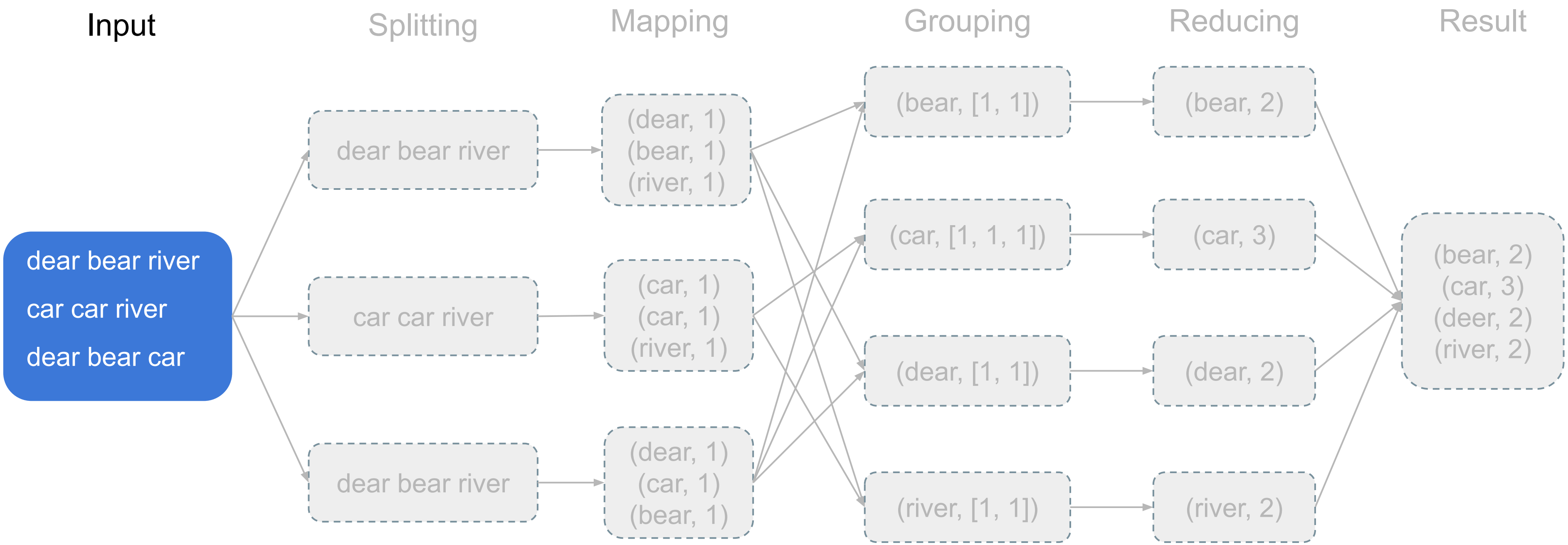
MapReduce
Simplified Data Processing on Large Clusters

Presented by: Wentao Yuan, Prashant Rangarajan

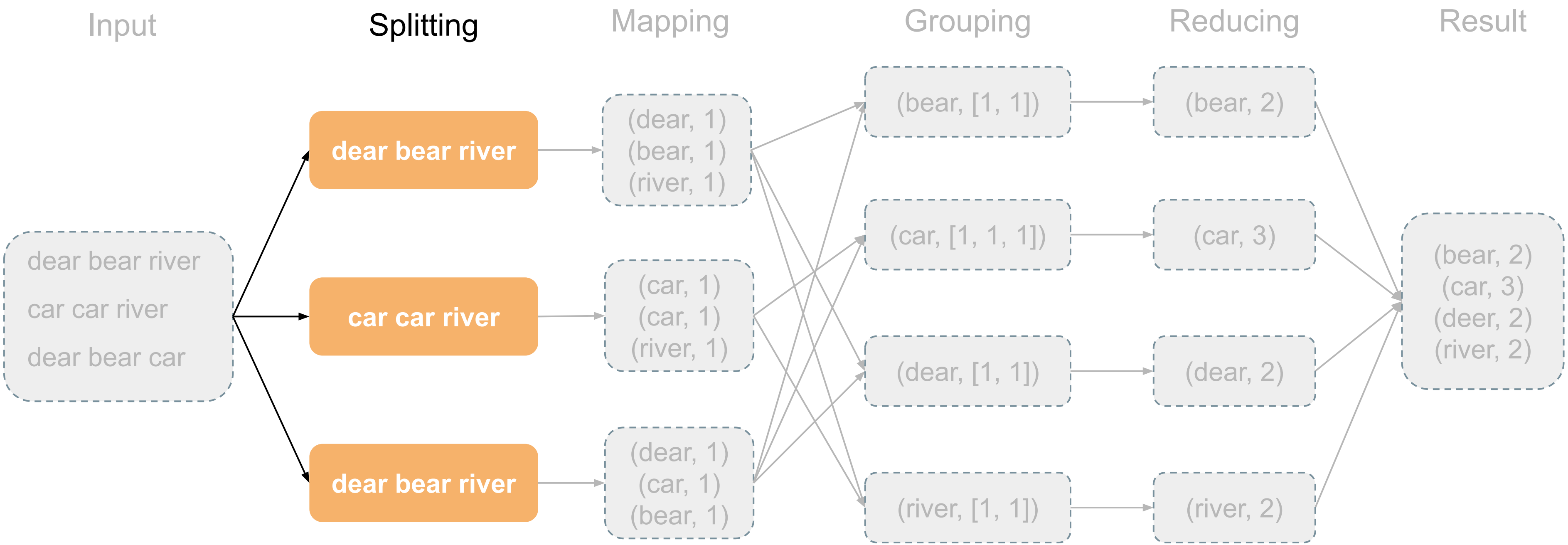
MapReduce

- Interface inspired by functional language (e.g. Lisp)
- Map
  - processes a key/value pair to generate a set of intermediate key/value pairs
- Reduce
  - merges all intermediate values associated with the same intermediate key
- Library handles the rest
  - parallelize computation
  - distribute data
  - handle failures
  - Balance load

