Distributed Processing

Ricardo Vilaça
HASLab
University of Minho
rmvilaca@di.uminho.pt

Roadmap

- MapReduce
- DataFlow
- Apache Spark

Distributed Processing

- Expressing the computation separately from distribution
- Challenges:
  - Balancing the load
  - Minimizing data movement
- Batch processing
  - Takes a large amount of input data, runs a job to process it, and produces some output data
  - Jobs often take a while
  - The primary performance measure of a batch job is usually throughput

MapReduce

- Important building block to build reliable, scalable, and maintainable applications
- Fairly low-level programming model
- Inspiration from functional programming
- Takes one or more inputs and produces one or more outputs
- Read and write files on a distributed file system
- Can implicitly parallelize a computation across many machines
  - handle the complexities of moving data between machines
- Computation near the data

Job Execution Steps

1. Read a set of input files, and break it up into records
2. Call the mapper function to extract a key and value from each input record
3. Sort all of the key-value pairs by key
4. Call the reducer function to iterate over the sorted key-value pairs
   - If there are multiple occurrences of the same key, the sorting has made them adjacent in the list, so it is easy to combine those values without having to keep a lot of state in memory

Job Execution

- MapReduce is a programming framework with which you can write code to process large datasets
- Scheduler tries to run each mapper on one of the machines that stores a replica of the input file
- The mapper and reducer are each a Java class that implements a particular interface

