CompSci 401: Cloud Computing

Spark

Prof. Ítalo Cunha

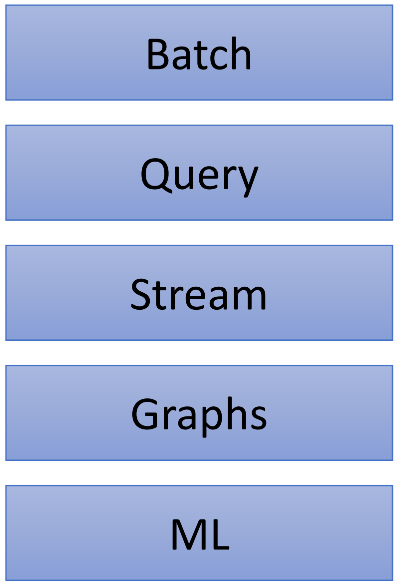
昆山杜克大学
DUKE KUNSHAN UNIVERSITY

Programming Paradigms for Big Data

- Different solutions for different types of Big Data analysis
  - Batch Processing
    - Large datasets, extract-transform-load (ETL) tooling, DataFrames
  - Interactive Queries (SQL and others)
    - IPython, Jupyter, DataFrames
  - Stream Processing
    - Network security monitoring, tweet processing, log mining
  - Graph Analysis
  - Machine Learning (many different sub-paradigms)
    - Spam detection, image processing, genome sequencing, model construction

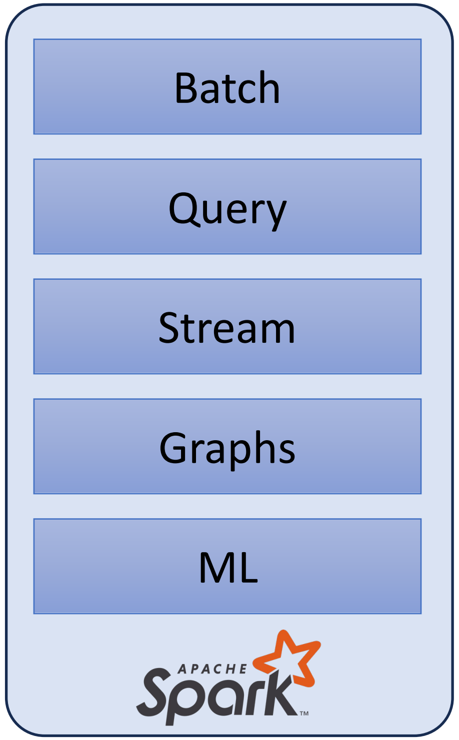
One specialized engine per task

- Pro: Specialized engines with high-performance
- Con: Hard to integrate across engines
  - Some applications cannot be expressed in a single engine



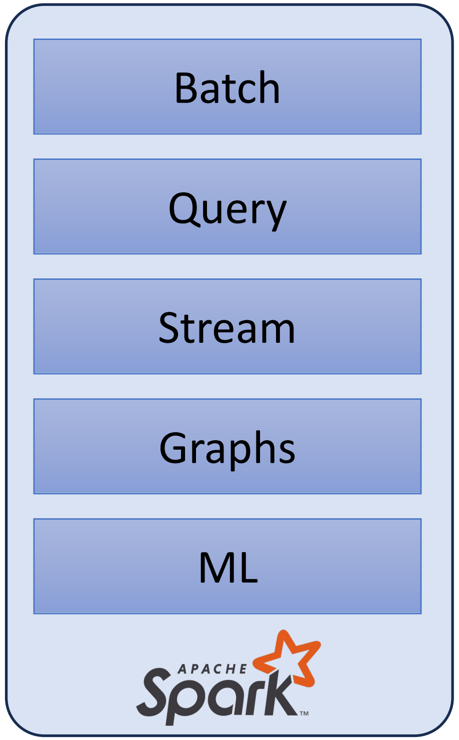
Spark: Integrate Multiple Paradigms

- Pro: Single engine with support for multiple paradigms
- Con: Possibly lower performance due to non-specialized engine



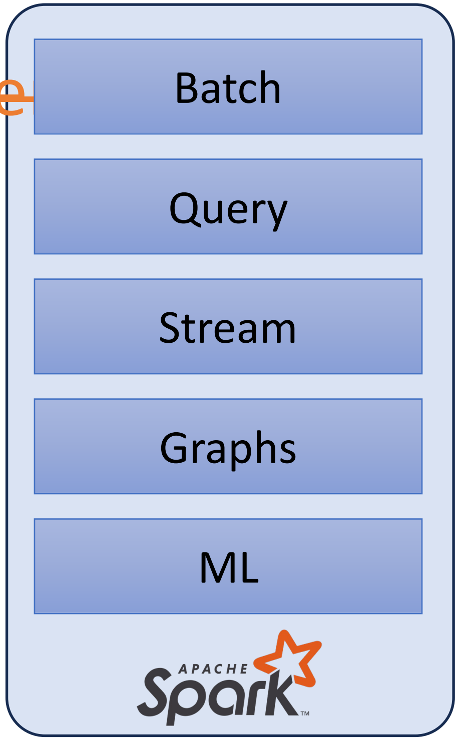
Spark: Integrate Multiple Paradigms

- Pro: Single engine with support for multiple paradigms
- ~~Con: Possibly lower performance due to non-specialized engine~~
  - Spark achieves high performance
  - Uses similar implementation to specialized engines



