Big Data Processing

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: Computing Systems and Concurrency

Lecture 19

Marco Canini

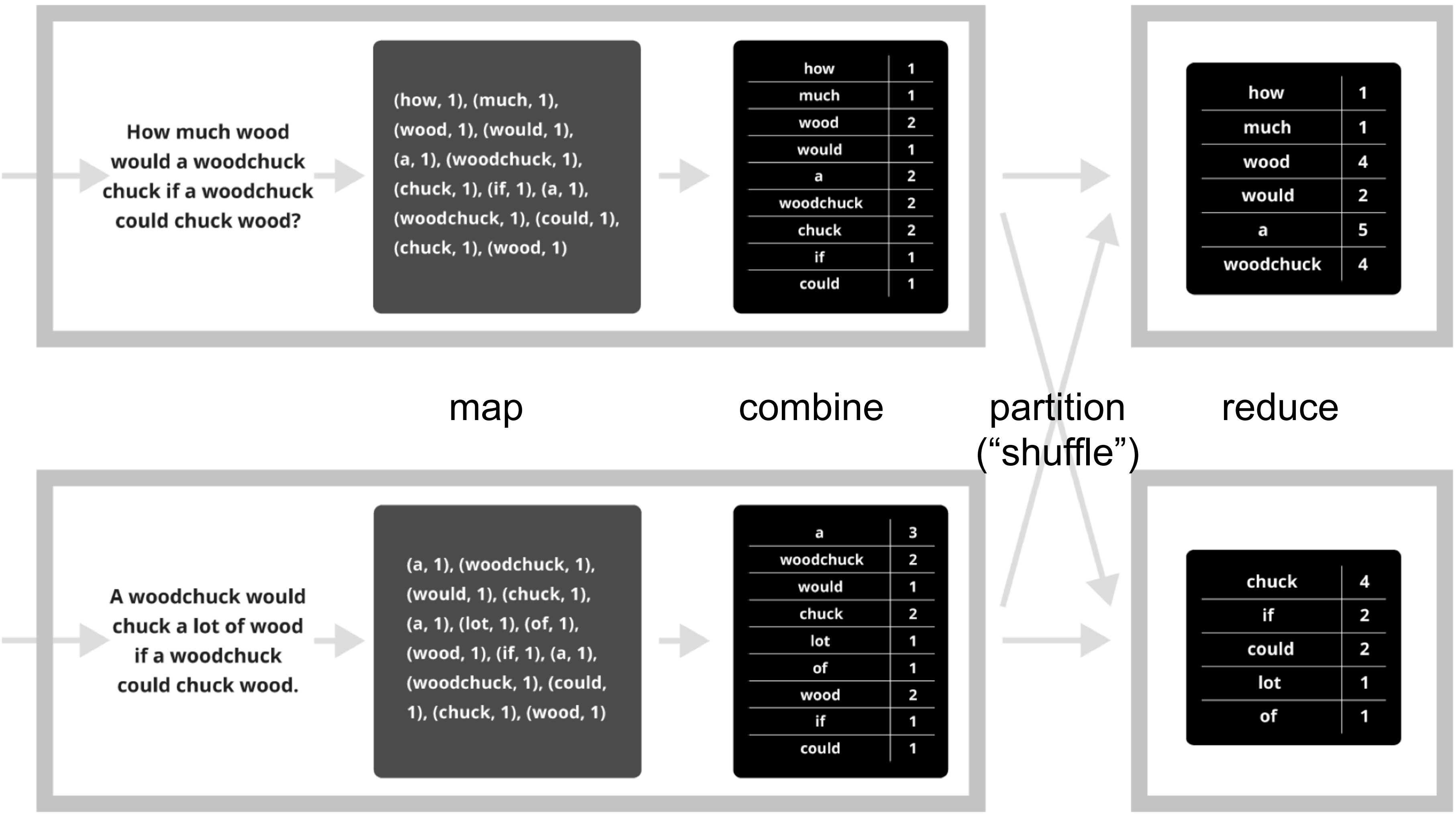
Credits: Michael Freedman and Kyle Jamieson developed much of the original material.
Selected content adapted from Wyatt Lloyd.