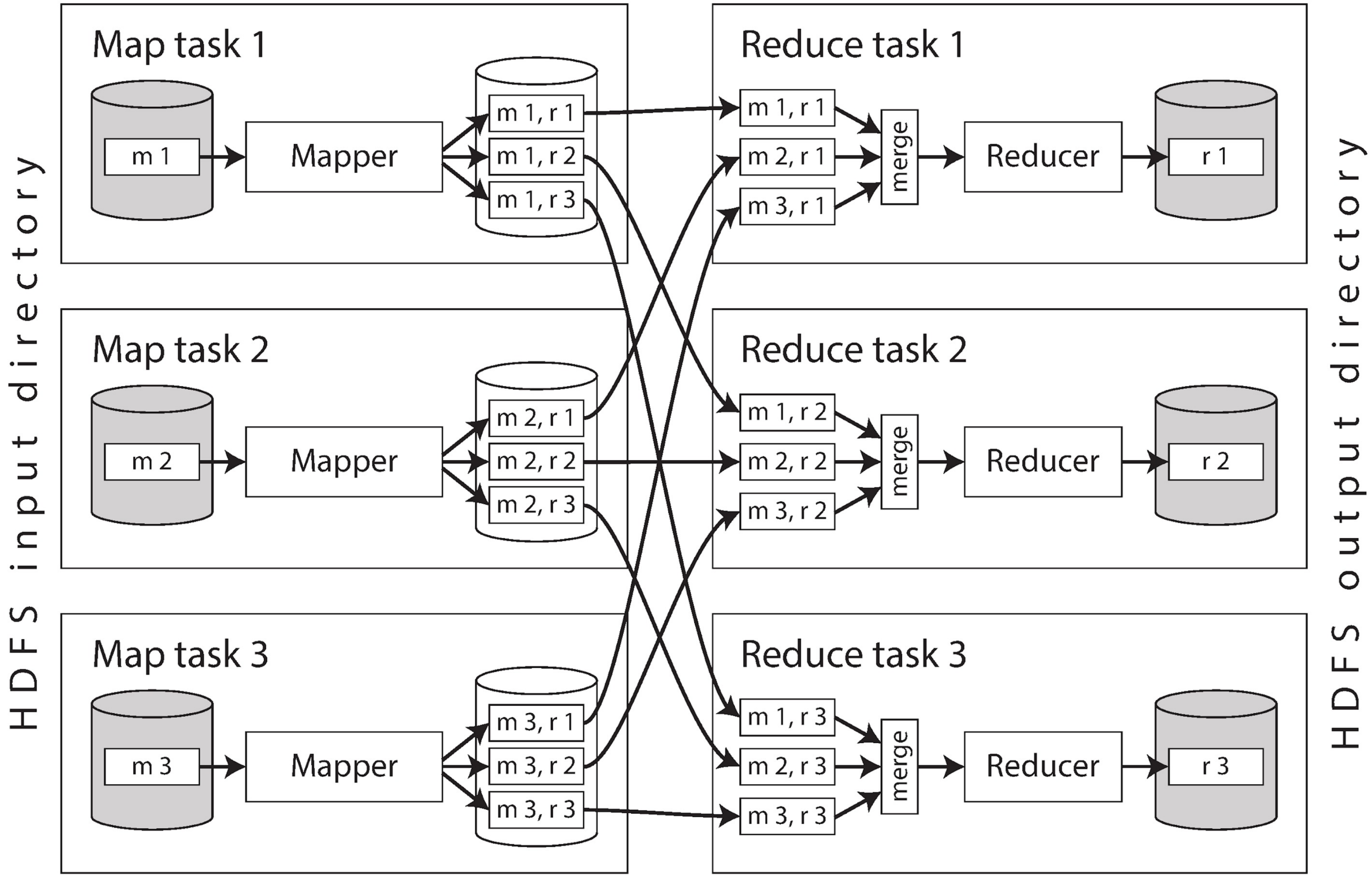


Image from M. Kleppmann(2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

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 reduce function
  - Similar to aggregation function in SQL

Wordcount example

```
class WordCountMap(Mapper, MapReduceBase):

        def map(self, key, value, output, reporter):




class Summer(Reducer, MapReduceBase):
        def reduce(self, key, values, output, reporter):
```

Wordcount example

```
class WordCountMap(Mapper, MapReduceBase):
    one = IntWritable(1)
    def map(self, key, value, output, reporter):
        for w in value.toString().split():
            output.collect(Text(w), self.one)

class Summer(Reducer, MapReduceBase):
    def reduce(self, key, values, output, reporter):
    sum = 0
    while values.hasNext():
        sum += values.next().get()
    output.collect(key, IntWritable(sum))
```

Distributed Execution

- The framework first copies the code (e.g., JAR files in the case of a Java program) to the machines that are running MapReduce tasks
- Starts the map task and begins reading the input file, passing one record at a time to the mapper callback
- The output of the mapper consists of key value pairs
- Can tolerate the failure of a map or reduce task without it affecting the job as a whole by retrying work at the granularity of an individual task

