CPSC 416 Distributed Systems

Winter 2022 Term 2 (March 23, 2023)

Tony Mason (fsgeek@cs.ubc.ca), Lecturer

Logistics

Teaching Assistants

Andy Hsu (andy.hsu@alumni.ubc.ca)

Hamid Ramezanikebrya (hamid@ece.ubc.ca)

Jonas Tai (jonastai@student.ubc.ca)

Cathy Yang (kaiqiany@student.ubc.ca)

Office Hours

Remember: Use Piazza for **all** official course-related communications

- Not on Piazza? Not official.
- Canvas "comments/messages" **are not monitored**

Office Hours:

Who	When	Where
Tony	Monday 14:00-15:00 Wednesday 16:00-17:00	Discord
Andy	Thursday 19:00-20:30	Discord
Hamid	Friday 16:30-18:00	Kaiser 4075
Jonas	Thursday 13:00-14:00	Online (see Piazza)
Cathy	Friday 09:00-10:30	X237

Self-Assessment

This week

- DP3 Implementation Report (Thu @ 23:59)

Next week

- Capstone Status Report (Tue @ 17:00)
- DP3: Peer Review Implementation Reports (Thu @ 17:00)
- Note: no self-assessment activity

Note:

- You are strongly encouraged to collaborate with others on this
- You should use tools at your disposal to answer these questions
- **Do not forget to submit it.**

Today’s Failure

Software Upgrade Failures in Distributed Systems

This is not about just a *single* failure, but a common class of failures.

> Upgrade is one of the most disruptive yet unavoidable maintenance tasks that undermine the availability of distributed systems. Any failure during an upgrade is catastrophic, as it further extends the service disruption caused by the upgrade. The increasing adoption of continuous deployment further increases the frequency and burden of the upgrade task. In practice, upgrade failures have caused many of today's high-profile cloud outages. Unfortunately, there has been little understanding of their characteristics.

Understanding and Detecting Software Upgrade Failures in Distributed Systems

[Presented at SOSP 2021]

Cassandra	HBase	HDFS	Kafka	MapReduce	Mesos	Yarn	ZooKeeper
44	13	38	7	1	8	8	4

Table 1. Numbers of upgrade failures we analyzed.

By analyzing upgrade failures of 123 failures, the authors created:

- Insight into severity, root causes, exposing conditions, and fix strategies
- DUPChecker: a testing framework for upgrade class failures
    - It identified 20 previously unknown upgrade failure causes in 4 distributed systems

	Failure	From	To	C.?	Cause
Cassandra	15794	3.11.4	4.0	✓	Data-syntax Incomp.
	16258	3.11.6	4.0		Data-syntax Incomp.
	16301	3.11.9	4.0	✓	Code Incompatibility
	16292	3.0.0	3.2.0		Data-syntax Incomp.
	16257	2.1.0	2.2.0		Data-syntax Incomp.
	16264	2.0.0	2.1.0		Data-semantics Incomp.
	16265	2.0.0	2.1.0		Data-syntax Incomp.
	16266	2.0.0	2.1.0	✓	Data-syntax Incomp.
	16267	1.1.0	1.2.0	✓	Data-semantics Incomp.
	16268	1.1.0	1.2.0		Data-syntax Incomp.
	16269	1.1.0	1.2.0		Data-syntax Incomp.
HBase	25239	2.3.3	3.0		Broken Upgrade Op.
	24430	2.2	2.4		Broken Dependency
	24556	2.2	2.3	✓	Broken Dependency
	25238	2.2.0	2.3.3	✓	Data-syntax Incomp.
	25259	2.1.1	2.2.0		Broken Upgrade Op.
	25260	2.0.6	2.1.1		Broken Upgrade Op.
Kafka	10041	1.1	2.4	✓	Broken Dependency
Hive	24440	2.3.7	3.0.0		Data Syntax Incomp.
	24493	2.1.1	2.3.7		Upgrade Operation

Table 5. *DUPTester*'s result on real-world systems. Failure number is the report ticket number on JIRA. *C.?*: whether the bug is already confirmed by developers.

