Data Management in the Cloud

APACHE SPARK – PART 2

THANKS TO M. ZAHARIA

Key Idea: Resilient Distributed Datasets

Resilient Distributed Datasets (RDDs)

- Immutable collections, usually partitioned
- Defined via transformations (map, filter, join, …)
- Fault tolerant – can be reconstructed on failure

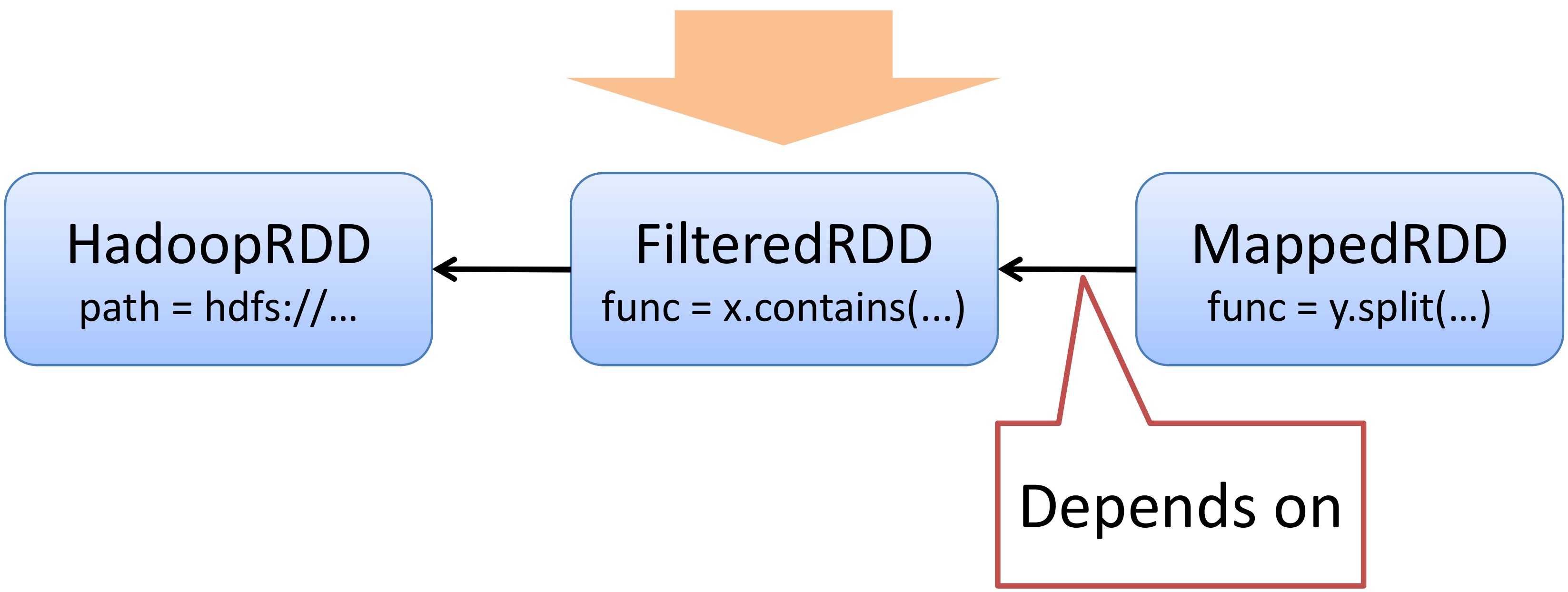
  Or checkpointed
- Can persist, in memory or on disk
  - Connect sequences of transformations
  - Use same dataset for multiple tasks

Fault Tolerance

Each RDD tracks the series of transformations used to build it (its *lineage*) to recompute lost data

E.g:

```
messages = spark.textFile("hdfs://...")
                .filter(x => x.contains("error"))
                .map(y => y.split('\t')(2))
```



Narrow vs. Wide Dependencies

- *Narrow* dependency: Slice of result RDD depends on a slice of the parent RDD

  Examples: `map, filter`

- *Wide* dependency: Slice of a result RDD depends on many slices of the parent RDD