Tasks

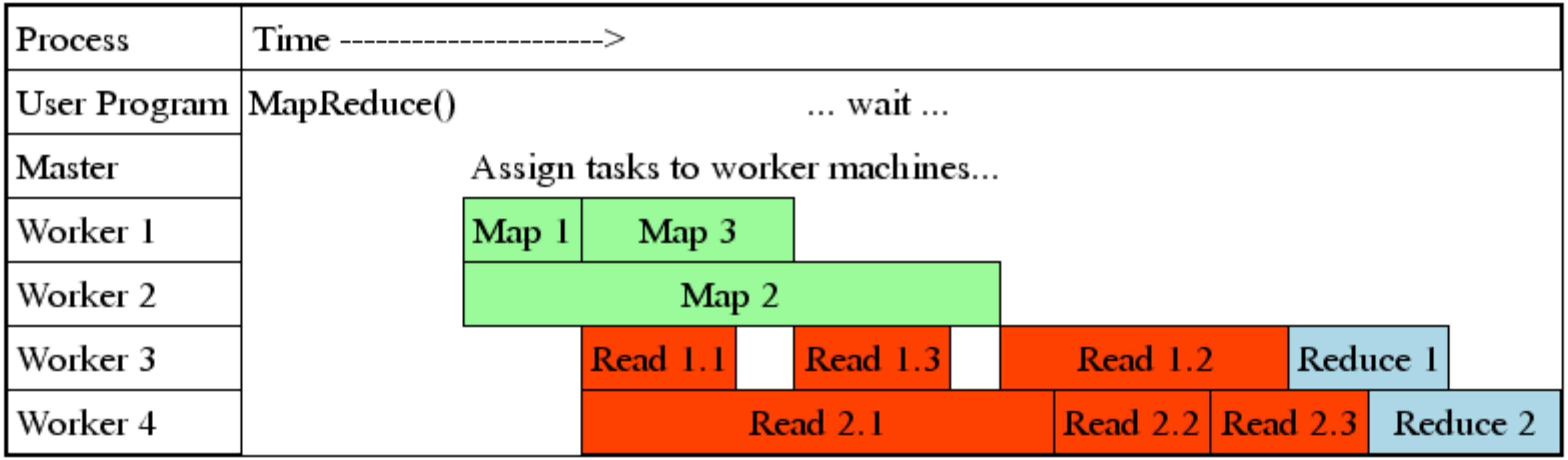


Image from https://research.google.com/archive/mapreduce-osdi04-slides/index.html

Partition

- The number of map tasks is determined by the number of input file blocks
- The number of reduce tasks is configured by the job author
- To ensure that all key-value pairs with the same key end up at the same reducer, the framework uses a hash of the key to determine which reduce task should receive a particular key-value pair

Partition

- The key-value pairs must be sorted
  - Sorting is performed in stages.
  - Each of these partitions is written to a sorted file on the mapper’s local disk
  - Whenever a mapper finishes reading its input file and writing its sorted output files, the MapReduce scheduler notifies the reducers that they can start fetching the output files from that mapper. The reducers connect to each of the mappers and download the files of sorted key-value pairs for their partition.
  - The process of partitioning by reducer, sorting, and copying data partitions from mappers to reducers is known as the shuffle

Partition

- The key-value pairs must be sorted
  - The reduce task takes the files from the mappers and merges them together, preserving the sort order.
  - Thus, if different mappers produced records with the same key, they will be adjacent in the merged reducer input.
  - The reducer is called with a key and an iterator that incrementally scans over all records with the same key (which may in some cases not all fit in memory). The reducer can use arbitrary logic to process these records, and can generate any number of output records.
  - These output records are written to a file on the distributed file system (usually, one copy on the local disk of the machine running the reducer, with replicas on other machines).