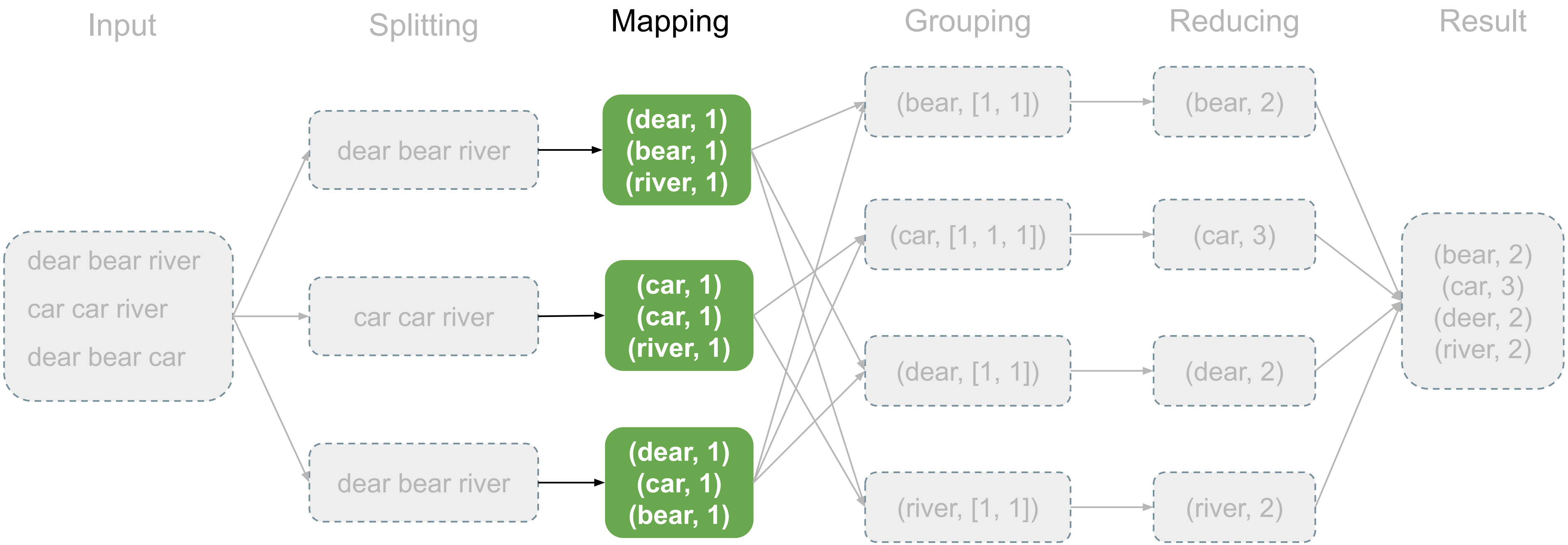
Example: Word Count



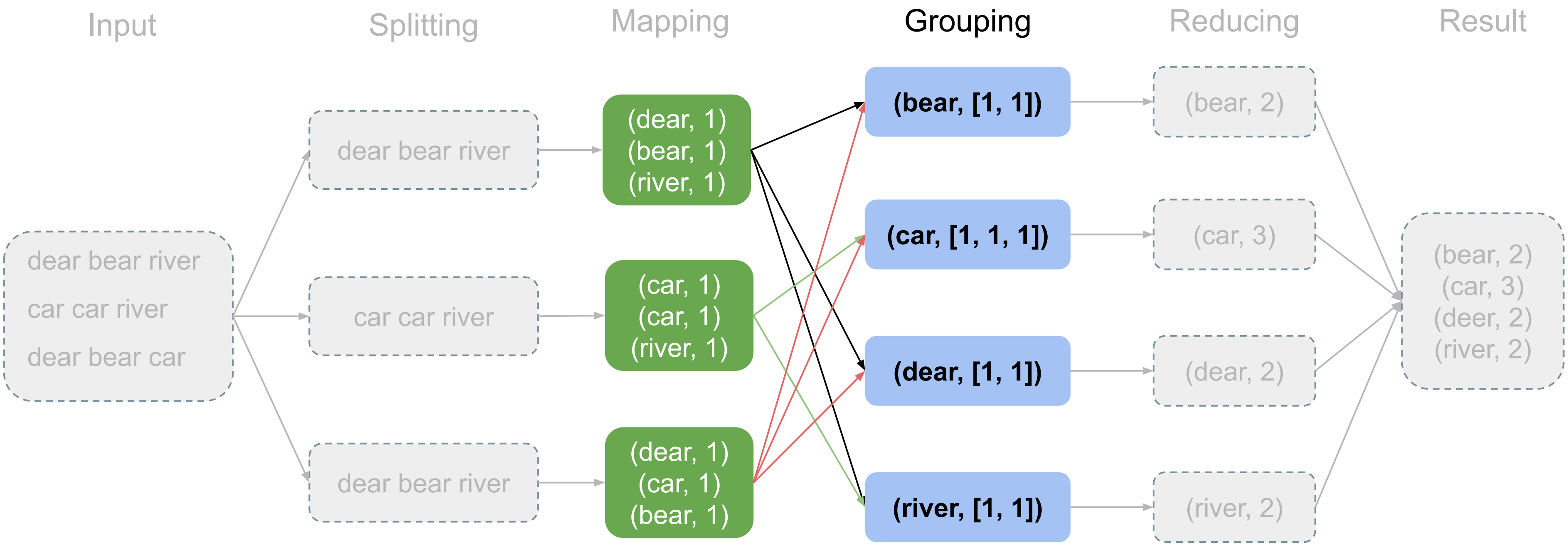
Example: Word Count



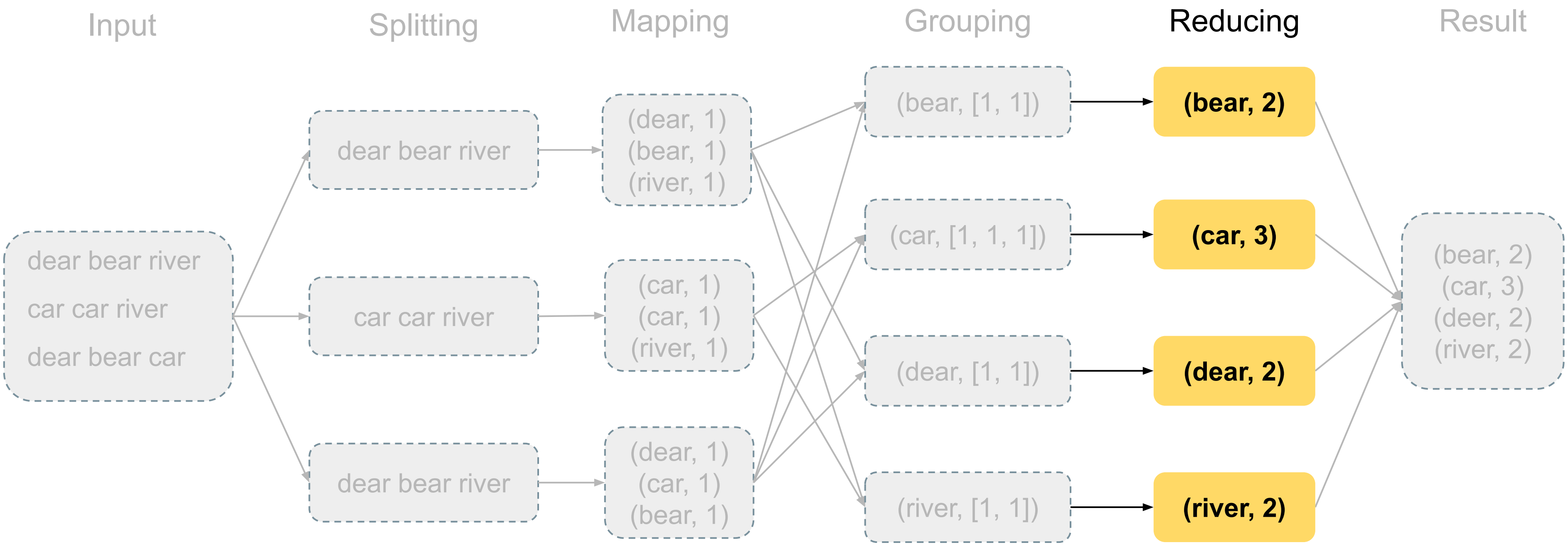
Example: Word Count