Spark: Integrate Multiple Paradigms

- Pro: Single engine with support for multiple paradigms
  - Resilient Distributed Datasets (RDDs) as building block
  - Much easier to integrate different Big Data solutions
    - For example, no saving intermediate results to disk
- ~~Con: Possibly lower performance due to non-specialized e~~
  - Spark achieves high performance
  - Uses similar implementation to specialized engines



A vantagem de combinar soluções

A vantagem de combinar soluções

waze

Resilient Distributed Datasets

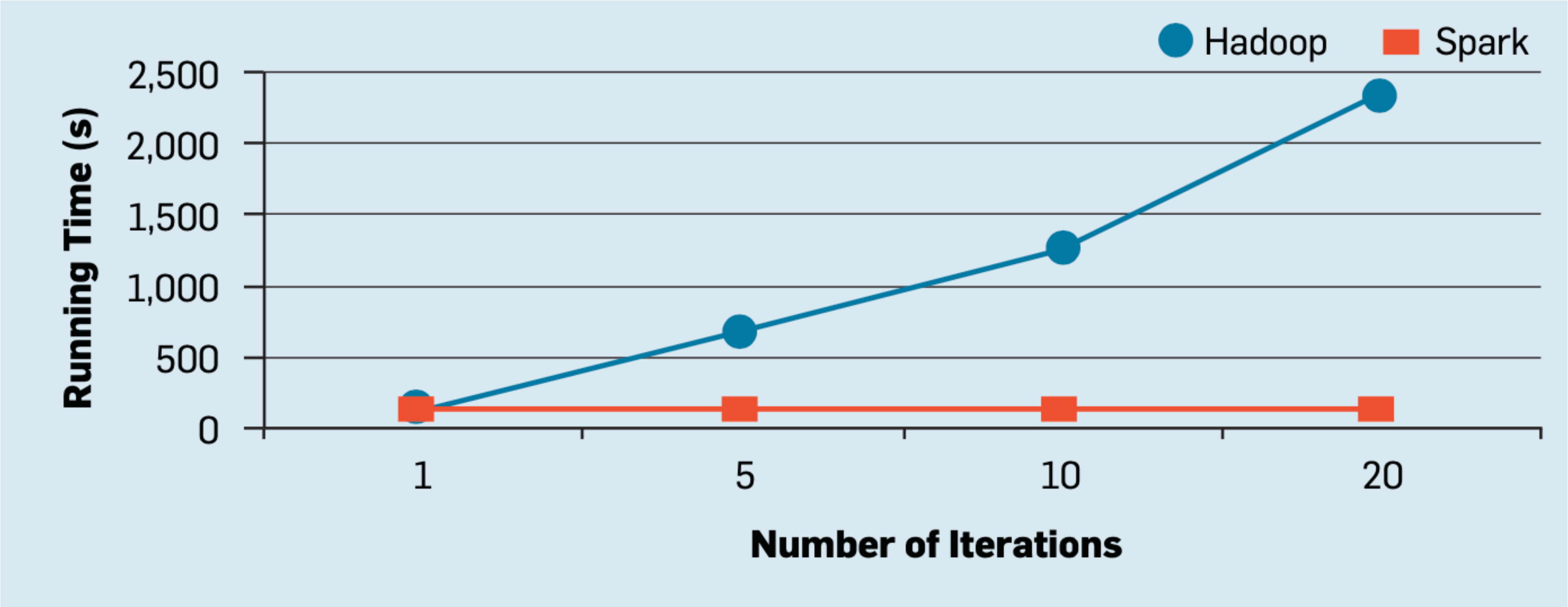
- High-performance
- General computation
- Lazy evaluation
- Ephemeral
- Lineage-based reconstruction
- Data sharing across different solutions
  - In-memory

Resilient Distributed Datasets

- High-performance
- General computation
- Lazy evaluation
- Ephemeral
- Lineage-based reconstruction
- Data sharing across different solutions