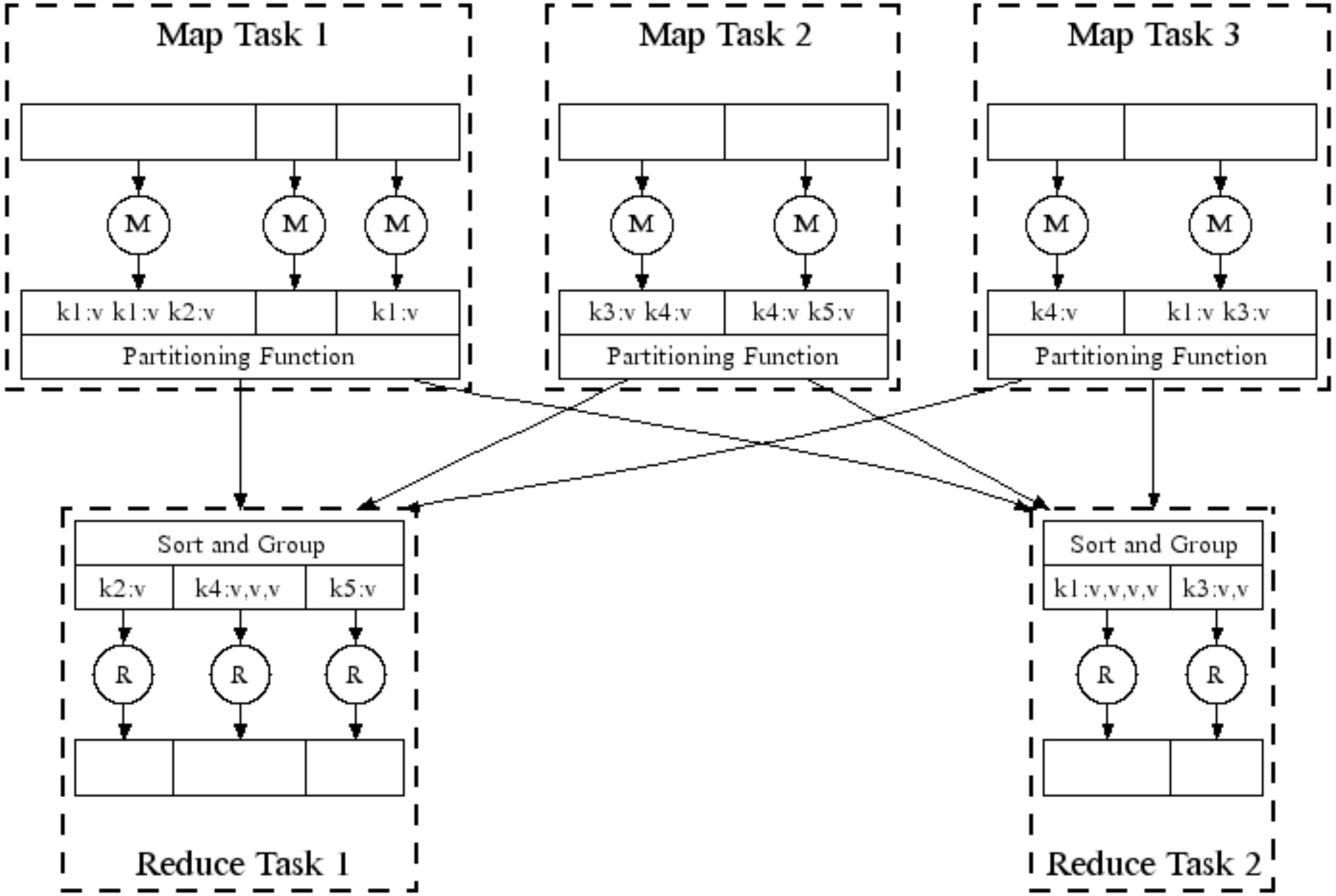


Image from https://research.google.com/archive/mapreduce-osdi04-slides/index.html

Workflows

- The range of problems you can solve with a single MapReduce job is limited
- It is very common for MapReduce jobs to be chained together into workflows
  - The output of one job becomes the input to the next job
  - Chaining is done implicitly by directory name
  - A batch job’s output is only considered valid when the job has completed successfully
  - One job in a workflow can only start when the prior jobs have completed successfully.
- Several workflow schedulers for Hadoop

