Candoia: A Platform for Building and Sharing Mining Software Repositories Tools as Apps

Nitin Mukesh Tiwari

Department of Computer Science
Iowa State University
nmtiwari@iastate.edu

- Problem
  - Building easily customizable, adoptable and applicable mining software repository tools
- Solution
  - An ecosystem which offers suitable abstractions and computational means to realize the process for building and sharing MSR tools as apps.
- Evaluation
- Related works, Conclusion, & Future Work
  - Existing open source tools and frameworks
  - Open source datasets

Goal

- Reduce the efforts required to build MSR tools
- Ease the process of adopting, customizing and sharing MSR tools
- Allow users to run third-party tools more securely

Scenario 1: MSR Tool Building and Sharing

User wants to build a tool for mining

- Source code
  - Java source code
- Version control systsm(VCS)
  - Subversion (SVN)
- Bug Information
  - BugZilla Bug Tracker

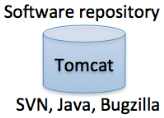


Figure: Software Repository Data

Scenario 1: Tool Building and Sharing

Build a tool for mining the source code, version data, and bugs

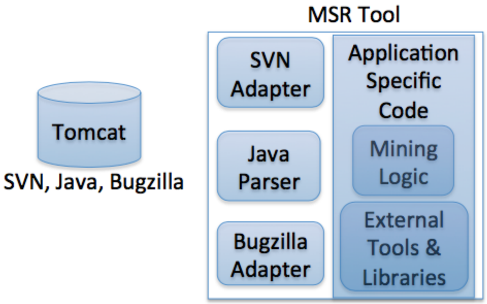


Figure: Repository Mining Tool Building

Scenario 1: Tool Building and Sharing

Share the built tool with other researchers and practitioners

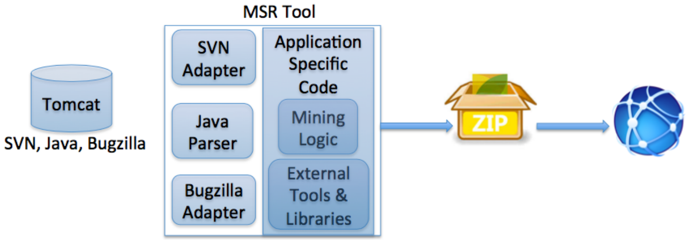


Figure: Complete process of building and sharing tool

Challenges: Scenario 1 MSR Tool Building and Sharing

- User is required to build necessary data preperation tools

Challenges of building and sharing MSR tool

- User is required to build necessary data preperation tools
- Tool are tightly coupled to other tools and particular SCM systems

Scenario 2: Adopting a shared tool

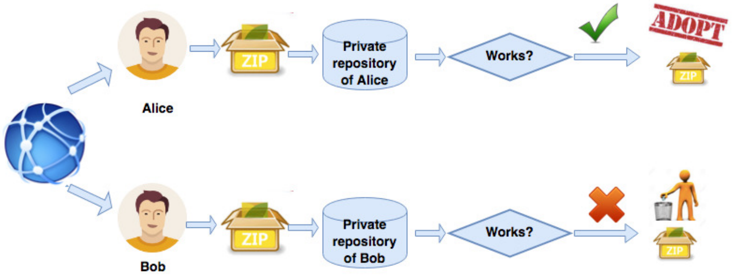


Figure: Repository Mining Tool Building

Scenario 2: Adopting a shared MSR tool

Why Alice was able to adopt but not Bob?
&
What are the possible points of failure?

How MSR tools are build?

- MSR tools are build for specific project setting.
- A project setting defines types and sources of various MSR artifacts [1].

[1]MSR artifacts include Revision history from version control system (VCS), Source code of programming language(s), Bug data from bug trackers, Project metadata, users and teams data from forges

Potential Points of failure

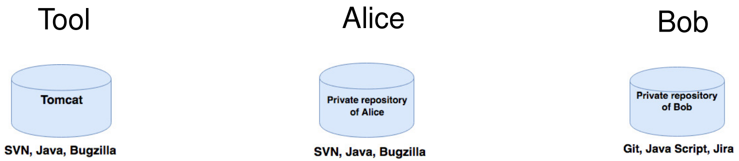
- MSR tools are build for specific project setting.
- A project setting defines types and sources of various MSR artifacts [2].

