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

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica

Presentation by -

Zone Li, Shayar Shah, Aryan Pariani, Sai Anoop Avunuri, Wei Zhou

Georgia Tech

Introduction

Problem – Inefficiencies in Data Reuse for Cluster Computing

- **Need for Data Reuse** - Many modern applications need to **reuse intermediate results** across multiple computations / jobs:
  - **Iterative machine learning** algorithms: K-means, Logistic Regression
  - **Graph algorithms**: PageRank
  - **Interactive Data Mining**: where a user needs to run ad-hoc queries on the same data subset
- **Limitations of Existing Frameworks** - Limited and inefficient for **data reuse**:
  - **High Cost of External Stable Storage** (e.g. **MapReduce**, **Dryad**):
    - Rely on external stable storage (e.g. distributed file systems) rather than memory
    - Substantial overheads due to disk I/O, data replication
  - **Limitations of Specialized Frameworks** (e.g. **Pregel**, **HaLoop**):
    - Developed for iterative jobs
    - Limited to very specific patterns; lack generalization

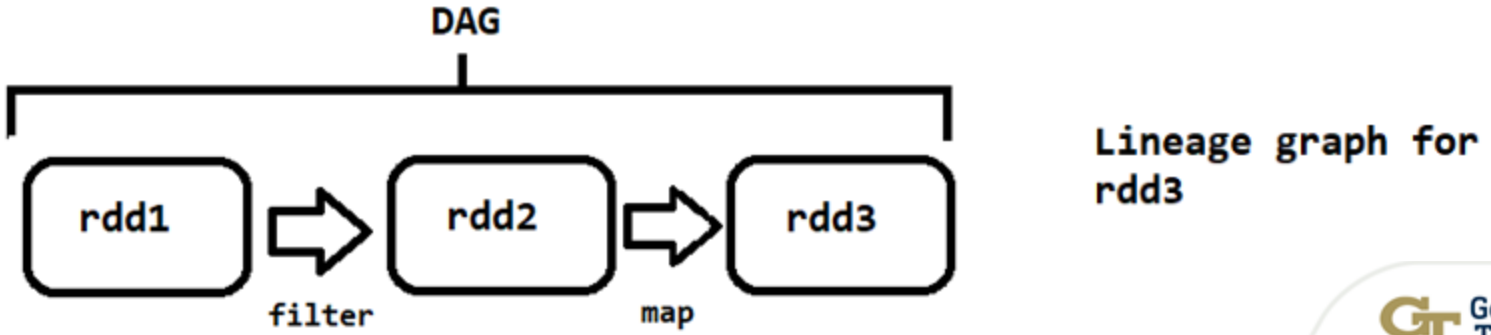
Solution – Resilient Distributed Datasets (RDDs)

What are RDD's?

- **Immutable collections** of objects

- **Distributed in memory** (faster access / write speeds than stable storage) across different nodes in a cluster.

- Equipped with **lineage** info for each RDD - Sequence (or DAG) of transformations applied to RDD's to compute



Benefits of RDD's

- **Why not other in-memory storage solutions?** Example: Piccolo, DSM
  - Apply **fine-grained updates** to mutable state (e.g. individual cells in a table)
  - **Inefficient Fault Tolerance**: Need to replicate intermediate data / log multiple updates across nodes
- **RDD's go above and beyond in solving this inefficiency!**
  - **Efficient Fault Tolerance**:
    - **Coarse-Grained Transformations**: RDD's use scalable operations applied to many data items at once, rather than multiple fine-grained updates
    - **Fast Recovery via Lineage**: Each RDD logs transformation lineage for efficient recomputation of lost partitions, avoiding replication overheads.
  - **Flexibility + Generalizability**: Can express different programming models that otherwise only require separate systems (e.g. MapReduce, SQL) and also new applications like interactive data mining

Resilient Distributed Datasets (RDDs)

Introduction to RDDs

- RDDs are **read-only, partitioned collections of records** designed for distributed computing.
  - Created through **deterministic transformations** on data in stable storage (e.g., HDFS) or from other RDDs.
  - Central abstraction of the Spark framework.
  - **Actions**: Operations (that trigger computation on RDDs and) requiring return results to the user. Examples include count, collect, and save.
  - **Transformations**: Operations that create new RDDs from existing ones (lazy i.e. don't execute until an action is called). Examples include map, filter, and join.
- **Analogy**: Think of lineage graphs in RDDs as **recipes**.
- **Purpose**:
  - To **optimize data reuse** in memory for iterative processes (e.g. ML, graph algorithms).
  - Enable efficient **fault recovery** by reconstructing data from transformations instead of full data replication.

