Distributed Computing

A-15. Apache Spark

References

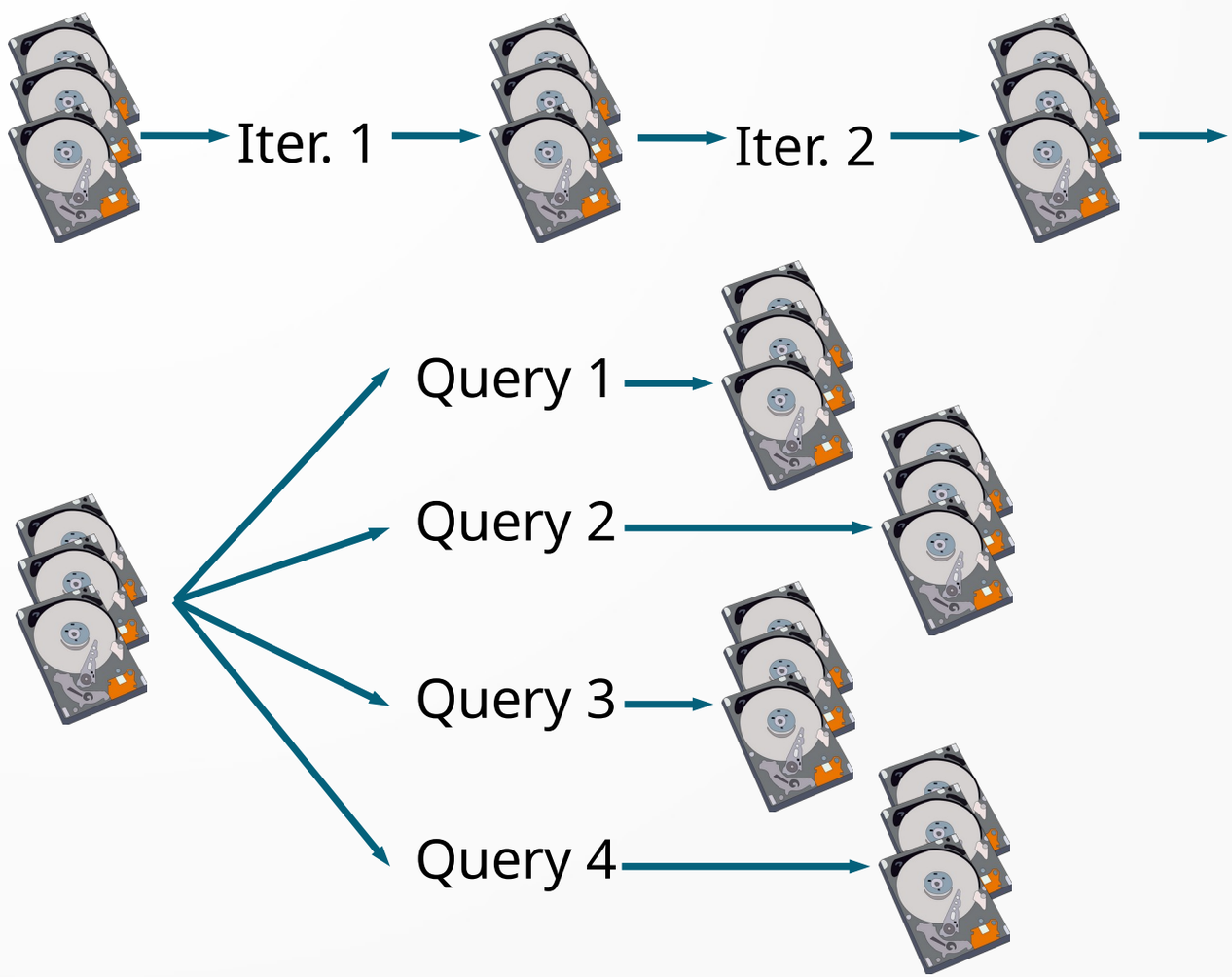
- The paper and presentation by Zaharia et al. At USENIX NSDI 2012: *"Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing"*
- The Spark website: https://spark.apache.org

MapReduce: Virtues & Shortcomings

- MapReduce has been a big improvement for “big data” on large clusters of unreliable machines
- However, it’s less than perfect for important use cases
  - Multi-stage applications (e.g., iterative machine learning, graph processing)
  - Interactive ad-hoc queries
- Several specialized frameworks were designed to handle these cases