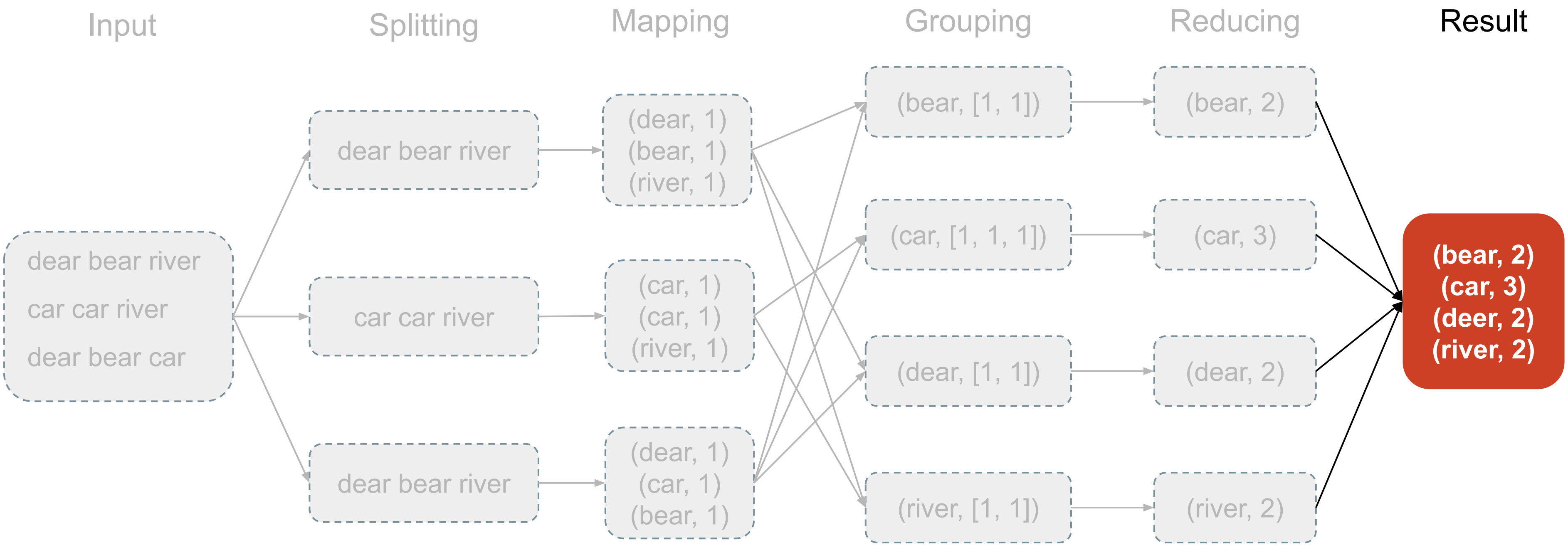
Example: Word Count



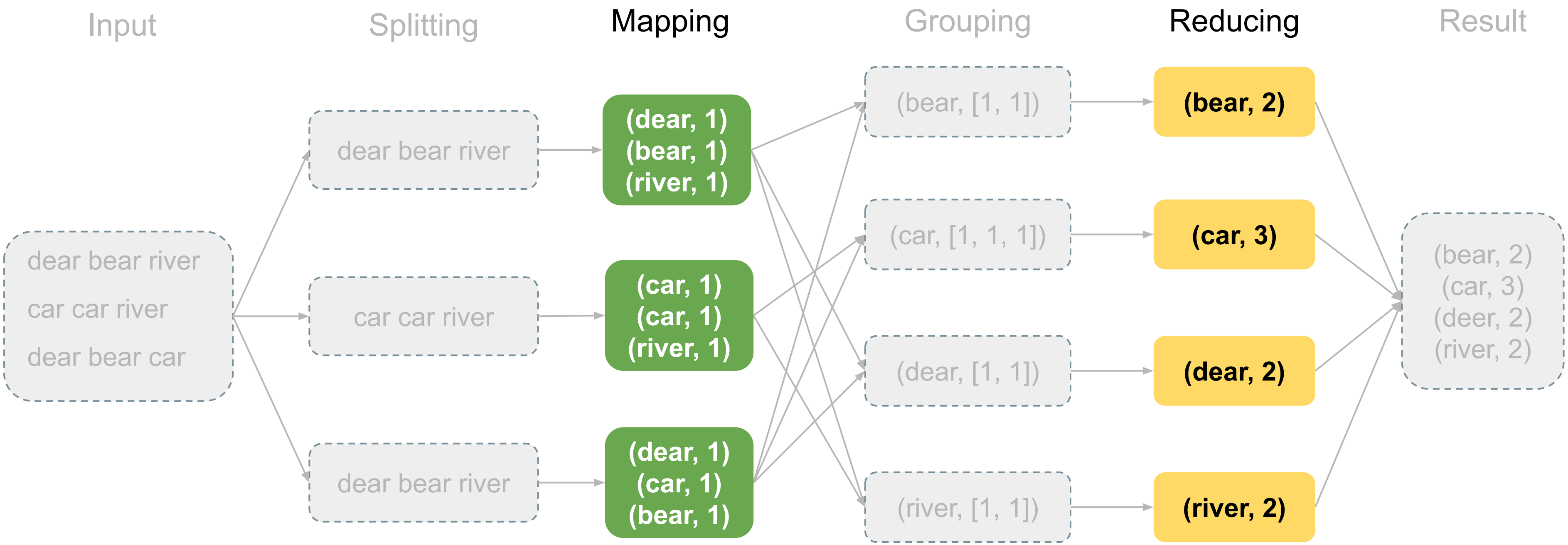
Example: Word Count



Example: Word Count

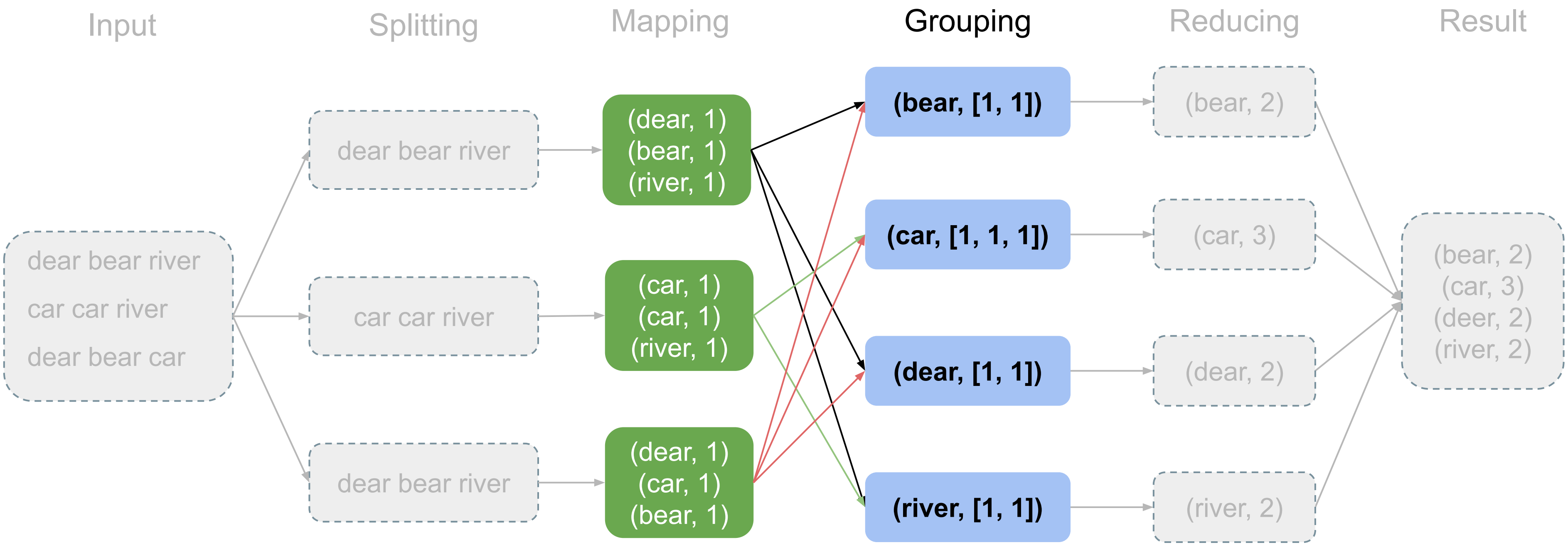


