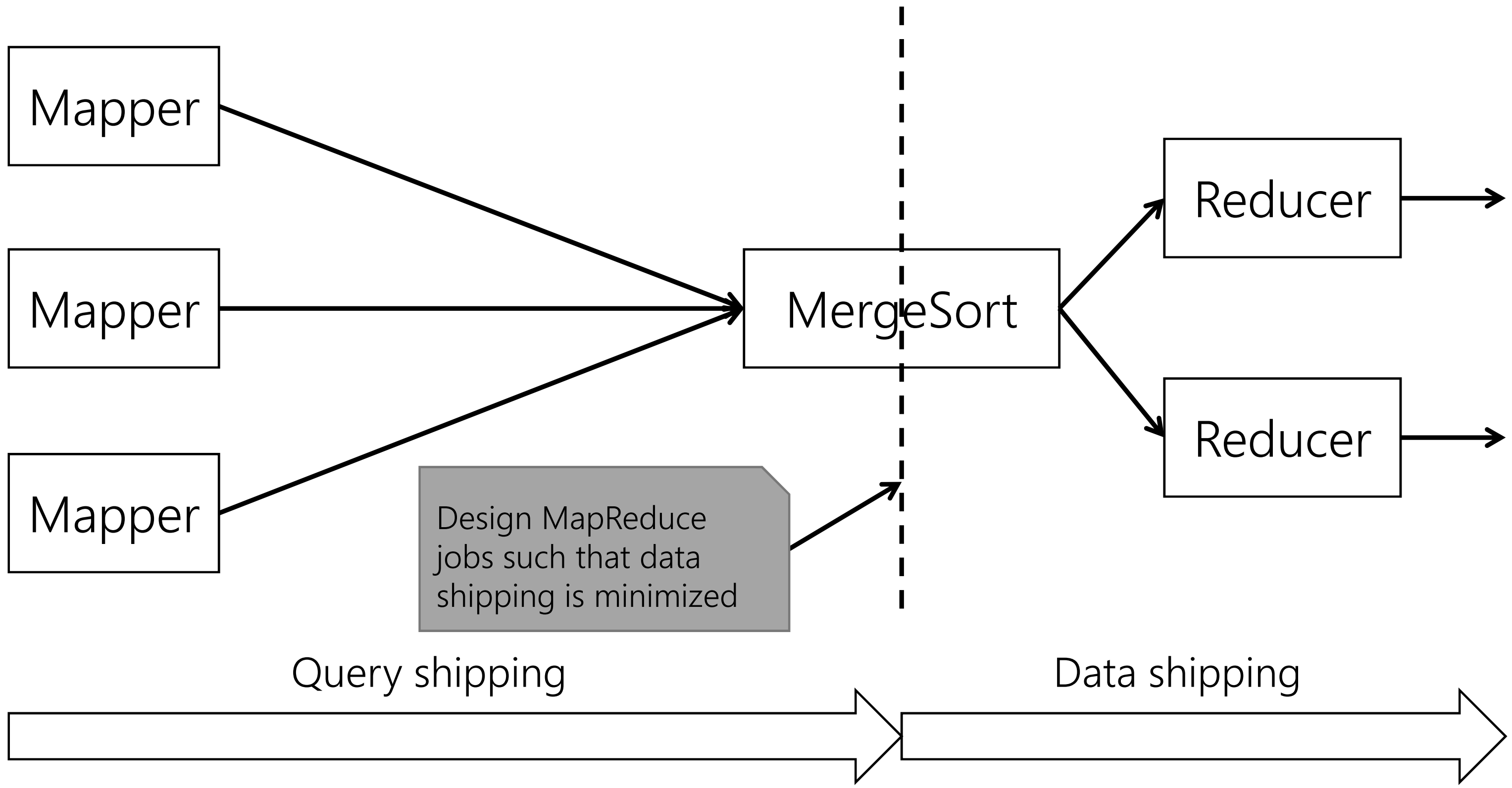
- Number of Mappers
  - One per split in the input (default one chunk)
    - To exploit data parallelism: $10*N < M < 100*N$
  - Mappers should take at least a minute to execute
    - Split size depends on the time to process data
- Number of Reducers
  - Many can produce an explosion of intermediate files
    - For immediate launch: 0.95*N*MaxTasks
    - For load balancing: 1.75*N*MaxTasks

Fault-tolerance mechanisms

- Worker failure
  - Master pings workers periodically (heartbeat)
    - Assumes failure if no response
  - Completed/in-progress map and in-progress reduce tasks on failed worker are rescheduled on a different worker node
    - Use chunk replicas
- Master failure
  - Since there is only one, it is less likely it fails
    - Keep checkpoints of data structure

