CS 582: Distributed Systems

Fault Tolerance in Spark

Dr. Zafar Ayyub Qazi

Fall 2024

Acknowledgements: Slides use some material from Matei Zaharia (UC Berkeley)

Why Spark?

- Interesting fault tolerance story
- Popular open-source project
  - Opensource at Apache, 50+ companies contributed to it
  - Big startup: Databricks
- Widely used
  - https://databricks.com/customers
  - More than 5000 customers
- Works well for many big data apps

Spark Background

Spark Background

- Arose from an academic setting
  - Amplab @ UC Berkeley
- Project Leader: Matei Zaharia
  - Back then: PhD student at UC Berkeley advised by Ion Stoica & Scott Shenker
  - Got ACM doctoral thesis award
  - Now: Professor at Stanford University and CTO Databricks
- This paper was published in 2012 in NSDI
- Open sourced from day one …

Our Focus: Fault Tolerance in Spark

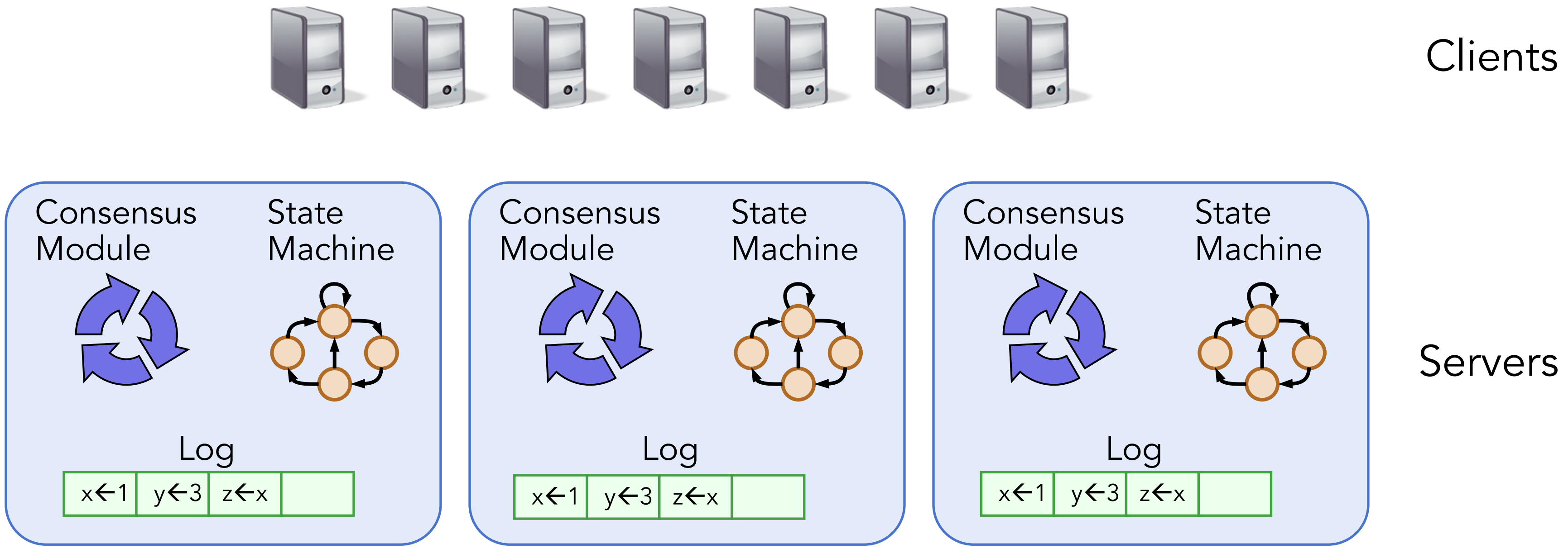
- Also provides a programming model and execution strategy

Our general fault tolerance story

- Replicate
  - If a server fails, have copies on backup servers
- Store data in persistent storage
  - If a server fails, we don't lose data

Replicated State Machine → Replicated Log

Replicated State Machine → Replicated Log



- Paxos/Raft/PBFT
  - Replicate <u>log of operations</u> on different servers
  - Store log of operations & state in persistent storage

Spark Context: Big Data Processing

- Considers applications that use large amounts of data
  - Think about ML, data mining applications
- Many of these are multi-stage and interactive applications
  - Iterative machine learning & graph processing
  - Interactive data mining

Challenges

1. Writing/Reading to persistent storage is slow
    - 10-100x slower than memory*
2. Replicating data can be slow
    - Takes time if you do it synchronously
        - i.e., you wait for operations to be replicated before proceeding

How to achieve both fault tolerance + speed?