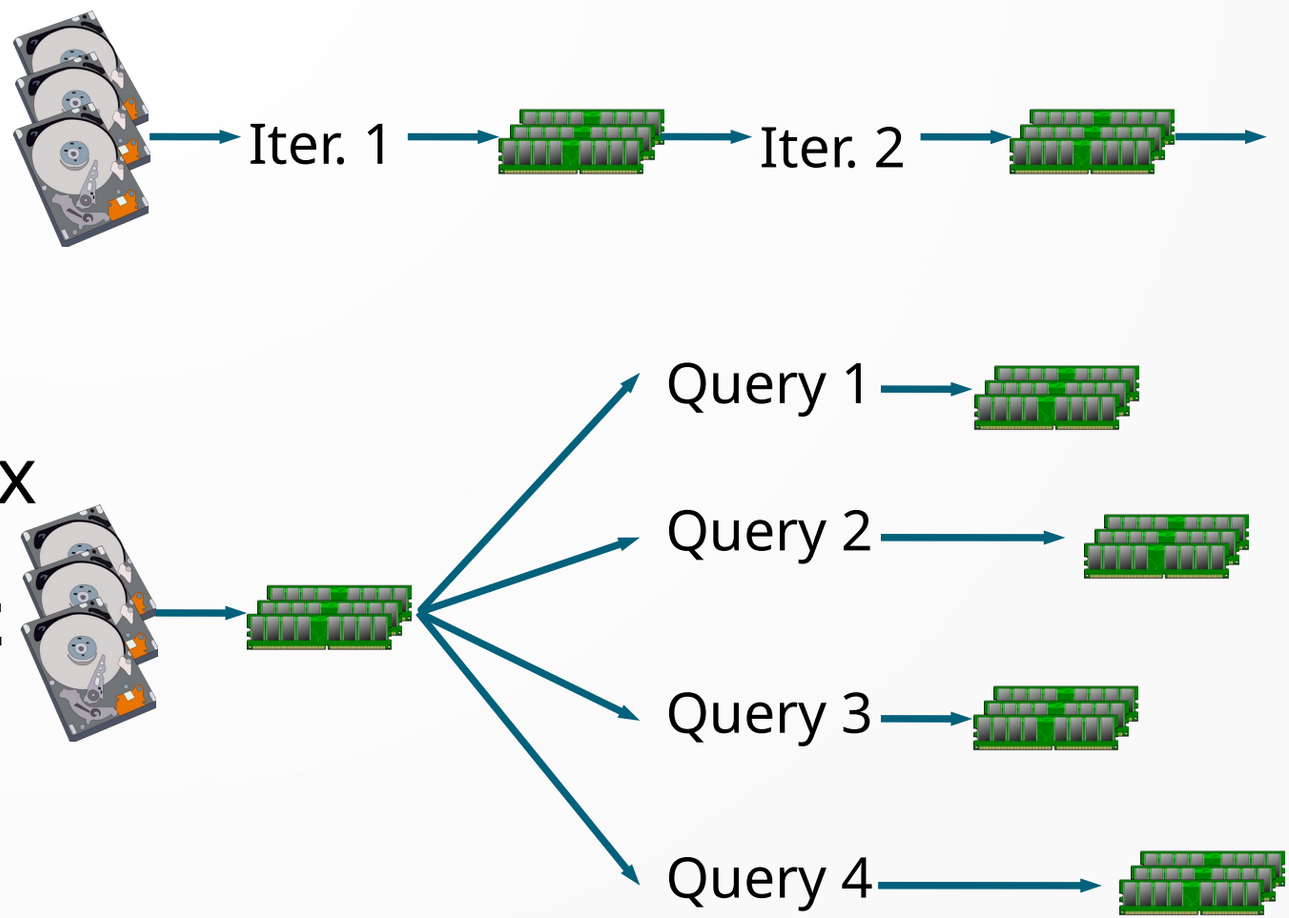
Everything On Disk

- Each MapReduce job (i.e., something that has "one shuffle phase") has to **read and write from disk**
- This is the solution to deal with unreliability: write everything on disk and replicate it
- However, this is **slow**