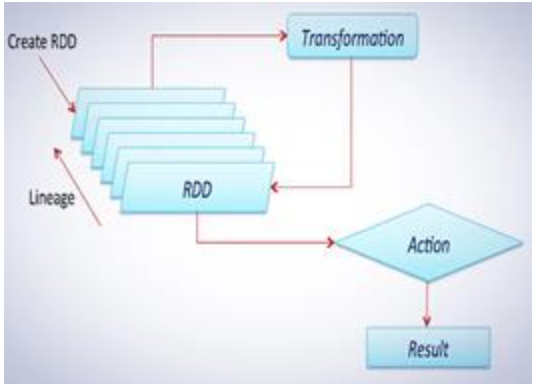


Image Cred: https://www.dataneb.com/post/sparkrdd-transformations-and-actions-example

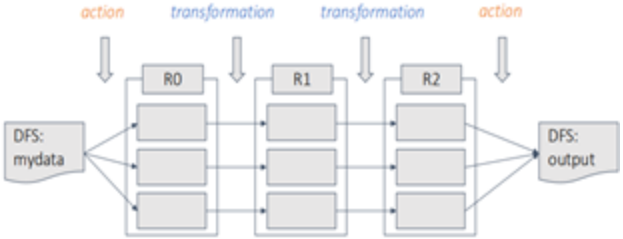


Image Cred: https://kexinrong.github.io/fa24-cs6400/assets/lectures/lec18-dqp.pdf

Key Properties of RDDs

- **Immutability**:
  - Once created, an RDD cannot be modified.
  - Simplifies fault tolerance, as RDDs can track their transformations in a lineage graph.
  - **Note**: create new RDDs representing different versions of data, simulating updates.
- **Deterministic Transformations**:
  - Generated through operations like map, filter, and join, known as coarse-grained transformations.
  - This allows Spark to apply the same operation to many data items simultaneously, optimizing processing.
- **Lazy Evaluation**:
  - RDD transformations are not immediately executed.
  - Spark builds a logical plan of transformations. Only executes when an action is called.
  - This lazy evaluation approach helps Spark to optimize execution plans, minimizing computation and memory use.
- **Persistence and Partitioning**:
  - **Persistence**: Users can choose to store and persist certain RDDs in memory for faster reuse, critical for iterative algorithms.
  - **Partitioning**: RDDs can be partitioned across nodes, improving data locality and reducing data shuffling during distributed processing.

Image Cred: https://realpython.com/python-lazy-evaluation/

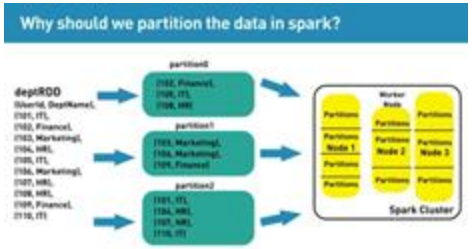


Image Cred: https://youtube.com/@BigDataElearning

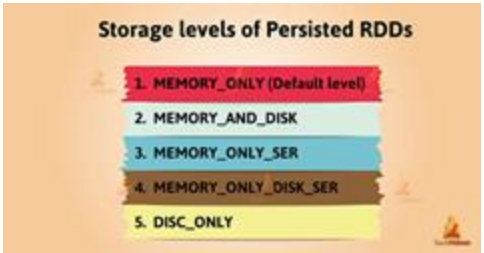


Image Cred: https://techvidvan.com/tutorials/persistence-and-caching-mechanism/

Example: Console Log Mining

- **Spark Programming Interface**:
  - Spark provides a language-integrated API for defining RDD transformations (e.g., map, filter) and actions (e.g., count, collect).
  - Transformations are *lazy*, creating a logical plan that Spark executes only when an action is called.

- **Example Query:** Return the time fields of errors mentioning 'HDFS'
  - Define lines RDD from HDFS.
  - Filter for "ERROR" messages → Create errors RDD.
  - Persist errors in memory for reuse.
  - Further transformations: filter for HDFS errors, then map to extract timestamp.
  - Finally, action `collect` is invoked to return all the time fields in the final RDD created after the map transformation.

