Today's content

- Resilient Distributed Datasets(RDDs) ------ Spark and its data model

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

By Matei Zaharia, etc.

Presented by 齐琦
海南大学

Abstract

- Resilient Distributed Datasets (RDDs)
  - A distributed memory abstraction; perform in-memory computations on large clusters in a fault-tolerant manner.
- Motivation
  - Current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools
- Expressive enough to apply to wide range of computations
- Spark, implemented RDDs.

Introduction

- Cluster computing frameworks
  - MapReduce, Dryad, etc. , widely adopted for large-scale data analytics
  - High-level operators for parallel computations, hide work distribution and fault tolerance
- Lack abstractions leveraging distributed memory
  - Emerging applications: reuse intermediate results across multiple computations

Introduction, contd.

- Data reuse
  - Iterative machine learning, graph algorithms
  - PageRank, K-means clusters, logistic regression
- Interactive data mining
  - Multiple ad-hoc queries on same data subset

Introduction, contd.

- For most frameworks, intermediate results between computations, written to external stable storage system (a distributed file system)
  - Overheads due to data replica, disk I/O, serialization, which dominate app exec time.

Introduction, contd.

- Specialized frameworks for data reuse
  - Pregel – iterative graph computations
  - HaLoop – iterative MapReduce interface
- Only support specific computation patterns, with implicitly data sharing for these patterns
  - Not provide abstractions for more general reuse, e.g. load datasets in memory and run ad-hoc queries across them.

Introduction, contd.

- New abstraction – resilient distributed datasets (RDDs)
  - Efficient data reuse in broad range of applications
  - Fault-tolerant
  - Parallel data structures
  - Optimize data placement by controlling their partitioning
  - Rich operators for data manipulation

challenge

- Define a programming interface providing fault tolerance efficiently
- Systems (existing abstractions for in-memory storage on clusters) before
  - Distributed shared memory
  - Key-value stores
  - Databases
  - Piccolo
  - Its interface based on fine-grained updates to mutable state(table cells)
  - Only way providing fault tolerance – replicate data across machines; log updates across machines
    - Expensive for data-intensive workloads
    - Copy large data over cluster network, bandwidth far lower than RAM; substantial storage overhead

RDD's approach

- Interface based on coarse-grained transformations (map, filter, join)
  - Fault tolerance efficiency: logging transformations (its lineage) used to build a dataset, rather than actual data
  - Lost data recovered quickly (by other RDDs), without costly replication

RDD's advantages

- Good fit for many parallel applications
  - They naturally apply same operation to multiple data items
- Express efficiently many cluster programming models that been proposed as separate systems
  - Including MapReduce, DryadLINQ, SQL, Pregel, HaLoop
  - Iteractive data mining (existing systems not capture)
- Can accommodate computing needs that previously only by introducing new frameworks – power of RDD abstraction

Spark, implementation

- Spark
  - Implements RDDs
  - At UC Berkeley
  - Provides Language-integrated programming interface
  - In Scala programming language
  - Use Scala interpreter to interactively query big datasets

Evaluation

- Through both micro-benchmarks and measurements of user applications
- Up to 20X faster than Hadoop for iterative applications
- A real-world data analytics speed up by 40X
- Interactively scan a 1TB dataset , 5-7s latency
- Show its generality
  - Implemented Pregel and HaLoop on top of Spark (200 lines of code each)

Resilient Distributed Datasets (RDDs)

Resilient Distributed Datasets (RDDs) Abstraction

- Read-only (immutable), partitioned collection of records
- Create through deterministic operations on data in stable storage, or other RDDs
  - These ops – transformations ; map, filter, join
- RDDs not need materialized all times
- Users can control RDDs' persistence, and partitioning
  - Persistence – storage strategy(in-memory)
  - Partition RDD's elements across machines based on a key in each record; placement optimizations (hash-partitioned)

Spark programming interface

- Spark provides API for using RDDs
  - Similar to DryadLINQ, FlumeJava
  - Each dataset as an object
  - Transformations invoked as methods on the objects
- Operations
  - Transformations – define RDDs
  - Actions – return a value to application, or export data to a storage system; count, collect, saves
- Persist method
  - Which RDDs want to reuse in future operations
  - Keep in memory by default(can spill to disk)
  - Other strategies: on disk, replicate across machines

Example: Console Log Mining

- Query terabytes logs of web service to find error cause (stored in Hadoop filesystem HDFS)
- Uses Spark, load just error messages in logs into RAM across a set nodes, query them interactively

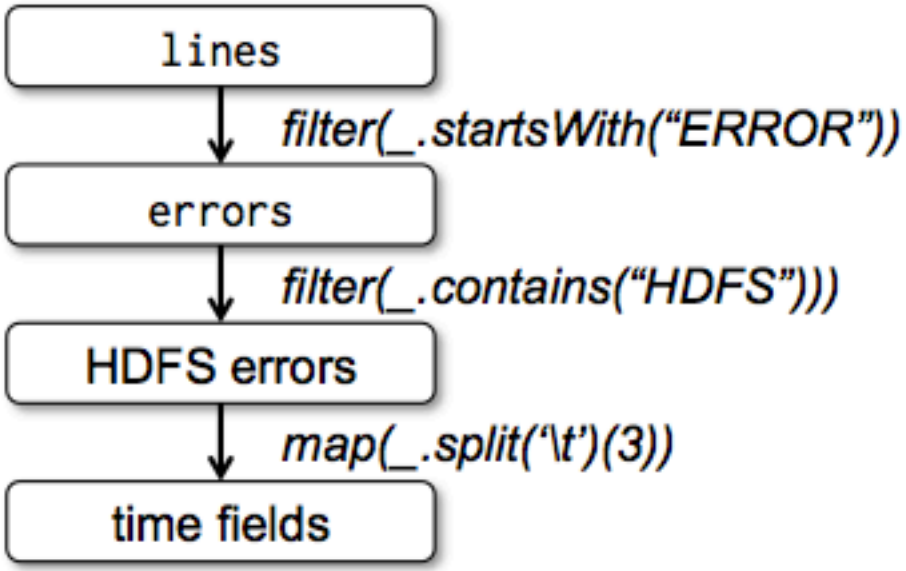


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

Example, contd.

1. Defines an RDD
2. Derives a filtered RDD
3. Asks errors to persist in memory, so it can be shared across queries

- Count error number
  - Errors.count()
- Lines, not loaded into RAM

Example, contd.