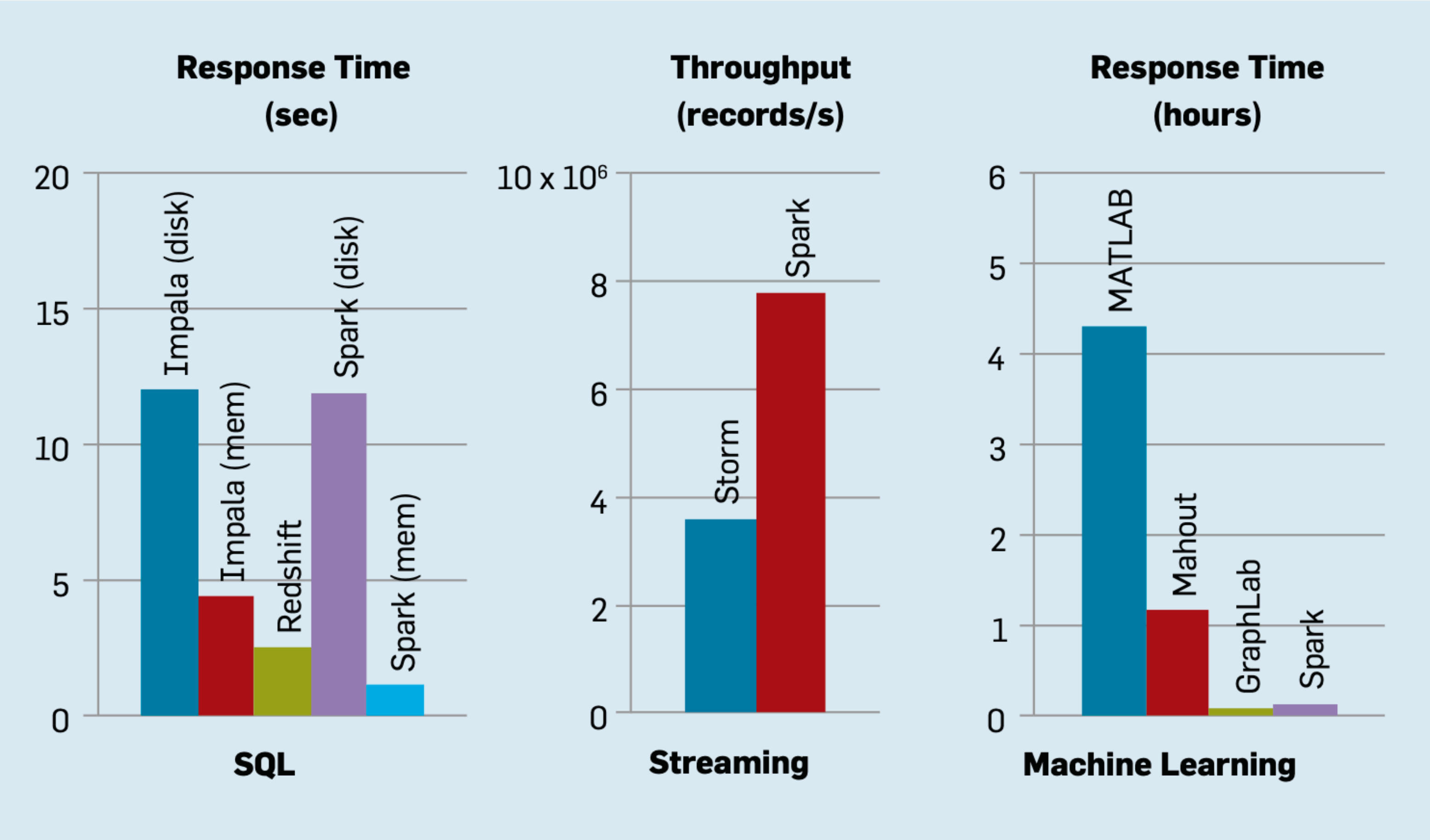
Spark Performance

- Spark beats Hadoop
  - Loads data only once instead of on each iteration



Spark Performance

- Spark is competitive with specialized systems

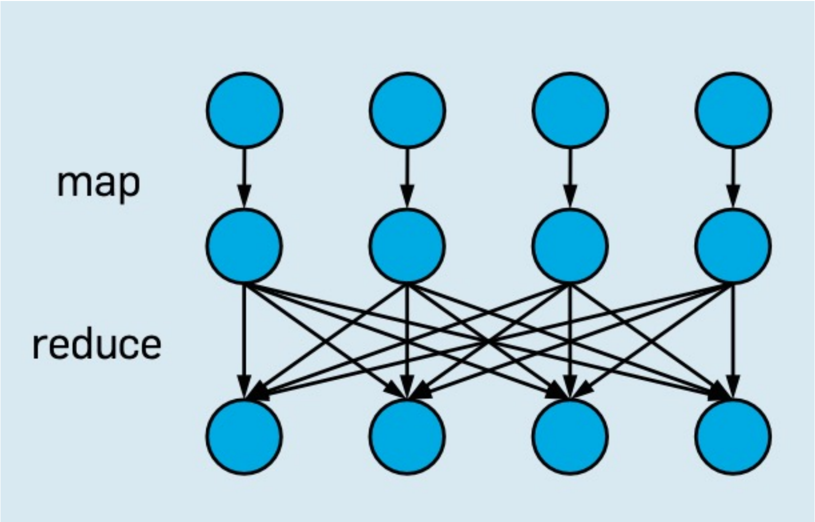


Resilient Distributed Datasets

- High-performance
- General computation
- Lazy evaluation
- Ephemeral
- Lineage-based reconstruction
- Data sharing across different solutions