*Latency numbers every programmer should know, check out: https://norvig.com/21-days.html#answers & https://colin-scott.github.io/personal_website/research/interactive_latency.html

Before Spark: MapReduce

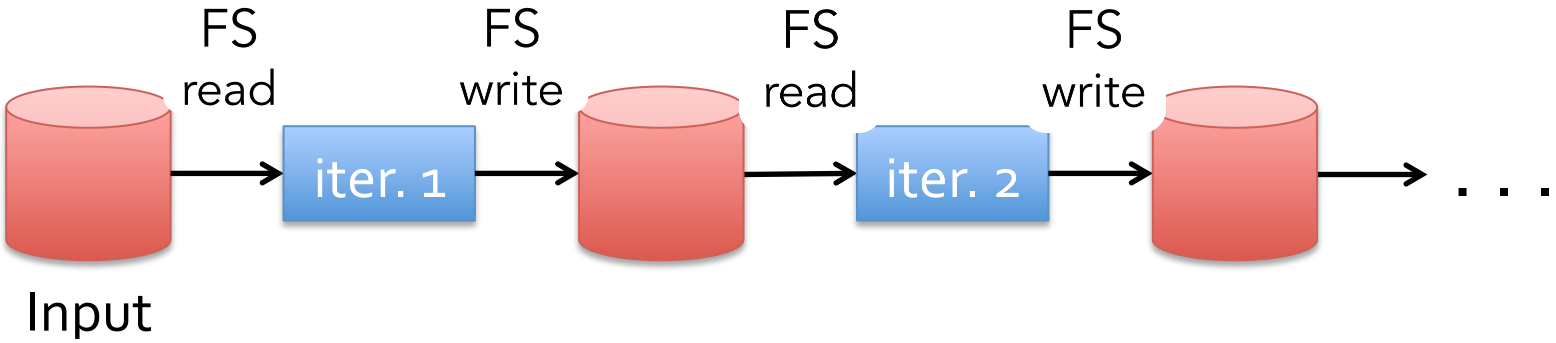
- Made life of programmers easy
  - Provides a simple programming model for big data analysis on large unreliable clusters
- MapReduce framework handles
  - Communication between nodes
  - Scheduling of tasks
  - Failures and stragglers

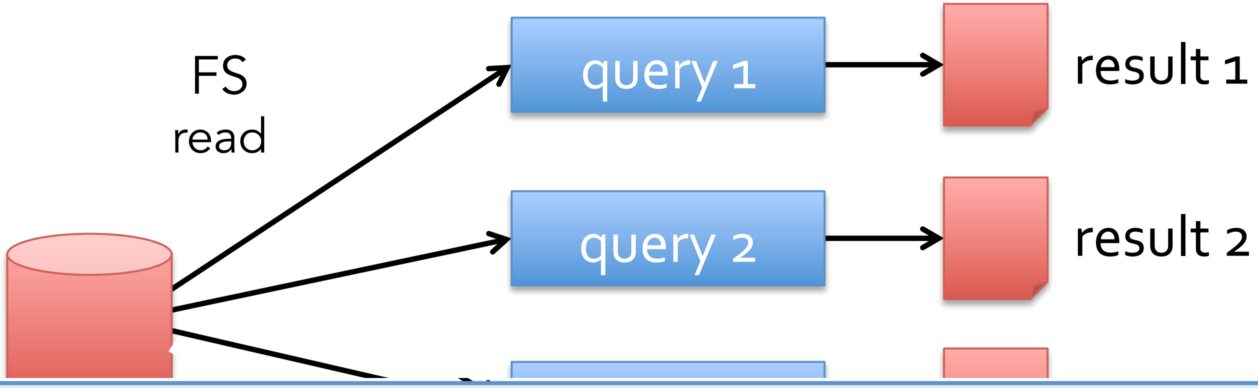
MapReduce Limitations

- But restricted programming model
  - Some apps don't fit well with MapReduce
- As soon as it got popular, users wanted more:
  - More complex, multi-stage applications (e.g., iterative machine learning & graph processing)
  - More interactive ad-hoc queries
- The only way to share data across jobs in MapReduce was through persistance storage

Examples

FS = Distributed File System





Slow due to replication and disk I/O,
but necessary for fault tolerance

A Strawman Solution

- Store and replicate the data in memory
    - E.g., data is stored in RAM, and replicated on the RAM of multiple servers
- Speed: get data from RAM which is fast
- Fault tolerance: if a server fails, use a copy from another server
- Any Issues?