Failure Cause

An MSR tool build for one project setting may not work for different project settings.

[2]MSR artifacts include Revision history from version control system (VCS), Source code of programming language(s), Bug data from bug trackers, Project metadata, users and teams data from forges

Failure: Mismatched Project Settings



Reusability of published MSR tools[3]

- Y = Replicable
- N = Not replicable
- P = Partially replicable

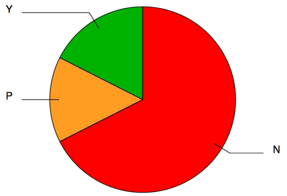
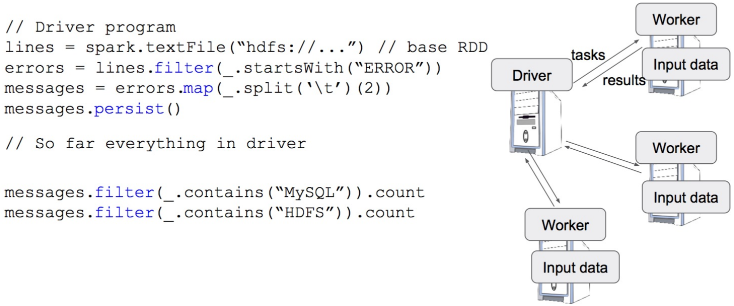


Figure: Replicability of *MSR*[3]

[3]Replicating MSR:A study of the potential replicability of papers, Gregorio Robles et.al, MSR 2010

Resilient Distributed Datasets (RDDs)

- Restricted form of distributed shared memory
- Immutable
- Only built through reading stable storage or coarse-grained deterministic **transformations** (`map`, `filter`, `join`, . . .)
- **Actions** (`count`, `collect`, `reduce`, . . .) either return value or write to storage