High-Level MapReduce Languages

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

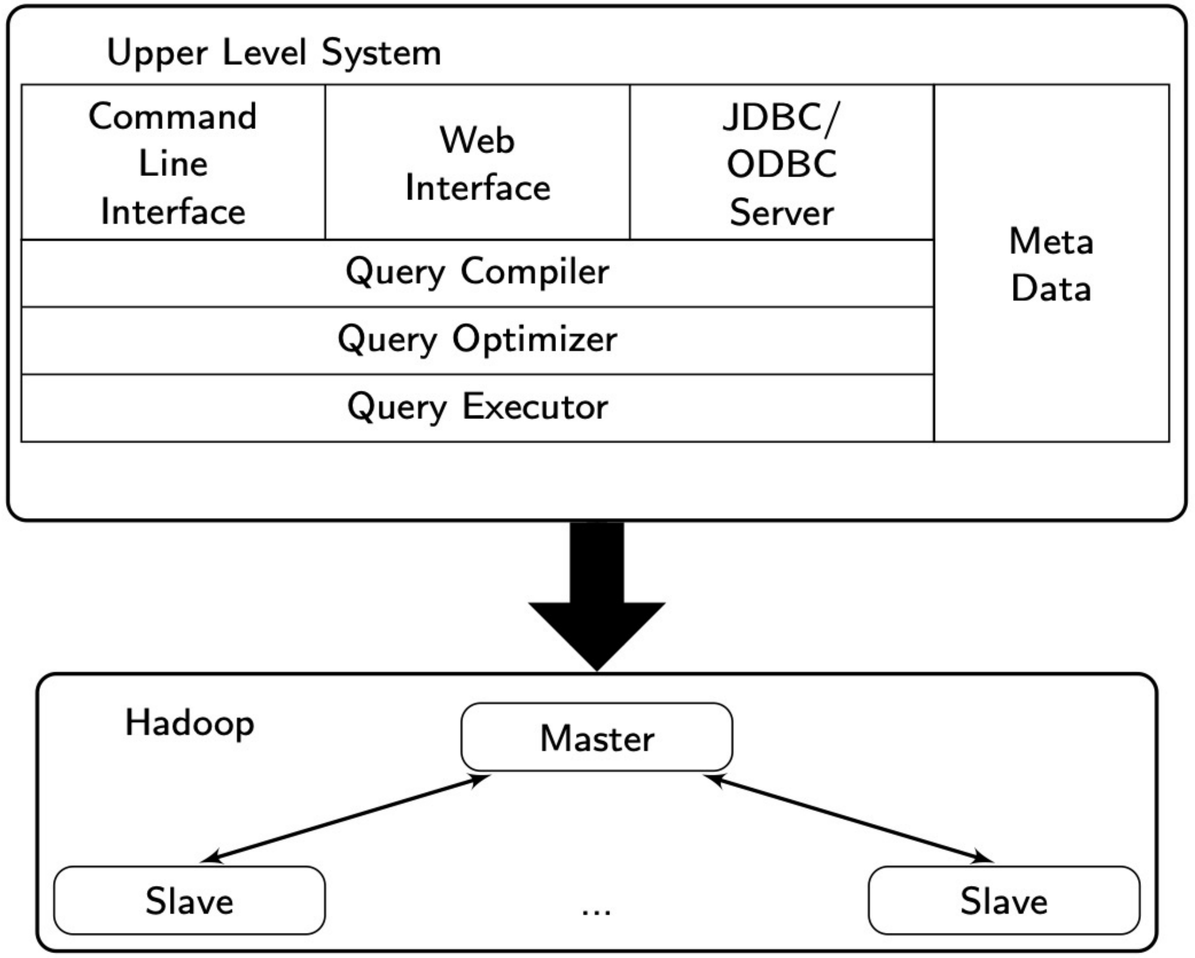


Image from M.T. Özsu & P. Valduriez (2020). Principles of Distributed Database Systems

Limitations

- Implementing a complex processing job using the raw MapReduce APIs is actually quite hard and laborious
- Other tools are sometimes orders of magnitude faster for some kinds of processing
- Several times the files on the distributed file system are simply **intermediate state**
  - a means of passing data from one job to the next.
- Fully materializing intermediate state has downsides
  - A MapReduce job can only start when all tasks in the preceding jobs have completed
  - Mappers are often redundant
  - Storing intermediate state in a distributed file system means those files are replicated
    - Overkill for such temporary data.