Lesson Goals

Distributed Data Analytics

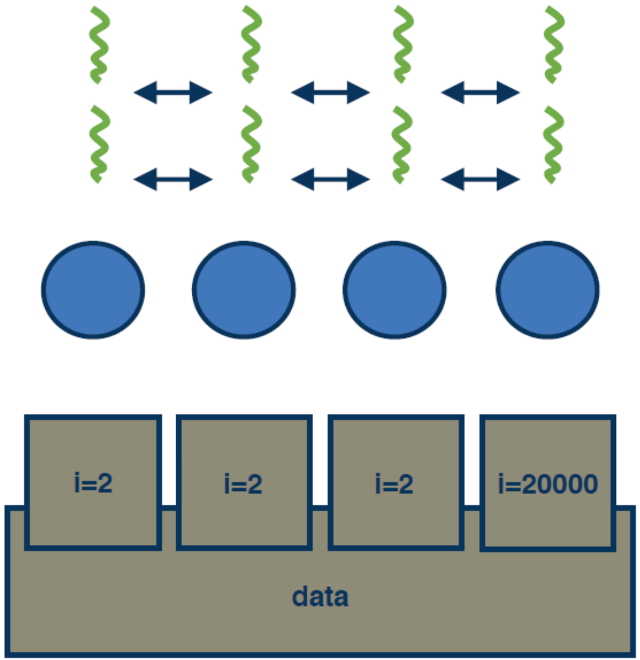
MapReduce

Spark (and RDDs)

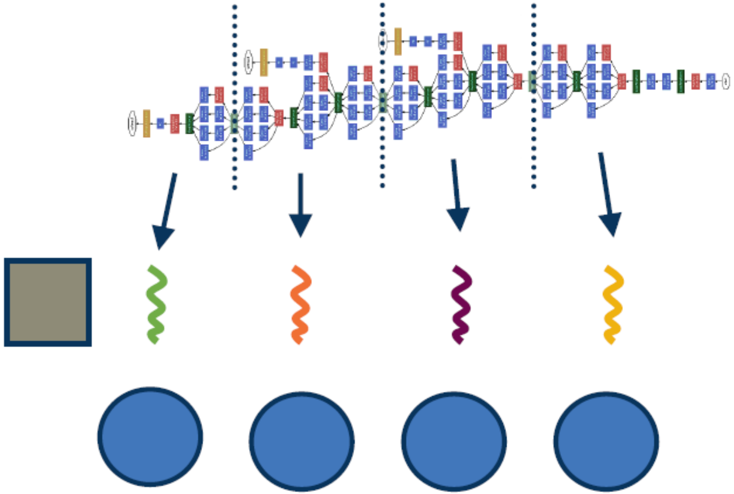
Common Techniques

Data Parallel (Divide & Conquer):

- Divide Data across nodes
- Load balancing, decomposition
- Messaging for data dependencies
- Application usage



Common Techniques



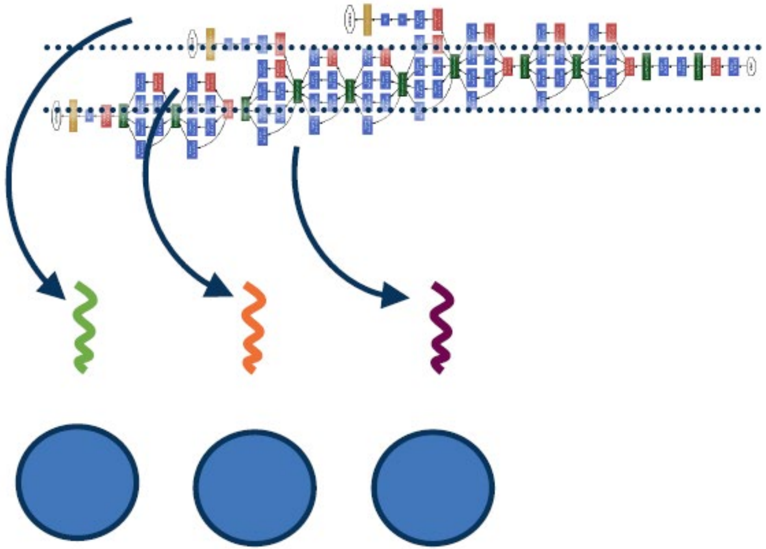
Pipelining

- Divide work into smaller tasks
  - Small number of tasks per node
  - Faster than generality
- Data streamed in chunks through task pipeline
- Increases throughput