Internal algorithm

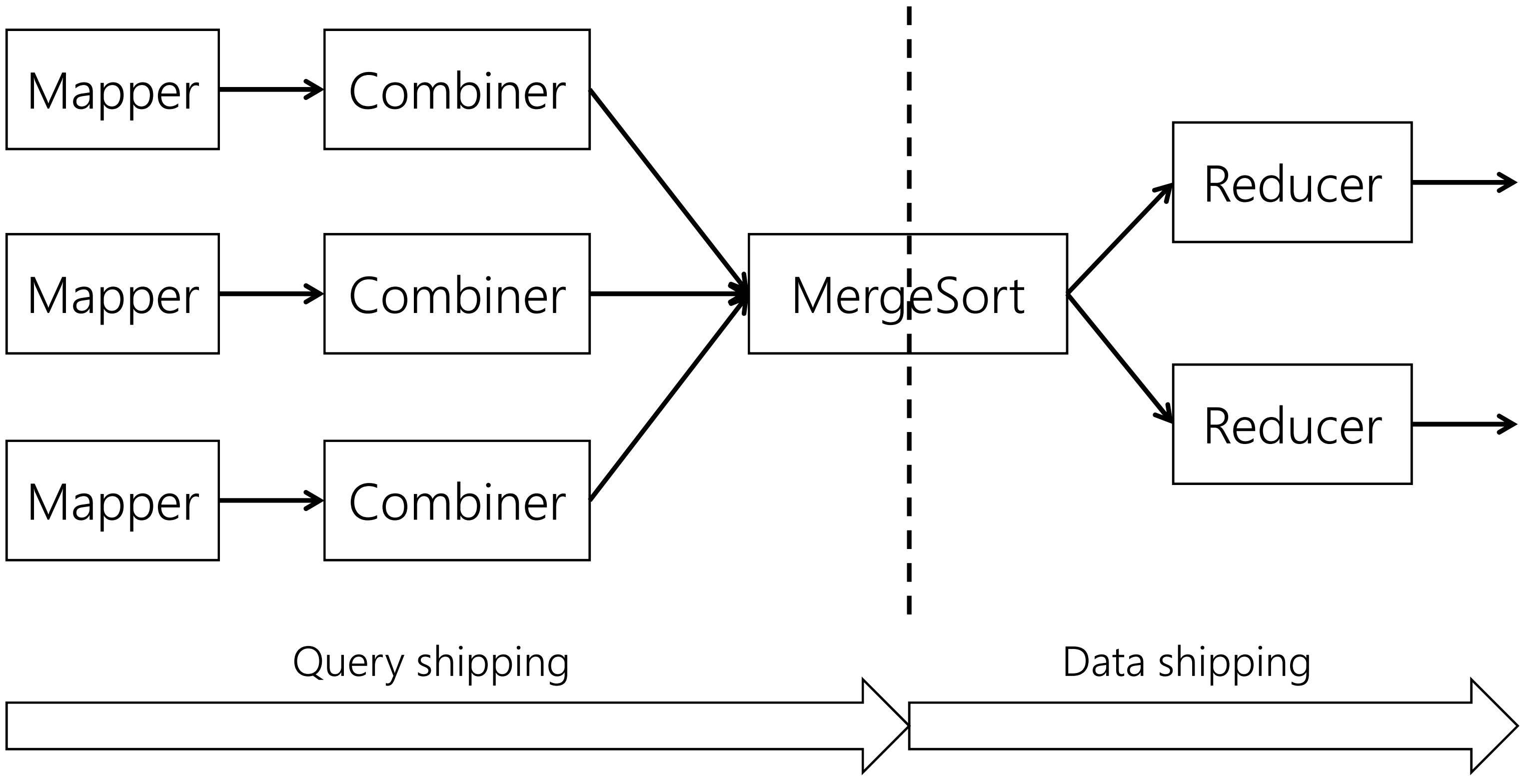
Query shipping vs. data shipping (I)