MapReduce Iterative Computation

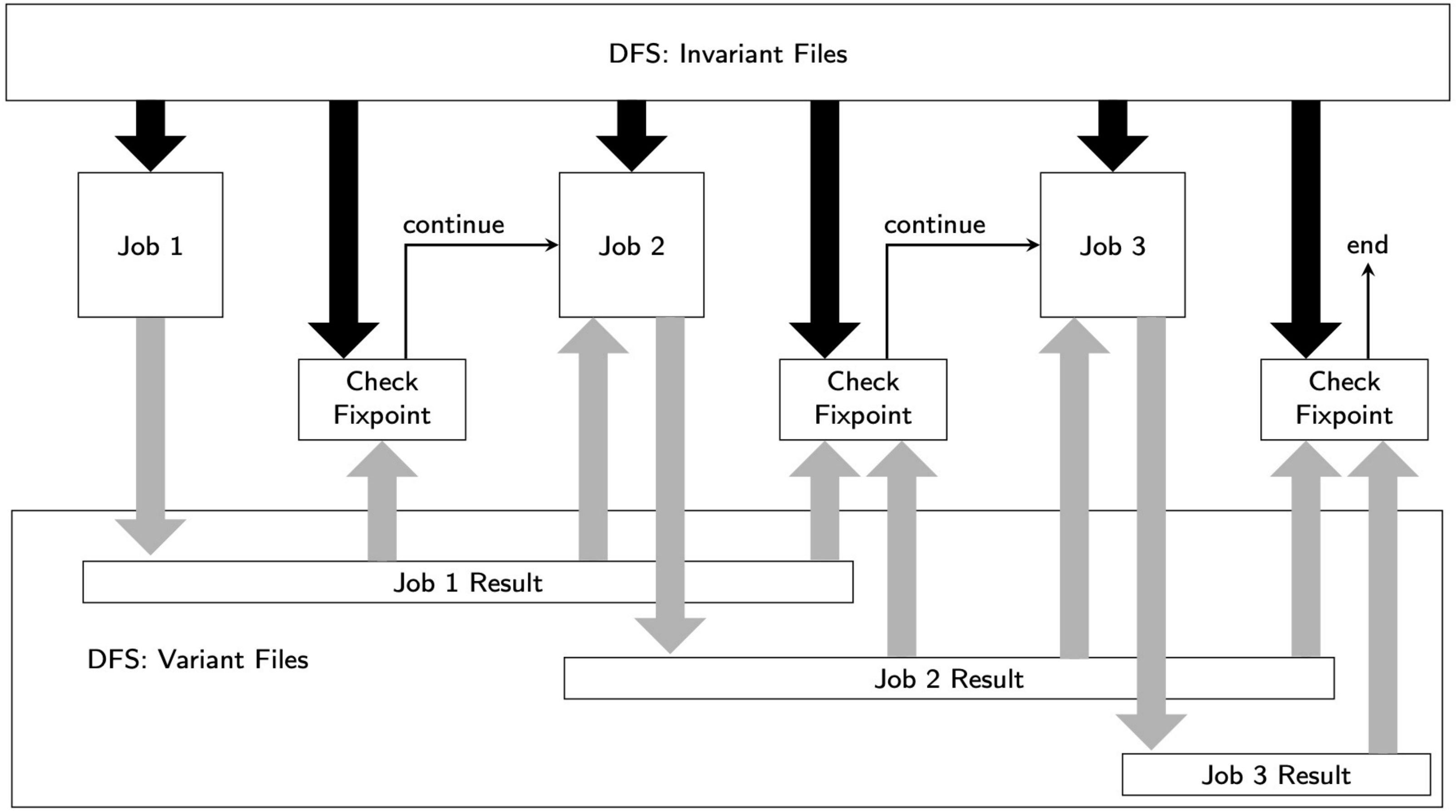


Image from M.T. Özsu & P. Valduriez (2020). Principles of Distributed Database Systems

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

Dataflow engines

- Several new execution engines for distributed batch computations were developed: Spark , Tez, Flink, DryadLINQ, Ray,
  - General-purpose execution environment for distributed, data-parallel applications
  - They handle an entire workflow as one job, rather than breaking it up into independent subjobs
  - They parallelize work by partitioning inputs, and they copy the output of one function over the network to become the input to another function
  - These functions need not take the strict roles of alternating map and reduce
  - Dataflow engines look much more like Unix pipes

