Title Page

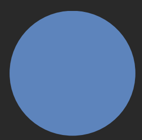
Team8

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

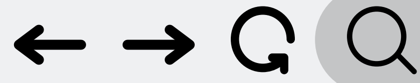
**Authors: Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica; University of California, Berkeley**

**NSDI'12, April 2012**

**Minjun Kim, Nabin Kim, Sandra Kurian, Seohee Yoon**



Title Page Background



# Efficient Management of Distributed Memory

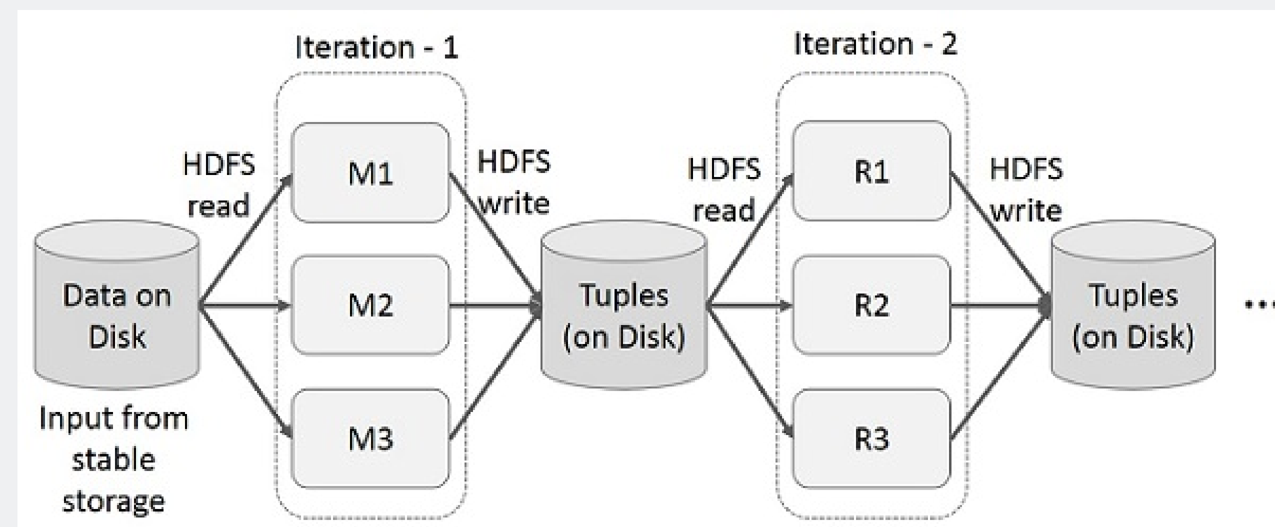



Figure 1.**Iterative** Operations on MapReduce

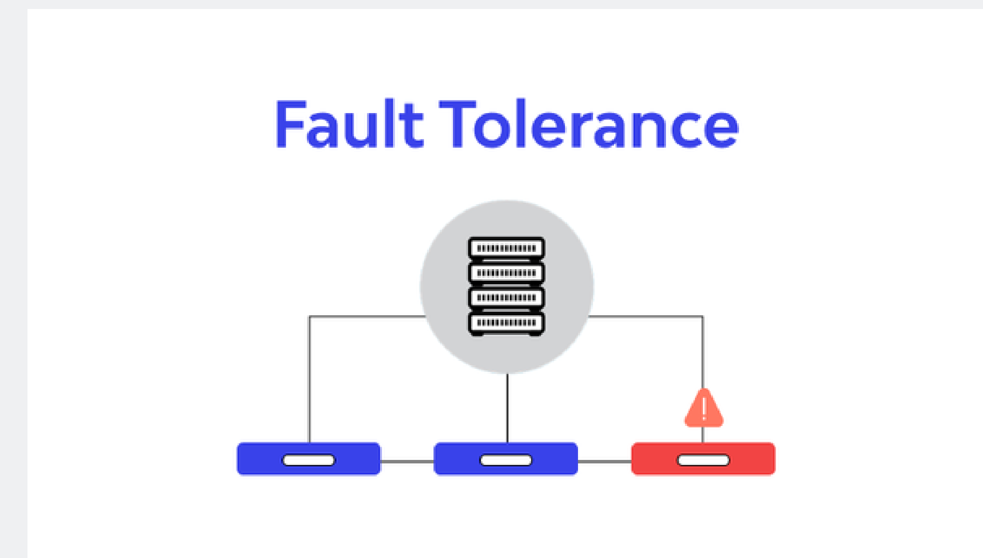



Figure 3. Fault Tolerance

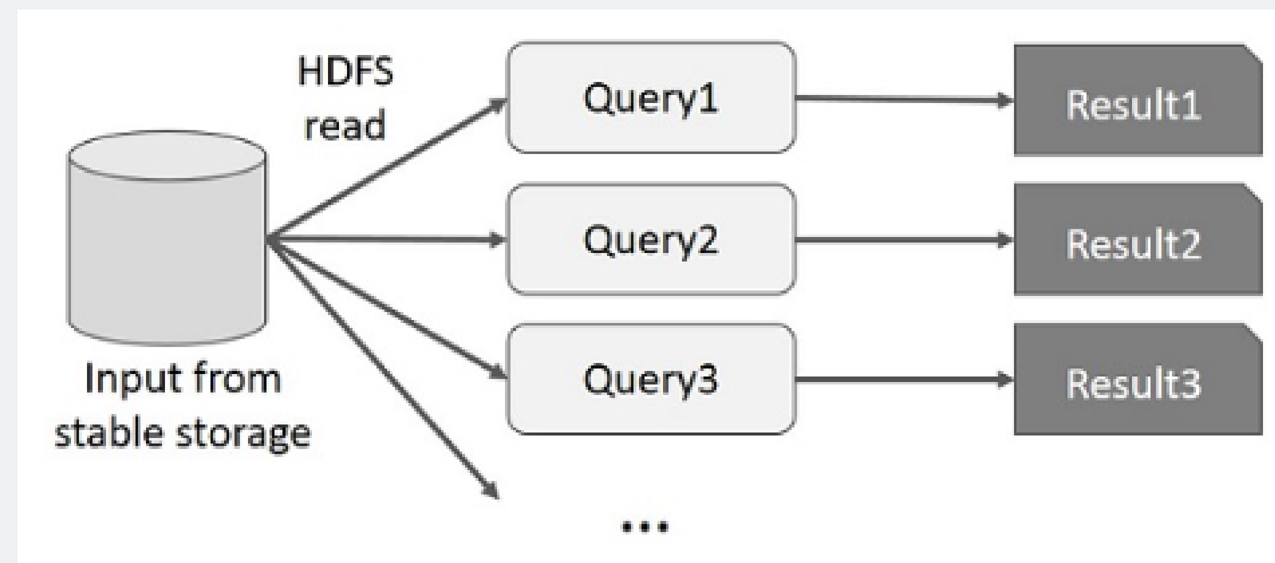
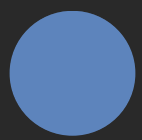
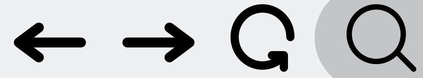



Figure 2. **Interactive** Operations on MapReduce

How to design a distributed memory abstraction that is **interactive**, **iterative**, and **fault-tolerant**?


Lee, I. (n.d.). Fault tolerance. Wallarm. Retrieved April 10, 2024, from https://www.wallarm.com/what/what-is-fault-tolerance.
Interactive Operations on MapReduce. (n.d.). Tutorials point. Retrieved April 10, 2024, from https://www.tutorialspoint.com/apache_spark/apache_spark_quick_guide.htm.
Iterative Operations on MapReduce. (n.d.). Tutorials point. Retrieved April 10, 2024, from https://www.tutorialspoint.com/apache_spark/apache_spark_quick_guide.htm.