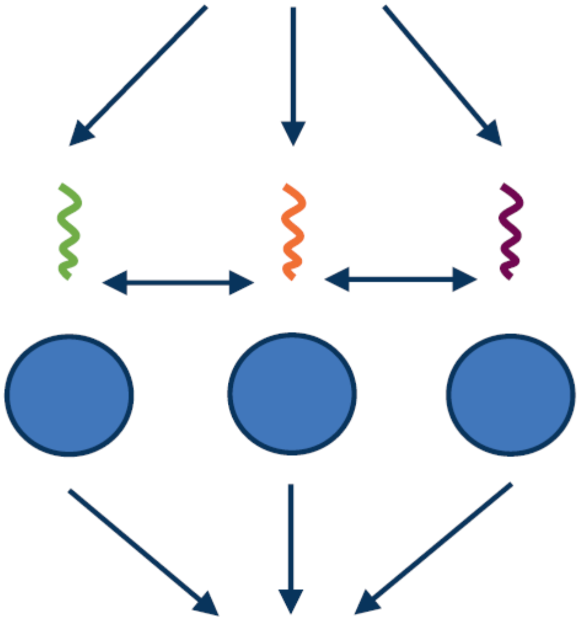
Common Techniques

Model Parallelism

- Divide state across nodes
- Less processing per node
- Input passed to all nodes
- Output combined from all nodes
- Must handle dependencies



Common Techniques



Model Parallelism

- Divide state across nodes
- Less processing per node
- Input passed to all nodes
- Output combined from all nodes
- Must handle dependencies

MapReduce

[MapReduce: Simplified Data Processing on Large Clusters, J. Dean, OSDI 2004.]

- Hadoop MapReduce
- AWS infrastructure

MapReduce

Input:

- Set of key-value pair records

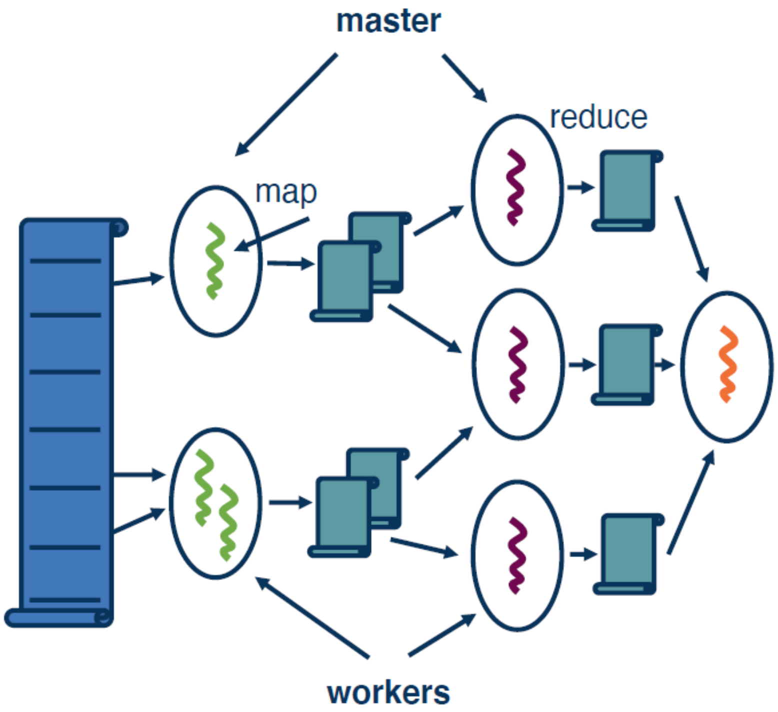
Map function

- Input: unique key-value pair
- Output: a new key-value pair

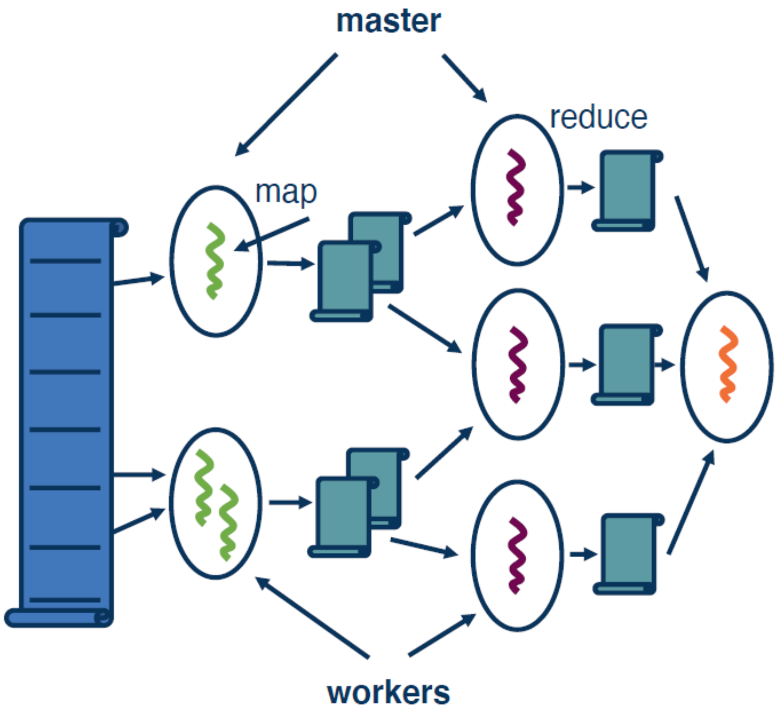
Reduce function:

- Input: output from map function
- Output: final result

Master: orchestrates workers, I/O, failure management



MapReduce



Wordcount example:

- Input: Collection of files

Map function:

- Input: File, key=filename, content=value
- Output: file with key=word, value=list of counts

Reduce function:

- Input: file with key=word, value=list of counts
- Output: list of words with total counts

Other examples:

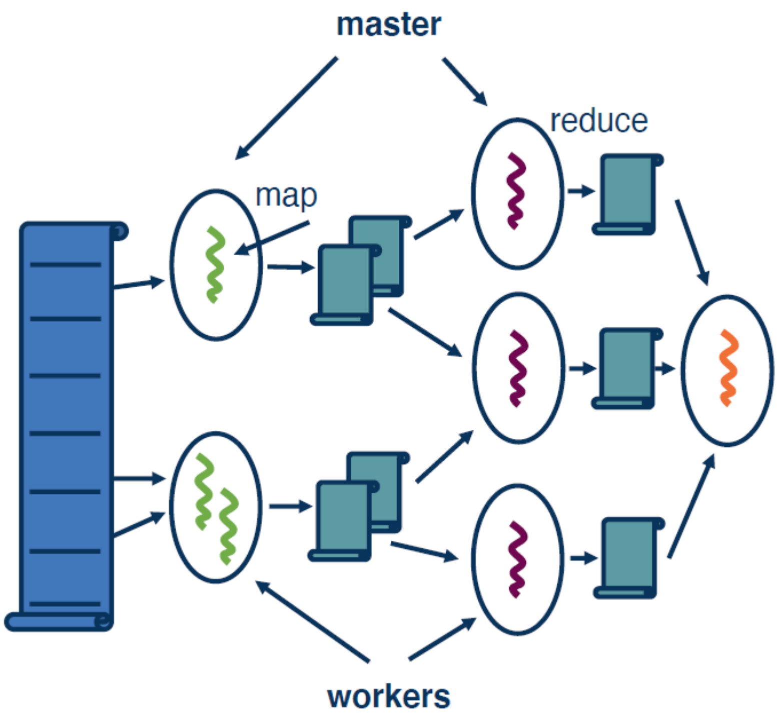
- URL access frequency, page rank, inverted word index

MapReduce

Combining Techniques:

- Data parallel: chunks to mappers
- Pipelining: mapper to reducer
- Model parallelism: reducers process parts of key space, combine

Dataflow model means flow of data determines execution



Map Reduce: Design Decisions

Master data structures:

- Tracking

Locality:

- Scheduling, placement of intermediate data

Task granularity:

- Finer granularity: more flexibility, management operation execution time
- Coarse granularity: lower management overhead

Fault tolerance:

- Master: standby replication
- Worker: detect failures or stragglers and re-execute