- Spark scheduler will pipeline the latter two transformations, send a set of tasks to compute them to nodes holding cached partitions of errors
- If partition lost, Spark rebuilds it by applying a filter on only corresponding partition of lines

```
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()

// Return the time fields of errors mentioning
// HDFS as an array (assuming time is field
// number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```

RDD Model's Advantages

- As a distributed memory abstraction; Compare against distributed shared memory (DSM)
  - DSM very general abstraction, but harder to implement in an efficient and fault-tolerant manner on commodity clusters

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?)

Table 1: Comparison of RDDs with distributed shared memory.

RDD model advantages, contd.

- RDDs can only be created through coarse-grained transformations, while DSM allows read and write to each memory location
  - RDDs' applications that perform bulk writes, but allows more efficient fault tolerance (using lineage)
  - Only lost partitions need to recomputed; can be in parallel on different nodes, no need of roll back whole program

Contd.

- Immutable nature mitigates slow nodes by running backup copies of slow tasks
  - Hard on DSM, two copies access same memory locations, interfere with each other's updates
- For bulk operations on RDDs, tasks can be scheduled based on data locality to improve performance
- Partitions not fit in RAM can be stored on disk, and provide similar performance to current data-parallel systems

RDDs limits

- Best suited for batch applications (batch analytics)
  - Apply same operation to all elements of a dataset
  - Each transformation as one step in a lineage graph; recover lost partitions without log large amount data
- Less suitable for applications that make asynchronous fine-grained updates to shared state
  - Storage system for a web application, or incremental web crawler
  - For these, more efficient to use traditional update logging and data check-pointing, e.g. database

Spark Programming Interface

- Spark provides the RDD abstraction through language-integrated API similar to DryadLINQ in Scala
  - Chose Scala due to its combination of conciseness and efficiency (due to static typing)

- Write a driver program that connects to a cluster of workers
- Driver defines RDDs and invokes actions on them

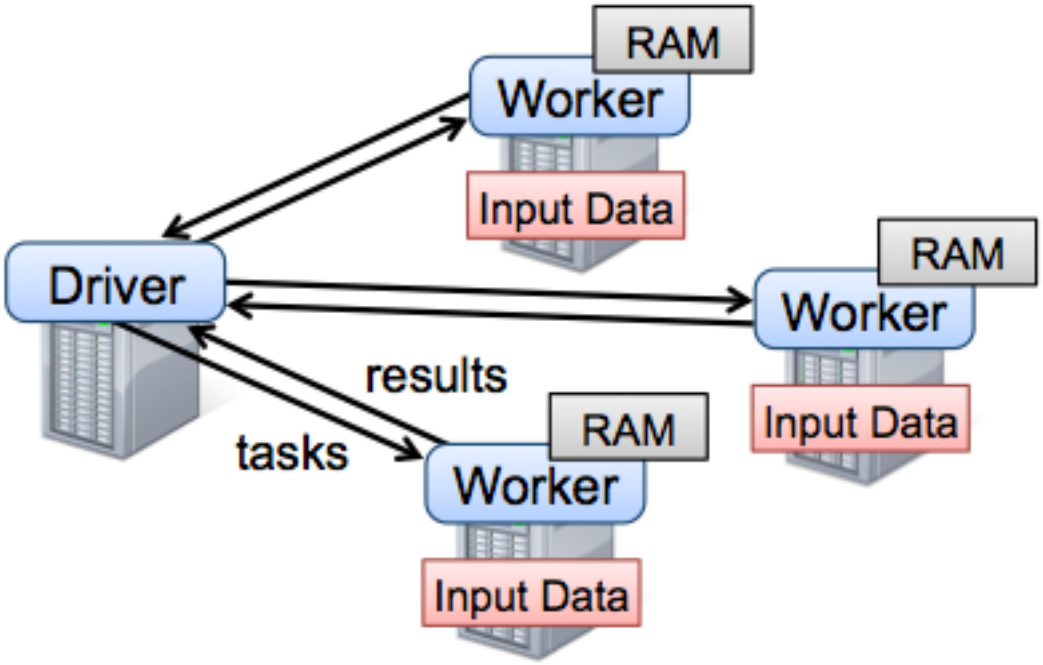


Figure 2: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

- RDDs themselves are statically typed objects parameterized by element type
  - RDD[Int], a RDD of integers
- Challenges to implement
  - Have to work around issues with Scala's closure objects using reflection

RDD operations in Spark

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

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

- Transformations
  - Lazy operations, define a new RDD
- Actions
  - A computation to return a value to the program, or write data to external storage
- Persist a RDD
  - Can get a RDD's partition order (by a Partitioner class)

Example applications: Logistic regression

- Machine learning algorithms
  - Many are iterative in nature (iterative optimization procedures)

```
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
}
```

Example2 : PageRank

- Iteratively updates rank for document by adding up contributions from documents that link to it.