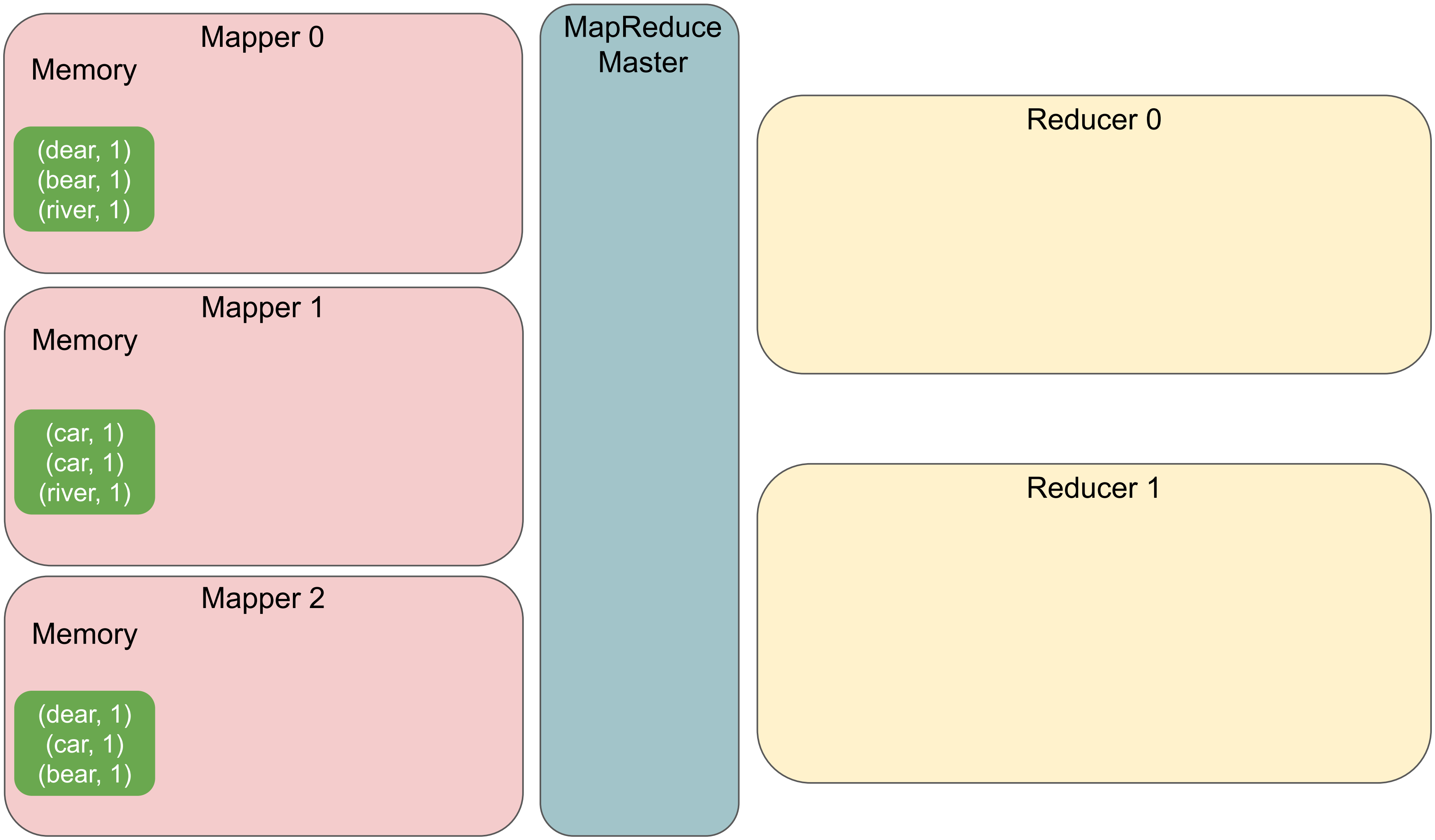
Example: Word Count

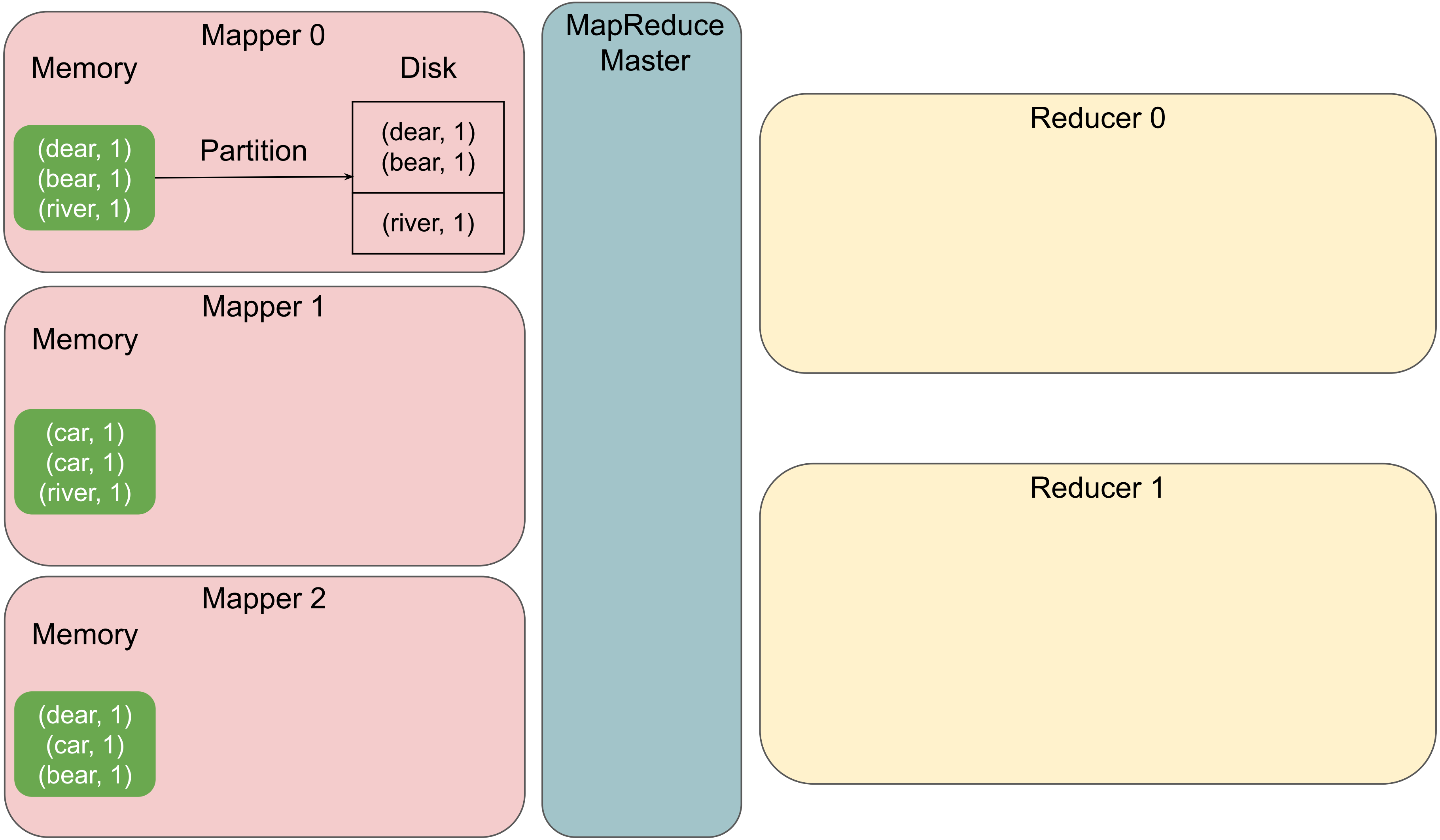


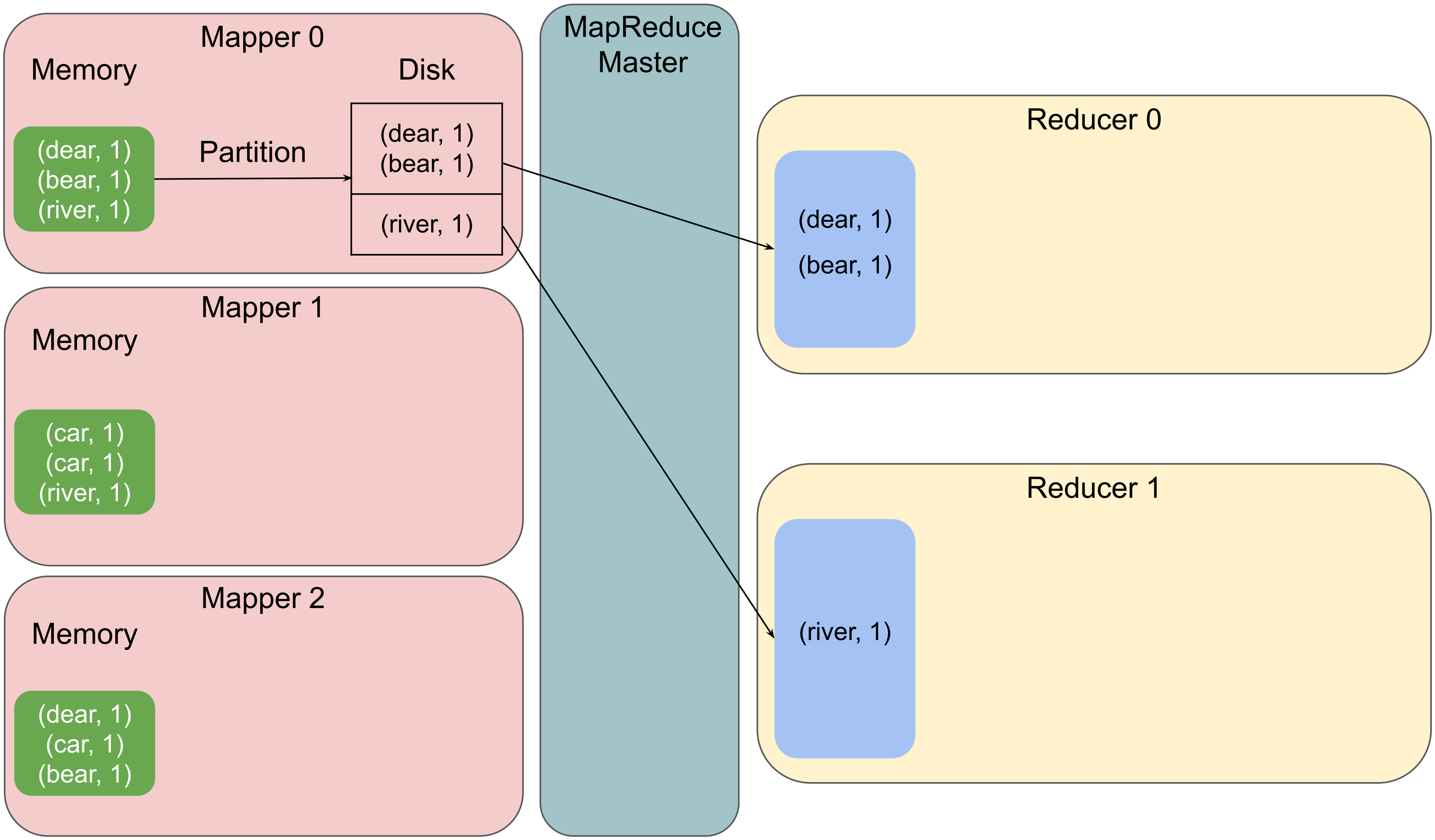
Executed in parallel without user's knowledge