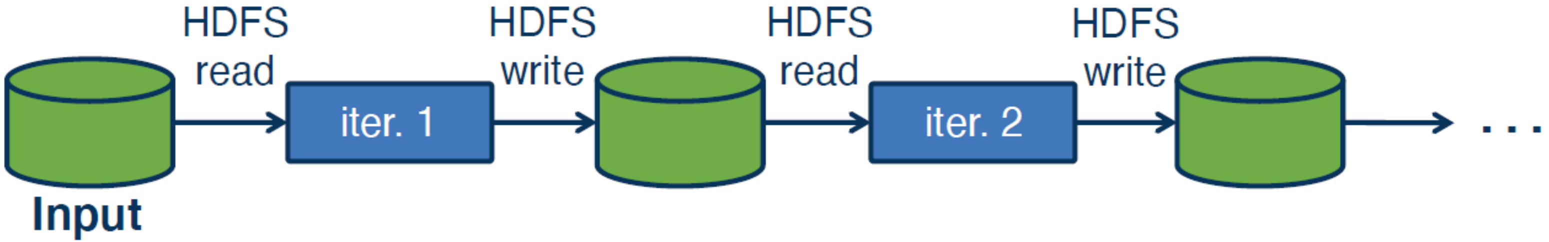
Failure semantics:

- Importance of Consistency and complete results

Backup tasks:

- Inevitable failures: speculative task backup

Map Reduce: Limitations



Failure inevitable: cannot re-execute entire operation

Fault-tolerant mechanism: requires intermediate data availability

- Serialiation to/from persistent storage
- Remote access and data movement

Data amplification:

- Intermediate data may be much larger than input
- Executions are iterative
- Storage level replication

System scale: cannot assume best-in-class storage devices

Spark

[Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing]

Faster analytics (10x) versus Hadoop

- Workloads: graph, streaming, SQL, Machine Learning, etc.
- Languages: Java, Python, Scala, etc.
- Platforms: AWS, Kubernetes, etc.

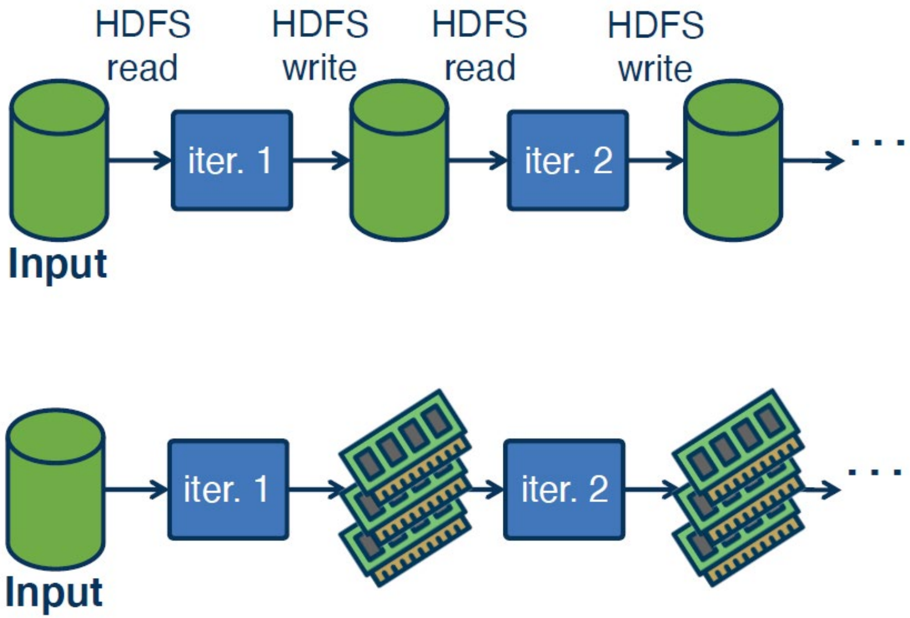
[Apache Spark]