- Spark's scheduler pipelines the `filter` and `map` transformations, dispatching tasks to nodes holding cached partitions of `errors`.
- **Fault Tolerance in Action**: If a partition of `errors` is lost, Spark can rebuild it by reapplying the original `filter` transformation on the corresponding partition of `lines`.

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```

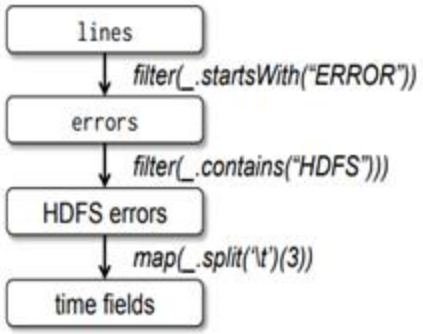


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

Image and Code Credits Resilient Distributed Datasets, M. Zaharia et al. 2012

Advantages over Existing *In-Memory* Storage Abstractions

Feature	Existing In-Memory Storage Abstractions (e.g., DSM, Key-Value Stores, Databases, Piccolo)	RDDs (Resilient Distributed Datasets)
Data Updates	**Fine-grained** updates to mutable state (e.g., individual cells or key-value pairs)	**Coarse-grained** transformations (`map`, `filter`, `join`) applied to entire datasets
Fault Tolerance Approach	• **Replication** of data / Logging of fine-grained updates across nodes • **Checkpoints** • **High Cost** (Expensive and slow) - Requires significant network bandwidth and storage	• Logging transformations (**lineage**) used to build the dataset and recompute lost partitions • **Lower Cost** (Fast and efficient) - Avoids data replication
Scalability for Data-Intensive Workloads	**Limited scalability** - High storage and network overhead - struggles with memory limitations	**High scalability** - Minimal storage and network overhead due to lineage-based recovery - can spill partitions to disk gracefully
Flexibility	**Low** - Separate frameworks needed for expressing different programming models (MapReduce, SQL, etc.)	**High** - Can express multiple programming models (MapReduce, SQL, etc.) and new applications like interactive data mining

Other Advantages of the RDD Model

Comparison with Distributed Shared Memory (DSM):

- **Straggler Mitigation**: RDDs' immutability allows for running backup tasks without data consistency issues, similar to MapReduce.

- **Work Placement/Data Locality**: RDDs support automatic data placement and task placement based on locality to keep them close, which enhance performance and minimizes latency.

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.

Image Credits: Resilient Distributed Datasets, M. Zaharia et al. 2012

Applications Not Suitable for RDDs

- **Suitable Applications**: RDDs are ideal for **batch processing** where the same operation is applied to all elements in a dataset (e.g., data mining, ML training).

- **Limitations**:
  - **Not Suitable for Fine-Grained Asynchronous Updates**: Applications requiring fine-grained, asynchronous updates for applications needing frequent updates are better served by systems designed for mutable data..
  - **Examples**: Real-time web applications and incremental web crawlers are better served by traditional databases or DSM-based systems.
  - **Alternatives**: Systems like **RAMCloud**, **Percolator**, and **Piccolo** handle such tasks better as they support fine-grained updates and checkpointing.

Spark Programming Interface

RDD Operations in Spark

- **Spark's RDD API**: Provides 2 types of RDD operations for manipulating and analyzing large datasets - **transformations** & **actions**
- **Transformations** (e.g., map, filter, join):
  - **Lazy** (don't execute immediately), **coarse-grained** operations that **produce a new RDD from one or more RDD's** (e.g. **RDD[T] ⇒ RDD[U]**)
  - Only executed when a **subsequent action is called**, for optimal computation via lineage
- **Actions** (e.g., count, collect, save): Produce user-facing outcomes
  - Trigger actual computation to **produce non-RDD results from RDD's** (e.g. **RDD[T] ⇒ U**, **RDD[T] ⇒ Seq[U]**) or **write a RDD to a storage system**
  - Execute lineage of all preceding transformations before action itself

```
val data = sc.textFile("input.txt")     // No execution yet
   .map(line => line.split(" "))        // Not executed
   .filter(words => words.length > 2)   // Still not executed
   .count()                             // Now it executes everything
```

Transformations & Actions

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.