Data-Parallel Computation

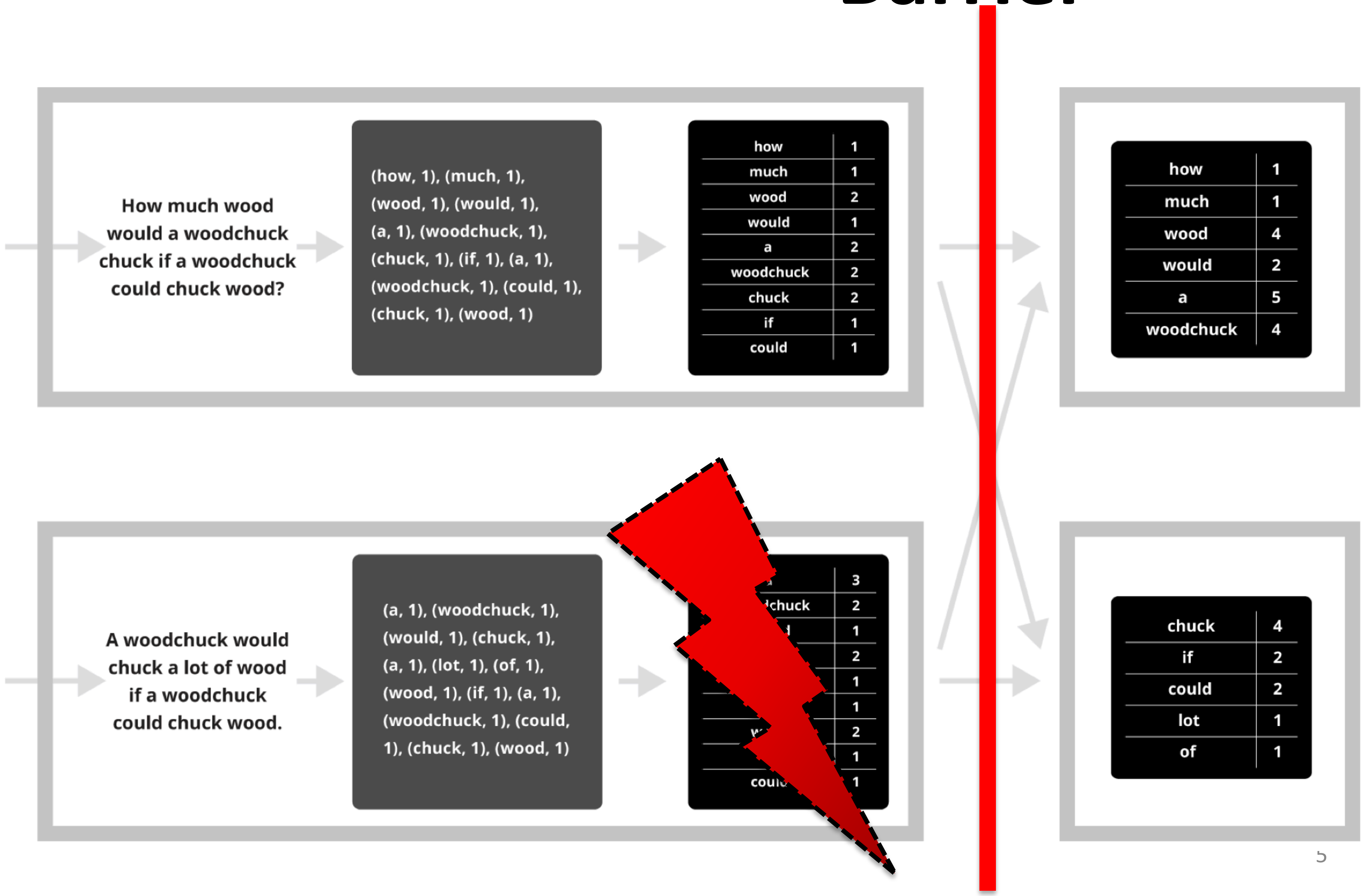
Ex: Word count using partial aggregation

1. Compute word counts from individual files
2. Then merge intermediate output
3. Compute word count on merged outputs

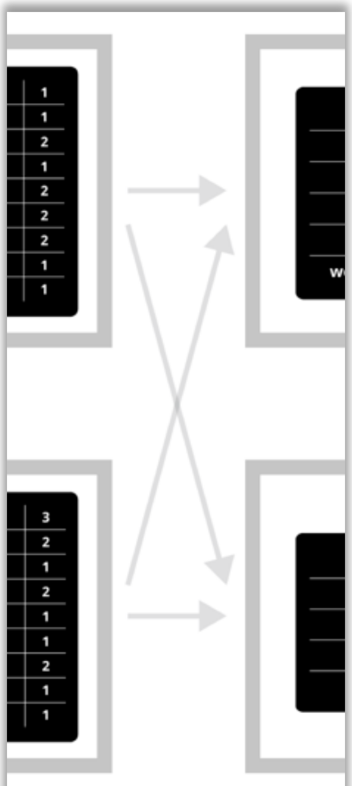
Putting it together…



Synchronization Barrier



Fault Tolerance in MapReduce



- Map worker writes intermediate output to local disk, separated by partitioning. Once completed, tells master node
- Reduce worker told of location of map task outputs, pulls their partition's data from each mapper, execute function across data
- Note:
  - "All-to-all" shuffle b/w mappers and reducers
  - Written to disk ("materialized") b/w *each* stage