Spark: Goals

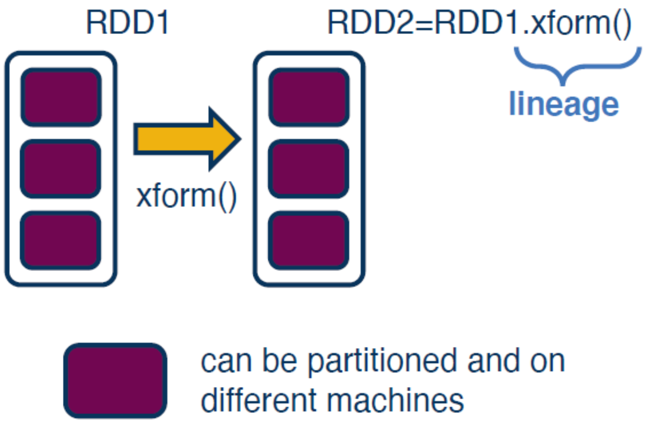
Allow in-memory data sharing

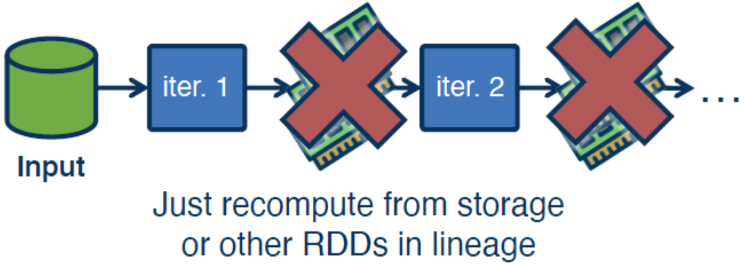
- Fast DRAM versus slow hard disk
- No serialization cost

Fault-tolerant



Resilient Distributed Datasets: Introduction





Immutable partitioned record collection

Created using transformations

- Operations on data in stable storage
- Map/join/filter on other RDDs

Used via actions (count, collect, save)