```
// Load graph as an RDD of (URL, outlinks) pairs

 val links = spark.textFile(...).map(...).persist()
 var ranks = // RDD of (URL, rank) pairs
 for (i <- 1 to ITERATIONS) {
   // Build an RDD of (targetURL, float) pairs
   // with the contributions sent by each page
   val contribs = links.join(ranks).flatMap {
     (url, (links, rank)) =>
       links.map(dest => (dest, rank/links.size))
   }
   // Sum contributions by URL and get new ranks
   ranks = contribs.reduceByKey((x,y) => x+y)
                .mapValues(sum => a/N + (1-a)*sum)
 }
```

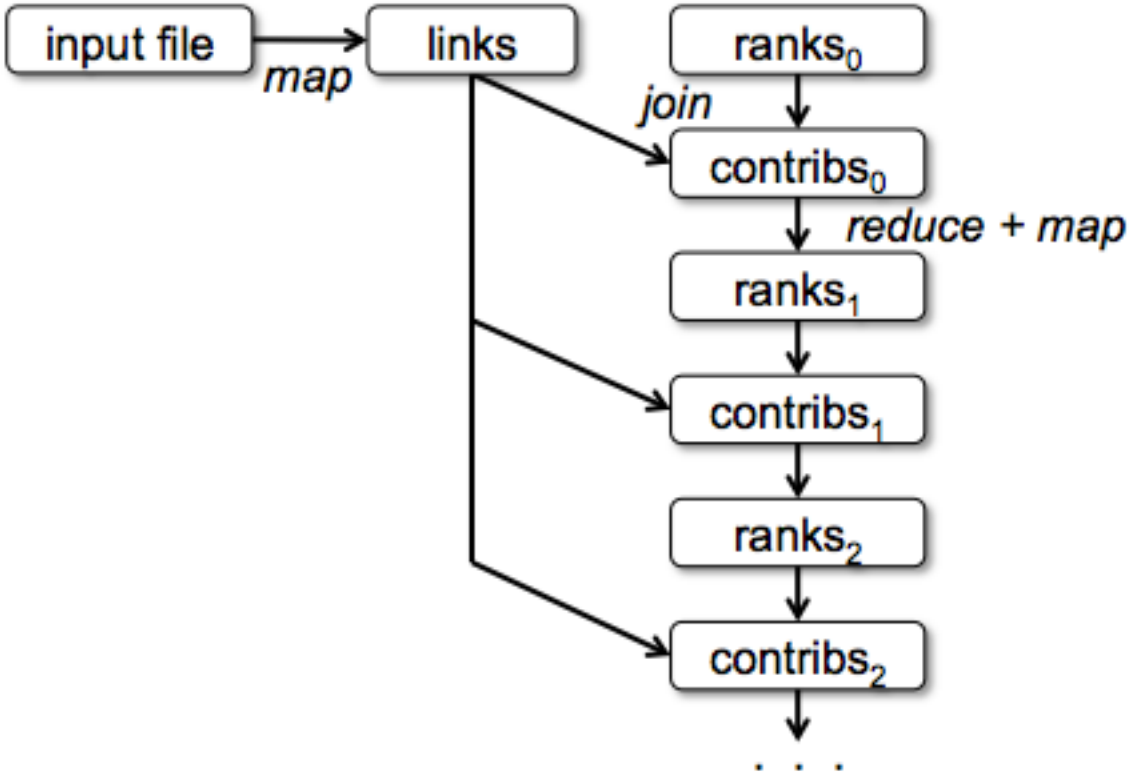


Figure 3: Lineage graph for datasets in PageRank.

- Optimize communication in PageRank by controlling partitioning of RDDs
  - Specify partitioning for links (e.g. hash-partition link lists by URL across nodes); then partition ranks in same way
  - So join operation between links and ranks requires no communication (as each URL's rank on same machine as its link list)
- Custom Partitioner class to group pages that link to each other together
- By calling partitionBy