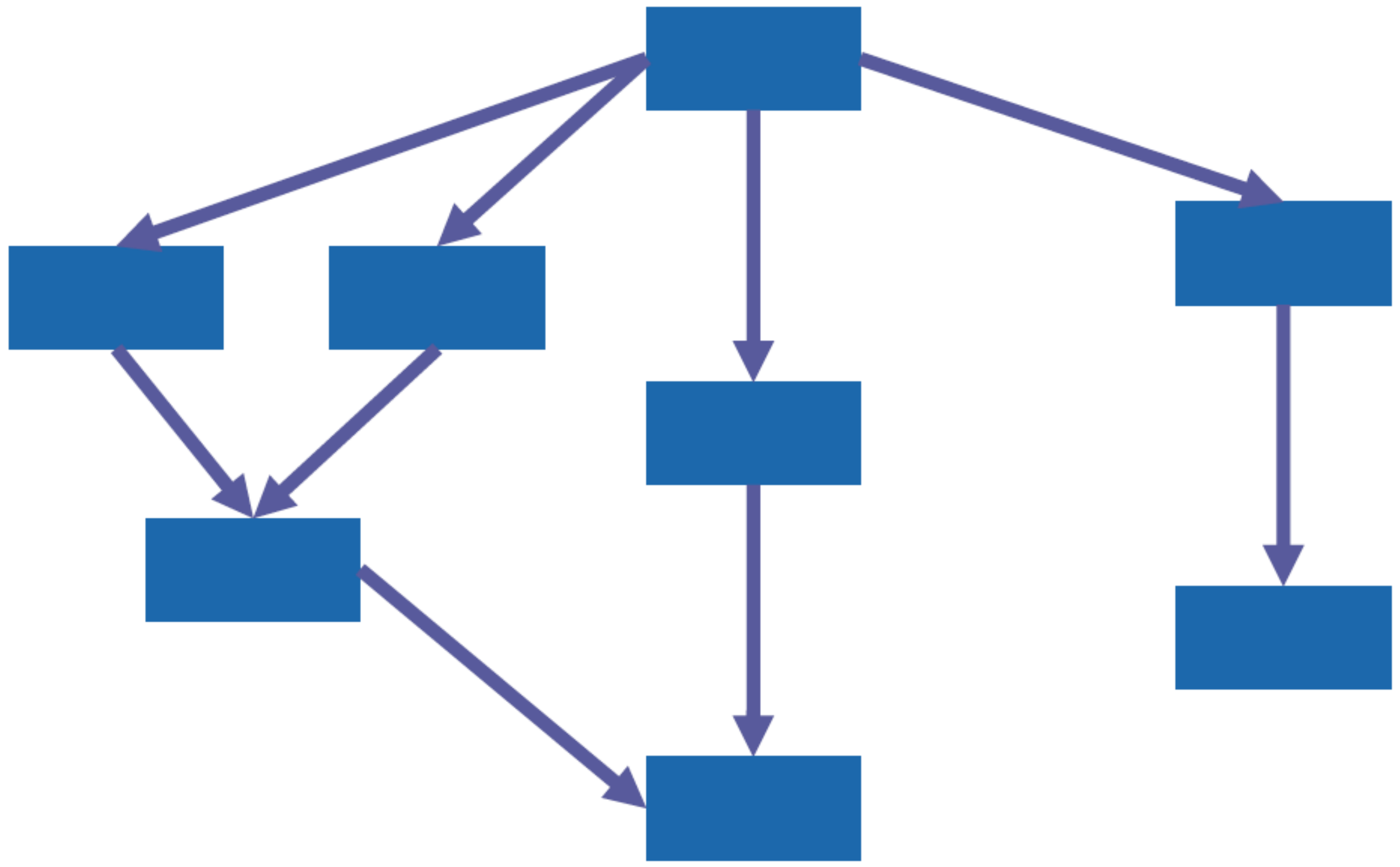
Dataflow engines

- Options for connecting one operator’s output to another’s input
  - Repartition and sort records by key, like in the shuffle stage of MapReduce
  - Take several inputs and to partition them in the same way, but skip the sorting
  - The same output from one operator can be sent to all partitions
    - broadcast hash joins
- The operators in a job are organized as a directed acyclic graph (DAG)

Dataflow engines

- Dataflow engines can be used to implement the same computations as MapReduce workflows
  - they usually execute significantly faster due to the optimizations
- The same processing code can run on either execution engine

DAG Processing



Advantages

- Expensive work such as sorting need only be performed in places where it is actually required
- There are no unnecessary map tasks
- Joins and data dependencies in a workflow are explicitly declared
  - the scheduler can make locality optimizations.

Advantages

- Intermediate state kept in memory or written to local disk
  - requires less I/O than writing it to HDFS
  - dataflow engines generalize the idea to all intermediate state
- Operators can start executing as soon as their input is ready
  - no need to wait for the entire preceding stage to finish before the next one starts
- Existing Java Virtual Machine (JVM) processes can be reused to run new operators
  - reduces startup overheads

Fault Tolerance

- Fully materializing intermediate state to a distributed file system in MapReduce makes fault tolerance fairly easy
  - if a task fails, it can just be restarted on another machine and read the same input again from the filesystem
- Dataflow avoid writing intermediate state to HDFS
  - Different approach to tolerating faults
    - If a machine fails and the intermediate state on that machine is lost, it is recomputed from other data that is still available
    - To enable this recomputation, the framework must keep track of how a given piece of data was computed
      - which input partitions it used, and which operators were applied to it