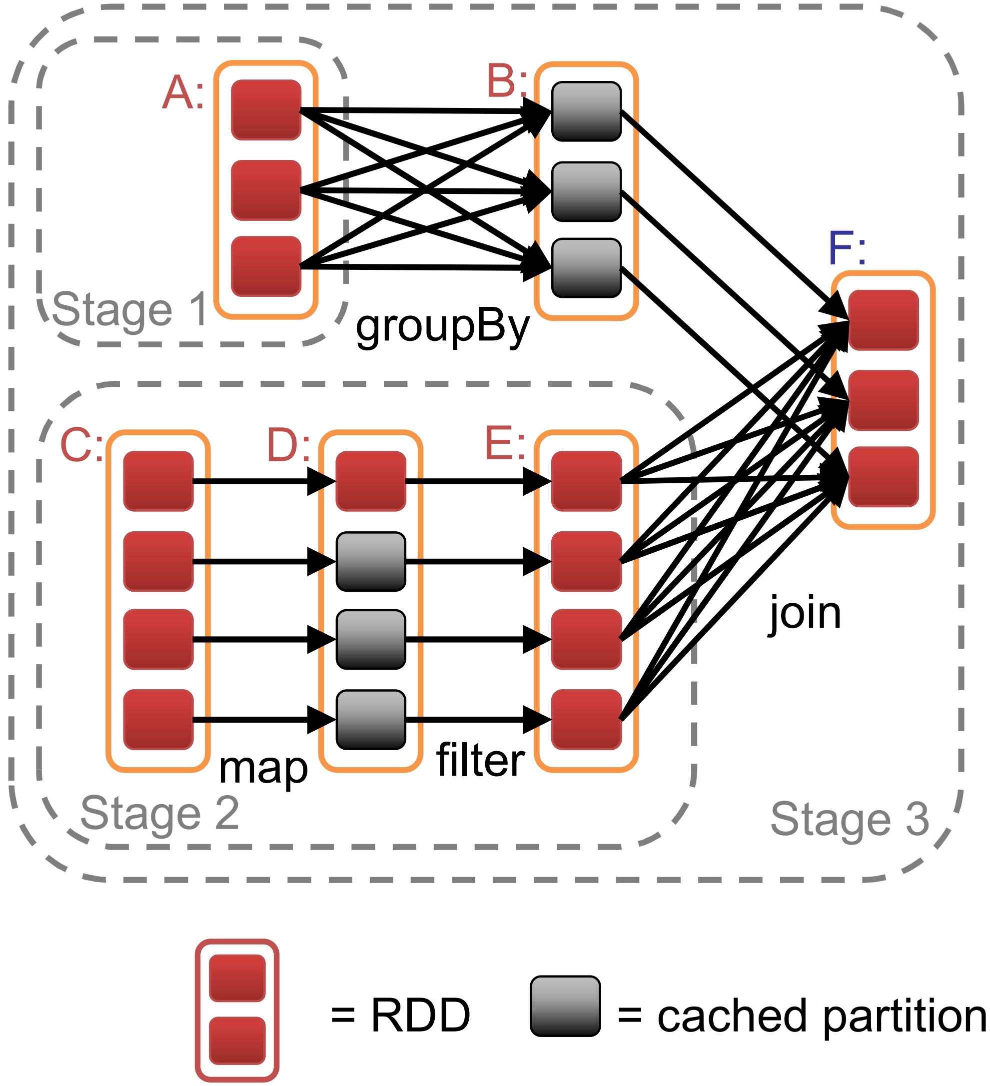
  Example: `groupBy`

`Join` might go either way, depending on whether partitioning is on join field.

Data heading into a wide dependency will be stored locally to help with re-generation.