Combiner

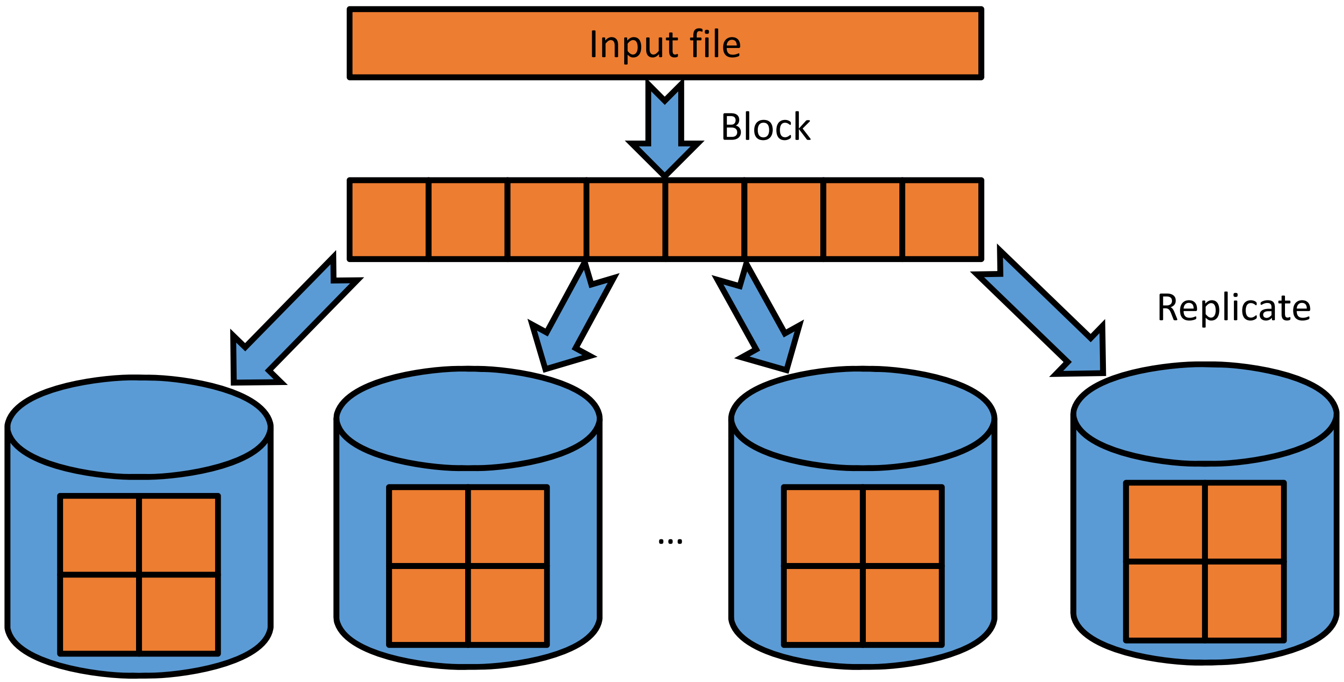
- Coincides with reducer function when it is:
  - Commutative
  - Associative
- Exploits data locality at the Mapper level
  - Data transfer diminished since Mapper outputs are reduced
    - Saving both network and storing intermediate results costs
- Only makes sense if $|I|/|O| >> \#CPU$
  - Skewed distribution of input data improves

Query shipping vs. data shipping (II)