Key Transformations:

- **map**: A one-to-one mapping function that transforms each item in RDD.
- **filter**: Selects elements based on a specified condition.
- **flatMap**: Similar to map but allows for one-to-many mapping, commonly used for splitting text.
- **join**: Combines two RDDs based on a common key (like an inner join)

Key Actions:

- **count**: Quickly gives the number of elements in an RDD
- **collect**: Returns a list of all the elements in an RDD
- **reduce**: Combines elements using a specified function, used for aggregations
- **save**: Outputs the RDD to external storage, enabling data persistence

Example Application - Logistic Regression

- **Algorithm Overview**: ML algorithm for binary classification (e.g., spam detection)
  - Uses **gradient descent** to find optimal weights by initializing weights **w** randomly and then **iteratively computing gradients** and updating **w** to minimize classification error on training data
- **Spark Program and Approach**:
  1. Load points from a text file; use **map transformation** to parse lines into a **Points RDD** and explicitly **persist** it in memory for **prioritized** reuse
  2. Initialize weight vector **w** randomly
  3. In each iteration of a loop:
     - **a.** **map transformation**: Compute gradients for each point **Points RDD** using a common formula with **w**, creating a **new RDD** (call it **PointGradients**) of per-point gradients
     - **b.** **reduce action**: Sum gradients for all points in **PointGradients RDD** to compute total **gradient**
     - a. Finally, update **w** by subtracting the **gradient**

```
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
}
```

Achieves up to 20x Speedup!

Representing RDDs

Attributes of an RDD

An RDD is represented in Spark using the following 5 attributes:

- A set of partitions, which are atomic pieces of the dataset
- A set of dependencies on parent RDDs
- An iterator computes the RDD using its parents
- Partitioner describes if dataset is hash/range partitioned
- Preferred location stores which node each partition is stored on

Dependencies

Narrow dependency means each partition of a parent RDD is used by at most one child partition. Wide dependency means each partition of a parent RDD is used by multiple child partitions.

Advantages of narrow dependencies:
1) Pipelining of transformations on one node. Wide dependencies need to reshuffle.
2) Recovery after node failure is fast since lost parent partitions can be recomputed in parallel on different nodes.

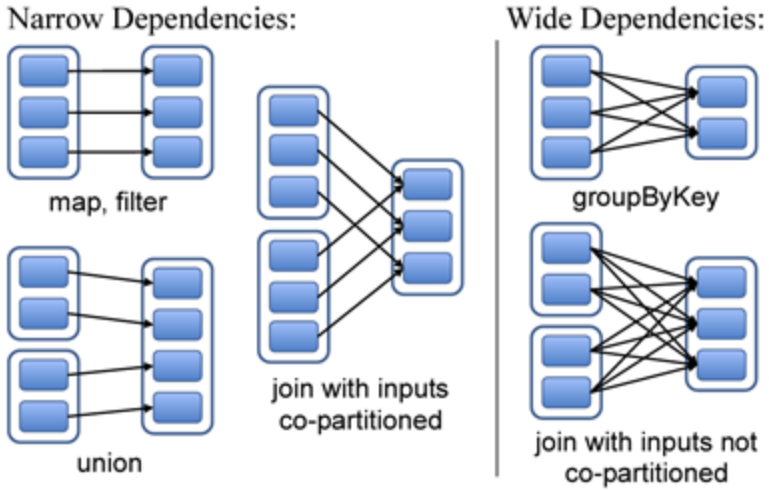


Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

A Few RDD Implementations

- HDFS files: Input RDDs. Each block of the file is a partition. Iterator reads the block and preferred location tells us which node has the block.
- Map: Creates a mappedRDD object by applying the function in its iterator to parent RDD partitions. Has same partition and preferred location as parent.
- Union: Creates an RDD that combines partitions from two RDDs, creating a narrow dependency.
- Sample: RDD stores a random number generator seed for each partition so that it can deterministically sample records.
- Join: Creates an RDD by joining two RDDs. Narrow dependency if parents are partitioned, else wide dependency.