In-Memory Processing

- What if things could be kept in RAM without the need of touching the disk every time?
- Huge speedup—10 to 100x
- The goal of Spark is to **get both this and fault tolerance**



Resilient Distributed Datasets (RDDs)

Resilient Distributed Datasets

- **Immutable**, **distributed** in-memory data structures
  - **Partitioned** among the machines in the cluster
- Only built through **coarse-grained** operations that process the whole RDD
  - Based on functional programming (`map`/`filter`/`join`…)
- Fault recovery performed using **lineage**
  - A log of all the operations done to get there
  - Recover lost partitions by **recomputing what's missing**
  - No cost if nothing fails

RDDs vs. Databases

- Databases & key-value stores handle **small updates** and **store everything on disk**
  - They're good for small modifications (transactions) that don't modify most of the state
- RDDs are efficient for **large operations**

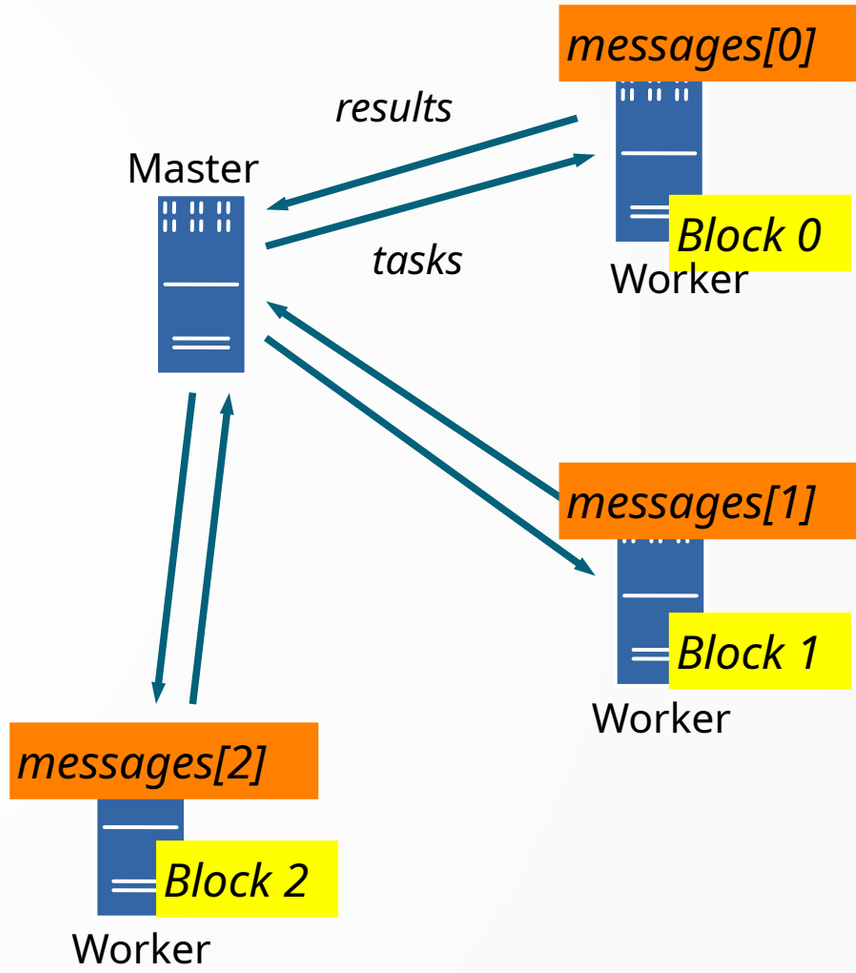
Example: Log Mining

```
lines = spark.textFile(“hdfs://…”)

def is_error(line):
    return line.startswith(‘ERROR’)
errors = lines.filter(is_error)

def get_message(line):
    return line.strip().split()[1]
messages = errors.map(get_message)
messages.persist()

messages.filter(lambda m: ‘foo’ in
m).count()
messages.filter(lambda m: ‘bar’ in
m).count()
```



Fault Recovery

- RDDs tracks **lineage** (i.e. dependencies) for each block

```
messages=textFile(...) \
      .filter(lambda x: 'error' in x) \
      .map(lambda x: x.split()[1])
```