```
links = spark.textFile(...).map(...)
             .partitionBy(myPartFunc).persist()
```

Representing RDDs

Design goals

- RDDs should (design goals):
  - Track lineage across wide range of transformations
  - Provide rich set of transformation operators, and let users compose them in arbitrary ways

Interface representing RDD

- Representing each RDD through a common interface that exposes five information
  - A set of partitions (atomic pieces of dataset)
  - A set of dependencies on parent RDDs
  - A function for computing dataset based on its parents
  - Metadata about partitioning scheme
  - Data placement

Interface summary

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

Table 3: Interface used to represent RDDs in Spark.

Dependencies between RDDs

- Two types
  - Narrow dependencies
  - Wide dependencies
- Narrow dependencies allow for pipelined execution on one cluster node
- Recovery after a node failure is more efficient with narrow dependency
  - Only lost parent partitions need to be recomputed, can be recomputed in parallel on different nodes
  - Wide deps: A single failed node might cause loss of some partition from all ancestors of an RDD, requiring a complete re-execution.

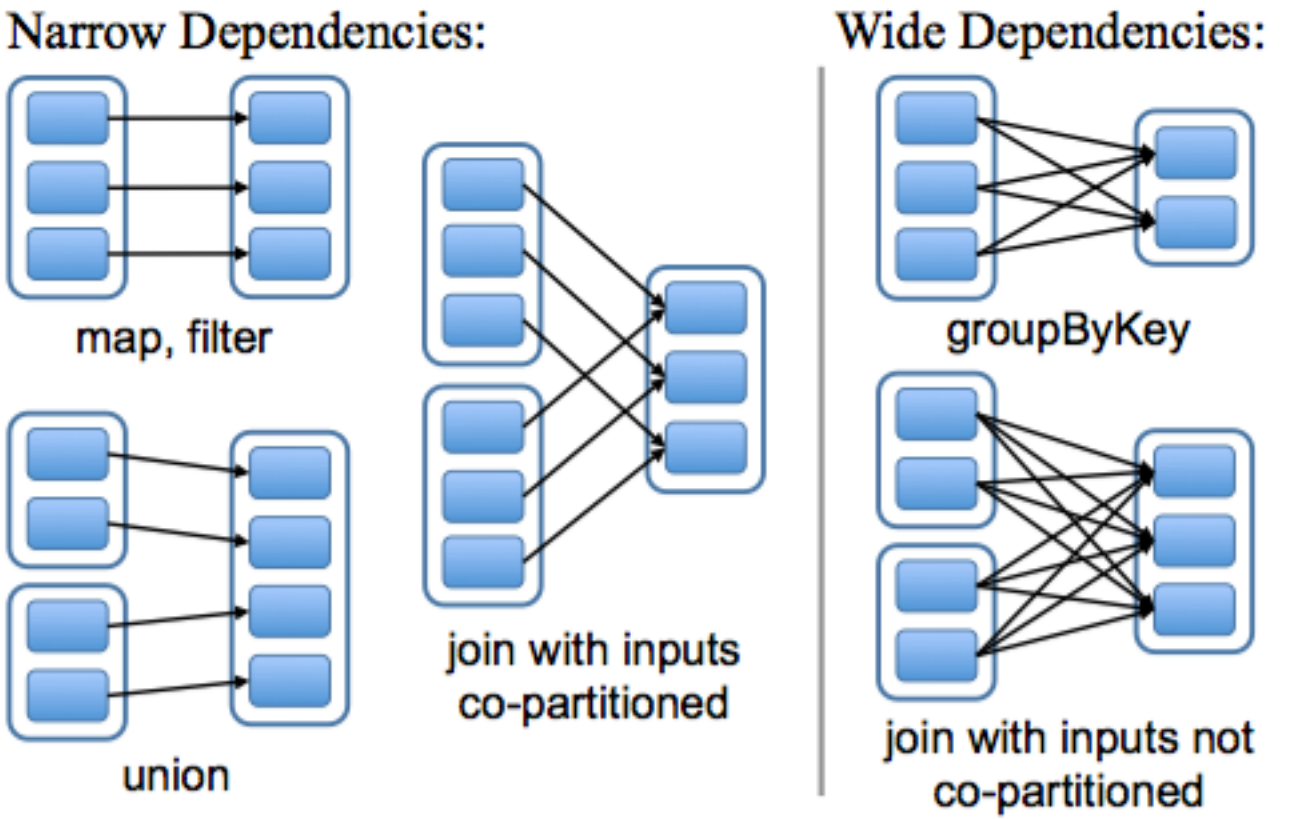


Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

Implementation

Overview

- Implemented Spark in 14,000 lines of Scala
- Runs over Mesos cluster manager, allowing it share resources with Hadoop, MPI and other applications
  - Each Spark program as separate Mesos app, with its own driver and workers
  - resource sharing between apps handled by Mesos
- Spark can read data from any Hadoop input source(e.g. HDFS or HBbase)
  - By Hadoop's existing input APIs

Job Scheduling

- When runs an action(e.g. count or save) on an RDD, scheduler examines RDD's lineage graph to build a DAG of stages to execute
  - Each stage contains as many pipelined transformations with narrow dependencies as possible
  - Stage boundaries are shuffle ops required for wide dependencies
  - Scheduler launches tasks to compute missing partitions from each stage, until computed target RDD

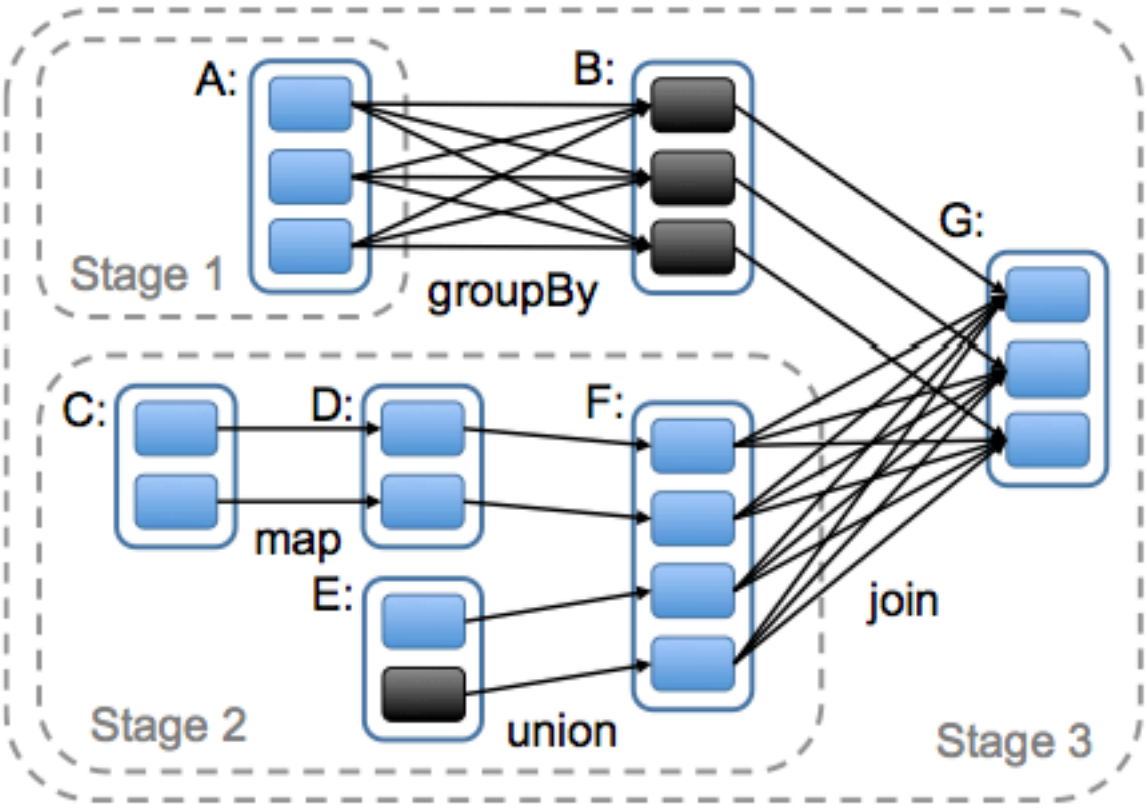


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

- Scheduler assigns tasks to machines based on data locality using delay scheduling
- If a task fails, re-run it on another node as long as its stage's parents are still available
  - If stages unavailable, resubmit tasks to compute missing partitions in parallel
- Not yet tolerate scheduler failures

Interpreter Integration

- How Scala interpreter works
  - Compile a class for each line typed by user, loading it into JVM, invoke a function on it
  - Types var x = 6 ; then println(x)
    - Define a class called Line1 containing x
    - Second line compiled to println(Line1.getInstance().x)

Changes to interpreter in Spark

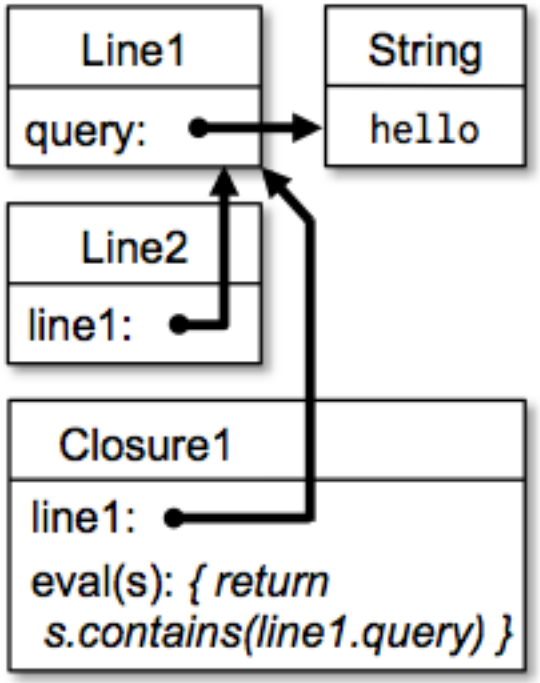
- Class shipping
  - Worker nodes fetch bytecode for classes on each line, and served over HTTP
- Modified code generation
  - To reference instance of each line object directly

Line 1:

```
var query = "hello"
```

Line 2:

```
rdd.filter(_.contains(query))
   .count()
```



a) Lines typed by user

b) Resulting object graph

Figure 6: Example showing how the Spark interpreter translates two lines entered by the user into Java objects.

Memory Management

- Options for storage of persistent RDDs
  - In-memory storage as deserialized Java objects (fastest performance)
  - In-memory storage as serialized data – memory-efficient representation than Java object graphs
  - On-disk storage – for too large to keep in RAM, but costly recompute on each use

Support for Checkpointing

- Lineage time-consuming for RDDs with long lineage chains, so helpful to checkpoint some RDDs to stable storage
- Long lineage graphs containing wide dependencies
  - A node failure may cause loss of some slice of data from each parent RDD, requiring a full recomputation
- Spark provides API for checkpointing, leaves decision of which data to checkpoint to users

Evaluation

Overview

- Run on Amazon EC2
- Spark outperforms Hadoop by up to 20x in iterative machine learning and graph applications
  - Through avoiding I/O and deserialization costs by storing data in memory as Java objects
  - Applications written by users perform and scale well
  - When nodes fail, Spark can recover quickly by rebuilding only lost RDD partitions
  - Spark can query a 1 TB dataset interactively with latencies of 5—7 seconds

Settings

- Used m1.xlarge EC2 nodes with 4 cores, 15GB RAM
- Used HDFS for storage, with 256MB blocks
- Before each test, cleared OS buffer caches to measure IO costs accurately

Iterative Machine learning applications

- Two applications: logistic regression, k-means
- Systems to compare
  - Hadoop (0.20.2 stable release)
  - HadoopBinMem
    - Converts input data into low-overhead binary format, stores in an in-memory HDFS instance
  - Spark: implementation of RDDs

- Ran both algorithms for 10 iterations on 100 GB datasets using 25—100 machines
- Iteration time of k-means dominated by computation; logistic regression less compute-intensive and more sensible to time spent in deserialization and I/O

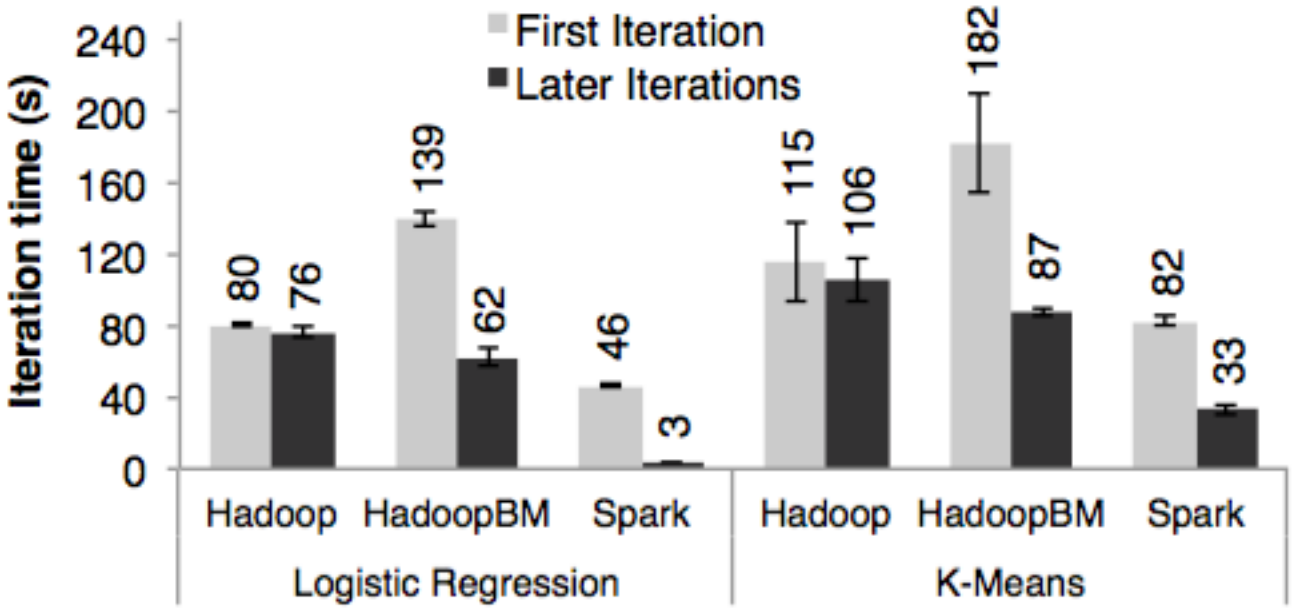


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

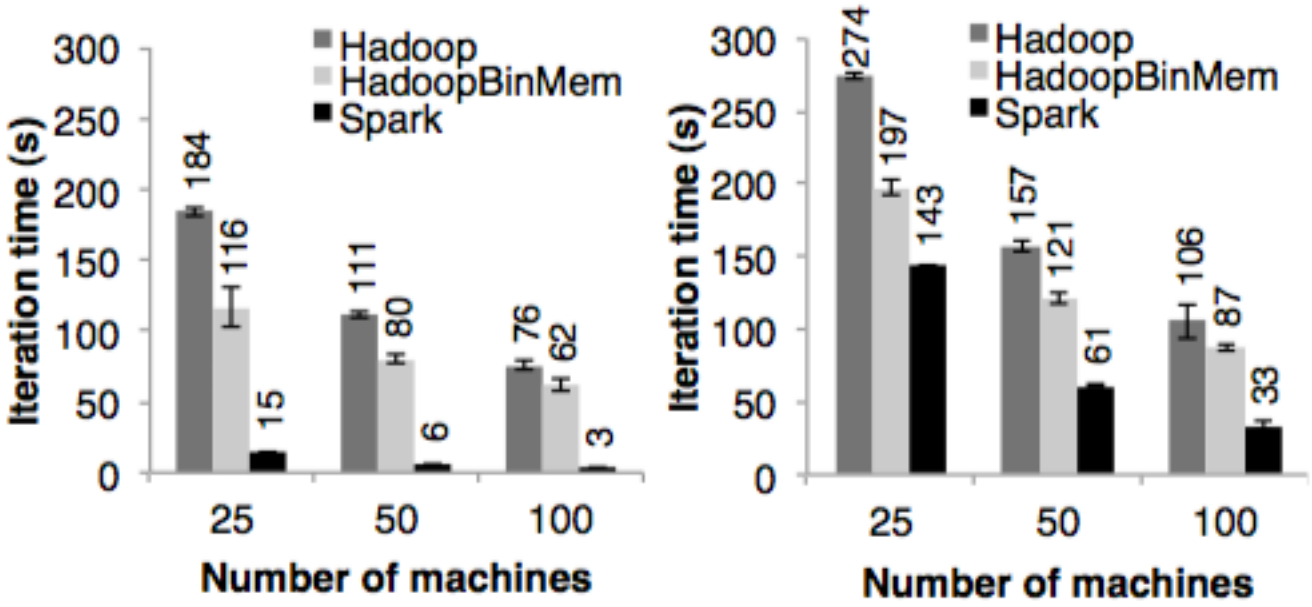


Figure 8: Running times for iterations after the first in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB.

- Spark outperformed even Hadoop with in-memory storage of binary data by a 20x margin
- Storing RDD elements as Java objects in memory, Spark avoids these overheads
  - Reading through HDFS introduced a 2-second overhead
  - Parsing text overhead was 7 seconds
  - Reading from in-memory file, converting pre-parsed binary data into Java objects took 3 seconds

PageRank experiment

- Dataset: 54GB wikipedia dump
- 10 iterations to process a link graph of 4 million articles
- In-memory storage alone provided Spark with a 2.4x speedup over Hadoop on 30 nodes
- Results also scaled nearly linearly to 60 nodes

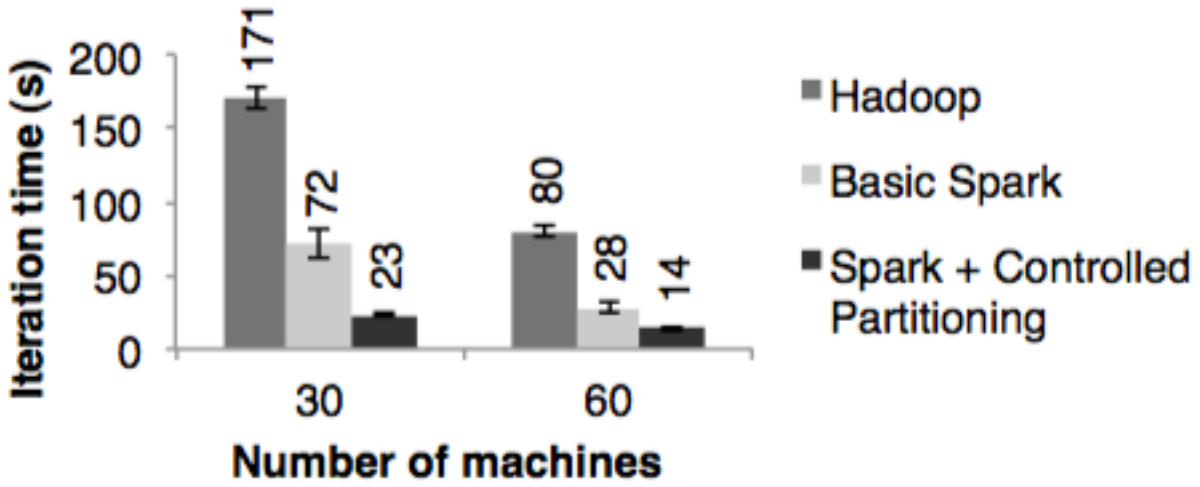


Figure 10: Performance of PageRank on Hadoop and Spark.

Fault Recovery