Task Scheduler

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



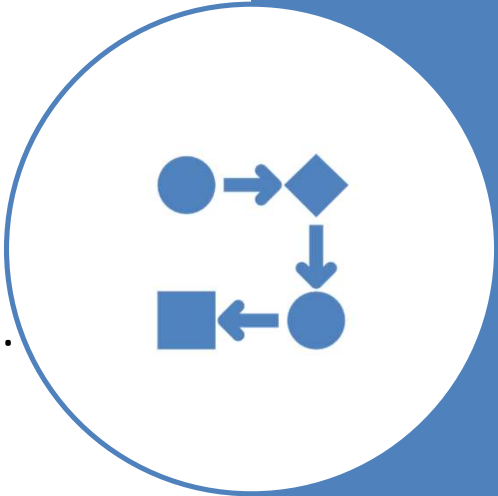
Participation Question

Objective:

– Work in groups to visualize how Spark tracks data transformations using lineage, and to discuss how fault recovery works in an RDD.

Instructions:

1. In your group, choose a simple Spark job (for example, reading a file, applying a filter, and then mapping the results).
2. Draw a quick diagram that shows:
   - Each step of the job as an RDD node
   - The dependencies between these nodes
3. Identify which transformation creates a narrow dependency (e.g., map or filter) and, if possible, add a wide dependency (e.g., a join) to discuss the difference.
4. Briefly discuss and note on your diagram how Spark would recover a lost partition using the lineage information (or how checkpointing might truncate a long lineage).



Representation of RDDs

Three representation choices for RDDs

- De-serialized Java objects in memory

  Fastest access

- Serialized object in memory

  Slower access, but less storage

- Disk storage

  If too large for main memory and costly to recompute

Checkpointing

If it might take a long time to recreate an RDD from its lineage, can checkpoint it.

- Use the `RELIABLE` tag
- Can use disk or in-memory replication
- Indicated when an RDD has a wide dependency on its parent(s)

Example: Logistic Regression

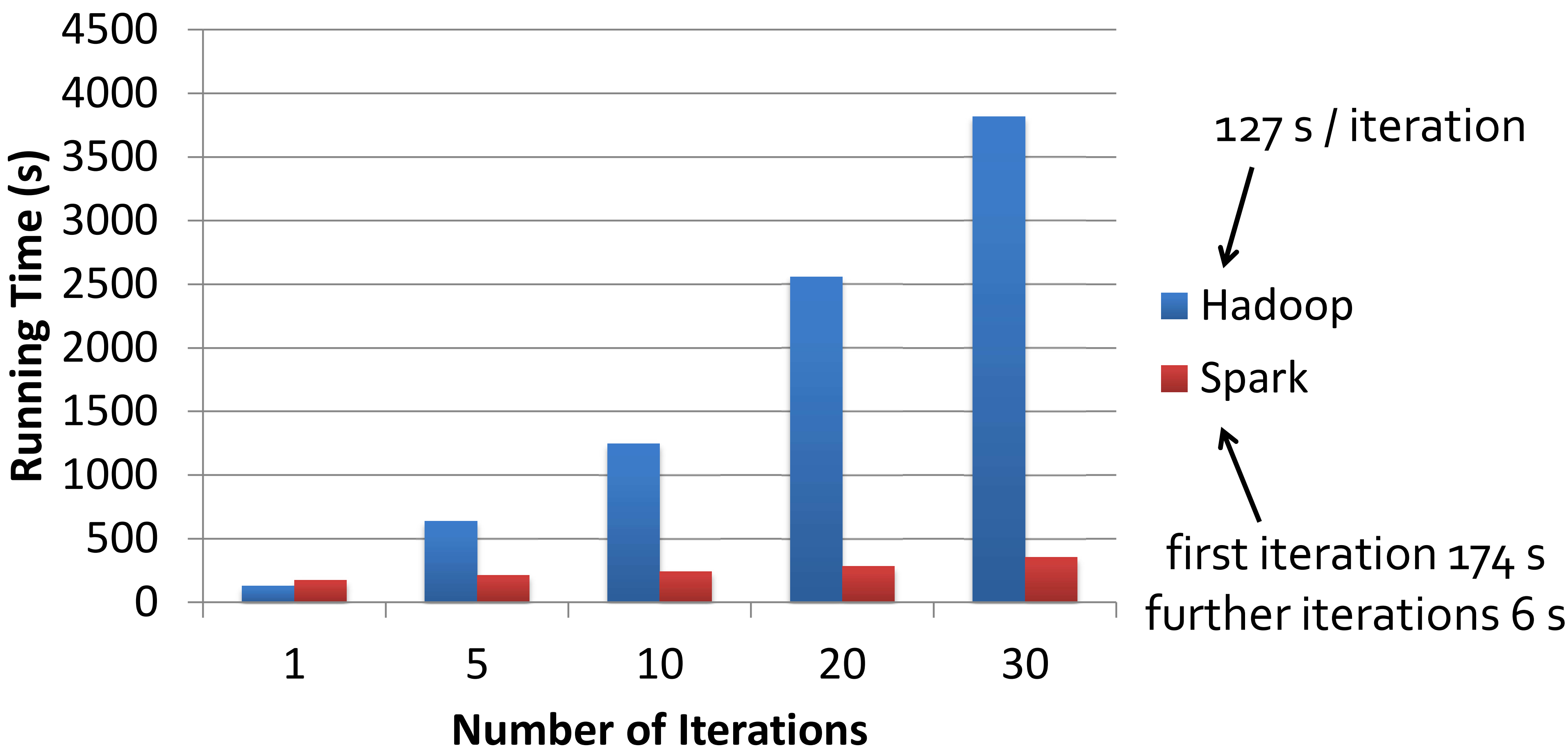
```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce((a,b) => a+b)
  w -= gradient
}

println("Final w: " + w)
```