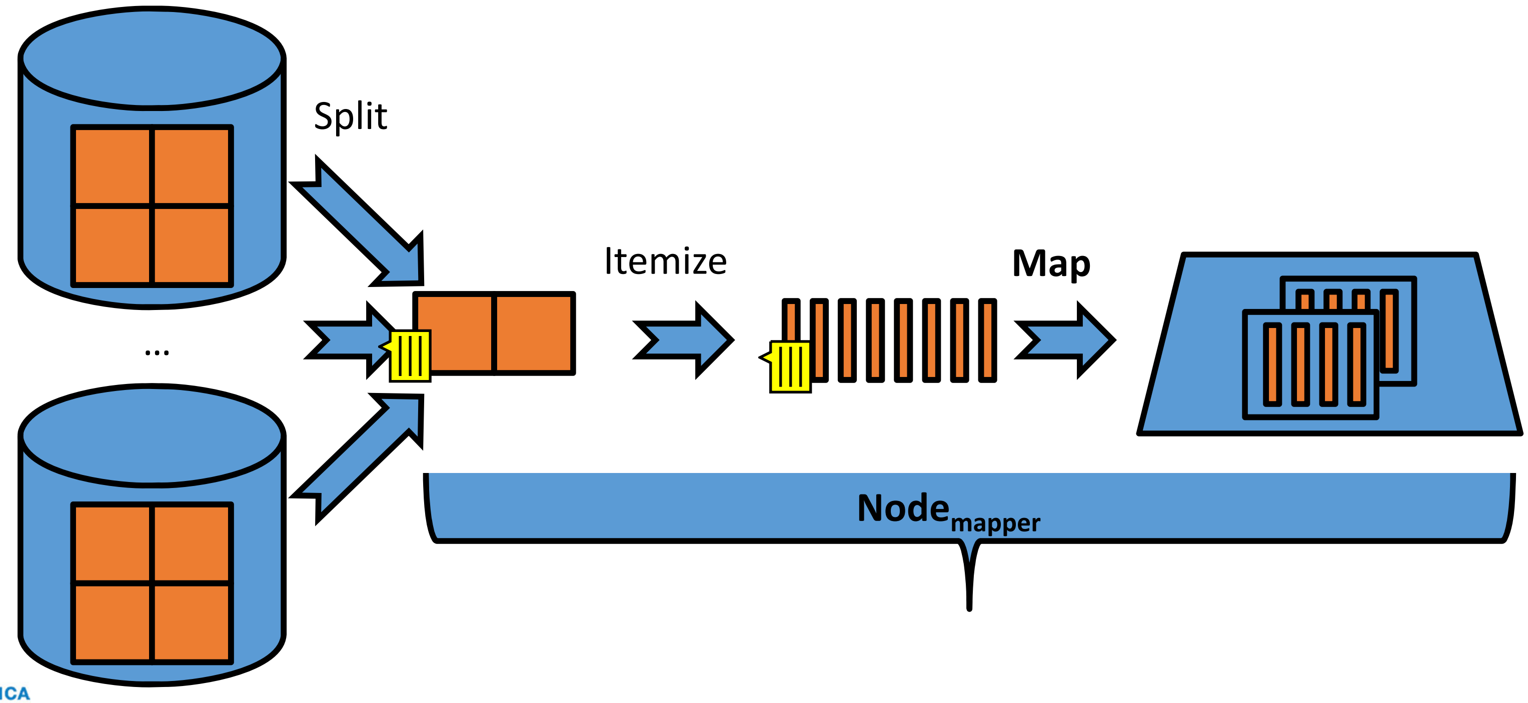
Algorithm: Data Load

1. Upload the data to the Cloud
   - Partition them into blocks
   - Using HDFS or any other storage (e.g., HBase, MongoDB, Cassandra, CouchDB, etc.)
2. Replicate them in different nodes



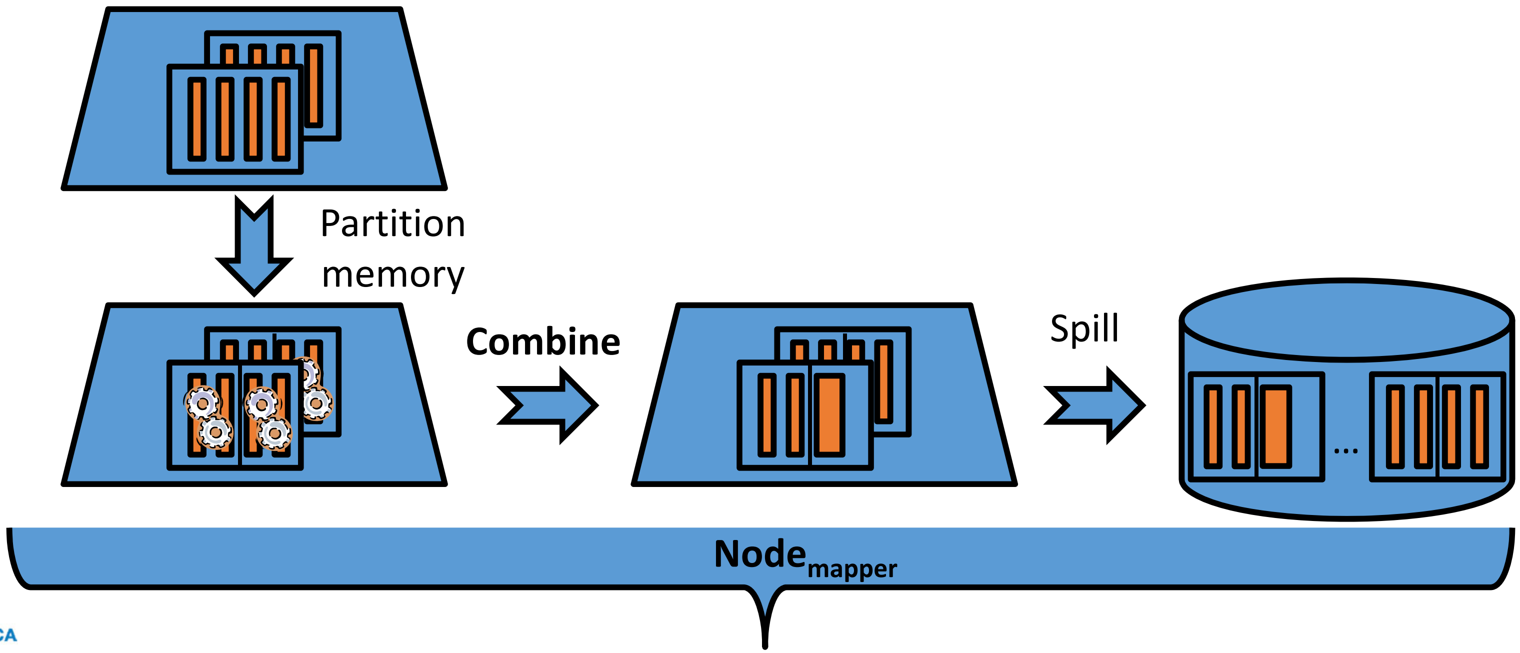
Algorithm: Map Phase (I)

3. Each mapper (i.e., JVM) reads a subset of blocks (i.e., split)
4. Divide each split into records
5. Execute the map function for each record and keep its results in memory
    - JVM heap used as a circular buffer