- Evaluated cost of reconstructing RDD partitions using lineage after a node failure in k-means application

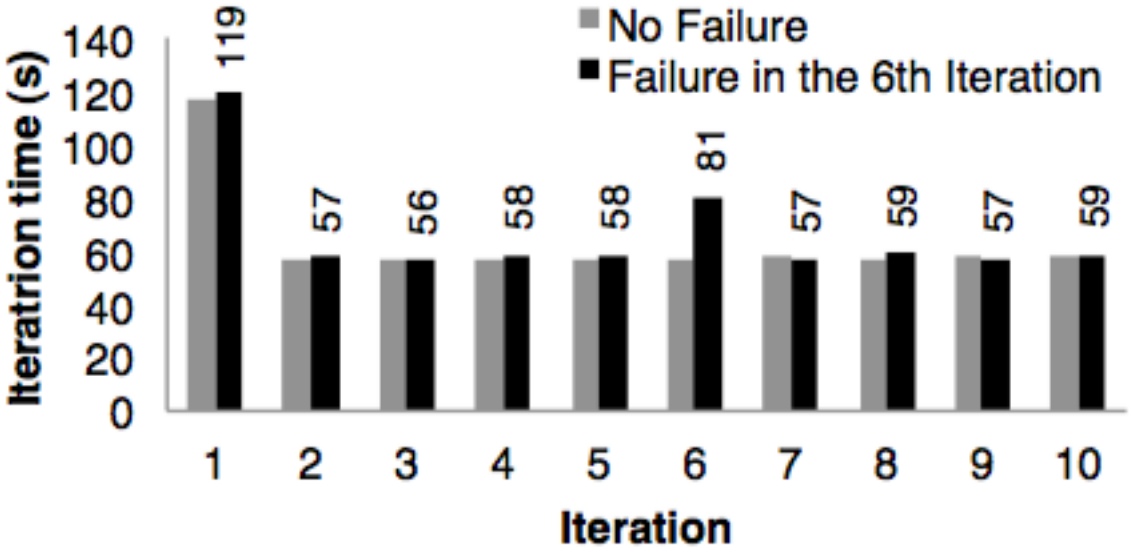


Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

Behavior with insufficient memory

- What if there's not enough memory to store a job's data
- Performance degrades gracefully with less space

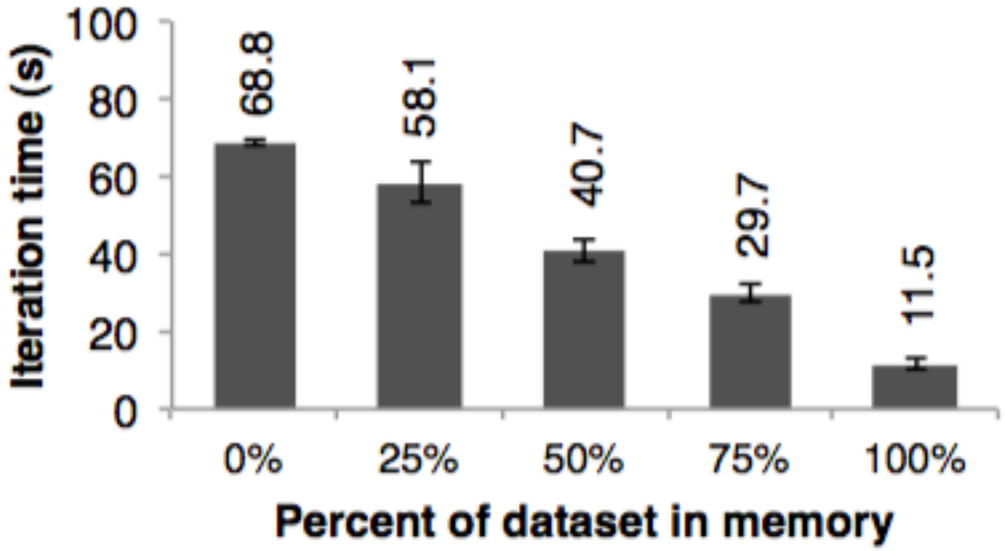
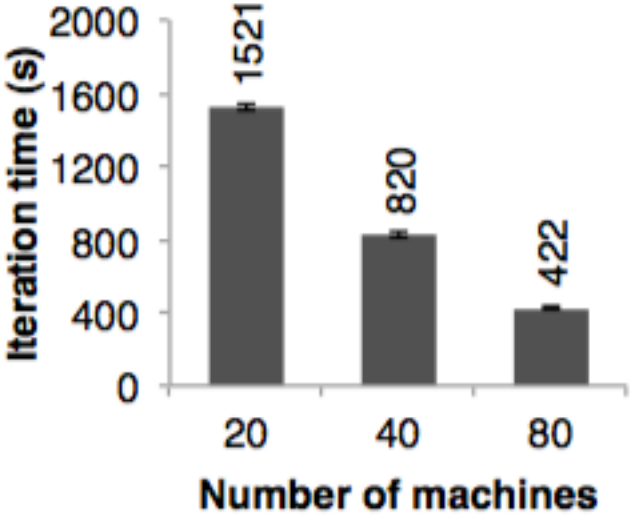


Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

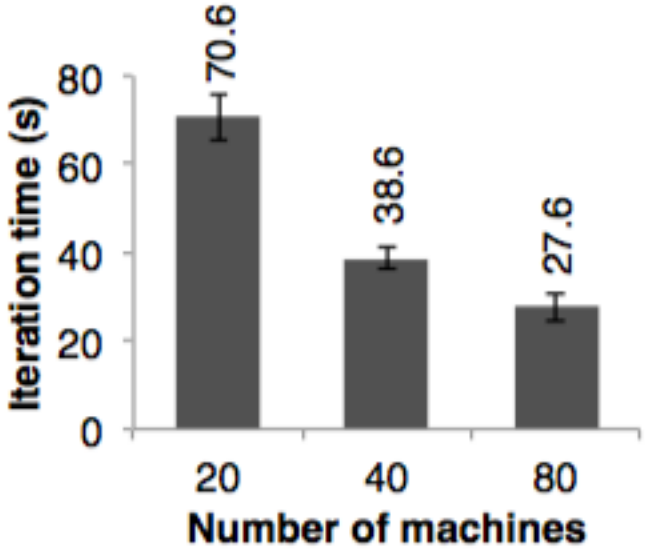
User applications built with Spark

- In-memory analytics
  - Conviva Inc, video distribution company, used Spark accelerate data analytics reports previously ran over Hadoop
  - Speed up report by 40x
  - A report on 200GB compressed data took 20 hours on Hadoop cluster, now runs in 30 minutes using only two Spark machines
  - Spark program only required 96 GB RAM, because only stored rows and columns matching user's filter in a RDD, not whole decompressed file

- Traffic Modeling
  - Mobile Millennium project at Berkeley, parallelized learning algorithm for inferring road traffic congestion from sporadic automobile GPS measurements
  - Source data, 10,000 link road network for a metropolitan area, 600,000 samples of point-to-point trip times
  - Traffic model estimates time taking to travel across road links
  - EM algorithm trained model, repeats two map and reduceByKey steps iteratively
  - Scales nearly linearly from 20 to 80 nodes with 4 cores each



(a) Traffic modeling



(b) Spam classification

Figure 13: Per-iteration running time of two user applications implemented with Spark. Error bars show standard deviations.

- Twitter Spam Classification
  - Monarch project at Berkeley used Spark to identify link spam in Twitter messages
  - Implemented a logistic regression classifier on top of Spark
  - Training a classifier over 50GB subset of the data: 250,000 URLs and 10to7 features/ dimensions
  - Scaling not as close to linear due to a higher fixed communication cost per iteration

Interactive Data Mining

- Analyze 1TB Wikipedia page view logs(2 years of data)
- Used 100 m2.4xlarge EC2 instances with 8 cores and 68GB of RAM each
- Queries
  - All pages
  - Pages with titles matching a given word
  - Pages with titles partially matching a word
  - Each query scanned the entire input data
- Took Spark 5—7 seconds
  - 170s from disk

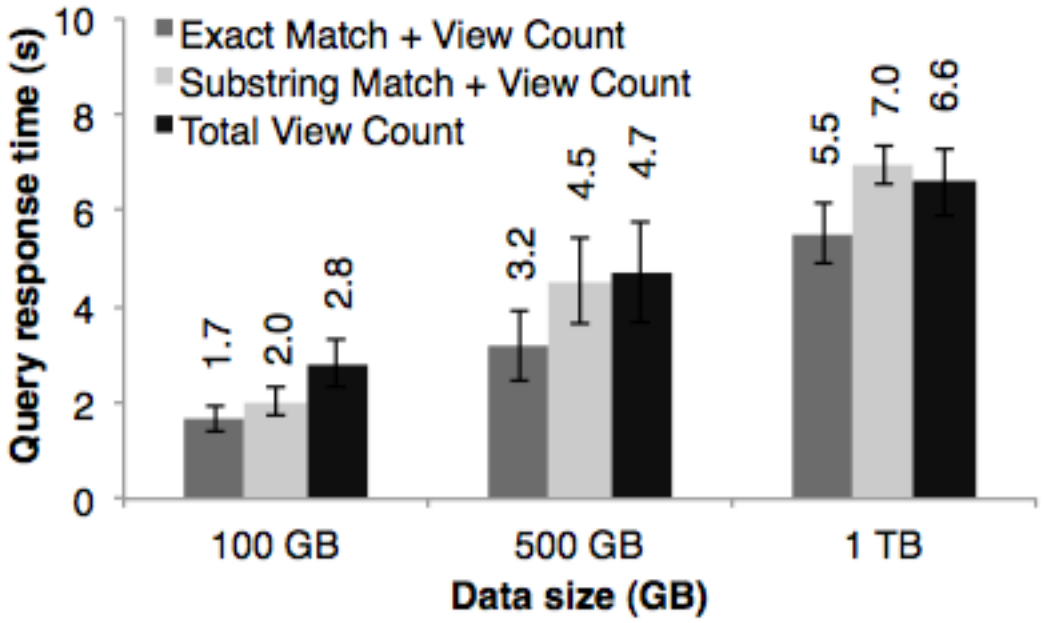


Figure 14: Response times for interactive queries on Spark, scanning increasingly larger input datasets on 100 machines.

Discussion

- RDDs immutable nature and coarse-grained transformations, suitable for a wide class of applications
- Can express many cluster programming models that so far been proposed as separate frameworks, allowing users tom compose these models in one program, and share data between them.

Expressing existing programming models

- RDDs can efficiently express a number of cluster programming models that have been proposed independently
- Also capture Optimizations
  - Keep specific data in memory
  - Partitioning to minimize communication
  - Recover from failures efficiently