Implementation

Job Scheduling

When an action is run on an RDD, the scheduler:

1. Looks at the **RDD's lineage graph** to build a **Directed Acyclic Graph (DAG)** of stages of as many pipelined transformations with narrow dependencies as possible
2. Then launches tasks to compute missing partitions until the target RDD has been computed

Upon task failure:

- **If stage parents are available**, it will simply be rerun on another node
- **If unavailable**, tasks need to be resubmitted to compute missing partitions in parallel

Due to scheduler optimization of tasks:

- Less need for shuffling data → enhanced performance

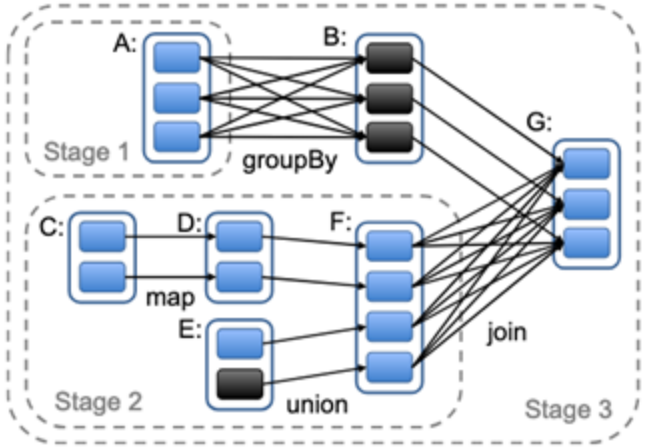


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case,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

Memory Management

Three options of storage of persistent RDD's are provided by Spark:

1. In-memory (RAM) storage as **deserialized Java object**
   - Fastest, since Java virtual machine can access each element natively
1. In-memory (RAM) storage as **serialized data**
   - Allows for user input on choosing a more memory-efficient representation when RAM is limited
   - May lower performance
1. On-disk storage
   - Useful are large RDDs, but costly to recompute on each use

Least Recently Used (LRU) eviction policy is employed at RDD level to manage limited memory

- Exception is made when RDD is the same as the one with the new partition
- Importance: most operations will require tasks to be run over entire RDD so in-memory partitions are likely to be reused

Support for Checkpointing

Why checkpoint to stable storage when RDDs can be recovered through their lineage?

- For RDDs with longer lineage chains, recovery can be time-consuming
- Especially useful for RDDs containing **wide dependencies** which would require full recomputation upon node failure
  - e.g. joining two tables on a key
- For more **narrow dependencies**, checkpointing is less necessary as recomputation is low-cost
  - e.g. filtering records of a single table

Easy to checkpoint because of the read-only nature of RDDs

- Compared to general shared memory

Evaluation

Iterative ML Applications

- The authors test Spark against Hadoop and HadoopBinMem, which converts input to binary format in first iteration and stores it in in-memory HDFS to avoid parsing in later iterations.

- Spark was 3x faster for K-means and 20x faster for logistic regression. This difference is because k-means is more compute intensive and I/O and serialization costs don't affect it as much.

- We also see that performance scales approximately linearly with number of nodes.

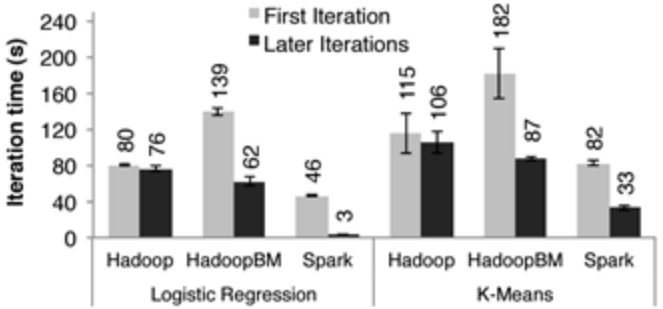


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

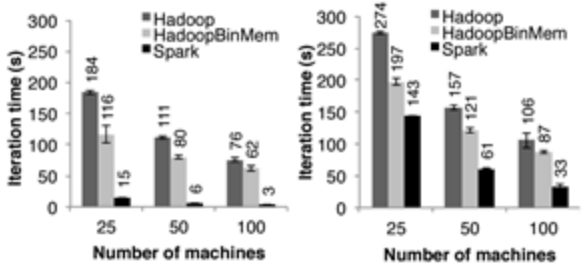


Figure 8: Running times for iterations after the first in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB.

Reasons for Speedup

The main reasons why Spark performed better were

1) Hadoop has operational overhead (a no-op Hadoop job takes 25 sec)

2) Hadoop has overhead when serving data (performs checksums)

3) Hadoop needs to deserialize binary data into Java objects before using.