Load data in memory once

Initial parameter vector

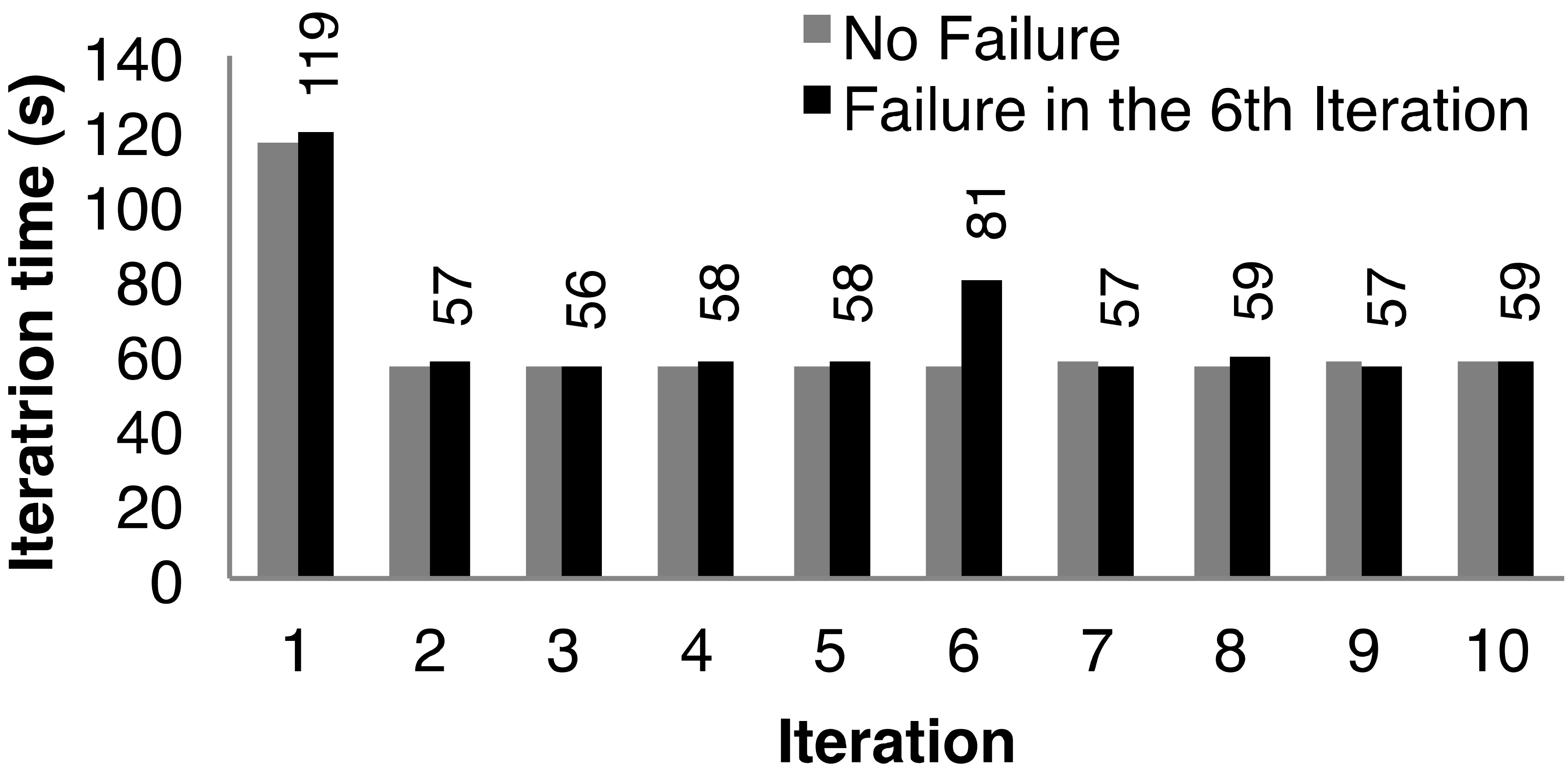
Repeated MapReduce steps
to do gradient descent