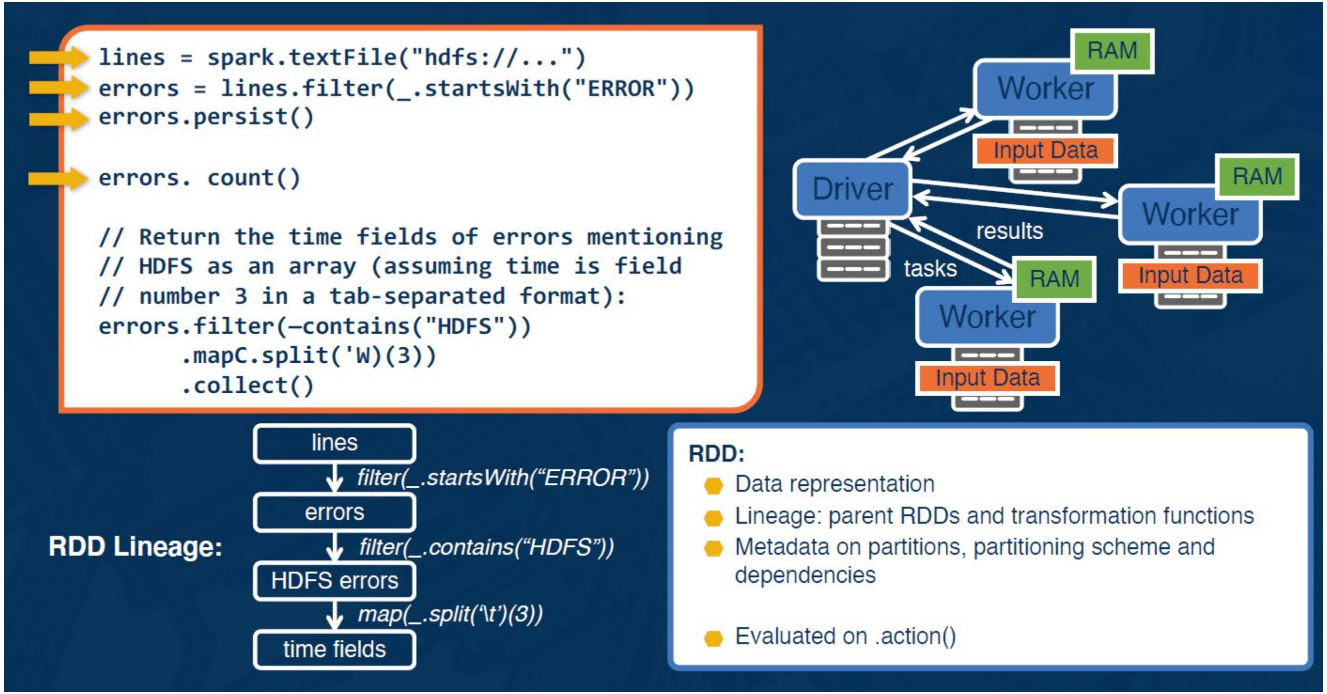
RDDs map back to source

- Compute partitions from data in stable storage

Users control persistence and partitioning

Resilient Distributed Datasets: Example

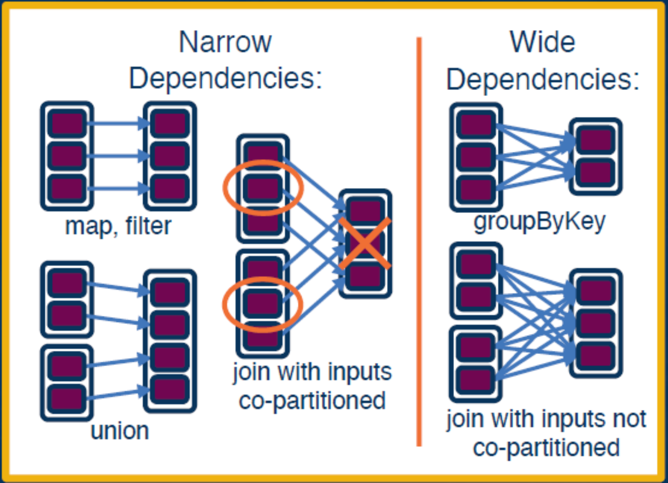
Console log mining example



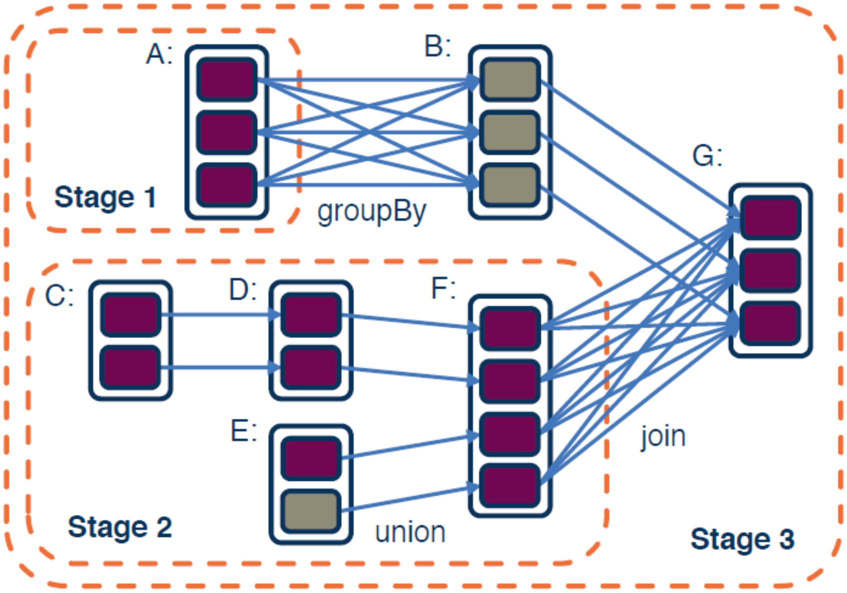
Resilient Distributed Datasets: Transformations & Actions

Transformations	$map(f : T \Rightarrow U)$	:	RDD[T] $\Rightarrow$ RDD[U]
	$filter(f : T \Rightarrow Bool)$	:	RDD[T] $\Rightarrow$ RDD[T]
	$flatMap(f : T \Rightarrow Seq[U])$	:	RDD[T] $\Rightarrow$ RDD[U]
	$sample(fraction : Float)$	:	RDD[T] $\Rightarrow$ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, Seq[V])]
	$reduceByKey(f : (V, V) \Rightarrow V)$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) $\Rightarrow$ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) $\Rightarrow$ RDD[(T, U)]
	$mapValues(f : V \Rightarrow W)$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, W)] (Preserves partitioning)
	$sort(c : Comparator[K])$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$partitionBy(p : Partitioner[K])$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
Actions	$count()$	:	RDD[T] $\Rightarrow$ Long
	$collect()$	:	RDD[T] $\Rightarrow$ Seq[T]
	$reduce(f : (T, T) \Rightarrow T)$	:	RDD[T] $\Rightarrow$ T
	$lookup(k : K)$	:	RDD[(K, V)] $\Rightarrow$ Seq[V] (On hash/range partitioned RDDs)
	$save(path : String)$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.