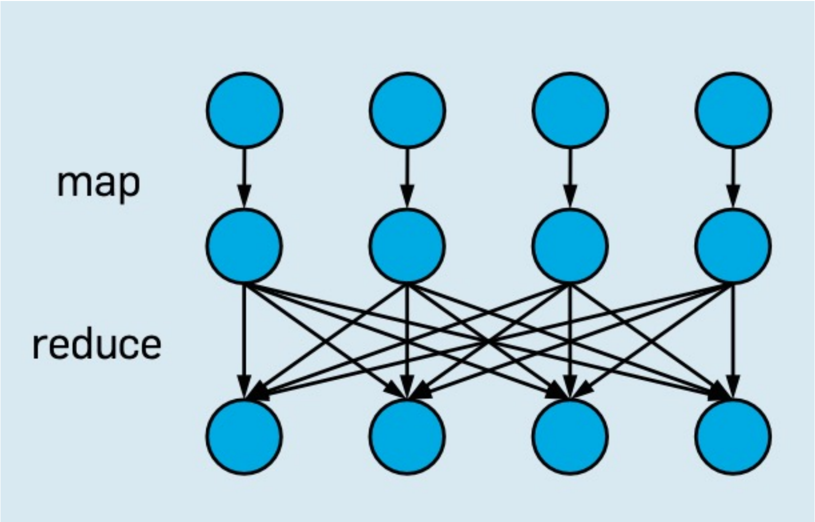
General Computation

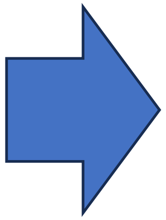
- MapReduce can express any computation

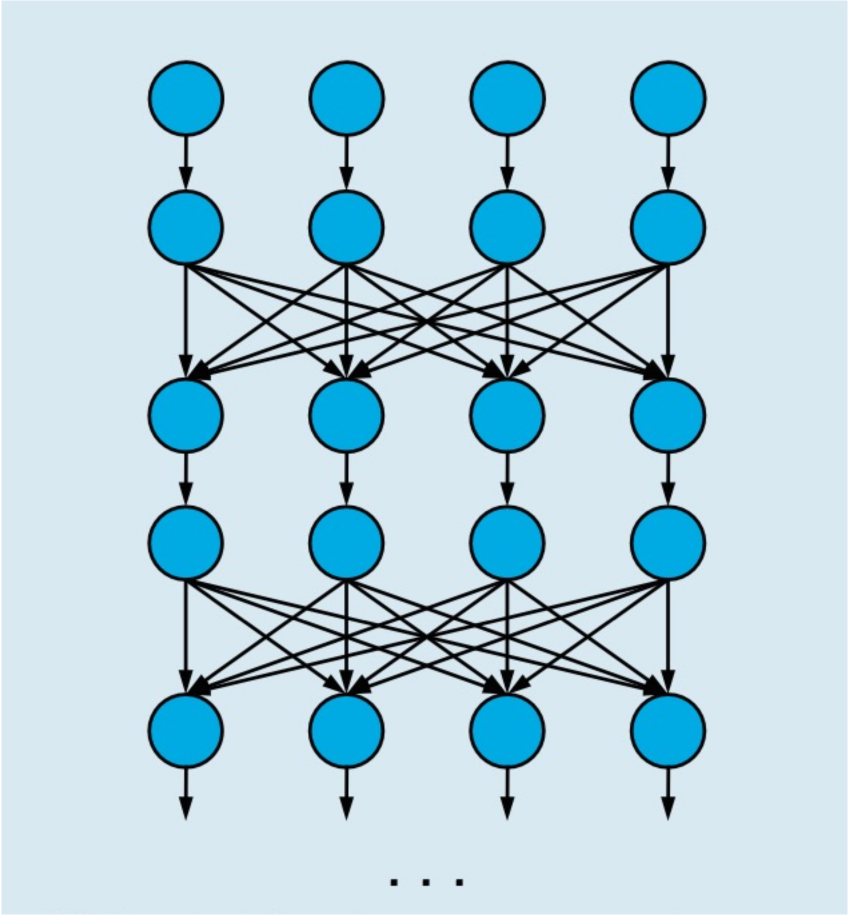


General Computation

- MapReduce can express any computation

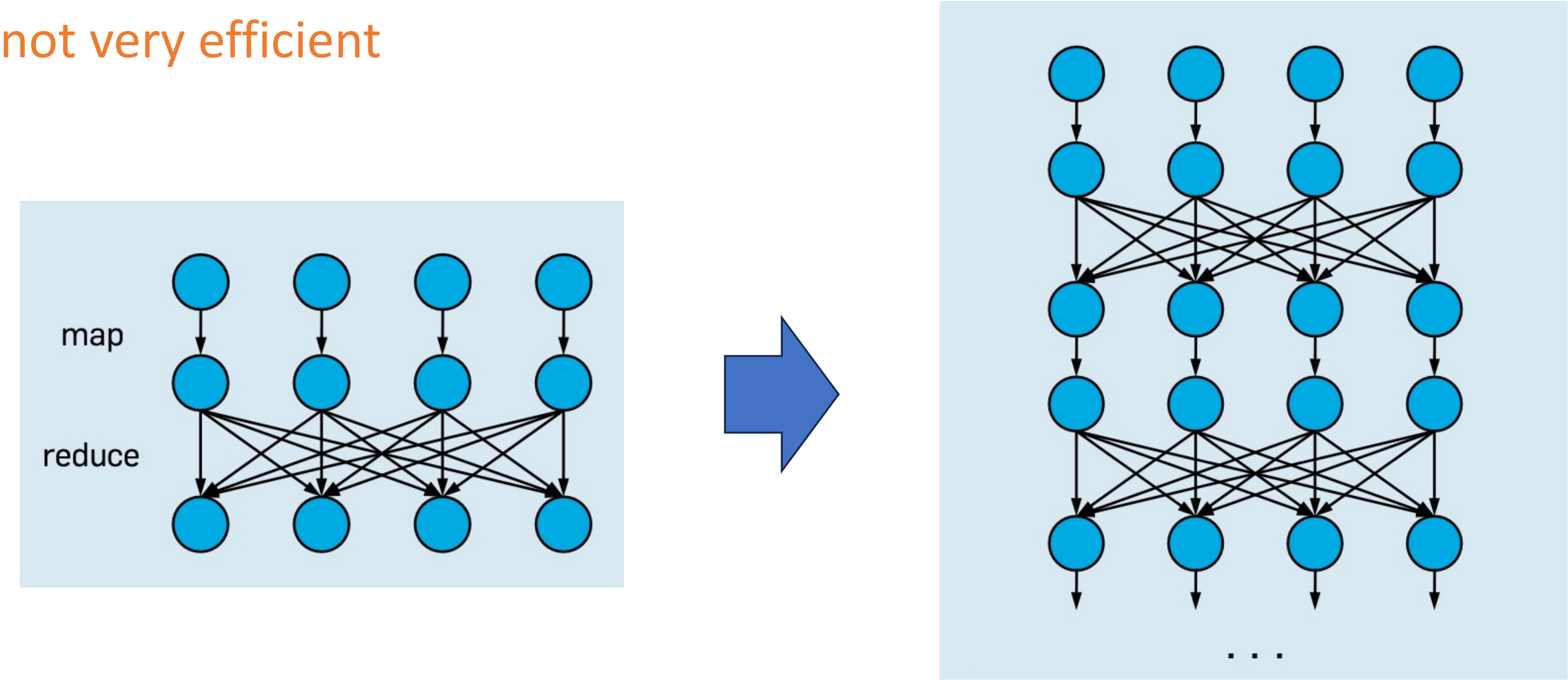






General Computation

- MapReduce can express any computation
- But not very efficient

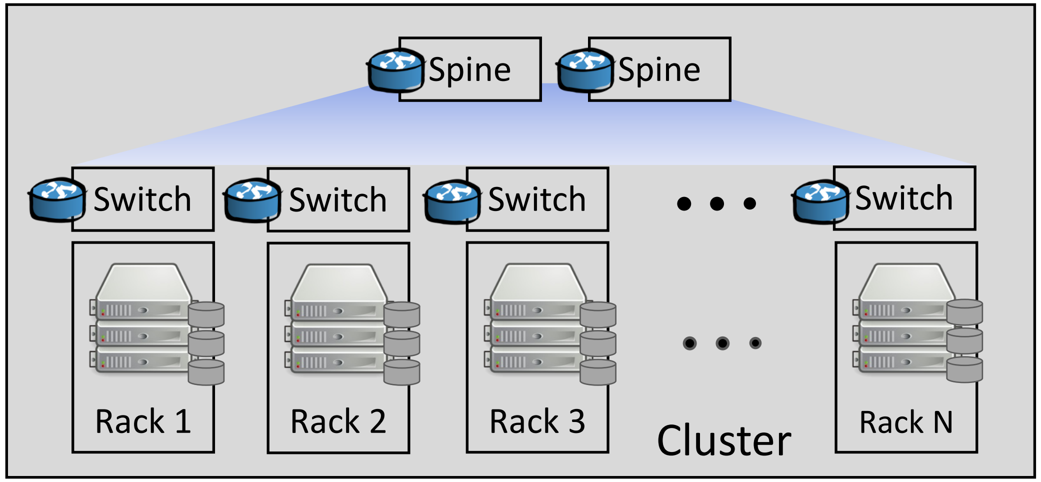


General Computation

- MapReduce can express any computation