Logistic Regression Performance



Fault-Recovery Test

On K-means



Environment, Languages

Ways to run it

- Local multicore: just a library in your program
- EC2: scripts for launching a Spark cluster
- Google Cloud Platform, with files or Bigtable
- Private cluster: Mesos, YARN, Standalone Mode

Languages

- APIs in Java, Scala and Python
- Interactive shells in Scala and Python

Data from HDFS, S3, Bigtable, Hbase, Cassandra

Understands Hadoop input formats

Generality of RDDs

Claims that RDDs are more general than other approaches

- Can express any distributed programming model (though maybe not always efficiently)
- Optimizations for to reduce network use and storage I/O can be readily applied to RDDs

Spark Libraries

- Dataframes: RDDs with schema
  - Spark SQL (was Shark)
  - SparkR
- MLib: Machine-learning library
- DStreams: Discretized streams

Can use all these libraries together.

Dstreams: Microbatches

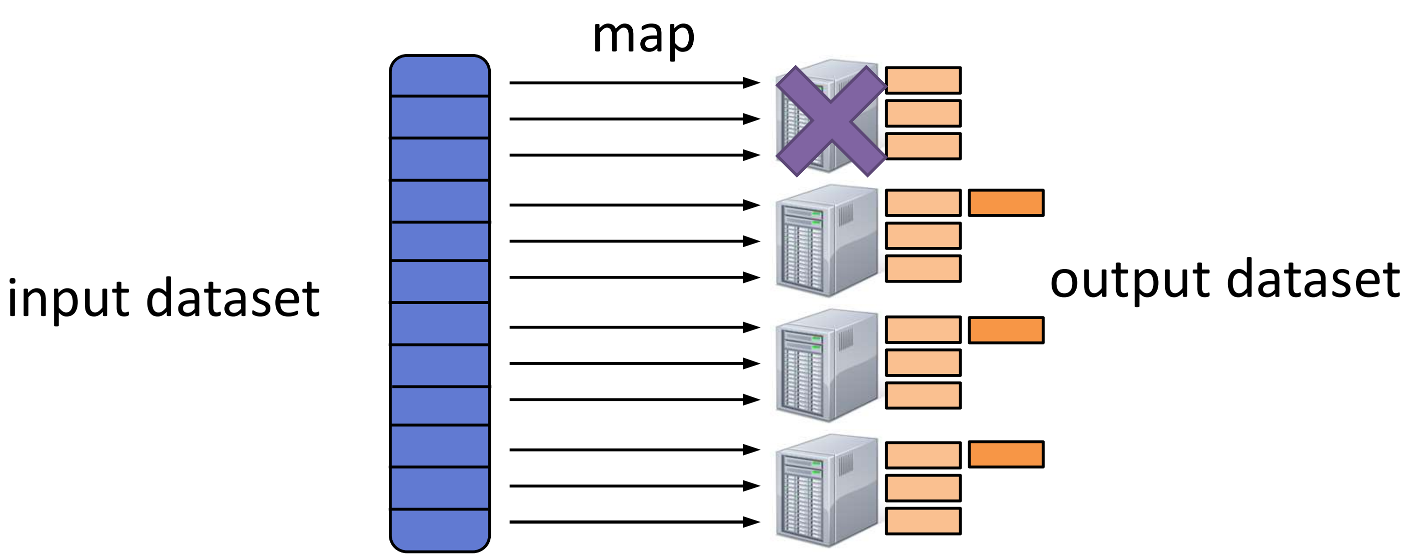
Run a streaming computation as dataflow over a sequence of RDDs that split up the stream