Example: Word Count

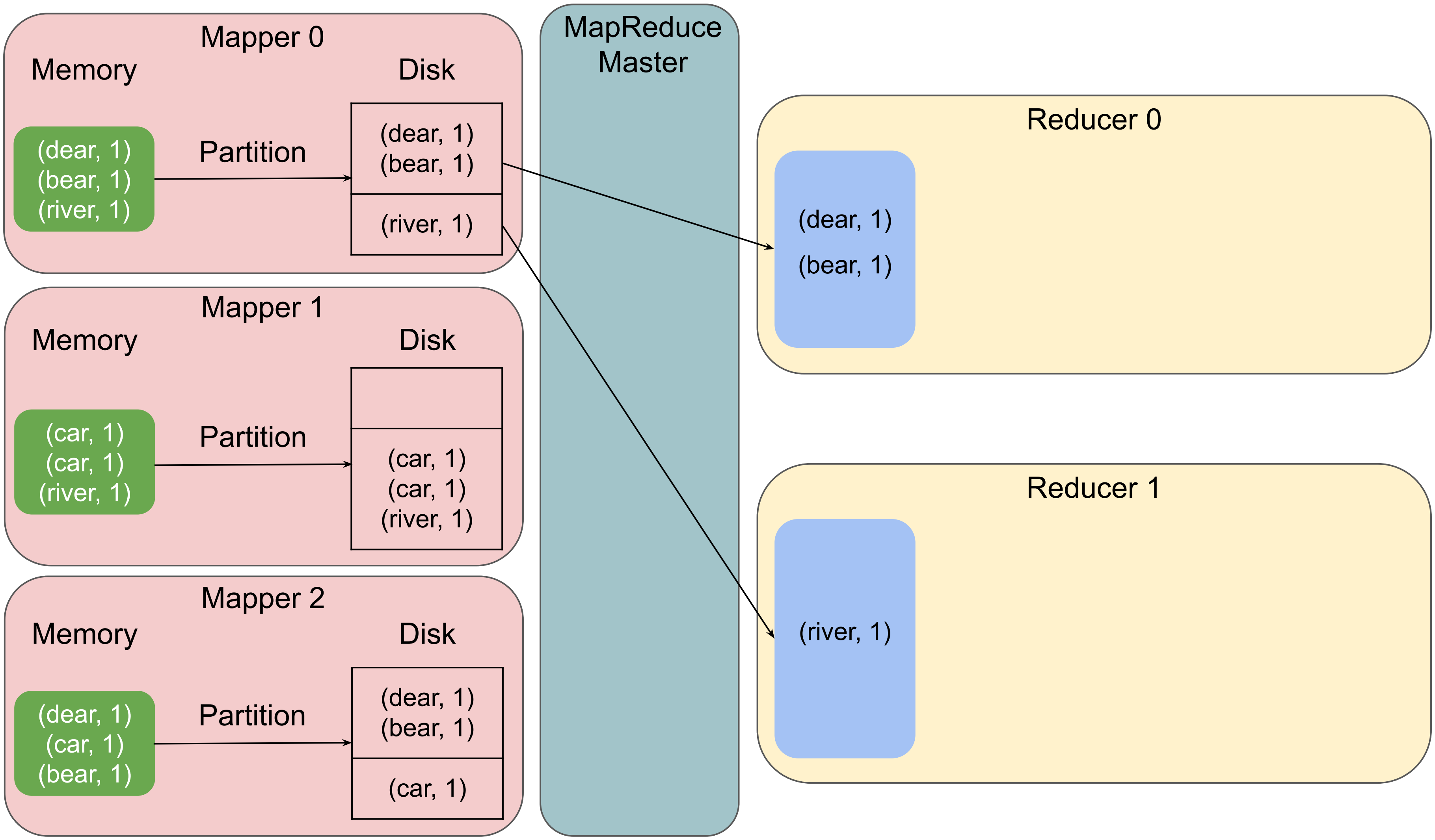


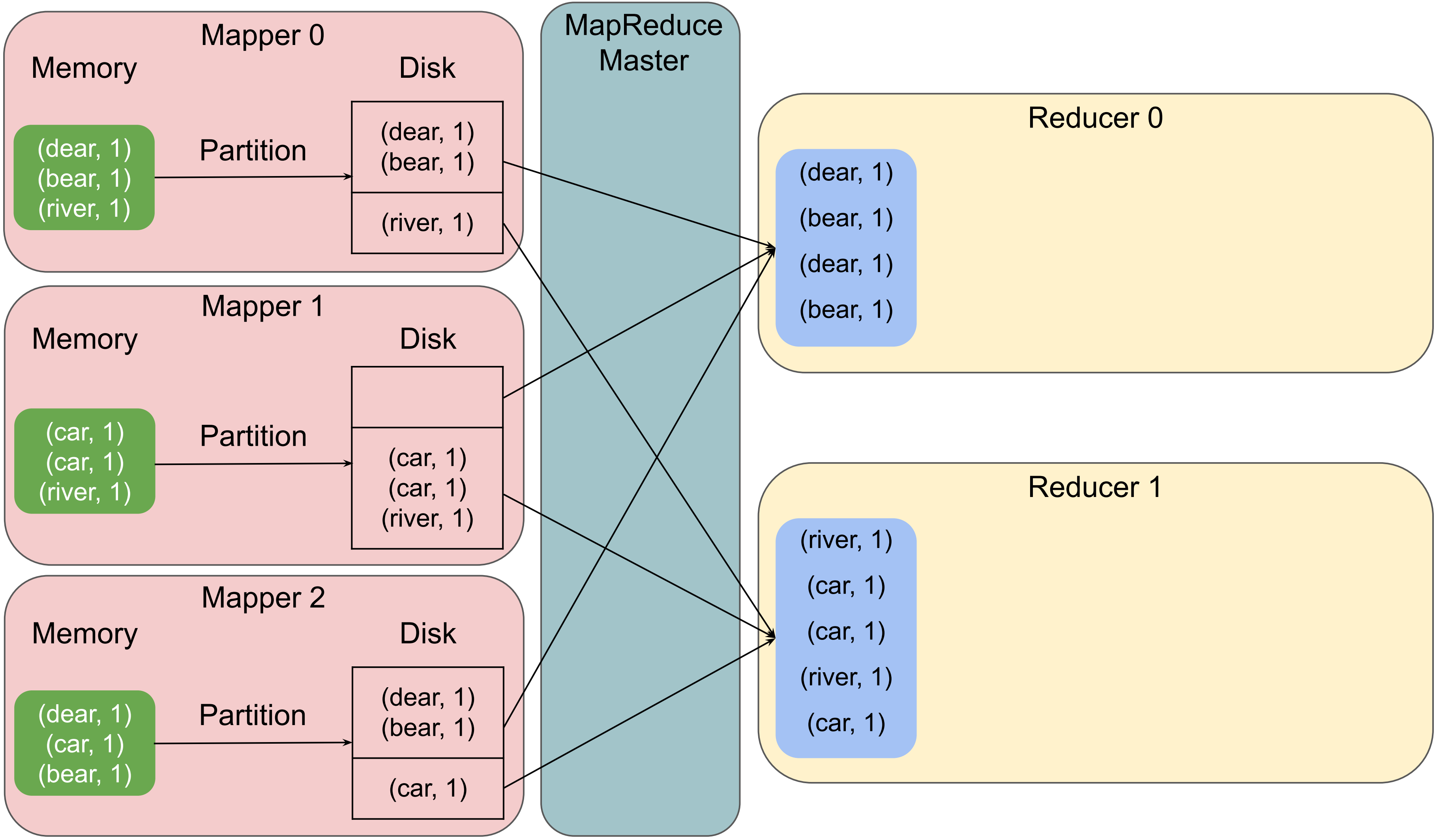

Mapper 0
Memory
(dear, 1)
(bear, 1)
(river, 1)
Mapper 1
Memory
(car, 1)
(car, 1)
(river, 1)
Mapper 2
Memory
(dear, 1)
(car, 1)
(bear, 1)
MapReduce Master
Reducer 0
Reducer 1