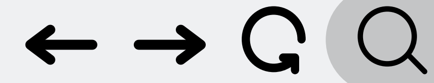

Title Page
Background



# RDD Abstraction

- Resilient Distributed Datasets (Spark, parallel data processing)
  - Creation:
    - From Stable Data.
    - From Other RDDs.
  - Transformations: Deterministic operations like map, filter, and join
  - Lineage: Records RDD derivation, enabling failure recovery
  - User Control: Management of persistence and partitioning

Title Page | Background | Motivation



# Spark Programming Interface

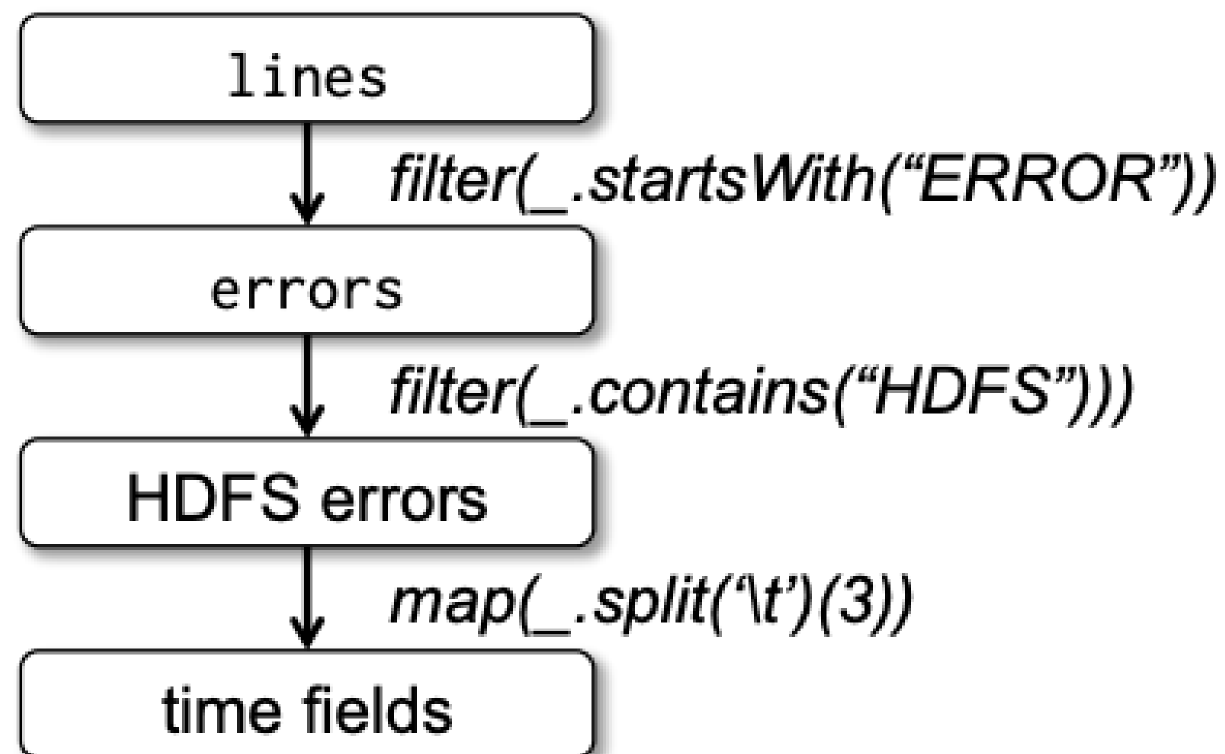



Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

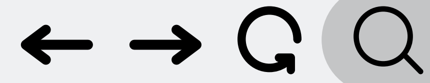
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

```
errors.count()
```

```
// 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()
```

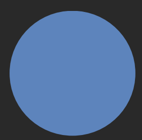
Figure 5, 6, 7. Code Snippet

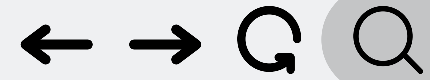


# RDD vs DSM (Distr. Shared Mem.)

| 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?) |

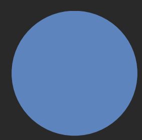
Table 1: Comparison of RDDs with distributed shared memory.