Generality vs Specialization

General Systems

- Can be used for many different applications
- Jack of all trades, master of none
  - Pay a generality penalty
- Once a specific application, or class of applications becomes sufficiently important, time to build specialized systems

MapReduce is a General System

- Can express large computations on large data; enables fault tolerant, parallel computation
- Fault tolerance is an inefficient fit for many applications
- Parallel programming model (map, reduce) within synchronous rounds is an inefficient fit for many applications

MapReduce for Google's Index

- Flagship application in original MapReduce paper
- *Q: What is inefficient about MapReduce for computing web indexes?*
  - "MapReduce and other batch-processing systems cannot process small updates individually as they rely on creating large batches for efficiency."
- Index moved to Percolator in ~2010 [OSDI '10]
  - Incrementally process updates to index
  - Uses OCC to apply updates
  - 50% reduction in average age of documents

MapReduce for Iterative Computations

- Iterative computations: compute on the same data as we update it
  - e.g., PageRank
  - e.g., Logistic regression
- *Q: What is inefficient about MapReduce for these?*
  - Writing data to disk between all iterations is slow
- Many systems designed for iterative computations, most notable is Apache Spark
  - Key idea 1: Keep data in memory once loaded
  - Key idea 2: Provide fault tolerance via *lineage*:
    - Save data to disks occasionally, remember computation that created later version of data. Use lineage to recompute data that is lost due to failure.

MapReduce for Stream Processing

- Stream processing: Continuously process an infinite stream of incoming events
  - e.g., estimating traffic conditions from GPS data
  - e.g., identify trending hashtags on twitter
  - e.g., detect fraudulent ad-clicks

- *Q: What is inefficient about MapReduce for these?*

Stream Processing Systems

- Many stream processing systems as well, typical structure:
  - Definite computation ahead of time
  - Setup machines to run specific parts of computation and pass data around (topology)
  - Stream data into topology
  - Repeat forever
  - Trickiest part: fault tolerance!

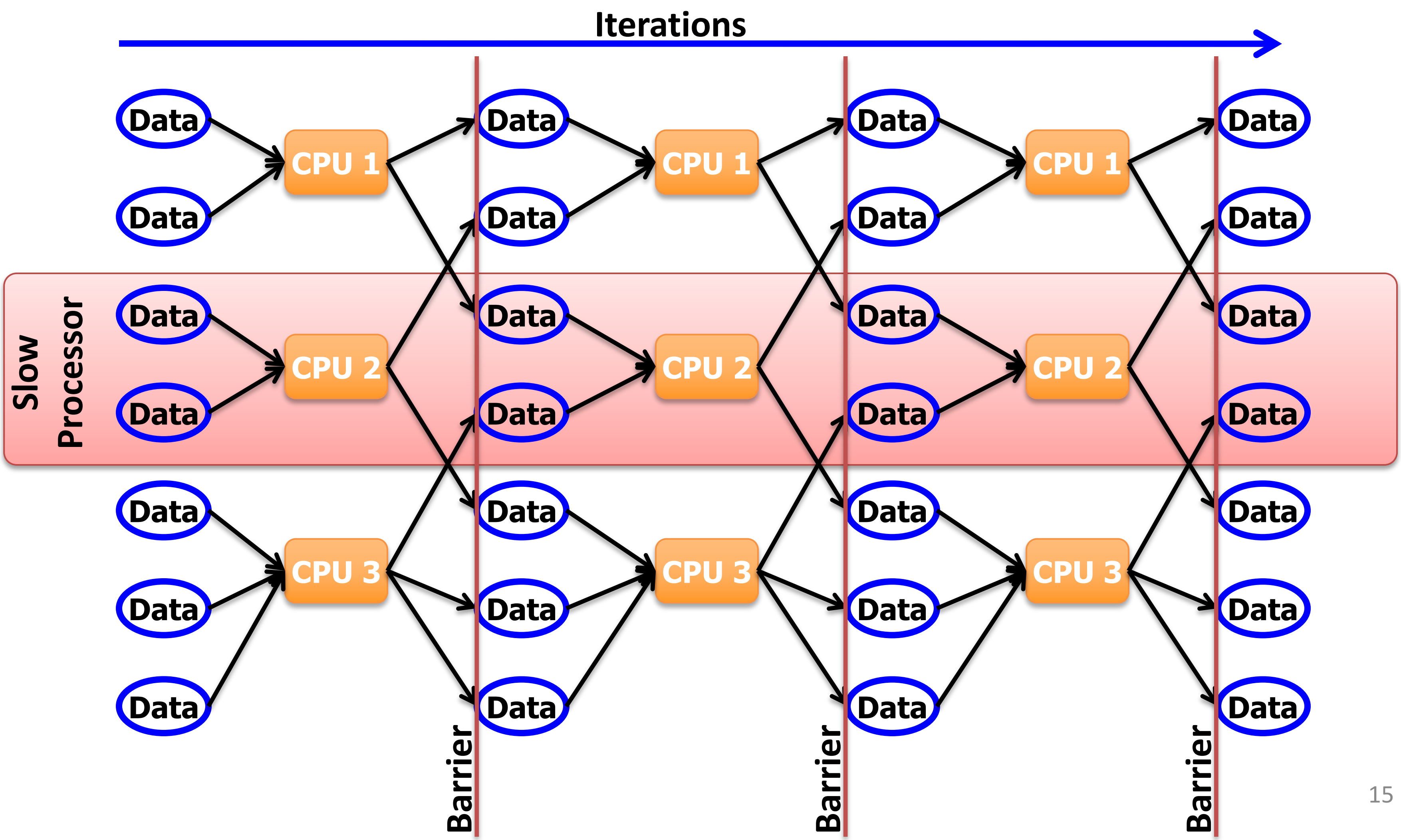
- Notably systems and their fault tolerance
  - Apache/Twitter Storm: Record acknowledgment
  - Spark Streaming: Micro-batches
  - Google Cloud dataflow: transactional updates
  - Apache Flink: Distributed snapshot