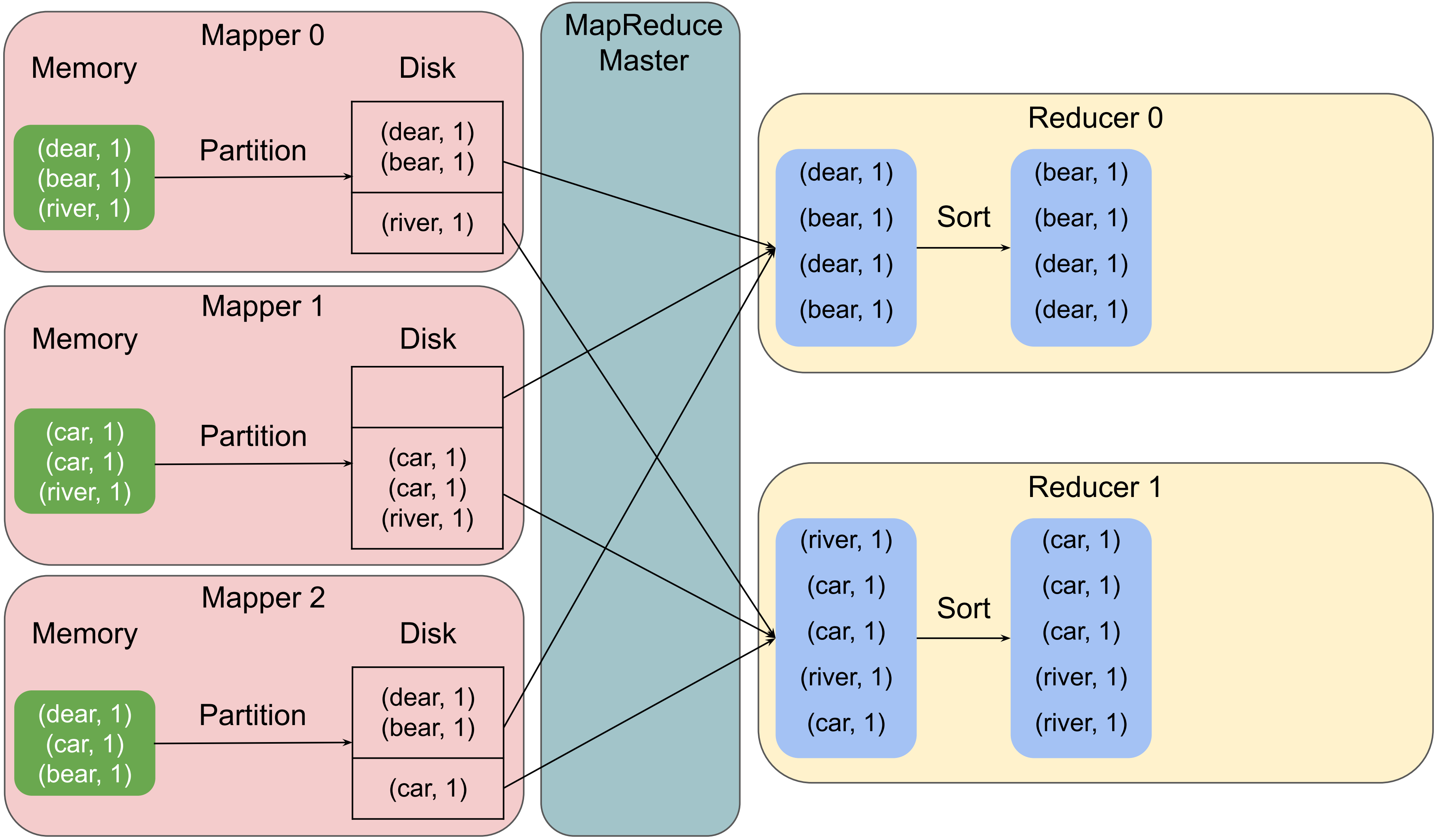

Mapper 0
Memory
Disk
(dear, 1)
(bear, 1)
(river, 1)
Partition
(dear, 1)
(bear, 1)
(river, 1)
Mapper 1
Memory
(car, 1)
(car, 1)
(river, 1)
Mapper 2
Memory
(dear, 1)
(car, 1)
(bear, 1)
MapReduce
Master
Reducer 0
Reducer 1

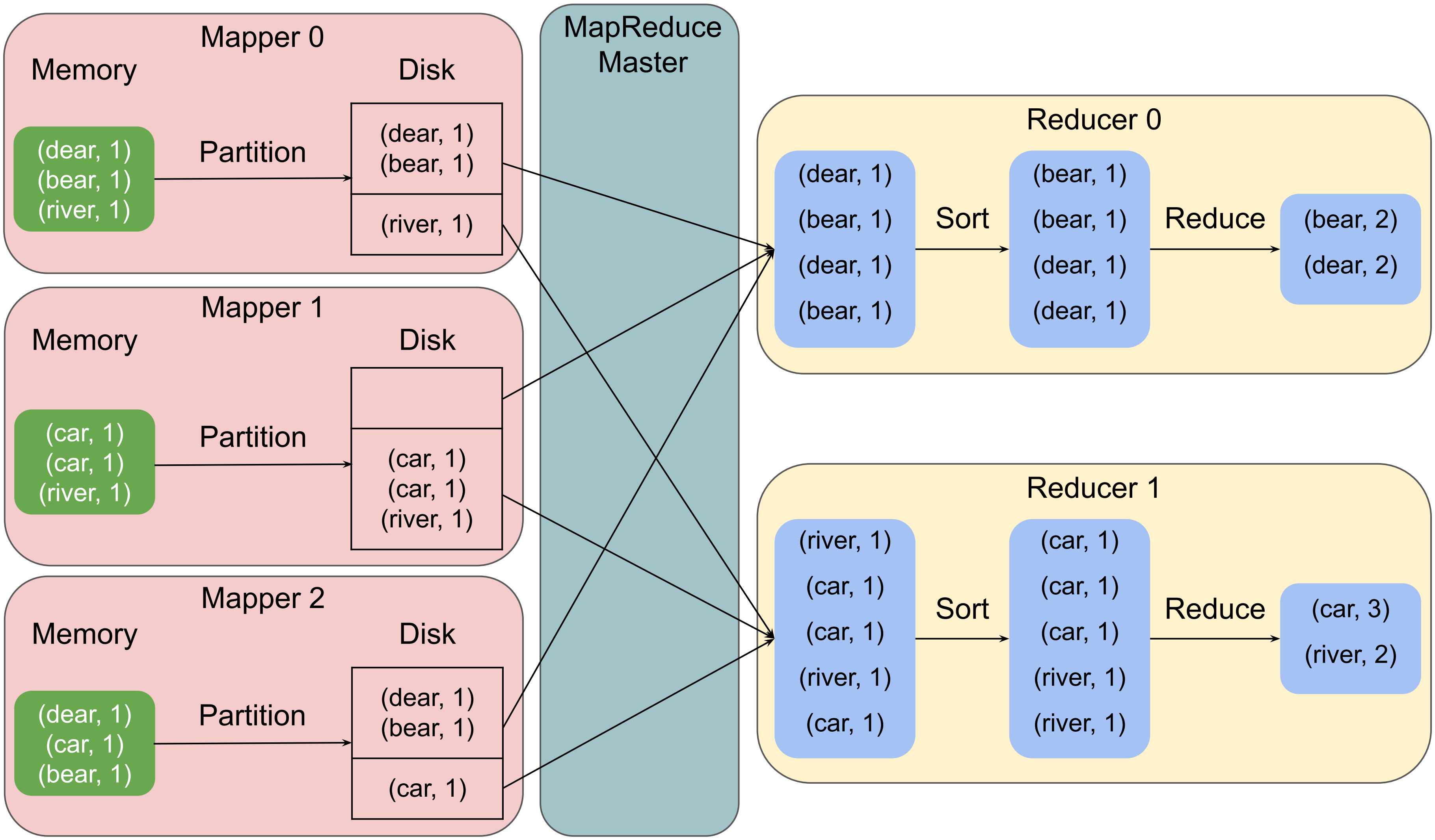


Mapper 0
Memory
Disk
(dear, 1)
(bear, 1)
(river, 1)
Partition
(dear, 1)
(bear, 1)
(river, 1)
MapReduce
Master
Reducer 0
(dear, 1)
(bear, 1)
Mapper 1
Memory
(car, 1)
(car, 1)
(river, 1)
Reducer 1
(river, 1)
Mapper 2
Memory
(dear, 1)
(car, 1)
(bear, 1)