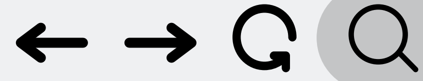




# Applications Not Suitable for RDDs

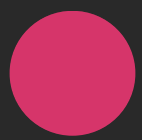
- RDDs unsuitable for asynchronous fine-grained updates to shared state
  - Multiple processes concurrently update data non-uniformly
  - Examples: Storage systems for web apps / Incremental web crawlers
- Traditional systems with update logging and data checkpointing are more efficient
  - Examples: RAMCloud, Percolator, Piccolo
- RDDs focus on efficient batch analytics rather than handling asynchronous updates

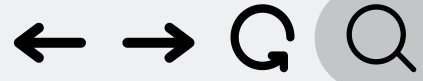




# Overview

- RDDs reduce the reliance on disk I/O and data replication for iterative algorithm and interactive data mining tools
- It stores data in memory and uses lineage information to reconstruct lost data
- Not only accelerates data processing but also simplifies fault recovery
- Spark with RDDs is 20x faster than Hadoop for iterative applications and speeds up data analytics reports by 40x





# Spark Programming Interface - Application

**Transformations**

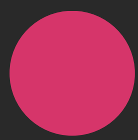
- Take existing RDD and produce new RDD
- Form Lineage graph for RDD's dependencies
- EX) map, filter, sample...

**Actions**

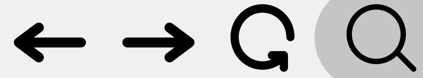
- Launch a computation to return a value
- EX) count, collect...

| | | |
|---|---|---|
| **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.







# Spark Programming Interface - Application

## Logistic Regression

- A common classification algorithm that searches for a hyperplane w that best separates two sets of points
- EX) Classify spam and non-spam emails
- Using a persistent RDD in Spark, we can reuse intermediate results across multiple computations, resulting in 20x speedup

```
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
}
```

Figure 8 Code Snippet

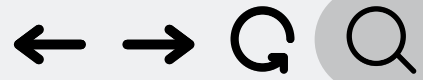
# Representing RDDs

## Spark proposes representing RDDs through interfaces

- A set of partitions
- Accessible nodes list from p
- 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 |

Table 3: Interface used to represent RDDs in Spark.