- But not very efficient
  - High latency between rounds
  - Well-defined round boundaries limit optimizations across rounds
  - On-disk replication of data across each step
    - No in-memory sharing
  - Fixed communication pattern

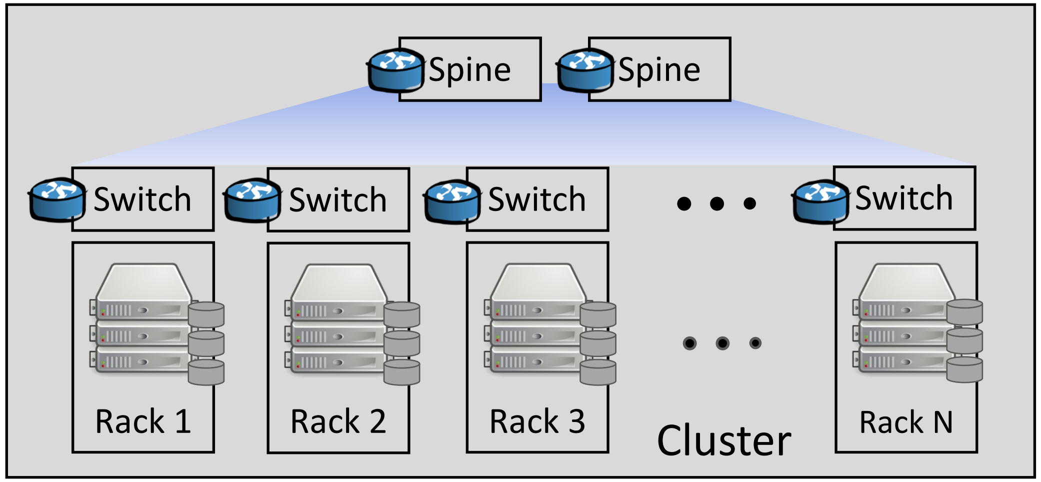
Hadoop hardware cluster model

- Hadoop assumes servers with compute, memory, *and disks*



Hardware Bottlenecks

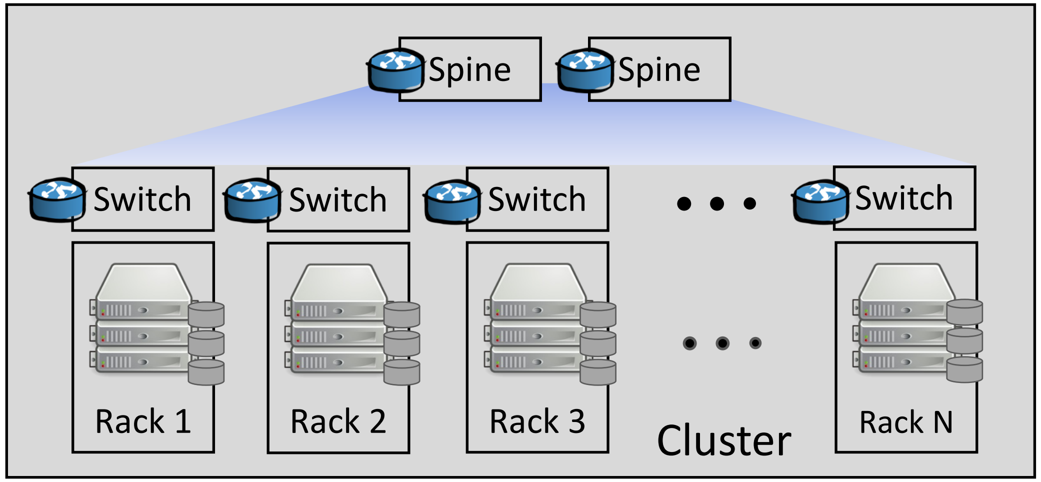
- Hadoop assumes servers with compute, memory, *and disks*