- Model stream computation as a sequence of stateless transformations over these batches. (Operator state must be explicitly output to next stage.)
- Same recovery scheme
- Try to make batch size as small as possible

Note: Latencies in 1-2s range

Parallel Recovery

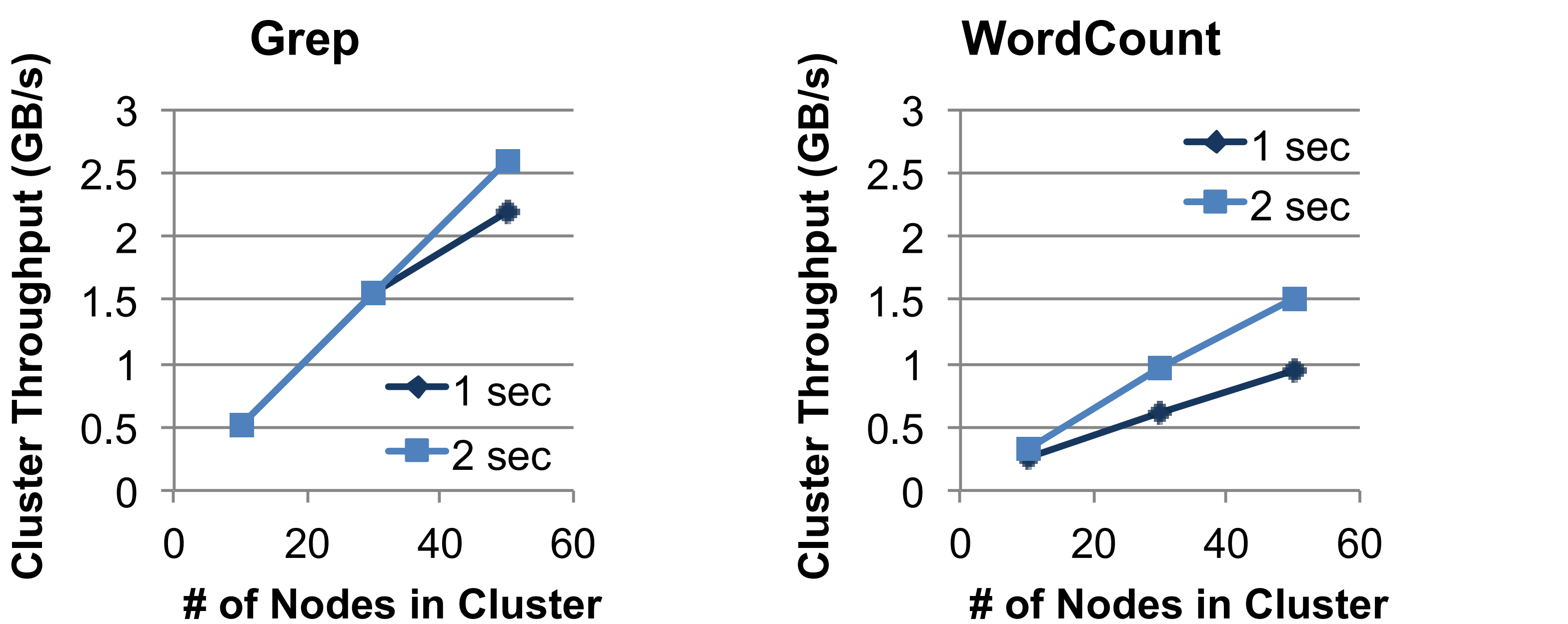
- Checkpoint state datasets periodically
- For node failure, recompute its slices **in parallel** on other nodes



