Paper by Matei Zaharia et al

Presented by Jiaxing Yang

Spark

- A unified analytics engine for general purpose data processing
  - Iterative Machine Learning
  - Page-rank computation
  - Data mining
  - Etc.
- The original team of the Spark project founded Databricks in 2013
- RDD is a core concept in Spark

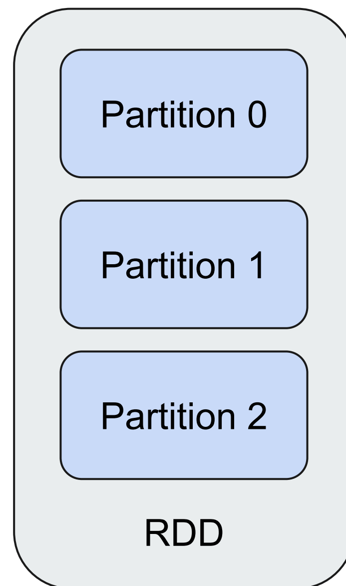
Motivation

- Cluster computing frameworks are widely adopted for large-scale data analytics
- MapReduce and Dryad lack abstractions for leveraging distributed memories
    - Shared data through disks are inefficient
    - Bad for tasks needs to reuse data
- Distributed shared memory, key-value stores, databases, and etc offer fine-grained shared state update interfaces
    - Fault tolerance requires replication -- expensive for data intensive tasks
- **Need an efficient, fault-tolerant method for data sharing through memory**

RDD Abstraction

RDD is a read-only, partitioned collection of records:

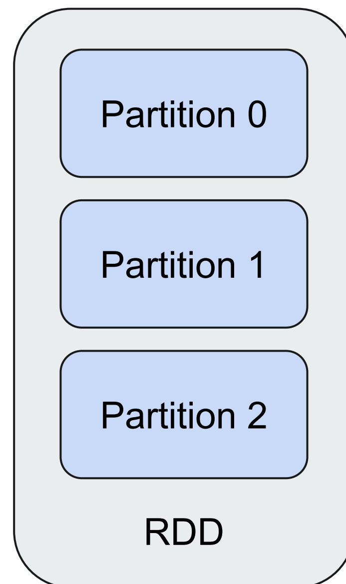
- Read-only: RDDs are immutable once generated
- Partitioned: An RDD consists of multiple partitions
  - Partitions can be stored by different machines



RDD Abstraction

RDDs can only be created through deterministic operations on either (1) data in stable storage or (2) other RDDs

- Such deterministic operations are called *transformations*
  - Coarse-grained operations
  - *map, filter, join, and etc.*
- An RDD has enough information called *lineage* to compute itself from stable data



Spark Programming Interface

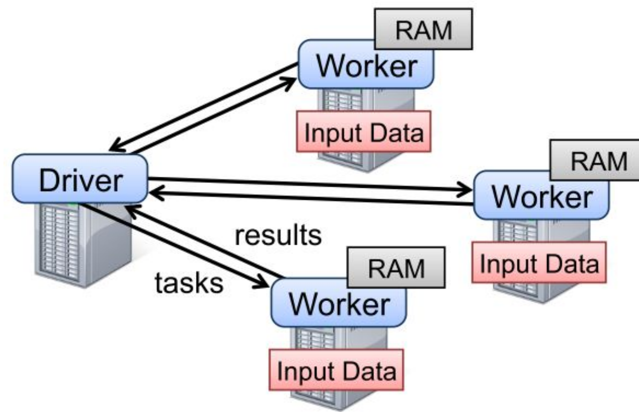
Spark exposes RDDs through a set of APIs:

- Define RDDs with *transformations*
- Use data by *actions* on defined RDDs
    - *count, collect, …*
- Execute *transformations* only when being used in an action
    - To pipeline *transformations*

Spark Programming Interface

To use Spark:

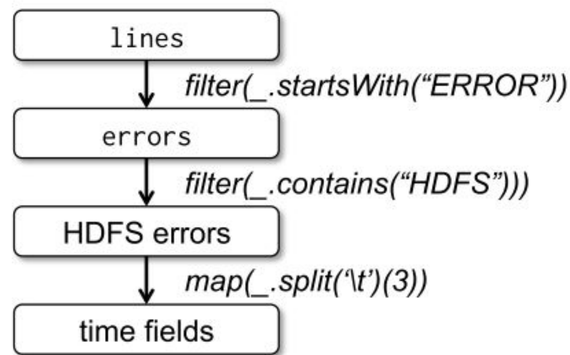
- Developers write a driver program
  - Connect to workers
  - Track RDDs' lineage
  - Assume no failure
    - Actually easy to be replicated



Spark Programming Interface

Example: Console Log Mining

```
1.   lines = spark.textFile("hdfs://...")
2.   errors = lines.filter(_.startsWith("ERROR"))
3.   errors.persist()
4.   errors.count()
5.   // Count errors mentioning MySQL:
6.   errors.filter(_.contains("MySQL")).count()
7.   // Return the time fields of errors mentioning
8.   // HDFS as an array (assuming time is field
9.   // number 3 in a tab-separated format):
10.  errors.filter(_.contains("HDFS")).map(_.split('\t')(3)).collect()
```



Spark Programming Interface

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

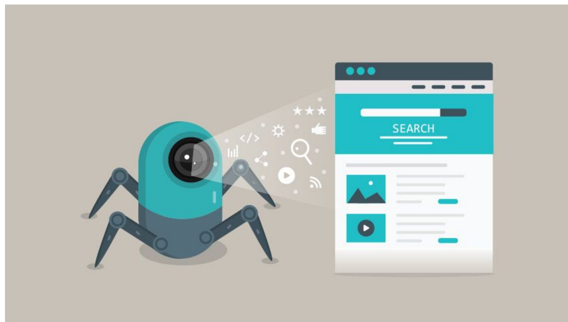
RDD Advantages

- Coarse-grained transformations allow efficient fault tolerance
    - No checkpoint required
    - Only lost partitions need recomputation
- RDDs' immutable nature enable slow nodes mitigation
    - Run backup tasks
    - Hard to do with DSM because copies of the task will access the same addresses and can interfere with each other

Aspect	RDDs	Distr. Shared Mem.
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Applications Not Suitable for RDDs

- Applications requiring fine-grained updates on shared state.
  - Web crawler



Representing RDDs

Spark proposes representing RDDs through an interface with:

- A set of partitions
- A set of dependencies on parent RDDs
- A function for computing the dataset from the parents
- Metadata about partitioning scheme and data placement

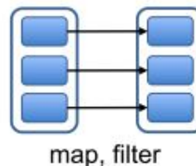
Operation	Meaning
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p*, *parentIters*)	Compute the elements of partition *p* given iterators for its parent partitions
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

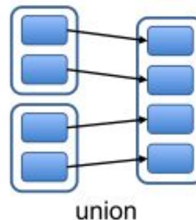
Representing RDDs

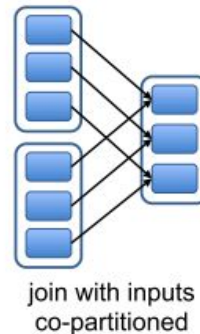
Dependencies between RDDs

- Narrow dependencies
  - One parent partition is used by at most one child partition
  - *map, filter, etc.*
- Wide dependencies
  - One parent partition is used by multiple child partitions
  - *join, etc.*

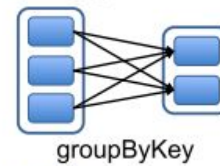
Narrow Dependencies:

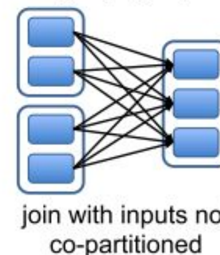






Wide Dependencies:



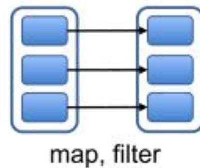


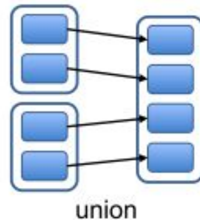
Representing RDDs

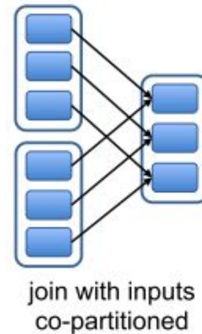
Narrow dependencies are preferred:

- Narrow dependencies:
  - Pipelined execution on one cluster node to compute the parent partition
  - Only the lost parent partitions need to be recomputed
- Wide dependencies:
  - All parent partitions are required to be available and and to be shuffled across nodes
  - A node failure may cause a complete re-execution

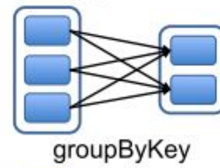
Narrow Dependencies:

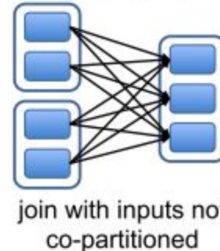






Wide Dependencies:



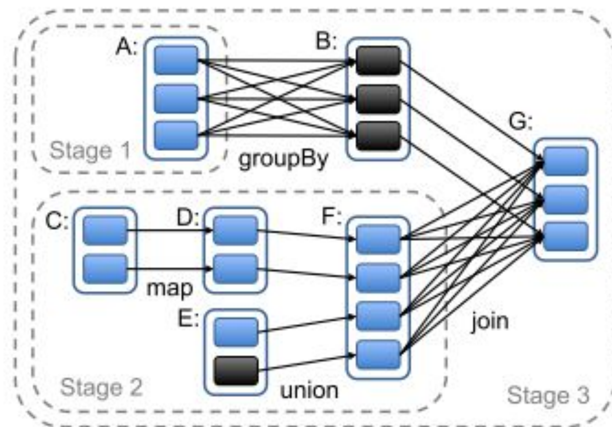


Spark Implementation

- ~14000 lines of Scala when paper published
  - Job scheduler
  - Spark interpreter
  - Memory management
  - Support for checkpointing

Job Scheduling

- Build a DAG of stages from RDD's lineage graph
  - Boundaries of stages are
    - wide dependencies
    - already computed partitions
- Assign tasks based on locality
  - Send a task to where the data are
- For wide dependencies, materialize intermediate records
  - Easier fault recovery



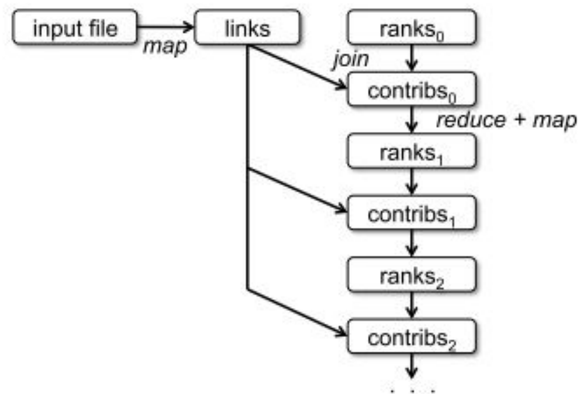
Memory Management

Three options for storage of persistent RDDs:

- In-memory, deserialized objects
  - Fastest performance
- In-memory, serialized data
  - Memory efficiency
- On-disk
  - Limited memory
- LRU for partition eviction
  - Evict a partition of the least recently used RDD
    - Unless is the same RDD of the newly computed partition