Mapper 0
Memory
Disk
(dear, 1)
(bear, 1)
(river, 1)
Partition
(dear, 1)
(bear, 1)
(river, 1)
Mapper 1
Memory
Disk
(car, 1)
(car, 1)
(river, 1)
Partition
(car, 1)
(car, 1)
(river, 1)
Mapper 2
Memory
Disk
(dear, 1)
(car, 1)
(bear, 1)
Partition
(dear, 1)
(bear, 1)
(car, 1)
MapReduce
Master
Reducer 0
(dear, 1)
(bear, 1)
Reducer 1
(river, 1)


Mapper 0
Memory
Disk
(dear, 1)
(bear, 1)
(river, 1)
Partition
(dear, 1)
(bear, 1)
(river, 1)
Mapper 1
Memory
Disk
(car, 1)
(car, 1)
(river, 1)
Partition
(car, 1)
(car, 1)
(river, 1)
Mapper 2
Memory
Disk
(dear, 1)
(car, 1)
(bear, 1)
Partition
(dear, 1)
(bear, 1)
(car, 1)
MapReduce
Master
Reducer 0
(dear, 1)
(bear, 1)
(dear, 1)
(bear, 1)
Reducer 1
(river, 1)
(car, 1)
(car, 1)
(river, 1)
(car, 1)


Mapper 0
Memory
Disk
(dear, 1)
(bear, 1)
(river, 1)
Partition
(dear, 1)
(bear, 1)
(river, 1)
Mapper 1
Memory
Disk
(car, 1)
(car, 1)
(river, 1)
Partition
(car, 1)
(car, 1)
(river, 1)
Mapper 2
Memory
Disk
(dear, 1)
(car, 1)
(bear, 1)
Partition
(dear, 1)
(bear, 1)
(car, 1)
MapReduce
Master
Reducer 0
(dear, 1)
(bear, 1)
(dear, 1)
(bear, 1)
Sort
(bear, 1)
(bear, 1)
(dear, 1)
(dear, 1)
Reducer 1
(river, 1)
(car, 1)
(car, 1)
(river, 1)
(car, 1)
Sort
(car, 1)
(car, 1)
(car, 1)
(river, 1)
(river, 1)



Mapper 0
Memory
Disk
(dear, 1)
(bear, 1)
(river, 1)
Partition
(dear, 1)
(bear, 1)
(river, 1)
Mapper 1
Memory
Disk
(car, 1)
(car, 1)
(river, 1)
Partition
(car, 1)
(car, 1)
(river, 1)
Mapper 2
Memory
Disk
(dear, 1)
(car, 1)
(bear, 1)
Partition
(dear, 1)
(bear, 1)
(car, 1)
MapReduce
Master
Reducer 0
(dear, 1)
(bear, 1)
(dear, 1)
(bear, 1)
Sort
(bear, 1)
(bear, 1)
(dear, 1)
(dear, 1)
Reduce
(bear, 2)
(dear, 2)
Reducer 1
(river, 1)
(car, 1)
(car, 1)
(river, 1)
(car, 1)
Sort
(car, 1)
(car, 1)
(car, 1)
(river, 1)
(river, 1)
Reduce
(car, 3)
(river, 2)

Handling Failures

- Master pings workers regularly, no response → failure
- If a mapper fails
  - All map tasks (completed or in progress) are rescheduled
  - Output of completed map tasks are written to local disk, access is lost if mapper fails
- If a reducer fails
  - Only in progress reduce tasks are rescheduled
  - Output of completed reduce tasks are written to global file system (has replication), access is not lost if reducer fails

What MapReduce is Good for

- Operations that access data sequentially
- Offline batch jobs
- Examples
    - Distributed Grep
    - Count of URL Access Frequency
    - Reverse Web-Link Graph
    - Inverted Index

What MapReduce is NOT Good for

- Operations that requires random data access
- Interactive, real-time applications
- Examples
  - Graphs (e.g. social network)
  - Monitoring consoles (e.g. Bloomberg)

Parallel Databases

- Gamma
  - horizontally partitioned relations → parallel scanning
  - Hashing-based parallel join/aggregate operations
  - Dataflow scheduling
- C-Store
  - Read-optimized database stored in column-oriented projections
  - Optimized for ad-hoc reads
- Both system evaluated on single multi-processor machines
  - How well do they tolerate node failure, network failure/congestion in data centers?

MapReduce: A Step Backwards?

Critique of MapReduce from a DBMS perspective (DeWitt and Stonebraker):

- Doesn’t use schemas, no separation of the schema from the application program
- Poor implementation - Lack of indices, skew
- Not a very novel concept
- Lack of modern features of a DBMS like views
- Incompatible with DBMS tools eg database design tools

Discussion

- What are the most powerful aspects of the MapReduce framework that have made it so popular today? What is the biggest disadvantage of the MapReduce model?
- Are there any optimizations you can make to reduce resources (energy, memory, compute, communication etc) used by MapReduce. Does your proposal introduce another complexity?
- What parts of the DeWitt and Stonebreaker's response critiquing MapReduce do you agree/disagree with?
- Do you see MapReduce being replaced by Parallel Databases in the future, or are they here to stay?

    - Discussion doc: https://tinyurl.com/cz8pbw5e

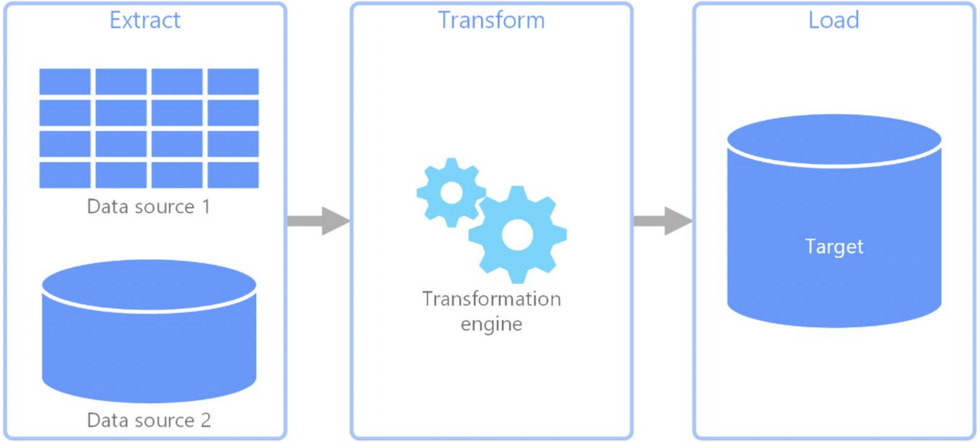
MapReduce	VS Parallel Database
- Can deal with Heterogeneous systems well - Semi-structured data - Easy to deploy and set up - Free, open source projects - Great for complex analysis - Useful for ETL tasks	- Great for pipelining - Scheduling - Parsing more efficient - Compression - Higher level language

Parallel Databases and MR - can they coexist?

Parallel DBMSs excel at efficient querying of large data sets

MRstyle systems excel at complex analytics and ETL tasks.

The two technologies can be complementary



Higher Level Interfaces built on MR

Pig: (by Yahoo!)

- Pig Latin: Find a balance between low level procedural programming of MR and higher level programming of SQL -
- For experienced programmers - perform ad-hoc analysis of extremely large data sets

Hive: (by Facebook)

- HiveQL language: SQL-like declarative language on top of Hadoop
- Tables, partitions, buckets, some primitive column types

Resilient Distributed Datasets

A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Zaharia et al, UC Berkeley

Resilient Distributed Datasets (RDDs)

Main applications:

- Iterative Algorithms
- Interactive Data Mining

Leverage distributed memory effectively, efficient fault tolerance.