# Representing RDDs

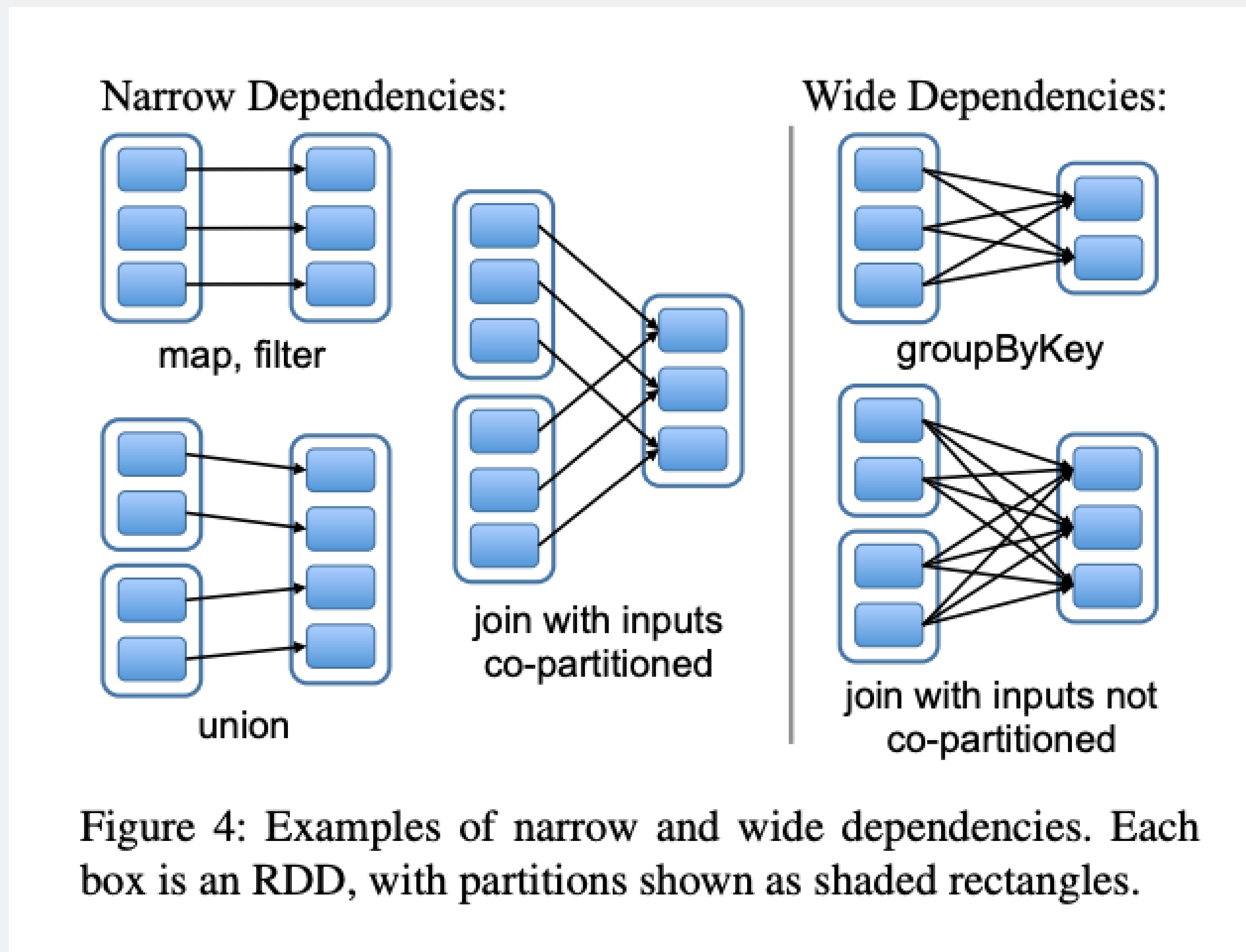
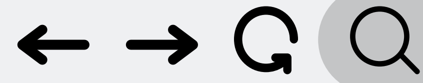


Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

- Narrow dependencies (Preferred)
  - A parent RDD is used by at most one child RDD
  - Pipelined execution on one cluster node to compute all parent partition
  - Only the lost parent partitions need to be recomputed
- Wide dependencies
  - A parent RDD is used by multiple child RDD
  - All parent partitions are required to be available
  - A single failed node might cause a complete re-execution
- Sketched Implementation
  - HDFS files, map, union, join