Another Possible design

- Only store input data in persistent memory
  - And replicate it
  - Store in a distributed file system,e.g., Google File System
- Store modified data in RAM
- Only replicate the log of updates
  - Store them in persistent storage & replicate them
  - If you lose modified data, just recreate by reapplying the updates to the orginal data
- Any Issues?

Spark Key Ideas

- Redesign storage interfaces
  - Instead of fine-grained storage interfaces, e.g., key/value storage system or SQL-based systems
  - Design coarse-grained operations
    - That apply to a number of data elements
      - So you have to log less data --> matters when dealing with large data
- A new data-sharing primitive
  - Resilient Distributed Datasets (RDDs):
    - Immutable and created through transformations
  - Store RDDs in memory + track graph of transformations
    - If a server fails, recreate lost data from the graph of transformations

RDDs

- Immutable, partitioned collections of records
- Can only be built through coarse-grained deterministic transformations (map, filter, join, …)
- Efficient fault recovery using *lineage*
    - Log one operation to apply to many elements
    - Recompute lost partitions on failure
    - No cost if nothing fails

Example

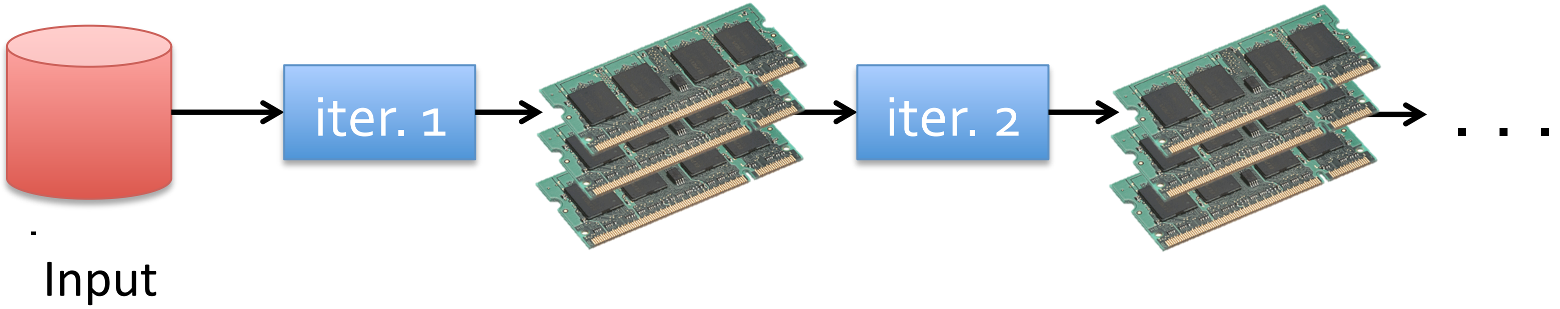
1. lines = spark.textFile(”gfs://…")
2. errors = lines.filter(_.startsWith("ERROR")) // lazy!
3. errors.persist() // no work yet
4. Errors.count() // an action that computes a result

- now errors are materialized in memory
- partitioned across many nodes
- Spark, will try to keep in RAM (will spill to disk when RAM is full)

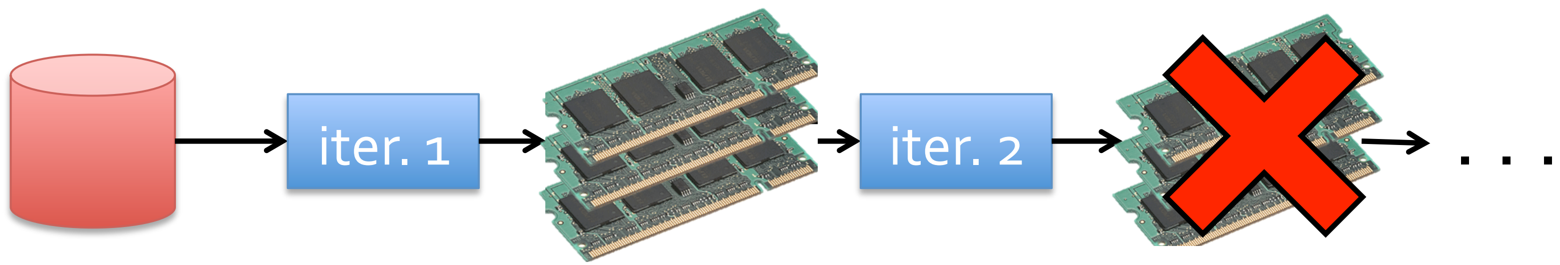
Reuse of an RDD

- errors.filter(_.contains("MySQL")).count()
- This will be fast because reuses results computed by previous fragment
- Spark will schedule jobs across machines that hold partition of errors