Support for Checkpointing

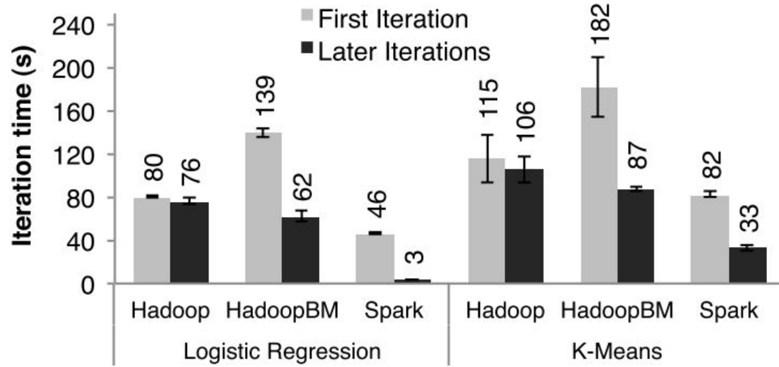
- Checkpointing is helpful for RDDs with very long lineage graphs and wide dependencies
    - Hard to recompute
- For RDDs with short lineage graphs and narrow dependencies, checkpointing may never be worthwhile
    - Efficient to recompute
    - Disk I/O and usage may be too expensive
- RDDs are read-only
    - Be written out in the background

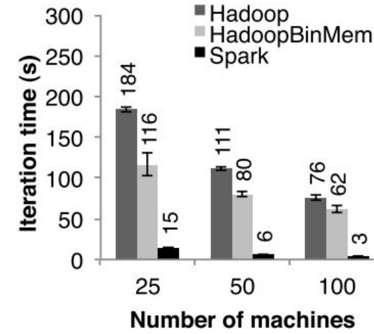


Evaluation

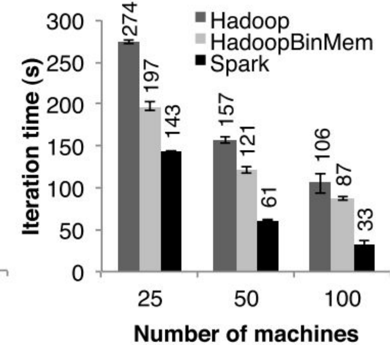
- Spark outperforms Hadoop by up to 20x in iterative machine learning and graph applications
    - Speedup comes from avoiding I/O and deserialization
- Customized applications perform and scale well with Spark
- Spark recover quickly when nodes fail
- Spark can be used to query 1-TB data in 5 - 7s