- Specialization is much faster, e.g., click-fraud detection at Microsoft
  - Batch-processing system: 6 hours
  - w/ StreamScope[NSDI '16]: 20 minute average

In-Memory Data-Parallel Computation

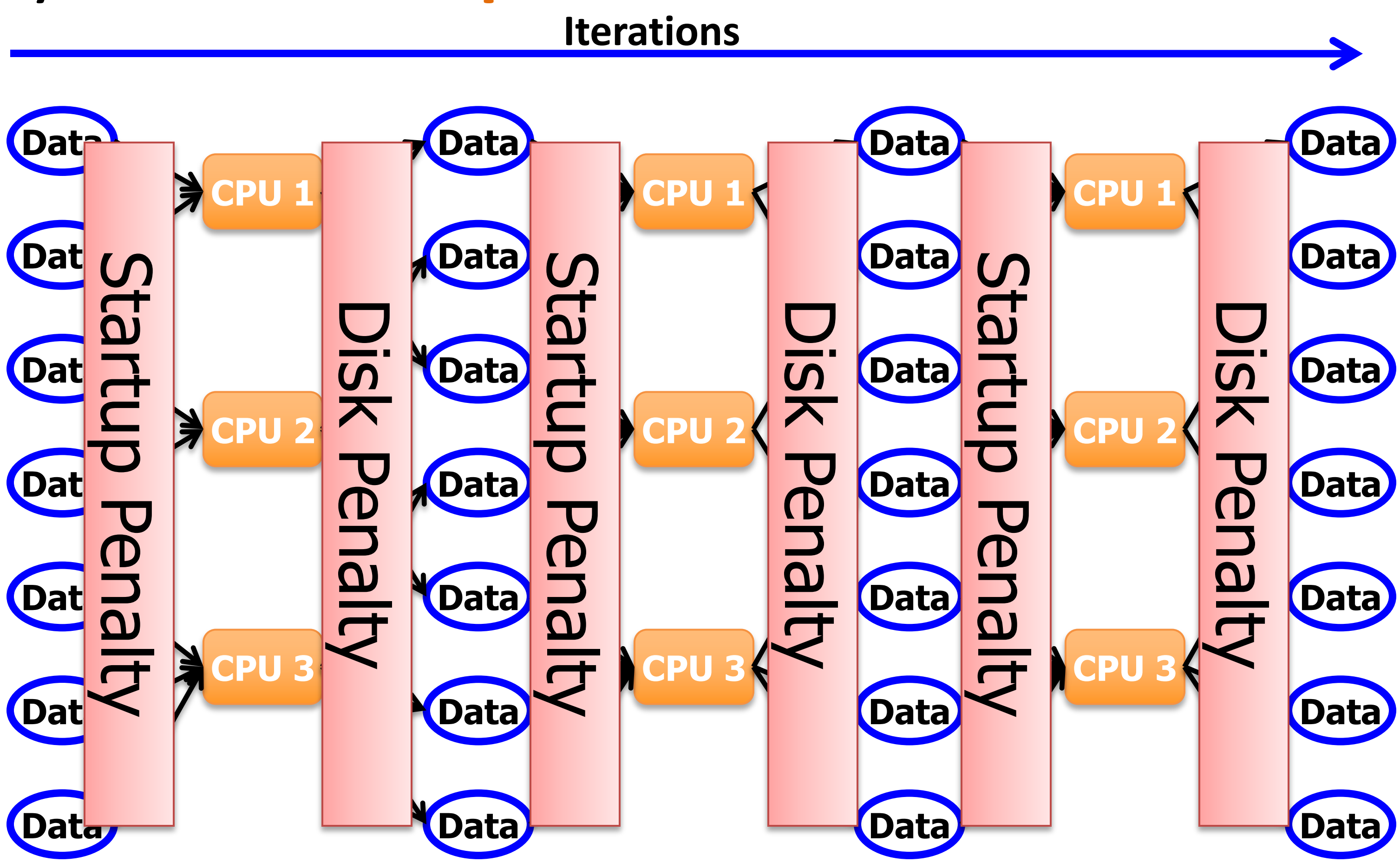
Iterative Algorithms

- MR **doesn't efficiently express** **iterative** algorithms:



MapAbuse: Iterative MapReduce

- System is **not optimized** **for iteration:**



Spark: Resilient Distributed Datasets

- Let's think of just having a big block of RAM, partitioned across machines…
  - And a series of operators that can be executed in parallel across the different partitions

- That's basically Spark
  - A distributed memory abstraction that is both fault-tolerant and efficient

Spark: Resilient Distributed Datasets

- Restricted form of distributed shared memory
  - Immutable, partitioned collections of records
  - Can only be built through *coarse-grained* deterministic transformations (map, filter, join, …)
  - They are called **Resilient Distributed Datasets** (RDDs)

- Efficient fault recovery using *lineage*
  - Log one operation to apply to many elements
  - Recompute lost partitions on failure
  - No cost if nothing fails