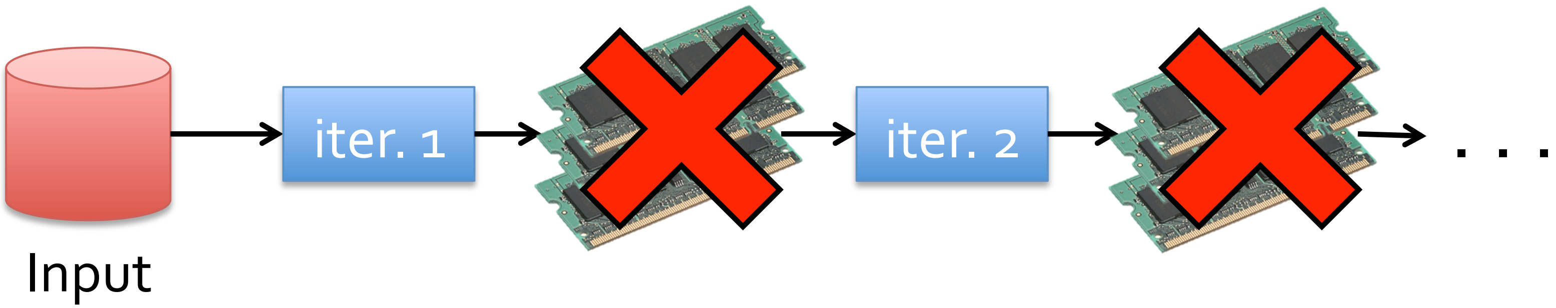
RDD Recovery



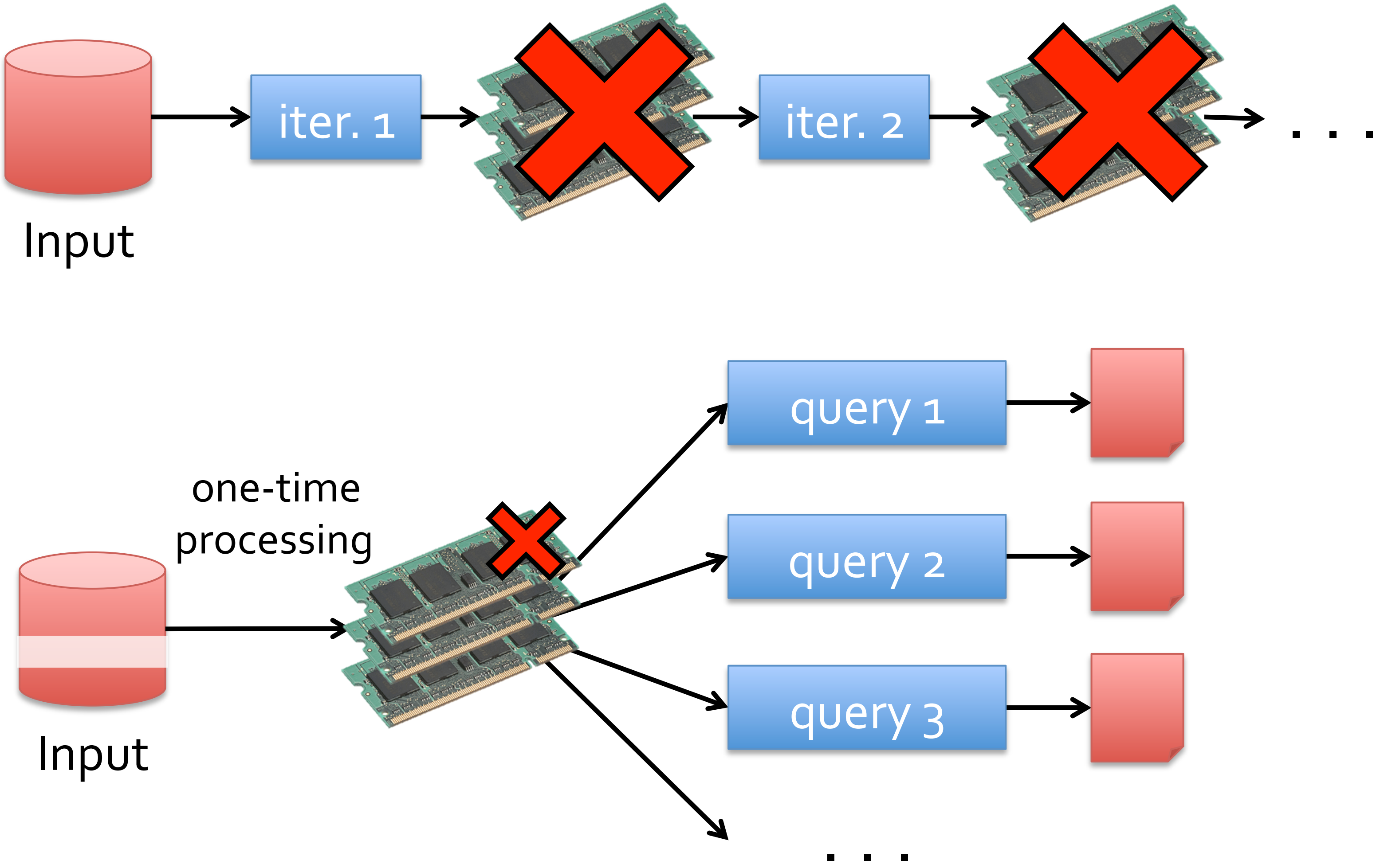
RDD Recovery



RDD Recovery



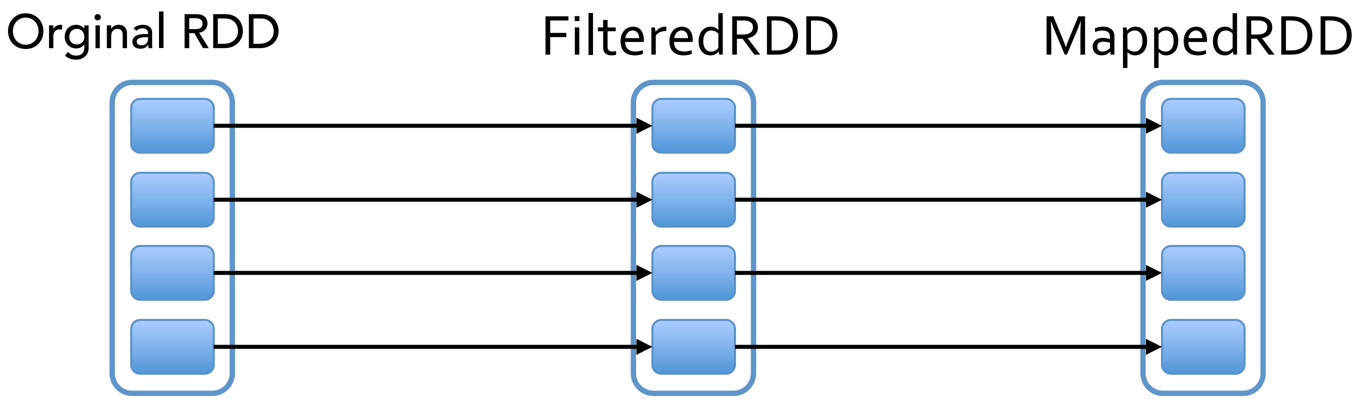
RDD Recovery



Fault Tolerance

- RDDs track the graph of transformations that built them (their *lineage*) to rebuild lost data

E.g.:
```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```



Lineage and Fault Tolerance

- Opportunity for efficient fault tolerance

- Let's say one machine fails
  - Want to recompute *only* its state
  - The lineage tells us what to recompute
  - Follow the lineage to identify all partitions needed

- Who tracks the lineage?

Generality

- Despite their restrictions, RDDs can express surprisingly many parallel algorithms
- Unify many current programming models
  - Such as MapReduce, Dryad, SQL, Pregel
- Support new apps that these models don't

Limitations

- Suited for batch applications
  - Apply the same operation to many elements of the dataset
  - Sparks remembers each transformation as one step
  - Can recover without having to log large amounts of data
- Not suitable for apps that require fine grained updates. For such apps more suitable to have
  - Key value stores
  - Classical databases

Discussion

- What happens if RAM is full?

Discussion

- What happens if the lineage chain is long?
    - Is it possible failure recovery may take a long time?

Spark Summary

- Keep intermediate data in memory
  - To provide fast access
  - But makes fault tolerance hard
- Proposes RDDs
  - Immutable → to simplify failure recovery
  - Instead of saving data (on persistence storage), track lineage graph
  - Lineage graph → graph of transformations
  - Transformations are course grained
    - Applied to several elements of data
    - Saves the data that needs to be logged; matters when dealing with huge amounts of data
  - Limitation: only supports coarse grained operations

Next Lecture

- Distributed Parameter Server