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

// So far everything in driver


messages.filter(_.contains("MySQL")).count
messages.filter(_.contains("HDFS")).count
```

Figure: Load error messages from a log into memory and then interactively search for various patterns.

- Transformations are lazy
  - Do not compute result until action is invoked
  - Useful

- Users can control two other aspects,
  - **Persistence**: keep an RDD in memory using `persist` or `cache`. Can spill the RDDs to disk if not enough RAM,
  - **Partitioning**: an RDDs elements be partitioned across machines based on a key in each record.
- In comparison with MapReduce,
  - map: a transformation that passes each dataset element through a function and returns a new RDD representing the results,
  - reduce: an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program

Efficient fault recovery using **lineage**

- On failure, log one operation to apply to many elements (e.g., `map` in previous example)
- Recompute lost partitions on failure
- No cost if nothing fails

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

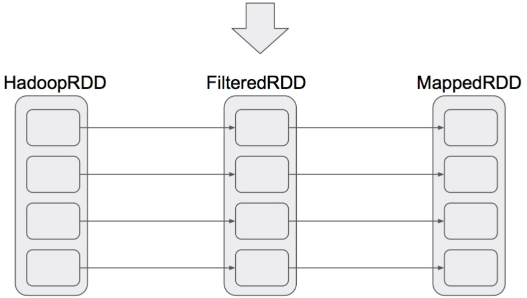
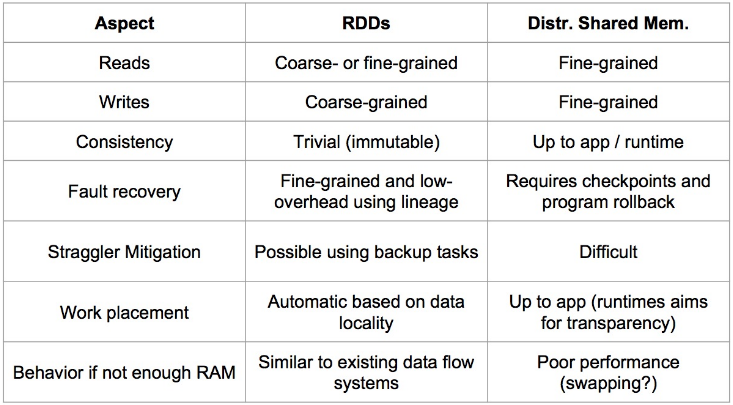


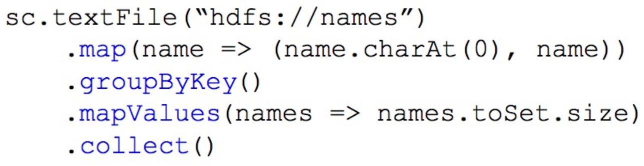
Figure: RDDs track their lineage (graph of transformations that built them) to rebuild lost data.



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 aims for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Figure: Comparison of RDDs with distributed shared memory.

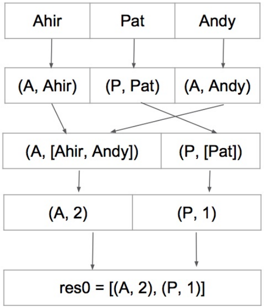
- Works best for batch applications that apply same operation to all elements of a dataset,
- Not well-suited for applications that do asynchronous fine-grained updates to shared state, e.g., storage system for a web application.



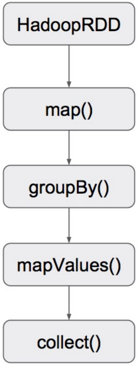
```
sc.textFile("hdfs://names")
    .map(name => (name.charAt(0), name))
    .groupByKey()
    .mapValues(names => names.toSet.size)
    .collect()
```

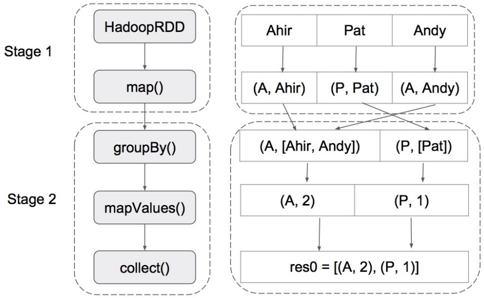
Figure: Find number of distinct names per “first letter”. This example is reproduced from [3].

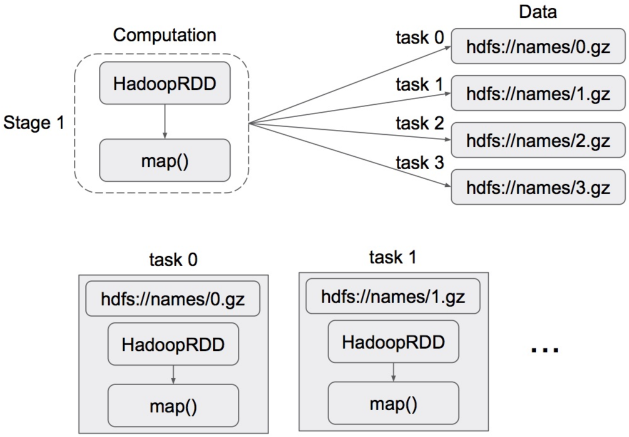
```
sc.textFile("hdfs://names")

    .map(name => (name.charAt(0), name))

    .groupByKey()

    .mapValues(names => names.toSet.size)