Fault Tolerance

- Spark uses the resilient distributed dataset (RDD) abstraction for tracking the ancestry of data
- Flink checkpoints operator state, allowing it to resume running an operator that ran into a fault during its execution
- When recomputing data, it is important to know whether the computation is deterministic
  - The solution in the case of non deterministic operators is normally to kill the downstream operators
  - Better to make operators deterministic
- Recovering from faults by recomputing data is not always the right answer

Apache Spark

- Spark is a distributed large-scale data processing engine that exploits in-memory computation and other optimizations
- One of the most popular data processing engine in the industry
- Many large companies use Spark at massive scale
- Spark extends the popular MapReduce model to efficiently cover a wide range of workloads, including interactive queries and stream processing
- Spark offers the ability to run computations in memory
- Every Spark application consists of a driver program that launches various parallel operations on a cluster

Apache 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

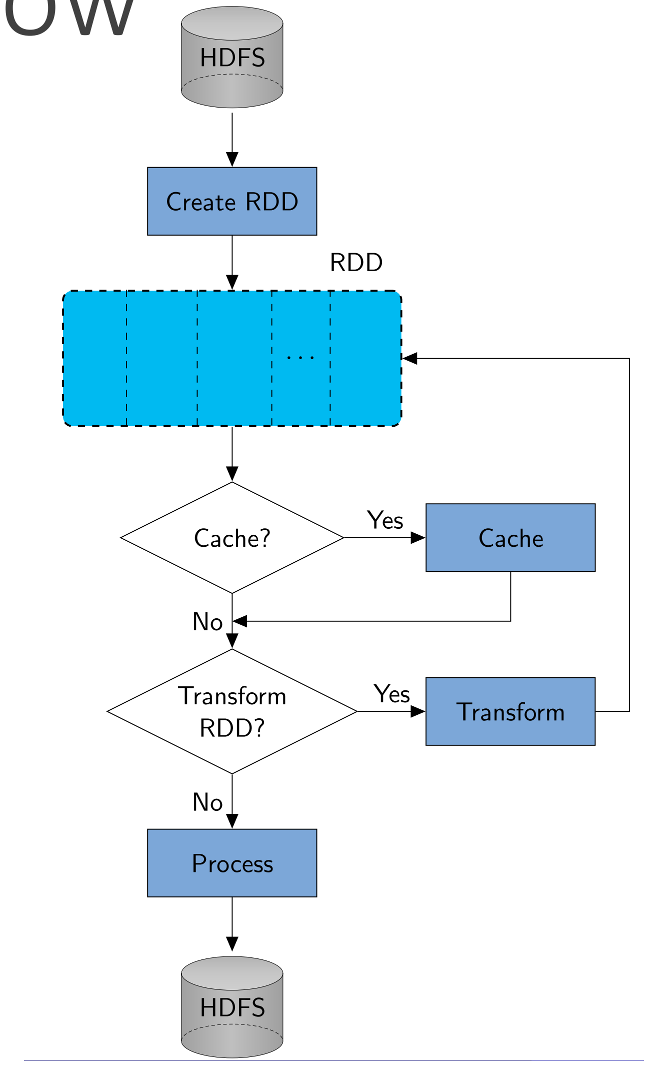


Image from M.T. Özsu & P. Valduriez (2020). Principles of Distributed Database Systems

Driver

- The driver program contains your application's main function and defines distributed datasets on the cluster, then applies operations to them.
- Driver programs access Spark through a SparkContext object, which represents a connection to a computing cluster.
- The driver communicates with a potentially large number of distributed workers called executors. The driver runs in its own process and each executor is a separate process. A driver and its executors are together termed a Spark application.

More information

- MapReduce: Simplified Data Processing on Large Clusters
- FlumeJava: Easy, Efficient Data-Parallel Pipelines
- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing
- Ray: A Distributed Framework for Emerging AI Applications