Faster recovery than upstream backup, without the cost of replication

Performance

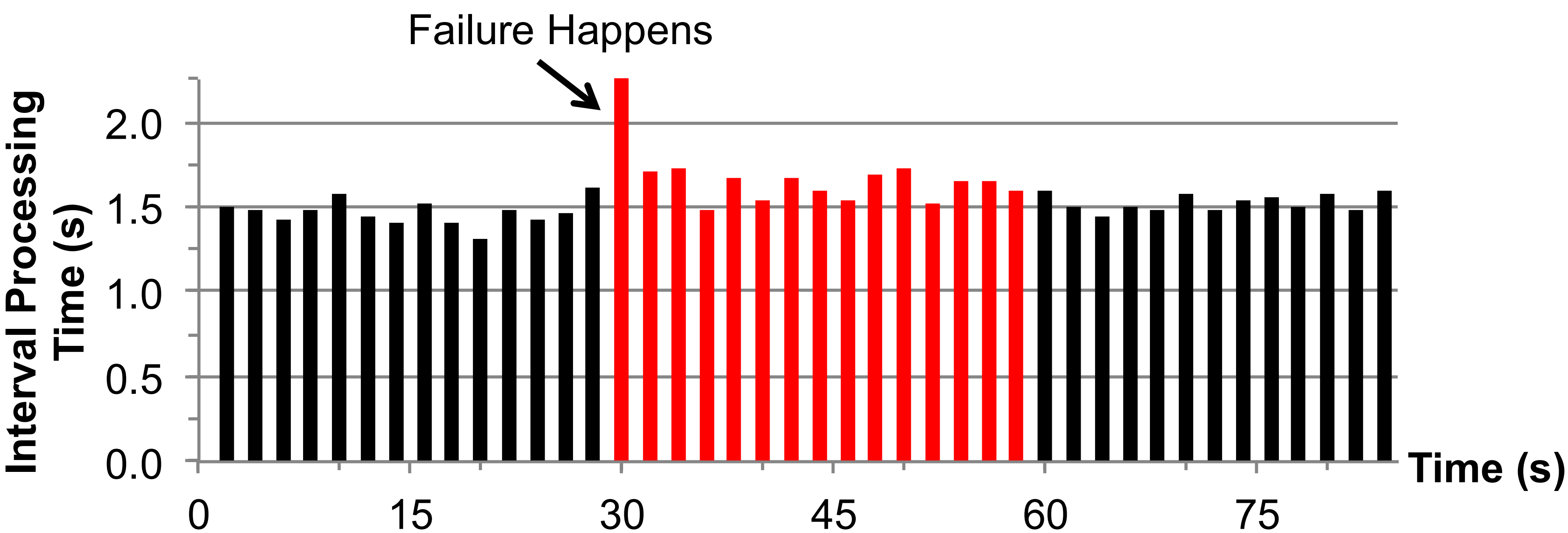
- Prototype built on the Spark in-memory computing engine can process **2 GB/s** (**20M records/s**) of data on 50 nodes at **sub-second** latency



Max throughput within a given latency bound (1 or 2s)

Failure Recovery

- Recovers from failures within **1 second**



Sliding WordCount on 10 nodes with 30s checkpoint interval

References

- M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, I. Stoica. ***Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing***, NSDI 2012.
- M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. ***Discretized Streams: Fault-Tolerant Streaming Computation at Scale***, SOSP 2013.
- M. Zaharia, R. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica. ***Apache Spark: A Unified Engine for Big Data Processing***, CACM, 59(11):56-65, November 2016.