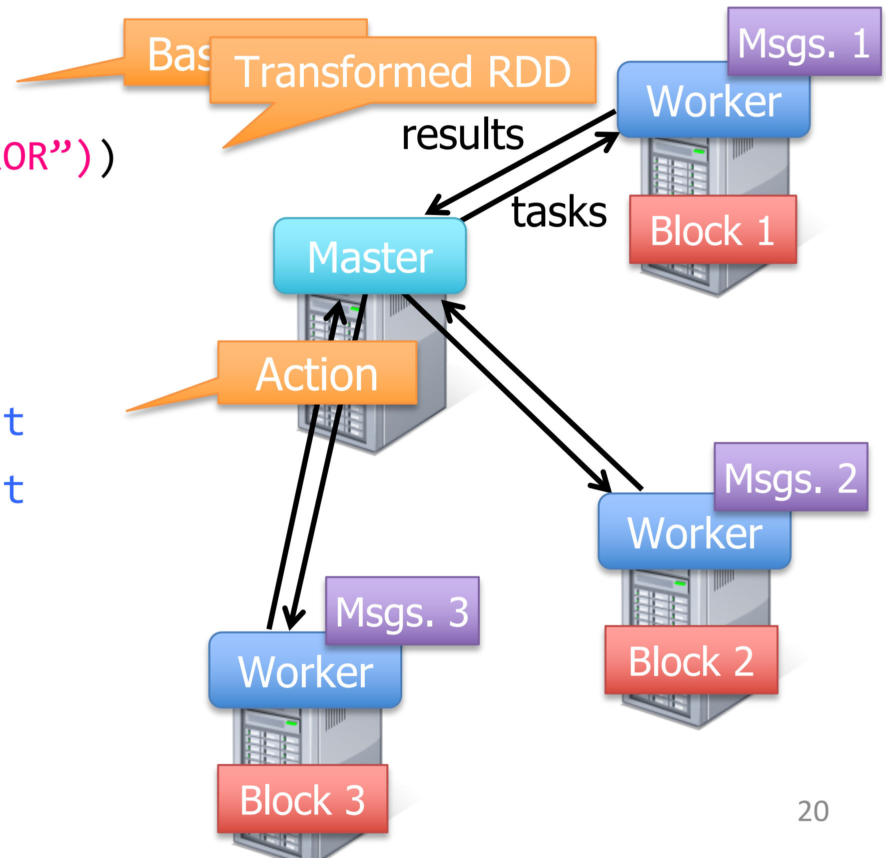
Spark Programming Interface

- Language-integrated API in Scala (+ Python)
- Usable interactively via Spark shell
- Provides:
  - Resilient distributed datasets (RDDs)
  - Operations on RDDs: deterministic *transformations* (build new RDDs), *actions* (compute and output results)