Iterative Machine Learning Applications





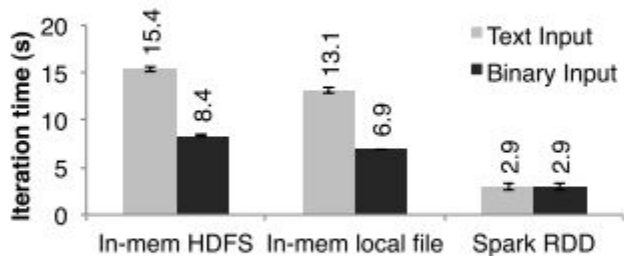
(a) Logistic Regression



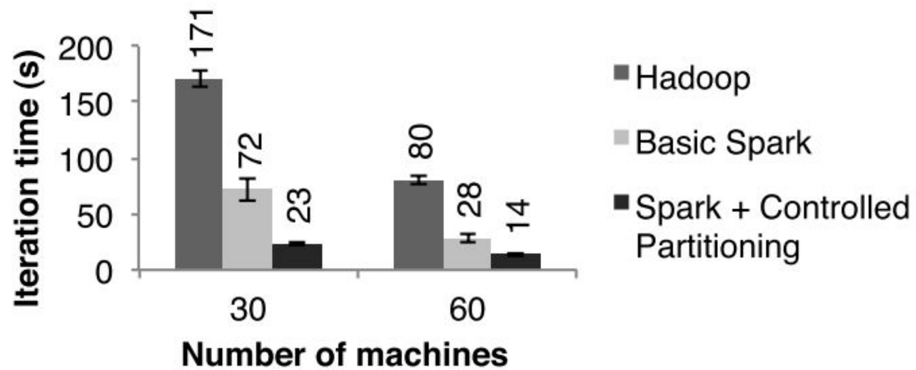
(b) K-Means

Iterative Machine Learning Applications

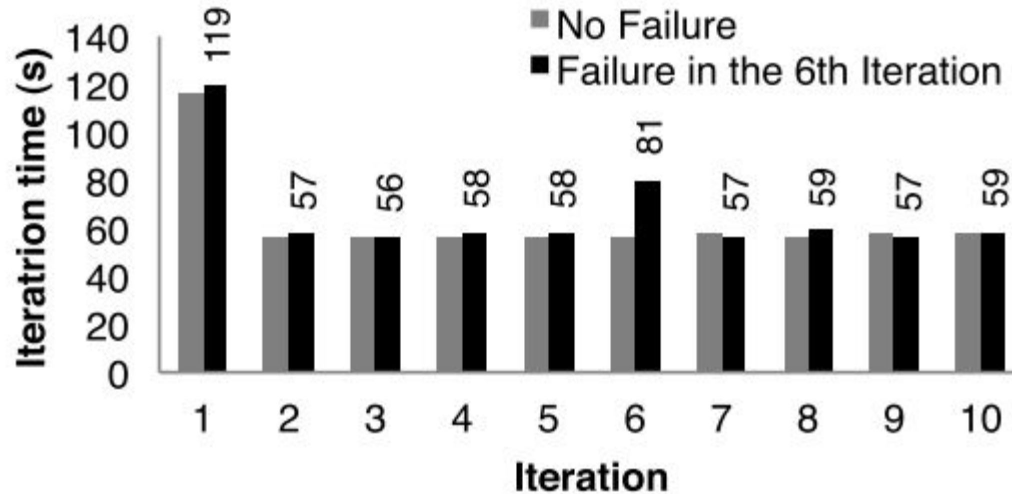
- Minimum overhead of the Hadoop software stack
  - ~25s minimum overhead
- Overhead of HDFS while serving data
  - Memory copies, checksum
- Deserialization cost to convert binary records to usable in-memory Java objects



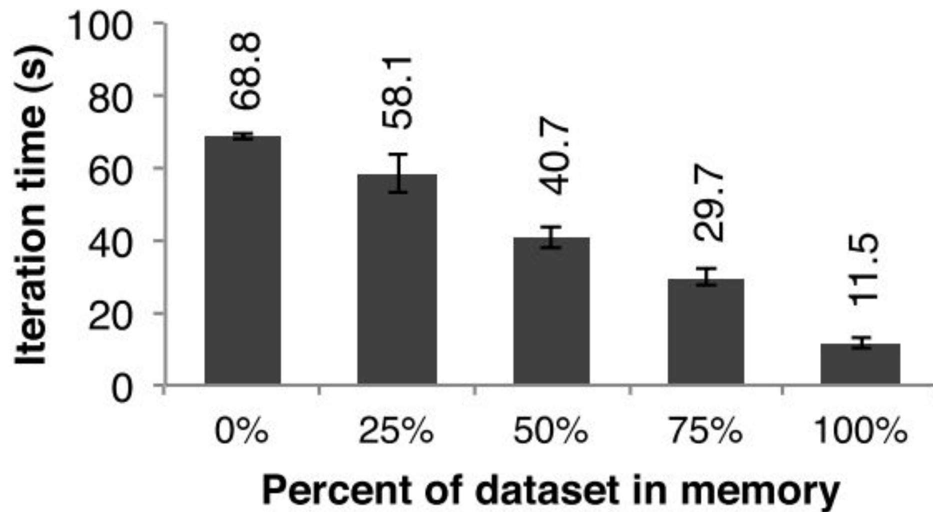
Page Rank



Fault Recovery

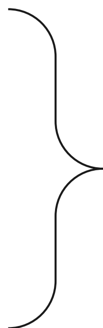


Behavior with Insufficient Memory



Expressing Existing Programming Models

- MapReduce
- DryadLINQ
- SQL
- Pregel
- Iterative MapReduce
- Batched Stream Processing



Apply the same operation to many records

Conclusion

- Resilient distributed dataset (RDD)
    - Efficient, general-purpose, fault-tolerant data abstraction
    - Can express a wide range of parallel applications
    - Use coarse-grained transformations to recover from faults
    - Implemented in Spark that outperforms Hadoop

The End

Thank You!