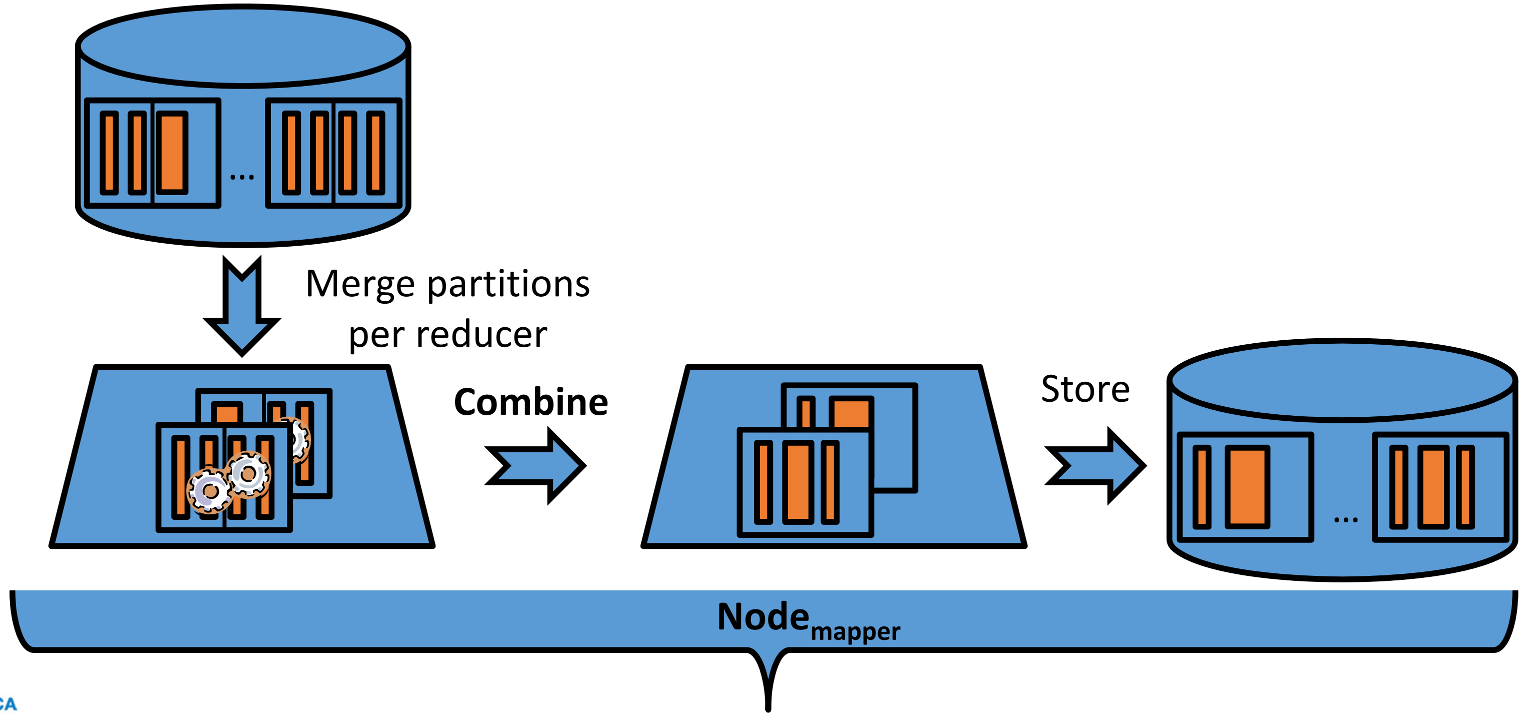
Algorithm: Map Phase (II)

6. Each time memory becomes full
   1. The memory is then partitioned per reducers
      - Using a hash function *f* over the new key
   2. Each memory partition is sorted independently
      - If a combine is defined, it is executed locally after sorting
   3. Spill partitions into disk (massive writing)