MapReduce:

Has to write to external storage system e.g distributed file system

Other specific frameworks eg Pregel, HaLoop - only work for specific computation patterns

RDD Abstraction

RDD -

- Read only/Immutable, spread across cluster
- Can control persistence and partitioning
- Caching dataset in memory
- Not materialized all the time, coarse grained operations

Two types of operations:

- Transformations - deterministic operations that define a new RDD, lazy evaluation
- Actions - return a value to the program or write data to external storage

Transformations	$map(f: \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$filter(f: \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$flatMap(f: \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$sample(fraction: \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
	$groupByKey()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
	$reduceByKey(f: (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$union()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]}$
	$join()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]}$
	$cogroup()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
	$crossProduct()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]}$
	$mapValues(f: \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
	$sort(c: \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$partitionBy(p: \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
Actions	$count()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Long}$
	$collect()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Seq[T]}$
	$reduce(f: (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{T}$
	$lookup(k: \mathrm{K})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{Seq[V]}$ (On hash/range partitioned RDDs)
	$save(path: \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.,* HDFS

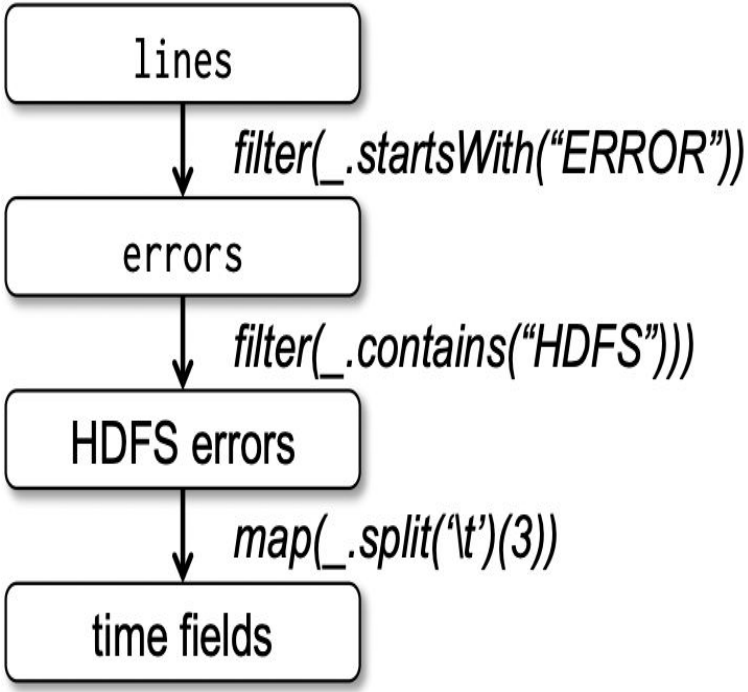
Fault Tolerance - Lineage

Console Log Mining

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

```
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()

// Return the time fields of errors mentioning
// HDFS as an array (assuming time is field
// number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```



RDDs vs DSM

Aspect	RDDs	Distr. Shared Mem.
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Table 1: Comparison of RDDs with distributed shared memory.

- Works well for applications which batch transform data
- Not so great for more fine grained applications.
- More memory intensive

Spark

Programming interface for RDDs based on Scala

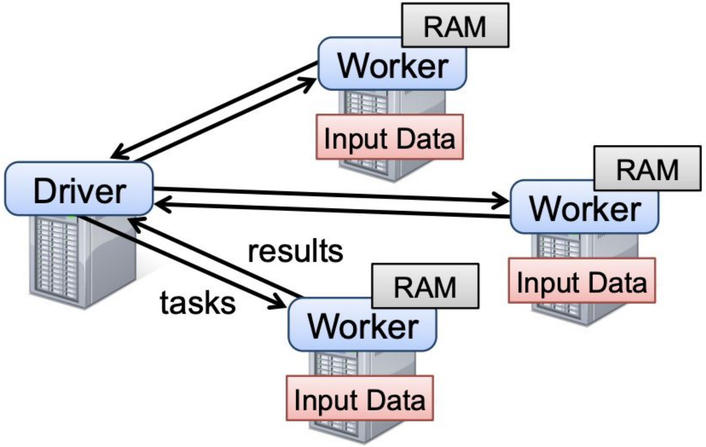
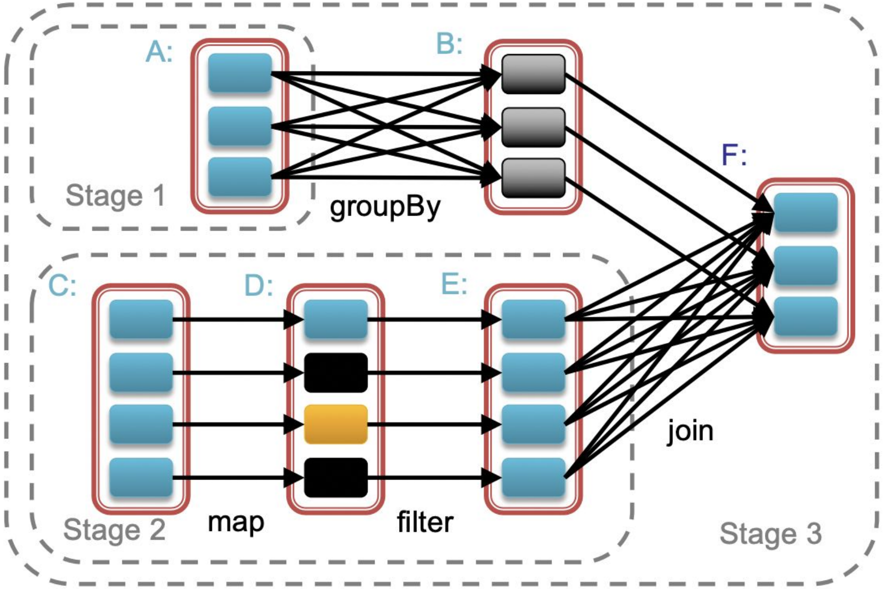
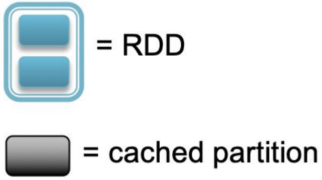


Figure 2: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

Task Scheduler: General DAGs



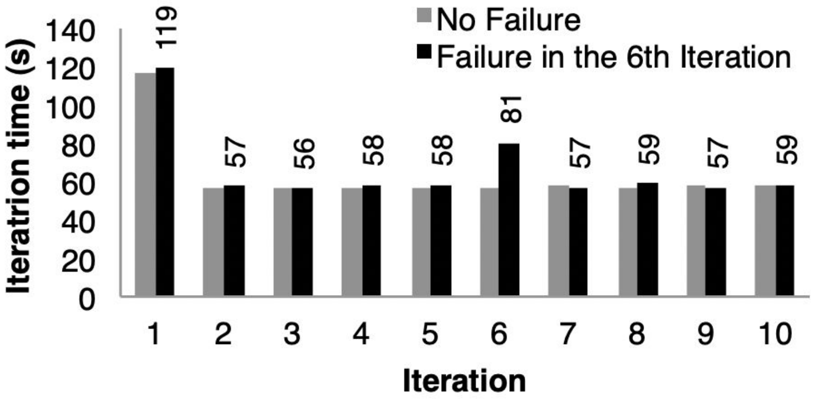
- Supports general task graphs
- Pipelines functions where possible
- Cache-aware data reuse & locality
- Partitioning-aware to avoid shuffles



Evaluation

Speed Up: Spark outperforms Hadoop by up to 20× in iterative machine learning and graph applications

Fault Recovery: Iteration times for k-means in presence of a failure



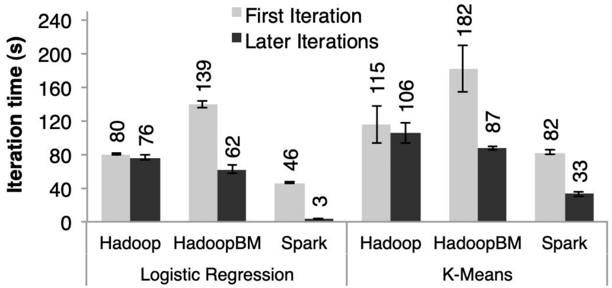
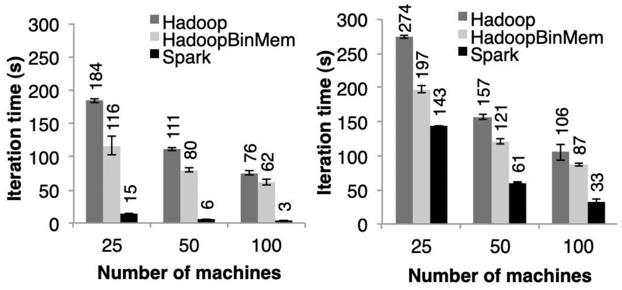


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.



(a) Logistic Regression (b) K-Means

User Applications Built with Spark

- In-Memory Analytics
- Traffic Modeling
- Twitter Spam Classification

Highly expressive - can do operations of different frameworks:

MapReduce, SQL, Pregel etc.