# Implementation - Job Scheduling

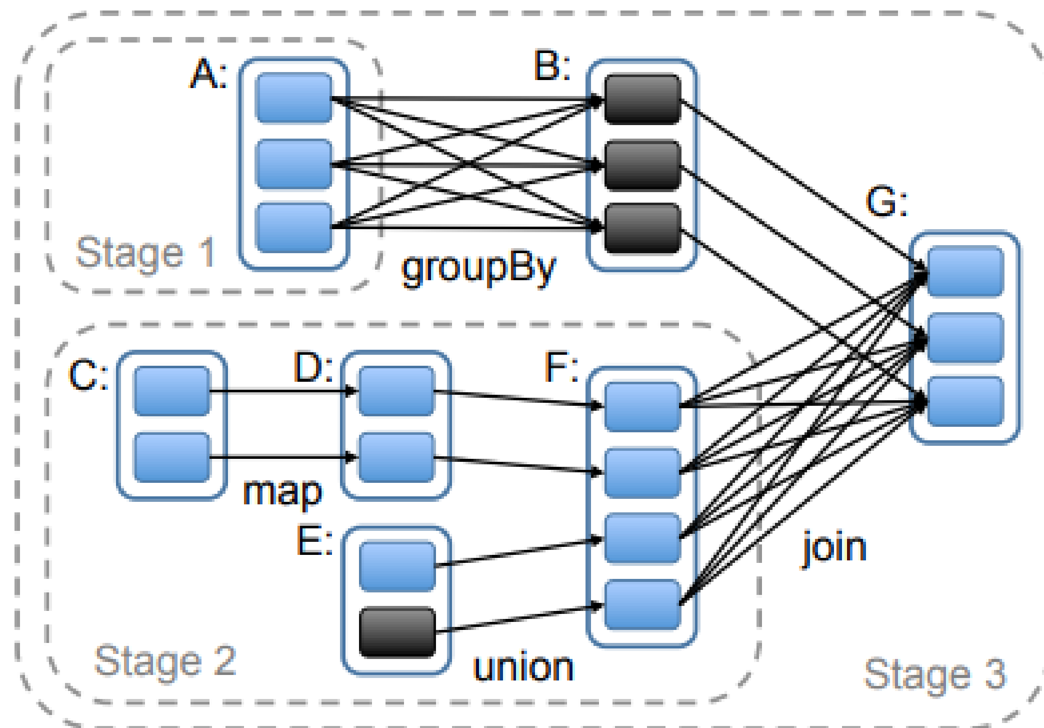



Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case,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

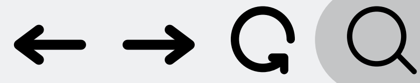
When an action is run, Spark's scheduler builds a Directed Acyclic Graph (DAG) of stages based on the RDD's lineage graph, as shown in Figure 5.

- Each of the stages contain as many pipelined transformations with narrow dependencies as possible.
- The boundaries of the stages are the wide dependencies and any already computed partitions.

The scheduler assigns tasks based on locality, using delay scheduling.

For wide dependencies, Spark materializes intermediate records making fault recovery easier.