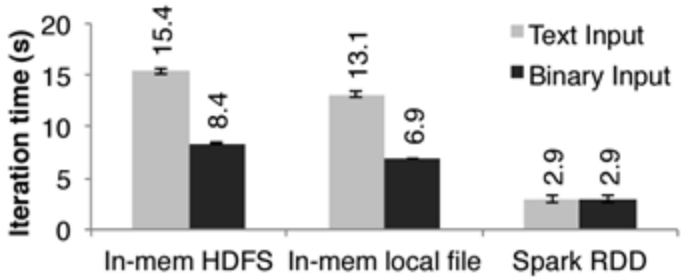


Figure 9: Iteration times for logistic regression using 256 MB data on a single machine for different sources of input.

Fault Recovery

- Before iteration 6, a node was killed, so its partitions were recalculated on other nodes in parallel, which is why it took longer. Afterwards, the nodes went back to normal iteration times.

- Checkpointing system would have required rerunning several iterations of the algorithm.

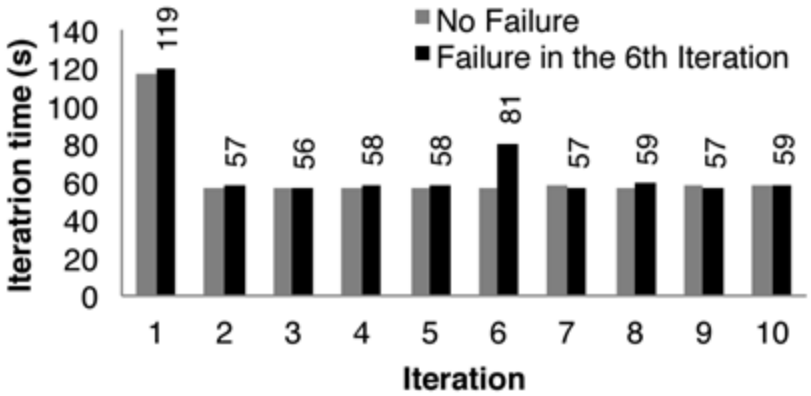


Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

Performance vs Cluster Memory

- Researchers wanted to see how the performance varies if the cluster doesn't have memory to store all of the RDD data in memory.

- We can see that performance degrades reasonably when we lower memory available.

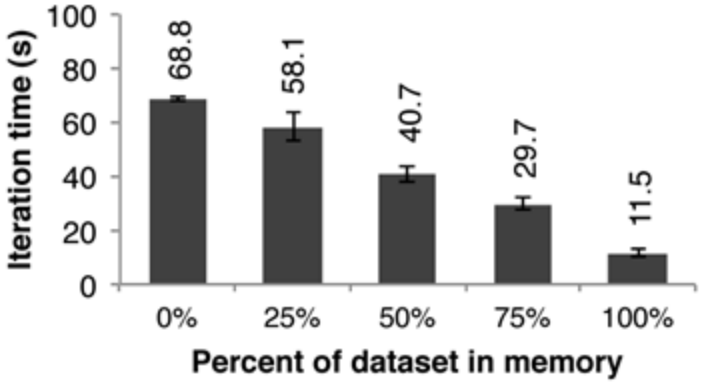


Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

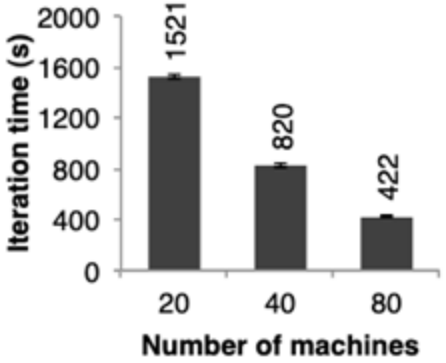
User Applications: Traffic Modeling

Conducted by the Mobile Millennium project at Berkley

- Parallelized a learning algorithm to infer road traffic congestions
  - 10,000 link road network
  - 600,000 samples of point-to-point trip times from sporadic automobile GPS data
- Estimates travel time across individual road links
  - Trained using an **expectation-maximization algorithm** by repeating the *map* and *reducebykey* steps iteratively

Demonstrates starks ability to perform rapid iterative computations

Scales nearly linearly from 20 to 80 nodes with 4 cores each



(a) Traffic modeling

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

Interactive Data Mining

1 TB of Wikipedia view logs (2 years)

Three queries were **tested on entire input data**

1. Total view count of all pages
2. Pages with titles ***exactly matching*** a given word
3. Pages with titles ***partially matching*** a given word

For context: querying 1TB file **on disk** took **170 seconds** (~3 min)