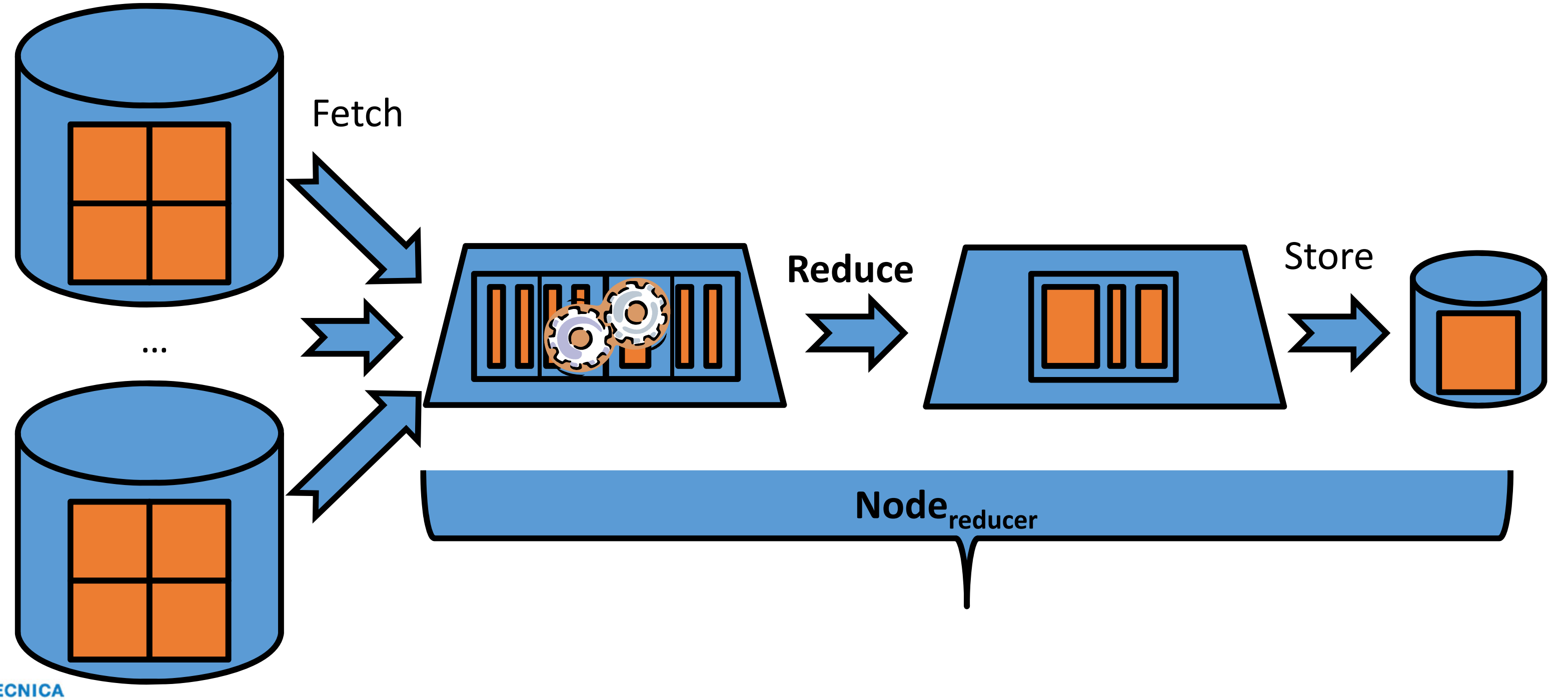
Algorithm: Map phase (III)

7. Partitions of different spills are merged
   - Each merge is sorted independently
   - Combine is applied
8. Store the result into disk