- 50GB/s RAM
- 1-2 GB/s disk
- 1.2 GB/s intra-rack bandwidth
- 0.2 GB/s average bandwidth inter-rack

Hardware Bottlenecks

- Hadoop assumes servers with compute, memory, *and disks*

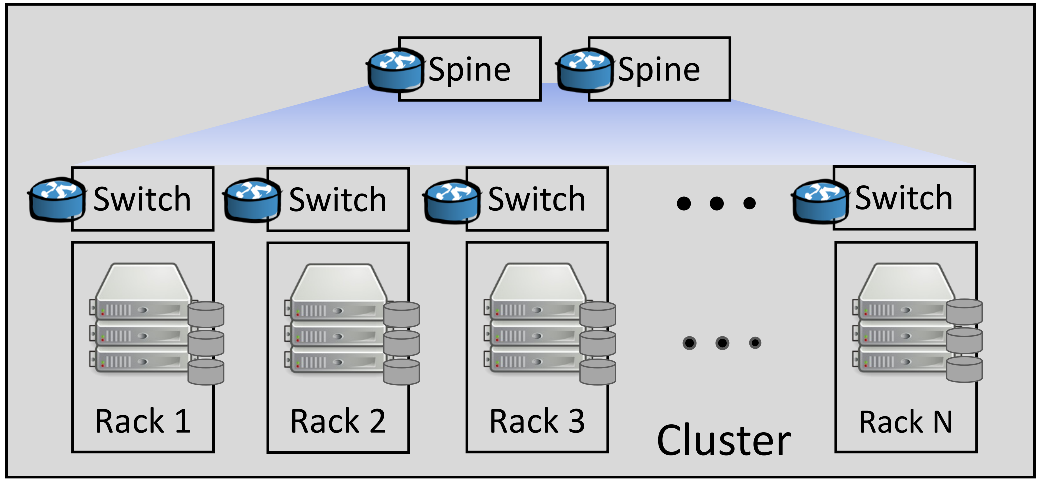


- 50GB/s RAM
- 1-2 GB/s disk
- 1.2 GB/s intra-rack bandwidth
- 0.2 GB/s average bandwidth inter-rack

Optimizing anything other than the bottleneck incurs minor benefits

RDDs provide flexibility

- Hadoop assumes servers with compute, memory, *and disks*



- 50GB/s RAM
- 1-2 GB/s disk
- 1.2 GB/s intra-rack bandwidth
- 0.2 GB/s average bandwidth inter-rack

Optimizing anything other than the bottleneck incurs minor benefits,
RDDs are general enough to allow optimization at hardware boundaries

Resilient Distributed Datasets

- High-performance
- General computation
- Lazy evaluation
- Ephemeral
- Lineage-based reconstruction
- Data sharing across different solutions

Creating RDDs

- From a file
- By “parallelizing” a data structure (array or dictionary)
  - Each slice is sent to a different node
- Transforming one RDD into another
- Persisting an RDD (cache and save)

Parallel Operations

- Functional operators
  - Map, filter, groupBy, partition
  - Collect, reduce
  - Foreach
  - Closures

Parallel Operations

- Functional operators
  - Map, filter, groupBy, partition
  - Collect, reduce
  - Foreach
  - Closures
- Shared variables
  - Broadcast variables (read-only)
  - Accumulators (append-only)

Parallel Operations

- Functional operators
  - Map, filter, groupBy, partition
  - Collect, reduce
  - Foreach
  - Closures
- Shared variables
  - Broadcast variables (read-only) → Stored on disk
  - Accumulators (append-only) → Sync only when task ends *without failure*

Lazy evaluation and ephemeral

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(
  s => s.startsWith("ERROR"))
println("Total errors: "+errors.count())
```

Resilient Distributed Datasets

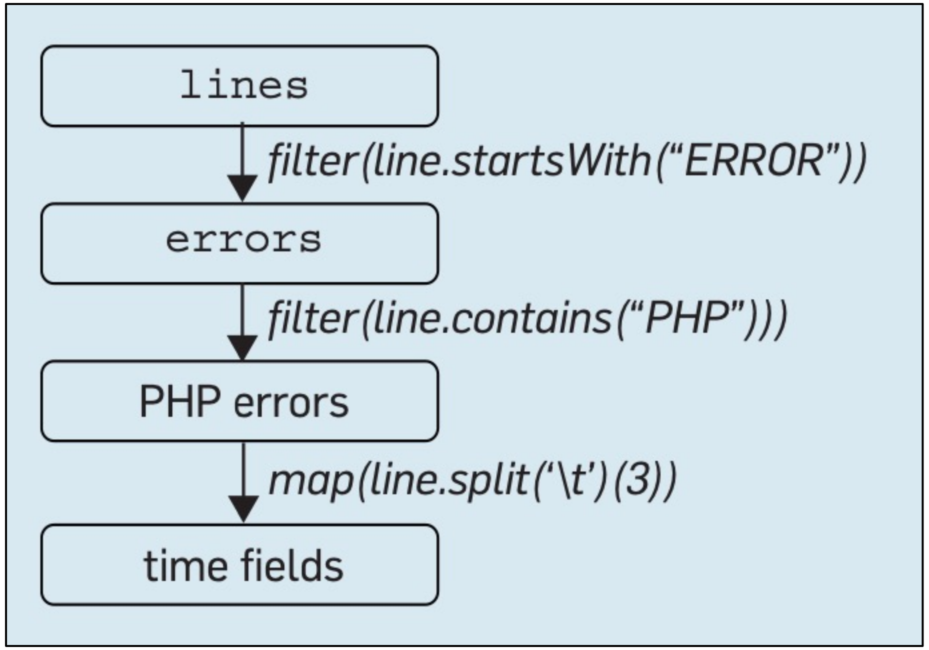
- High-performance
- General computation
- Lazy evaluation
- Ephemeral
- Lineage-based reconstruction
- Data sharing across different solutions

Lineage-based Reconstruction

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(
  s => s.startsWith("ERROR"))
println("Total errors: "+errors.count())
```