Title Page Background Motivation Overview Spark Pro Rep Implementation



# Implementation - Interpreter Integration

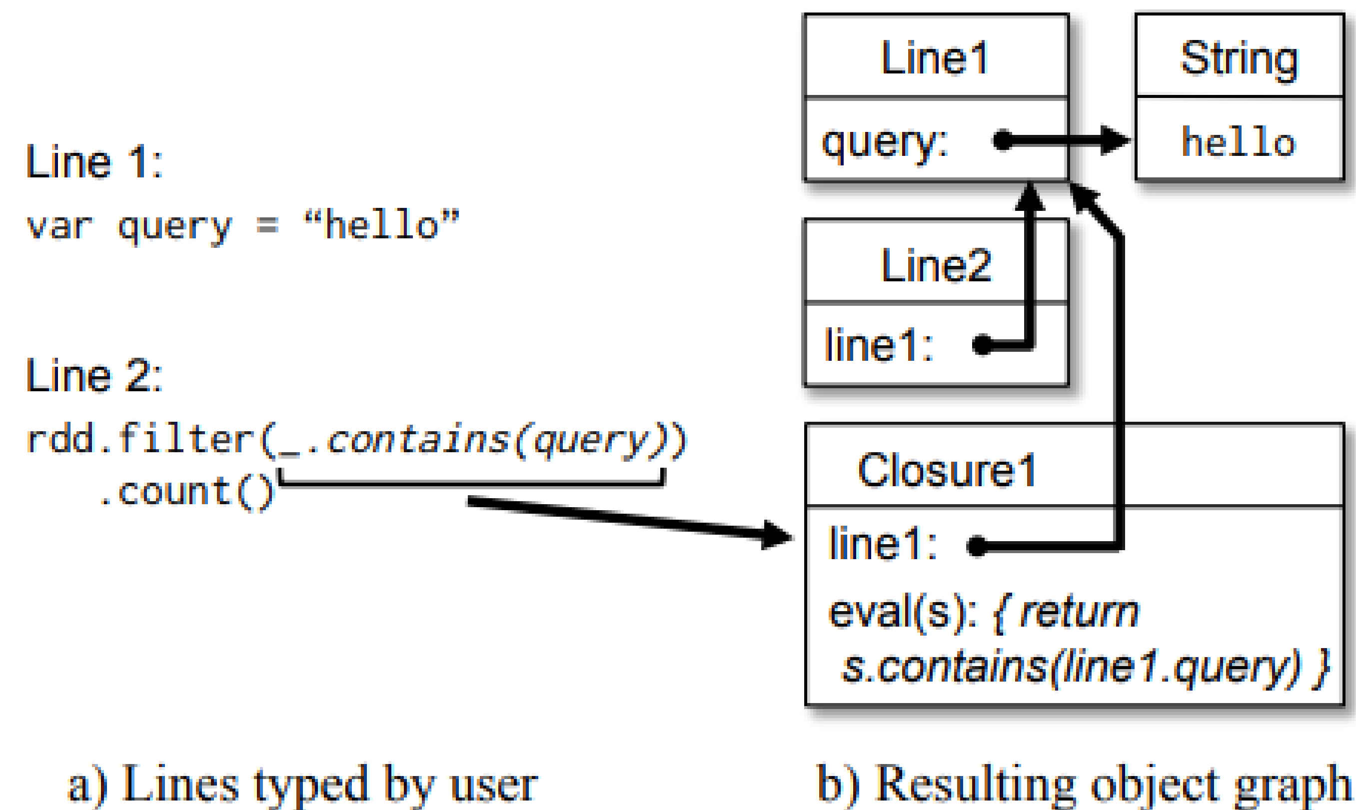



Figure 6: Example showing how the Spark interpreter translates two lines entered by the user into Java objects.

Spark has an in-memory Scala interpreter option to query big datasets.

Figure 6 shows how the modified interpreter translates a set of lines typed by the user to Java objects.

# Implementation - Memory Management

Spark provides three options for storage of persistent RDDs:

- In-memory storage as deserialized Java objects (fastest performance as Java VM can access RDD elements natively)
- In-memory storage as serialized data (memory-efficient representation compared to Java object graphs when space is limited.)
- On-disk storage (for RDDs that are too large to keep in RAM but costly to recompute on each use)

To manage the limited memory available, Spark uses an LRU (Least Recently Used) eviction policy at the level of RDDs.

When a new RDD partition is computed but there is not enough space to store it, Spark evicts a partition from the least recently accessed RDD, unless this is the same RDD as the one with the new partition. In that case, Spark keeps the old partition in memory to prevent cycling partitions from the same RDD in and out.