  - Control of each RDD's *partitioning* (layout across nodes) and *persistence* (storage in RAM, on disk, etc)

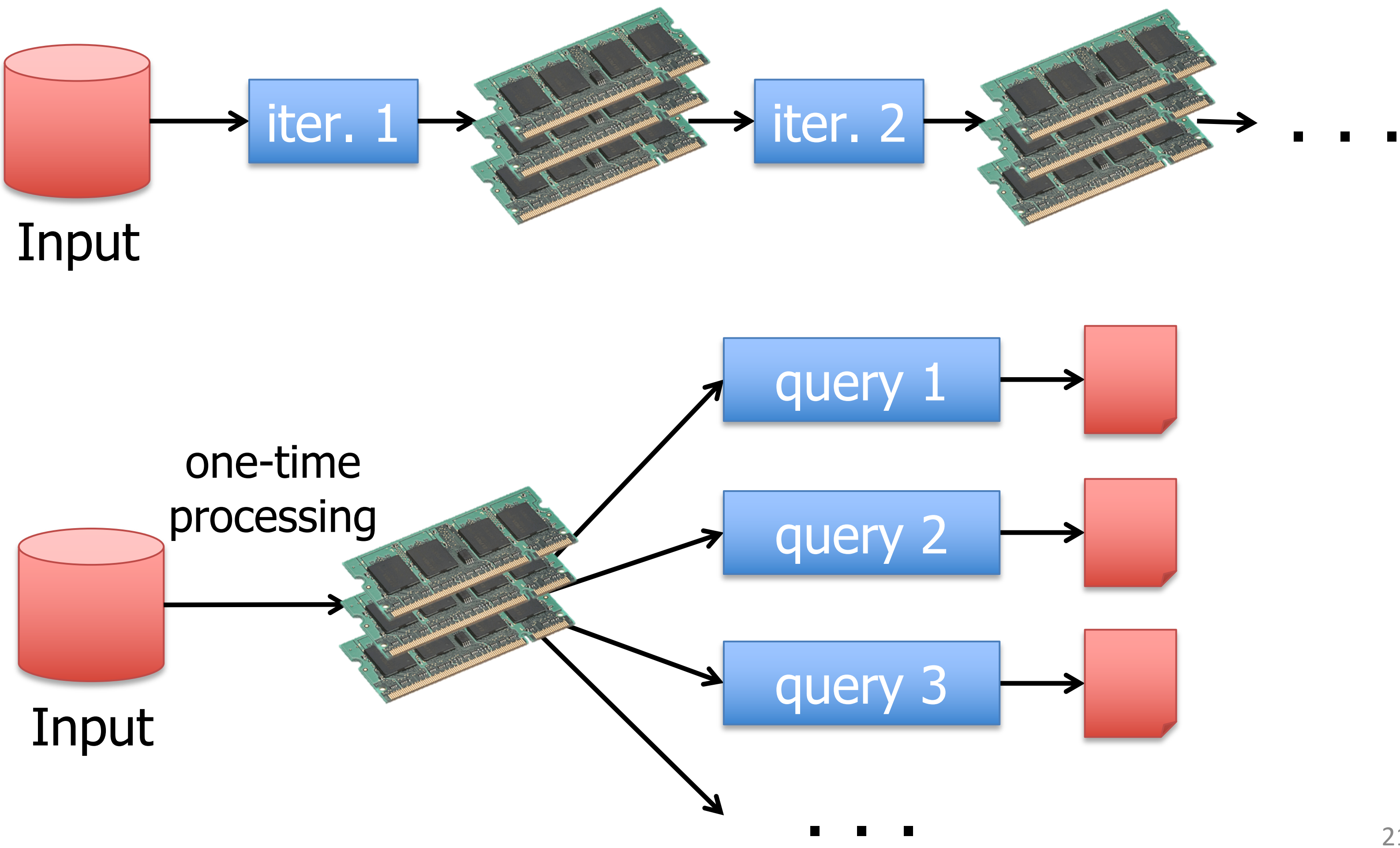
Example: Log Mining

- Load error messages from a log into memory, then interactively search for various patterns