    .collect()
```

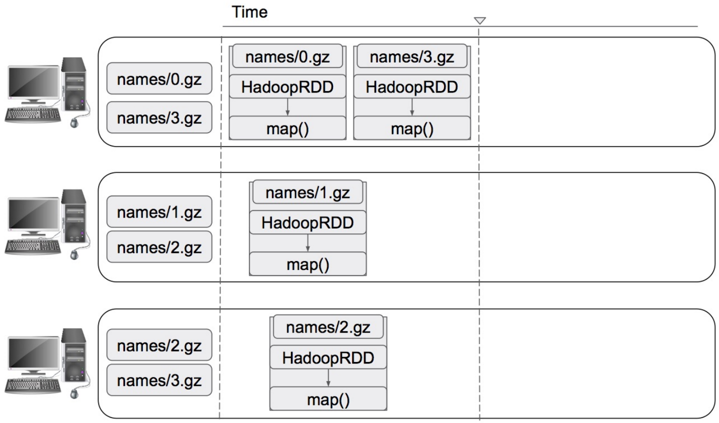


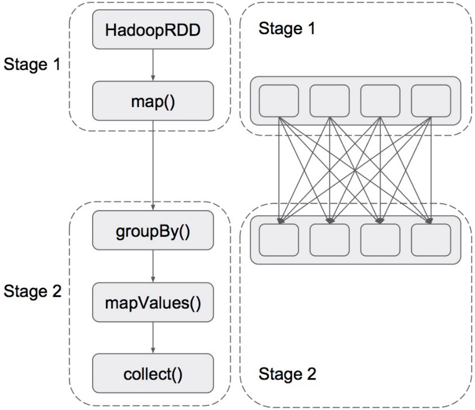
```
sc.textFile("hdfs://names")

    .map(name => (name.charAt(0), name))

    .groupByKey()

    .mapValues(names => names.toSet.size)

    .collect()
```

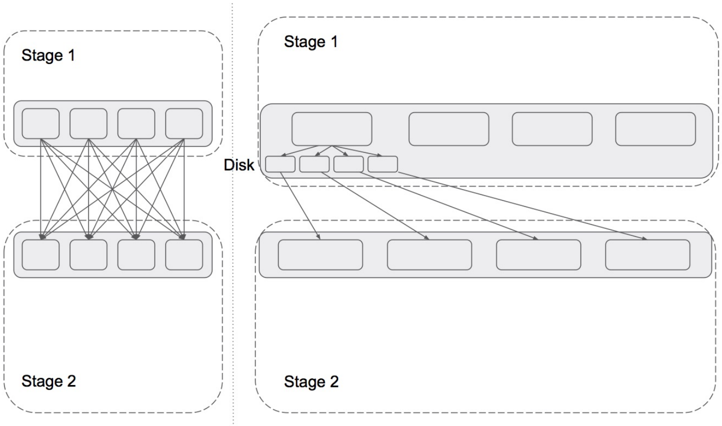



Stage 1
Stage 2
HadoopRDD
map()
groupBy()
mapValues()
collect()
Ahir
Pat
Andy
(A, Ahir)
(P, Pat)
(A, Andy)
(A, [Ahir, Andy])
(P, [Pat])
(A, 2)
(P, 1)
res0 = [(A, 2), (P, 1)]


Data
Computation
task 0
hdfs://names/0.gz
HadoopRDD
task 1
hdfs://names/1.gz
Stage 1
task 2
hdfs://names/2.gz
map()
task 3
hdfs://names/3.gz
task 0
task 1
hdfs://names/0.gz
hdfs://names/1.gz
HadoopRDD
HadoopRDD
map()
map()
...


Time
names/0.gz
names/3.gz
names/0.gz
HadoopRDD
map()
names/3.gz
HadoopRDD
map()
names/1.gz
names/2.gz
names/1.gz
HadoopRDD
map()
names/2.gz
names/3.gz
names/2.gz
HadoopRDD
map()


Stage 1
HadoopRDD
map()
Stage 2
groupBy()
mapValues()
collect()
Stage 1
Stage 2


Stage 1
Stage 2
Disk
Stage 1
Stage 2

```
val data = Array(1,  2,  3,  4,  5)
val distData = sc. parallelize (data)

//  add up the elements of the array
distData.reduce((a, b) => a + b)

//  specify  number of partitions
//  default :  sets  automatically  based on cluster
//  runs one task for  each partition  of  the  cluster
distData = sc. parallelize (data,  10)

val  distFile  = sc. textFile ("data. txt ")

//  add up the sizes of  all  the  lines
 distFile .map(s => s.length).reduce((a, b) => a + b)