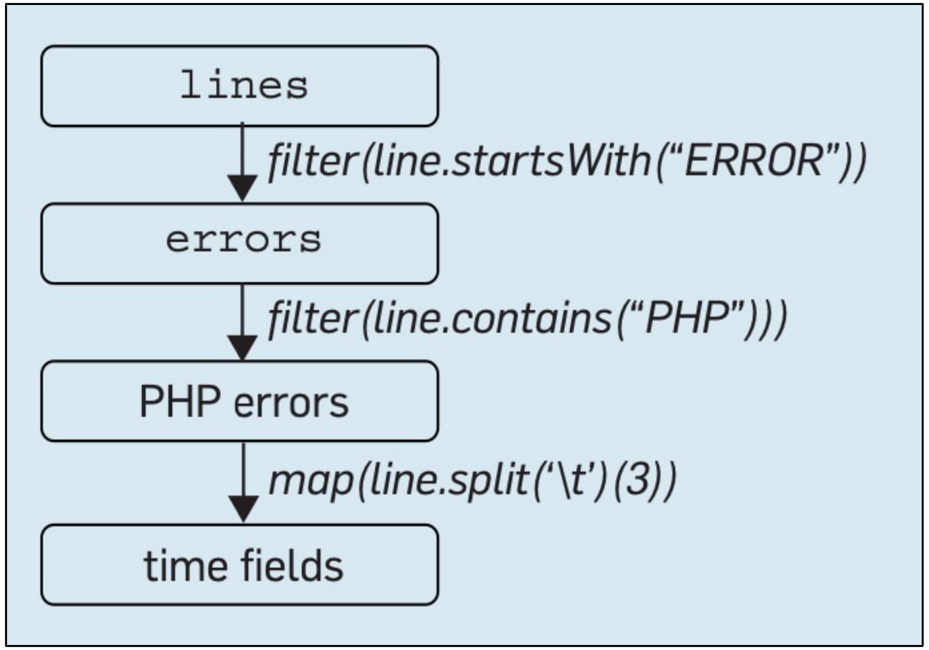
Lineage-based Reconstruction

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(
  s => s.startsWith("ERROR"))
println("Total errors: "+errors.count())
```



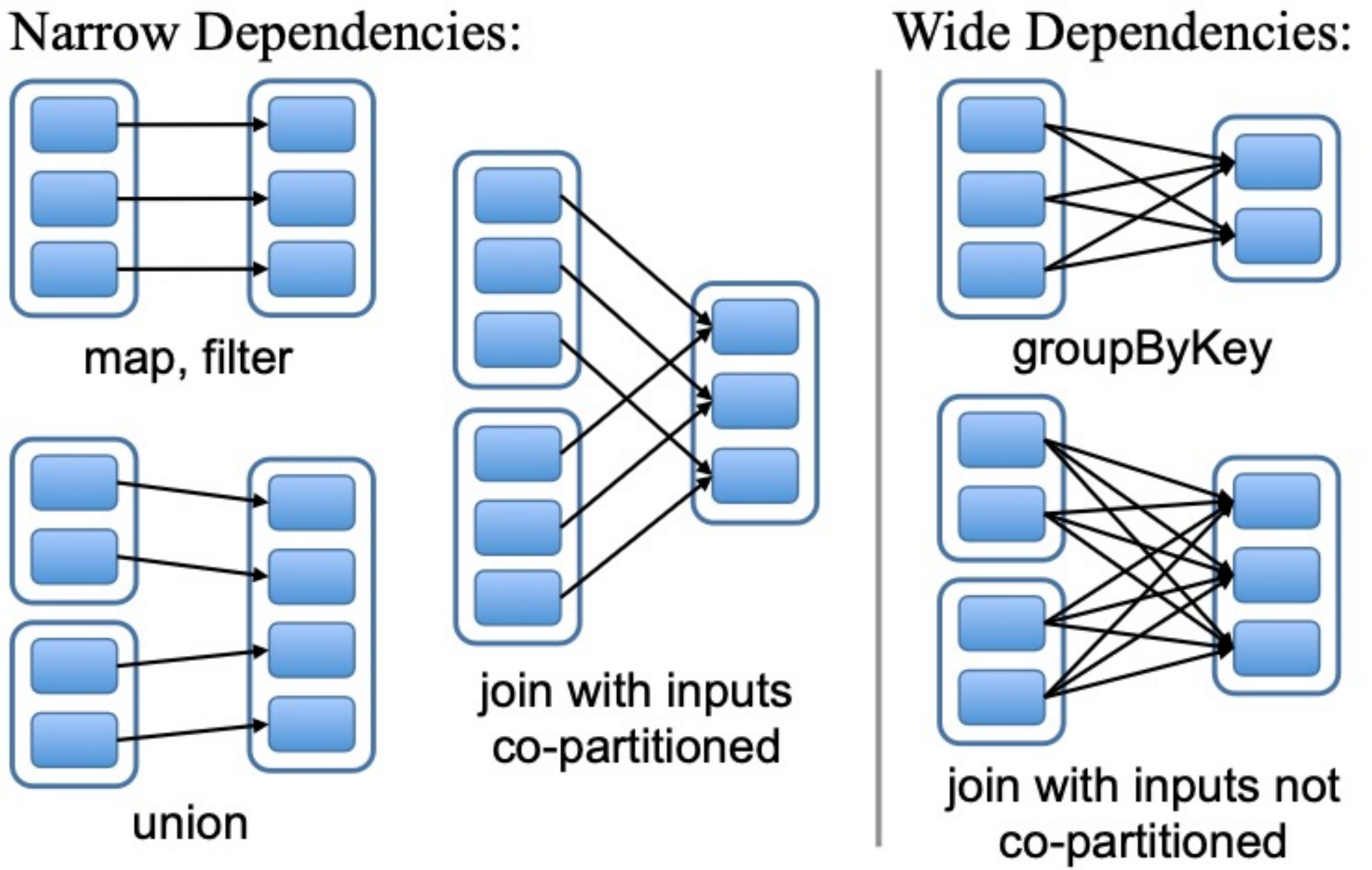
Lineage-based Reconstruction

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(
  s => s.startsWith("ERROR"))
println("Total errors: "+errors.count())
```