Expressible models by RDDs

- MapReduce
  - By flatMap, groupByKey, or reduceByKey
- DryadLINQ
  - All bulk operators that correspond directly to RDD transformations (map, groupByKey, join, etc.)
- SQL
  - Like DryadLINQ expressions, SQL queries perform data-parallel operations on sets of records

Contd.

- Pregel
  - Google's specialized model for iterative graph applications
  - A program runs series coordinated "supersteps"; on each one, each vertex runs a user function that update state associated with vertex, change graph topology, and send messages to other vertices for use in the next superstep
  - It can express many graph algorithms, shortest paths, bipartite matching, PageRank

Contd.

- Pregel
  - Pregel applies same user function to all vertices on each iteration
  - Thus, can store vertex states for each iteration in an RDD, perform a bulk transformation (flatMap) to apply function and generate an RDD of messages
  - Then join this RDD with vertex states to perform message exchange
  - Have implemented Pregel as 200-line library on top of Spark

- Iterative MapReduce
  - Such as HaLoop, Twister, provide iterative MapReduce model, user gives the system a series of MapReduce jobs to loop
  - HaLoop implemented as a 200-line library using Spark

- Batched Stream Processing
  - Incremental processing systems for applications that periodically update a result with new data
  - E.g. , statistics about ad clicks every 15 minutes should combine intermediate state from previous 15-minute window with data from new logs
  - They perform bulk operations, but store application state in distributed file systems
  - Placing intermediate state in RDDs would speed up their processing

Why RDDs able to express these diverse programming models?

- Restrictions on RDDs have little impact in many parallel applications (though RDDs can only be created through bulk transformations)
- Many parallel programs naturally apply the same operation to many records – make them easy to express
- Immutability of RDDs not an obstacle, because on can create multiple RDDs to represent versions of the same dataset
- Many today's MapReduce applications run over file systems that do not allow updates to files, such as HDFS

Why previous frameworks have not offered the same level of generality?

- Because those systems explored specific problems that MapReduce and Dryad do not handle well, such as iteration, without observing that the common cause of these problems was a lack of data sharing abstractions

Leveraging RDDs for Debugging

- Initially designed RDDs to be deterministically recomputable for fault tolerance; it also facilitates debugging
- By logging lineage of RDDs created during a job
  - Reconstruct these RDDs later and let user query them interactively
  - Re-run any task from the job in a single-process debugger, by recomputing RDD partitions it depends on

- Traditional replay debuggers for general distributed systems
  - Must capture or infer the order of events across multiple nodes
- Adds virtually zero recording overhead because only the RDD lineage graph needs to be logged