//  passing functions
object MyFunctions { def func1(s: String) :  String  = {   ...  }  }
myRdd.map(MyFunctions.func1)
```

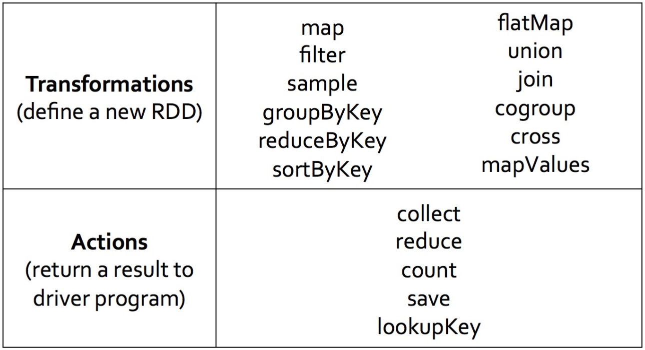
```
var counter = 0
var rdd = sc. parallelize (data)
rdd.foreach(x => counter += x)
 println ("Counter value: "  + counter)
```

- Behavior of the above code is undefined
- In local mode, works because counter is in the memory of driver
- In cluster mode, a new copy of counter is sent to each node in the cluster,
- In cluster mode, use accumulators to make it work.
- Closure: those variables and methods which must be visible for the executor to perform its computations on the RDD
- The closure is serialized and sent to each executor

Spark supports “Shared Variables” along with RDDs

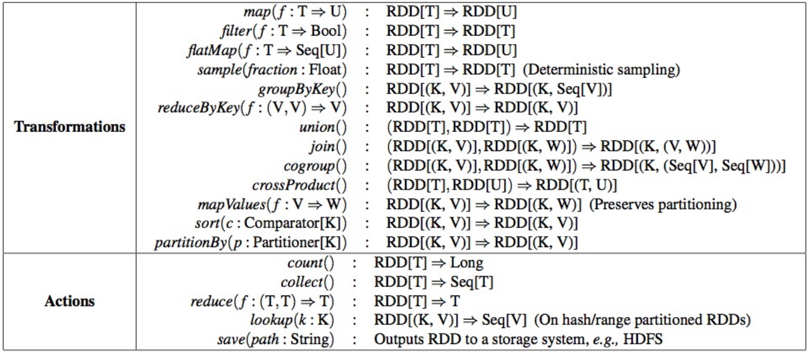
- closure (variable + function) runs in parallel as set of tasks in nodes
- generally ships copy of each variable to each task
- for sharing across tasks: i) broadcast variables, ii) accumulators
- broadcast: cache a value in memory on all nodes
- accumulators: variables that are only “added”, e.g., counters, sums.

```
val broadcastVar = sc.broadcast(Array(1, 2, 3))