Confirms that RDD framework makes Spark a powerful tool for interactive data mining

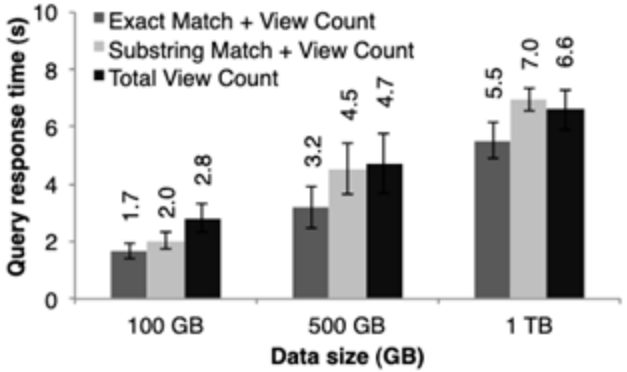


Figure 14: Response times for interactive queries on Spark, scanning increasingly larger input datasets on 100 machines.

Discussion

Expressing Existing Programming Models in Spark

RDDs can express a variety of programming models.

Programming Models Supported by RDDs in Spark:

- **MapReduce**: Expressed using flatMap and groupByKey operations.
- **DryadLINQ**: Direct correspondence to RDD transformations (e.g., map, groupByKey, join).
- **SQL**: Data-parallel operations naturally supported.
- **Pregel**: Iterative graph processing achieved by storing vertex states in RDDs, transforming RDDs through bulk operations and join with the vertex states to perform message exchanges.
- **Iterative MapReduce** (HaLoop, Twister): Achieved through consistent data partitioning across iterations.

Benefits: RDDs can capture the optimizations each framework provide, like efficient memory usage, minimized communication through partitioning, fault-tolerant data recovery via lineage

Leveraging RDDs for Debugging

Tracking RDD Lineage for Debugging:

- The deterministic recomputation ability of RDDs can assist in debugging.
- Lineage logs allow:
  - Reconstruction of RDDs for interactive querying.
  - Single-process task re-execution by recomputing dependent RDD partitions.

Key Advantages:

- Contrasts with traditional replay debuggers that require capturing complex event orders across distributed nodes.
- Near-zero recording overhead (only lineage graph needs to be logged).
- Enables efficient task re-execution and data inspection during the debugging process

Related Work

Prominent Related Systems and Concepts:

Traditional Data Flow Models:

- MapReduce and Dryad: Provide parallel processing capabilities but rely heavily on stable storage for data sharing. Overhead due to data replication, serialization, and I/O costs.
- Spark & RDDs: Use in-memory data sharing and efficient lineage-based recovery, improve both latency and throughput.

Specialized Iterative and Graph Processing Frameworks:

- Pregel, Twister, HaLoop: Focus on iterative computations with built-in data reuse for specific patterns (e.g., graph processing, iterative MapReduce loops).
- Spark & RDDs: Provides a more general-purpose approach that can express and optimize these specialized models while allowing broader flexibility.

Caching and Data Sharing Systems:

- Nectar: Automatically manages data sharing by identifying common subexpressions, but lack in-memory caching.
- Spark & RDDs: Offers explicit control over which datasets to persist in-memory, reducing I/O costs and providing more efficient data sharing.

Shared Mutable State Systems:

- Piccolo, DSM: Provide fine-grained access to shared state with checkpointing and rollback-based recovery.
- Spark & RDDs: Use coarse-grained transformations that simplify and accelerate fault recovery using lineage tracking, improving overall system scalability and fault tolerance.

Conclusion

Conclusion

RDDs as a Transformative Abstraction:

- Efficient data sharing across cluster applications.
- Supports iterative computations, graph processing, and data mining with minimal data replication and serialization overhead.

Spark's Impact on the Industry:

- Widely adopted in both academic and industrial applications, delivering substantial performance improvements.
- Interactive use-cases supported by fast, in-memory data processing.

Future of RDDs and Spark:

- Expected continued influence on scalable and interactive data analytics.
- Active developments focused on enhancing data caching, lineage-based debugging, and interactive processing capabilities

Study Questions

Explain how RDDs (Resilient Distributed Datasets) provide a more efficient data sharing mechanism compared to traditional distributed memory systems and discuss the benefits associated with using coarse-grained transformations for fault tolerance.

How does the job scheduler in Spark improve query performance using in-memory RDDs and dependencies?