Comparison with related work

- Data flow models like MapReduce, Dryad, and Ciel support operators for processing data but share it through stable storage systems
- RDDs represent a more efficient data sharing abstraction than stable storage because they avoid cost of data replication, I/O and serialization

Comparing contd.

- Previous high-level programming interfaces for data flow systems, like DryadLINQ, FlumeJava, provide language-integrated APIs, user manipulates "parallel collections" through ops like map and join
  - They cannot share data efficiently across queries
- Spark's API on parallel collection model, not claim novelty for this interface, but by providing RDDs as storage abstraction behind this interface

Contd.

- Pregel supports iterative graph applications, Twister and HaLoop are iterative MapReduce runtimes
  - They perform data sharing implicitly for pattern of computation they support, not provide a general abstraction that user can employ to share data of their choice among ops of their choice
- RDDs provide a distributed storage abstraction explicitly and can thus support applications those specialized systems do not capture, e.g. interactive data mining

- Some systems expose shared mutable state to allow user to perform in-memory computation
  - Piccolo – run parallel functions that read and update cells in a distributed hash table
  - Distributed shared memory(DSM) systems and key-value stores like RAMCloud
- RDDs provide a higher-level programming interface based on operators like map, sort, join, more complicated functions than those
- Those systems implement recovery through checkpoints and rollback, more expensive than lineage-based strategy of RDDs

Comparing with Caching systems

- Nectar, not provide in-memory caching (it places data in distributed file system), nor it let users explicitly control which dataset to persist and how to partition them
- Ciel and FlumeJava can cache task results but not provide in-memory caching or explicit control over which data is cached

Lineage

- Long been a research topic in scientific computing and databases
- RDDs provide a parallel programming model, where fine-grained lineage is inexpensive to capture, so can be used for failure recovery
- RDDs apply lineage to persist in-memory data efficiently across computations, with cost of replication and disk I/O

Relational databases

- RDDs are conceptually similar to views in a database, and persistent RDDs resemble materialized views
- Databases typically allow fine-grained read-write access to all records, requiring logging of operations and data for fault tolerance, additional overhead to maintain consistency
- Coarse-grained transformation model of RDDs not required with these overheads.

Sponsorships

Nightingale, and our reviewers for their feedback. This research was supported in part by Berkeley AMP Lab sponsors Google, SAP, Amazon Web Services, Cloudera, Huawei, IBM, Intel, Microsoft, NEC, NetApp and VMWare, by DARPA (contract #FA8650-11-C-7136), by a Google PhD Fellowship, and by the Natural Sciences and Engineering Research Council of Canada.