val accum = sc.accumulator(0, "My Accumulator")
sc. parallelize (Array(1,  2,  3,  4)) .foreach(x => accum += x)
```



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	

Figure: Transformations and actions available on RDDs in Spark [2].



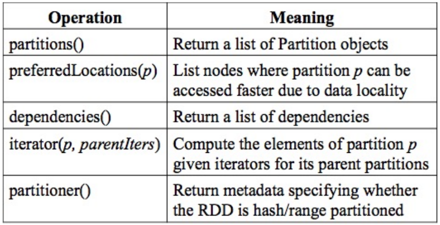
Transformations	*map*(*f* : T ⇒ U)	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V,V) ⇒ V)	RDD[(K, V)] ⇒ RDD[(K, V)]
	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

Figure: Transformations and actions available on RDDs in Spark [1].

- Persist RDDs using `persist()` or `cache()`
- Persisted RDDs can be stored using a storage level
- *MEMORY_ONLY* (default): store RDD as deserialized Java object in the JVM, RDD partitions that don't fit will be recomputed
- *MEMORY_AND_DISK*: similar to before but stores the partitions that don't fit on disk
- *MEMORY_ONLY_SER*: store RDD as serialized Java object
- *MEMORY_AND_DISK_SER*: similar but store on disk too
- *DISK_ONLY*: store the RDD partitions only on disk

Which storage level to choose?

- if RDD fits comfortably with default, leave it as is
- if not, try *MEMORY_ONLY_SER* and select fast serialization library
- don't spill to disk unless the functions that computed datasets are expensive (otherwise recomputing may be as fast as reading from disk)
- use replicated storage level for faster fault recovery
- spark automatically monitors cache usage on each node and drops out old partitions based on LRU. For manual, `RDD.unpersist()`



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

RDD interface exposes five pieces of information:

- a set of *partitions*, which are atomic pieces of the dataset
- a set of *dependencies* on parent RDDs
- a function for computing the dataset based on its parents
- metadata about its partitioning
- metadata about its data placement

Dependencies,

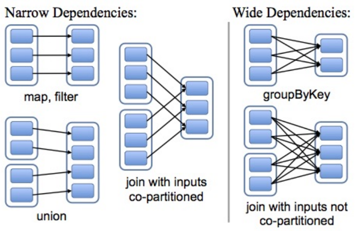

Narrow Dependencies:
map, filter
union
join with inputs co-partitioned
Wide Dependencies:
groupByKey
join with inputs not co-partitioned

Figure: Narrow and wide dependencies [1].

Dependencies are useful for,

- for staging (seen before, `groupByKey()`)
- recovery after node failure, easy with narrow and difficult with wide.

Main results,

- Spark outperforms Hadoop by up to 20x in iterative machine learning and graph applications (speedup mainly comes from avoiding I/O and deserialization costs by storing data in memory)
- When nodes fail, Spark can recover quickly by rebuilding only the lost RDD partitions
- able to query 1 TB dataset in 5-7 seconds

- Comparing Spark with Hadoop
  1. Iterative machine learning algorithms: logistic regression, k-means
  2. PageRank
- Fault recovery
- Behavior with insufficient RAM
- Scalability

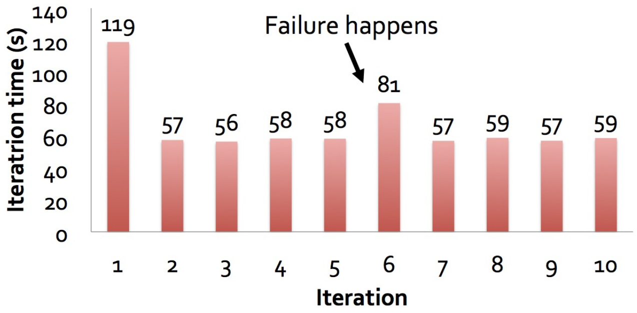


Figure: Failure Recovery [2].

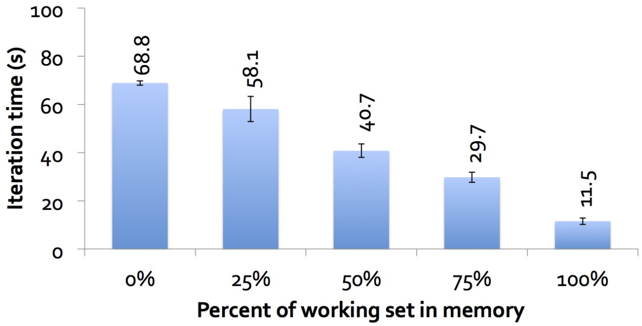


Figure: Behavior with Insufficient RAM [2].

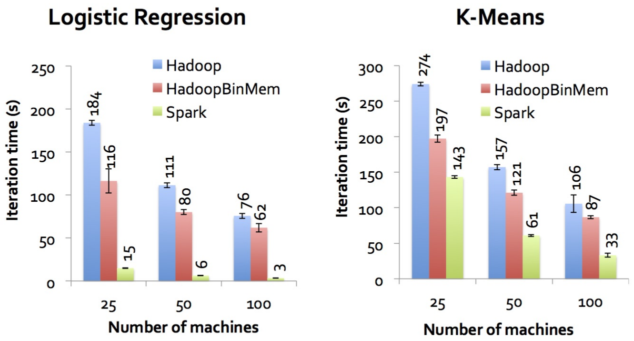


Figure: Scalability [2].

- RAMCloud, Piccolo, GraphLab, parallelDBs
  - Fine-grained writes requires replication for fault tolerance
- Pregel, iterative MapReduce
  - Specialized models, can't perform interactive mining
- DryadLINQ, FlumeJava
  - Similar to RDD, but cannot share datasets efficiently across queries
- Nectar
  - Automatic expression caching over distributed FS
- PacMan
  - Memeory cache for HDFS with writes over network/disk

Possible Errors in RDD based programming,

- Errors due to malformed or corrupt data
  - Use `filter` transformation to discard bad inputs
  - `map`, if it is possible to fix the bad input