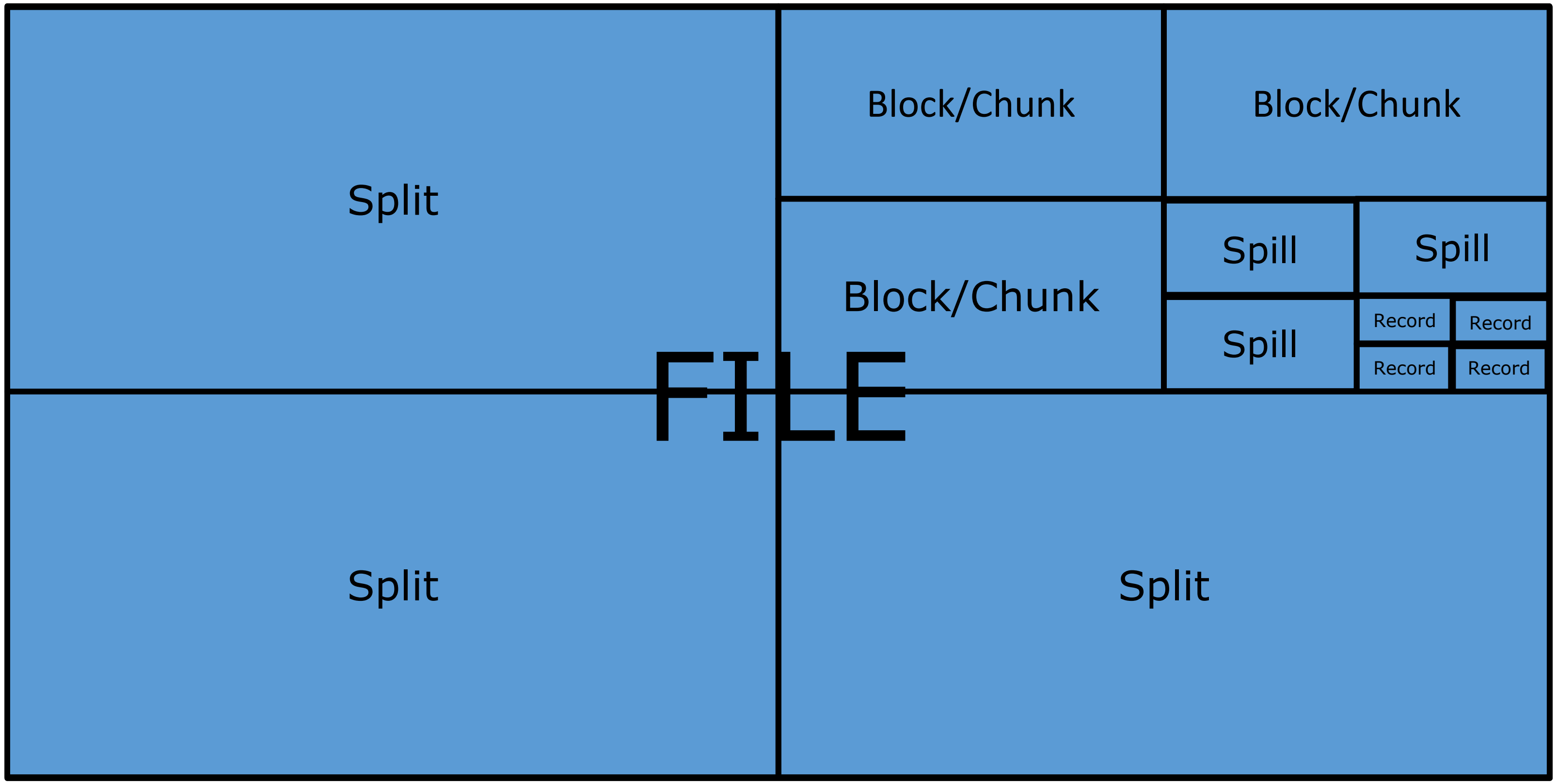


Algorithm: Shuffle and Reduce

9. Reducers fetch data through the network (massive data transfer)
10. Key-Value pairs are sorted and merged
11. Reduce function is executed per key
12. Store the result into disk