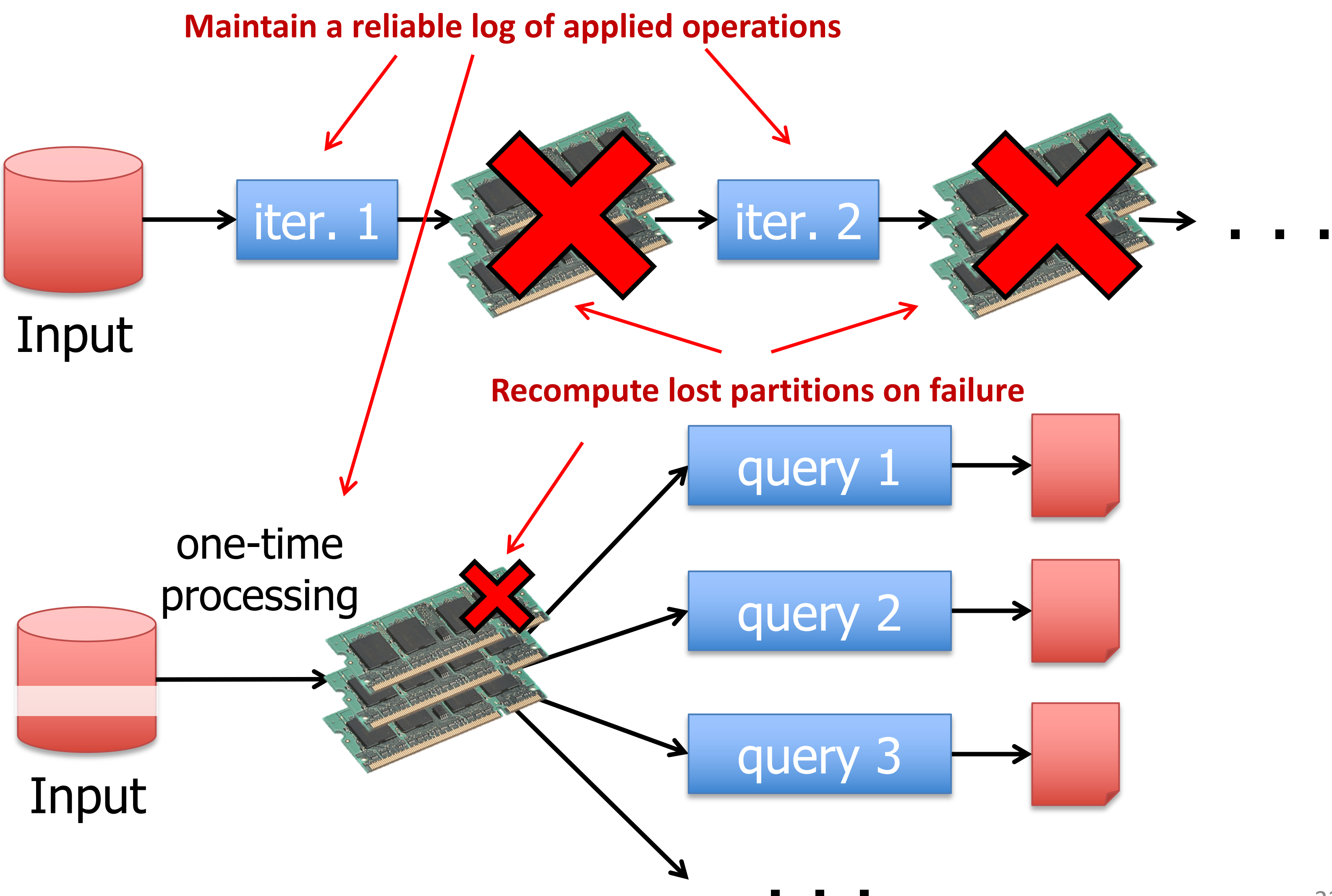
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
messages.persist()

messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
```



In-Memory Data Sharing



Efficient Fault Recovery via Lineage



Generality of RDDs

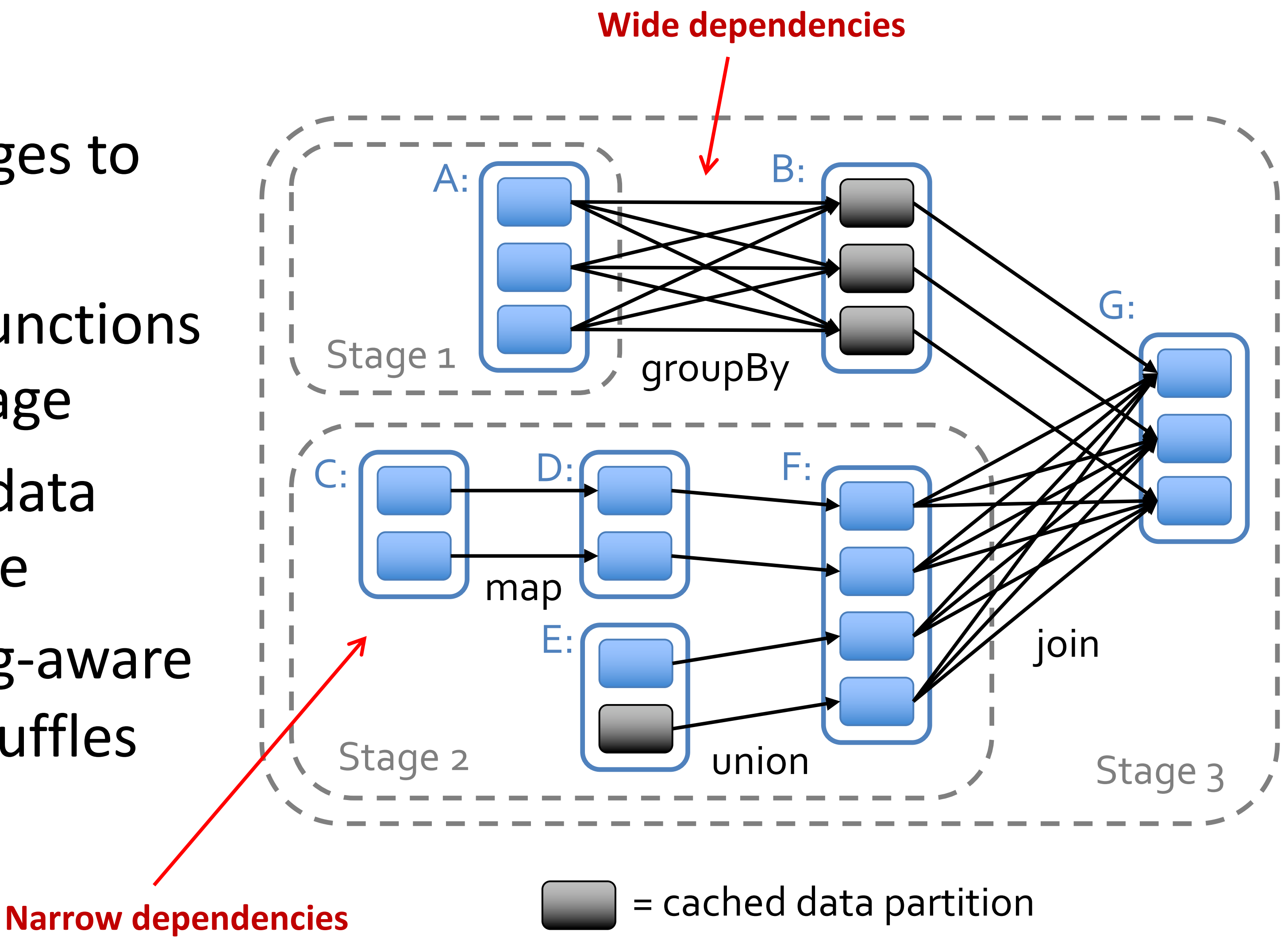
- Despite their restrictions, RDDs can express many parallel algorithms
  - These naturally *apply the same operation to many items*
- Unify many programming models
  - *Data flow models:* MapReduce, Dryad, SQL, …
  - *Specialized models* for iterative apps: BSP (Pregel), iterative MapReduce (Haloop), bulk incremental, …
- Support *new apps* that these models don't
- Enables apps to efficiently *intermix* these models

Spark Operations

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey take	

Task Scheduler

- DAG of stages to execute
- Pipelines functions within a stage
- Locality & data reuse aware
- Partitioning-aware to avoid shuffles



Spark Summary

- Global aggregate computations that produce program state
  - compute the count() of an RDD, compute the max diff, etc.
- Loops!
  - Spark makes it much easier to do multi-stage MapReduce
- Built-in abstractions for some other common operations like joins
- See also Apache Flink for a flexible big data platform