  - `flatmap`, try fixing the input but fall back to discarding
- Job aborted due to stage failure: Task not serializable
  - happens in the process of serializing the closure before sending it to workers, fail if the object is not serializable
  - cause: intialize a variable on the driver (master), but then try to use it on one of the workers
- UnsupportedOperation exception
  - Spark uses Scala's static type system and inference technique, however there are reports on exceptions triggered due to invalid operations
- OutOfMemory errors
  - caused by caching large reusable data

Some works,

- Spores and Silo
  - Potential hazards when using closures incorrectly are:
  - memory leaks
  - race conditions due to capturing mutable references
  - runtime serialization errors due to unintended capture of references
  - Spores: well-behaved closures with controlled environments that can avoid various hazards
  - Silo: Distributed Programming via Safe Closure Passing (uses spores in syntactic and type based restrictions)
- A Characteristic Study on Out of Memory Errors in Distributed Data-Parallel Applications

1) RDDs are simple and efficient programming model for a broad range of applications
2) Key to performance: it allows applications to avoid costly disk accesses by reliably storing the data in memory.
3) Spark implementation of RDDs,

- APIs in Scala, Java, Python and R
- Integrates HDFS, Amazon S3, Hive, HBase, Cassandra, etc.
- Can run on clusters managed by Hadoop YARN, Apache Mesos, also standalone
- High level library support, SparkSQL, Spark Streaming, MLlib (machine learning), GraphX (graph), R, etc.

[1] 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, April 2012

[2] Talk: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, by M. Zaharia, NSDI 2012, San Jose, CA, April 2012.

[3] Talk: A Deeper Understanding of Spark Internals, by Aaron Davidson (Databricks), Spark Summit 2014, San Francisco, CA, June 2014

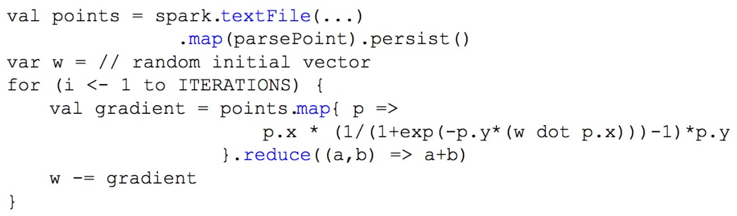
[4] Spark Programming Guide 1.6.0, URL: http://spark.apache.org/docs/latest/programming-guide.html

Questions?

http://www.cs.iastate.edu/~ganeshau/

Logistic Regression

Logistic regression,



```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
    val gradient = points.map{ p =>
                        p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
                   }.reduce((a,b) => a+b)
    w -= gradient
}
```

- `points` is a persistent RDD produced by `map` on text file
- repeatedly run `map` and `reduce` to compute gradient at each step by summing function of the current w.
- keeping `points` in memory across iterations yields 20x speedup.

Word Count

```
val textFile = spark.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Performance Tuning

Problems,

- Too few partitions
- Large per-key `groupBy()`
- Shipped all data across the cluster

Guidelines

- Ensure enough partitions (often 100 - 10000), mostly based on task execution time at least 100ms (upper bound), at least 2x number of cores in cluster (lower bound)
- Minimize memory consumption while sorting and large keys in groupBys
- Minimize amount of data shuffled

Resolution

- Increase `spark.executor.memory`
- Increase number of partitions
- Re-evaluate program structure

Example 2 Optimization

```
sc.textFile(“hdfs://names”)
    .map(name => (name.charAt(0), name))
    .groupByKey()
    .mapValues(names => names.toSet.size)
    .collect()
```

```
sc.textFile(“hdfs://names”)
    .distinct(numPartitions = 6)
    .map(name => (name.charAt(0), name))
    .reduceByKey(_ + _)
    .collect()
```