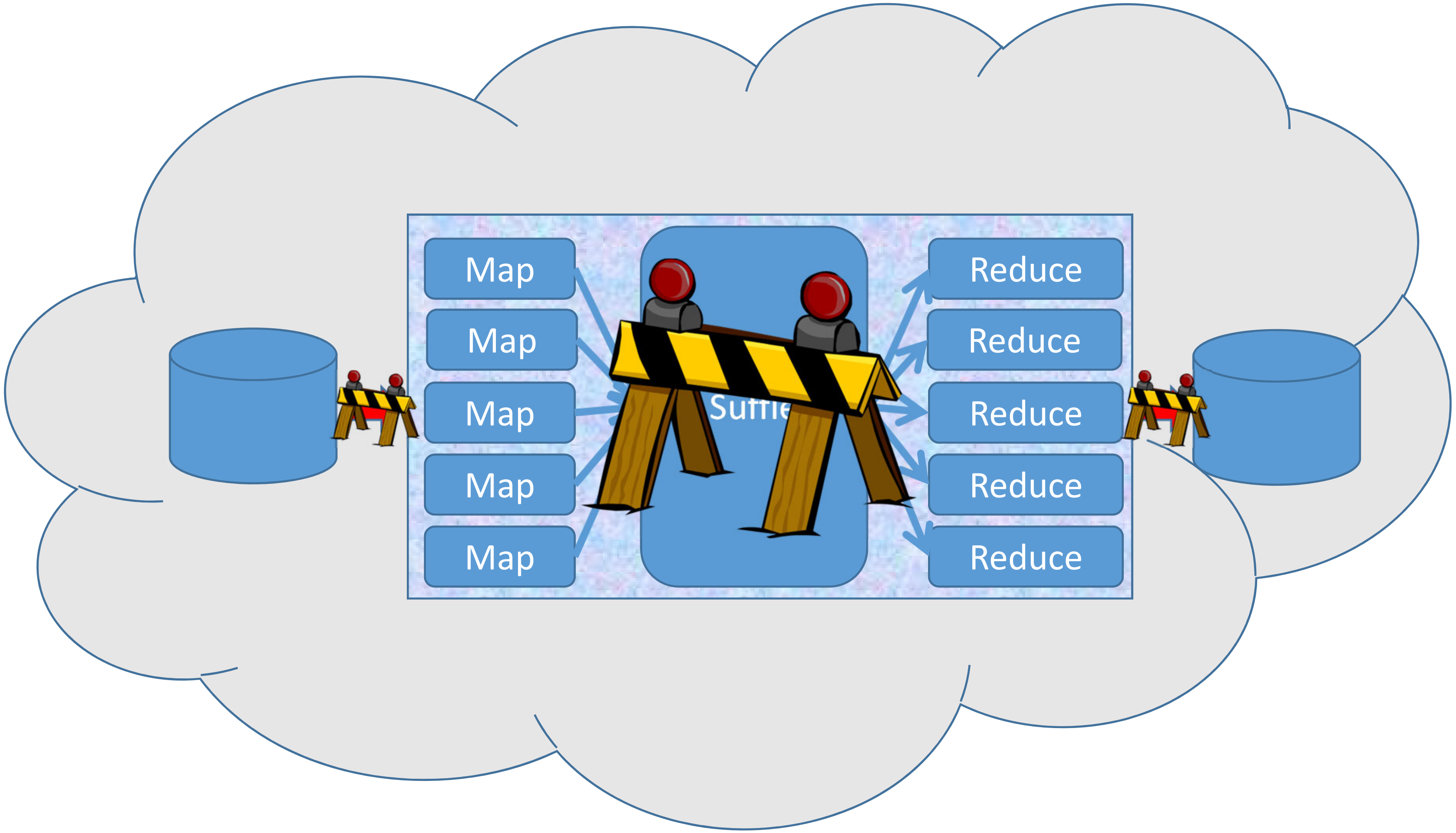


MapReduce objects

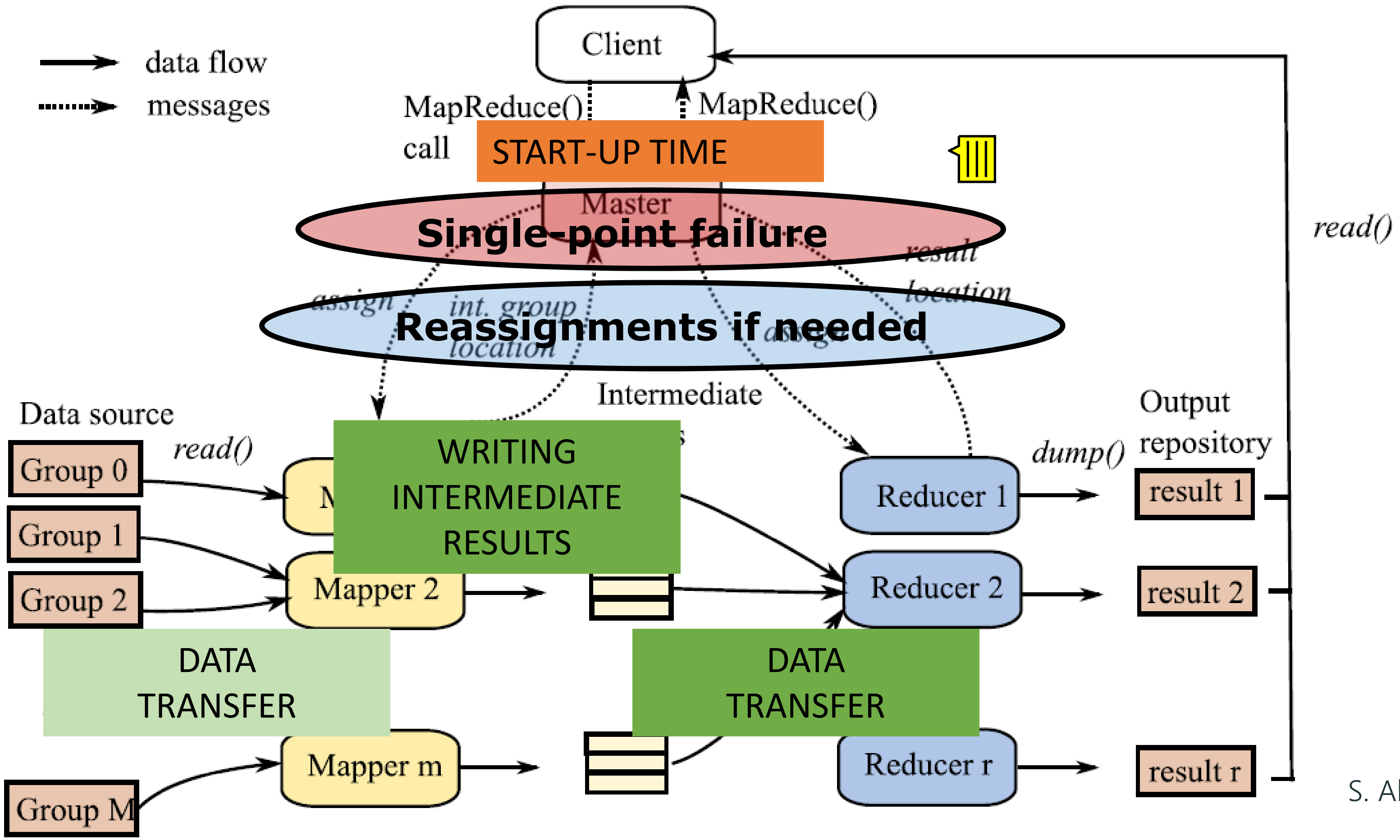


Record=Key-Value pair

Synchronization barriers



Tasks and Data Flows



S. Abiteboul et al.

Limitations

- Writes intermediate results to disk
  - Reduce tasks pull intermediate data
    - Improves fault tolerance
- Defines the execution plan on the fly
  - Schedules one block at a time
    - Adapts to workload and performance imbalance
- Does not provide transactions
  - Read-only system
    - Performs analytical tasks
- Does not benefit from compression during processing

Closing

Summary

- MapReduce architecture
  - Processes
  - Fault-tolerance mechanisms
  - Bottlenecks
    - Synchronization barriers
- MapReduce detailed algorithm
  - Query shipping
  - Data shipping
- MapReduce limitations

References

- J. Dean et al. *MapReduce: Simplified Data Processing on Large Clusters*. OSDI'04
- A. Pavlo et al. *A Comparison of Approaches to Large-Scale Data Analysis*. SIGMOD, 2009
- J. Dittrich et al. *Hadoop++: Making a yellow elefant run like a cheetah (without it even noticing)*. Proceedings of VLDB 3(1), 2010
- A. Rajaraman et al. *Mining massive data sets*. Cambridge University Press, 2012