Resilient Distributed Datasets: Scheduling Action Execution



Program defines dependencies

Actions:

- Directed acyclic graph (DAG)
- Minimize dependencies
- Optimize parallelism
- Limit I/O contention

Tasks assigned based on data locality

Spark: Goals

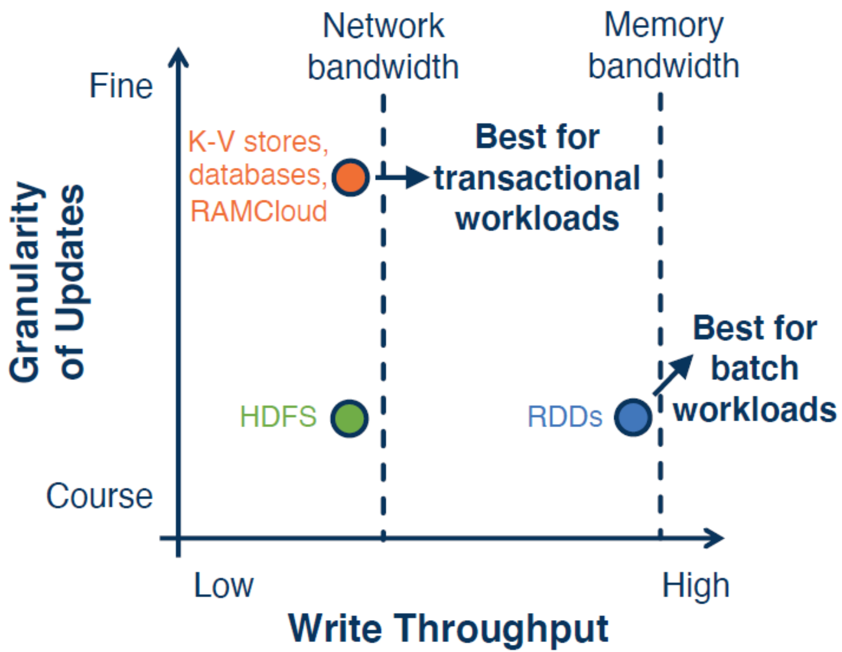
Data in memory?

- Distributed shared memory like runtime
  - Log updates
  - Persist lineage

Log coarse grained operations applied to all items in RDD elements

- (+) less data to persist in execution critical path
- (+) read data as low as once, less slow storage I/O
- (+) more control on locality
- (−) recovery time

Spark: Goals



Data in memory?

- Distributed shared memory like runtime
  - Log updates
  - Persist lineage

Log coarse grained operations applied to all items in RDD elements

Spark: Evaluation

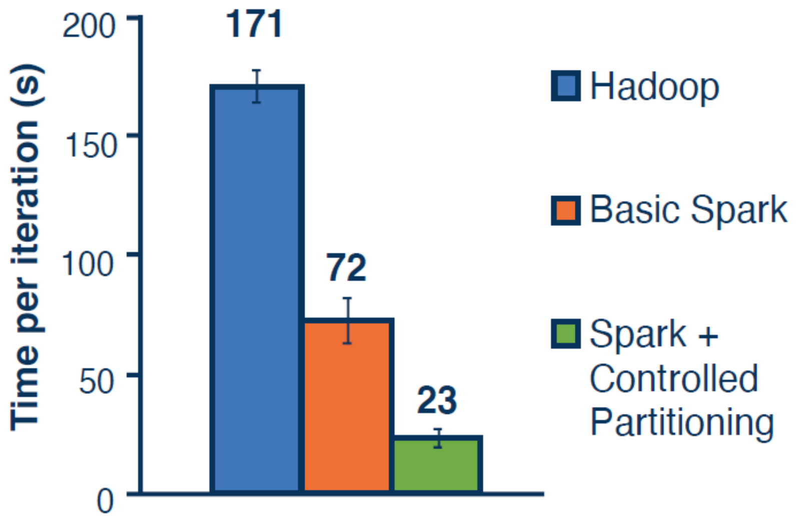
Up to 20x better than Hadoop

- Iterative
- Machine learning
- Graph applications

Analytics report generation 40x

Rapid failure recovery

1TB dataset queries with 5-7 second latencies



Lesson Review

Distributed Data Analytics

Systems for scalable data processing

MapReduce

Spark

Questions?

THE UNIVERSITY OF BRITISH COLUMBIA