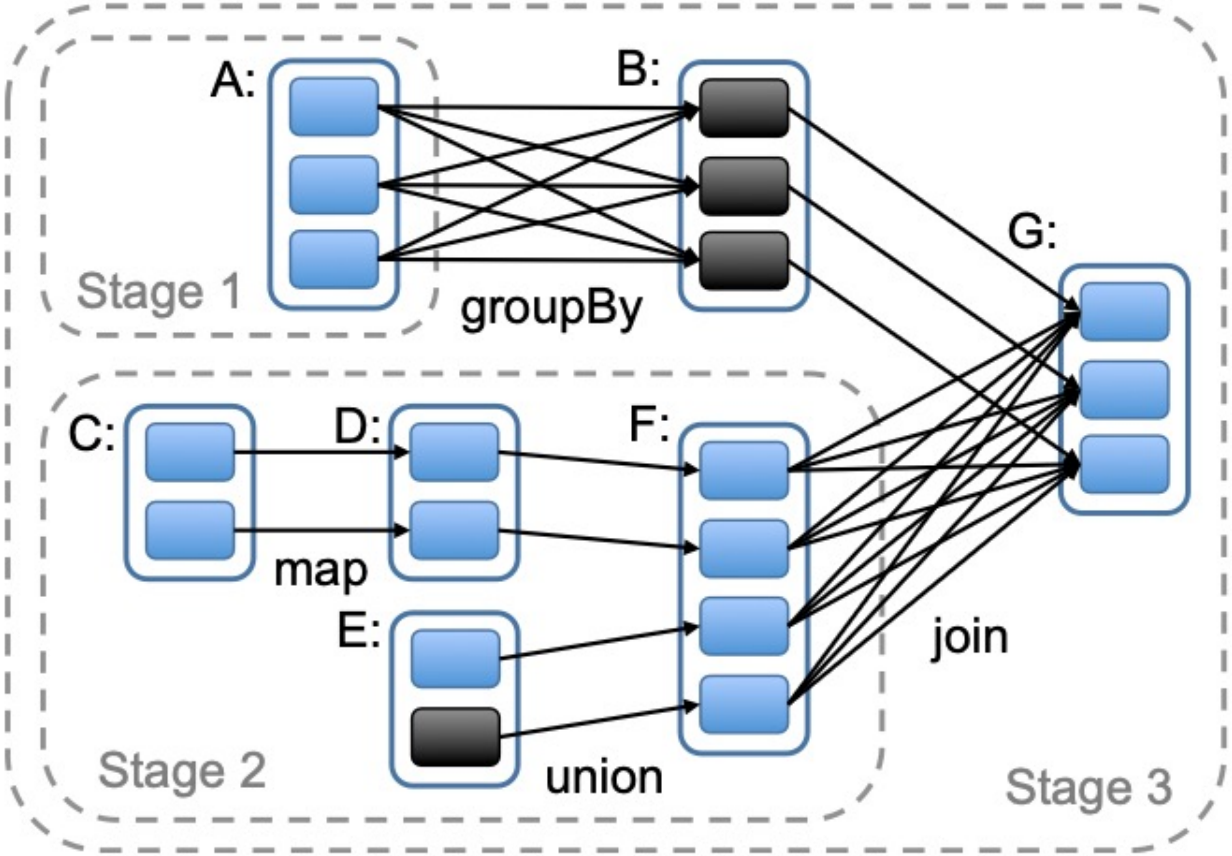
- Faster reconstruction (memory > disk)
- No persistence on disk

Narrow and Wide Dependencies



Implementation Details

Process Scheduling



- Stage 1 does not need to run
  - Results already in RAM
- Stage 2 pipelines narrow-dependencies
- Stage 3 runs afterwards because of wide dependency

Memory Management

- Aggressive caching of RDDs in memory
- LRU substitution
  - But prevent cycling of partitions of the same RDD

- Three ways of caching RDDs
  - In-memory, uncompressed
  - In-memory, serialized
    - Incurs deserialization overhead, amortized for complex computations
  - On disk