# Implementation - Support for Checkpointing

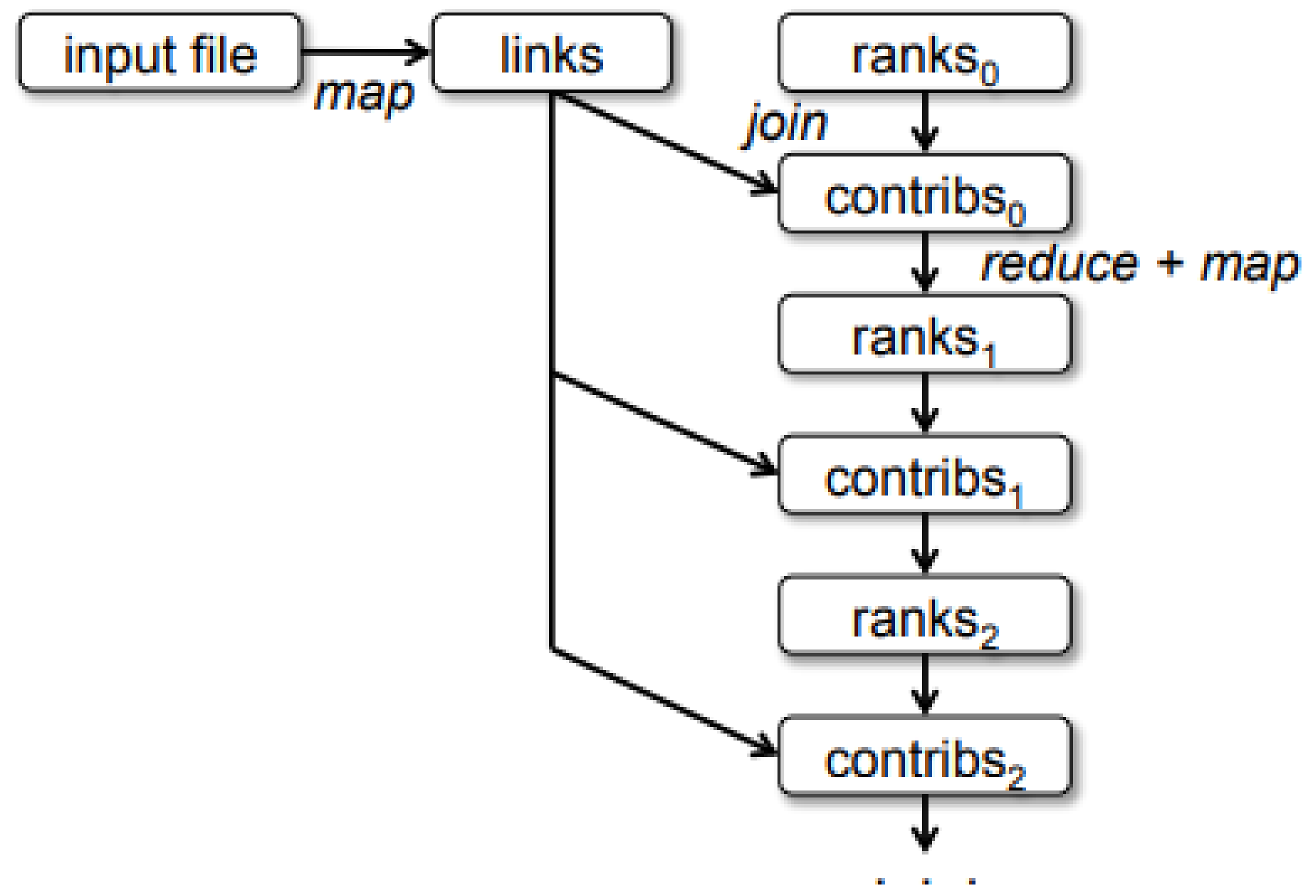
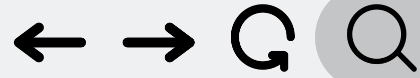



Figure 3: Lineage graph for datasets in PageRank.

Checkpointing is useful for RDDs with long lineage graphs containing wide dependencies. Figure 3 shows an example of a lineage graph for datasets in PageRank where checkpointing may or may not be helpful.

In these cases, a node failure in the cluster may result in the loss of some slice of data from each parent RDD, requiring a full recomputation. So, checkpointing will come in handy.

In contrast, for RDDs with narrow dependencies on data in stable storage, checkpointing may never be worthwhile.



# Evaluation

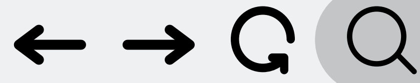
Previously computing frameworks were inefficient in:

- iterative algorithms
- interactive data mining tools

compared Spark against

- Hadoop (standard)
- HadoopBinMem (modified Hadoop)
    - input data is converted into a low-overhead binary format in the first iteration and stored in an in-memory HDFS instance
    - comparison highlights the benefits of in-memory storage and optimized data formats

evaluated through Amazon EC2



# Evaluation

Task: read text input from HDFS (Hadoop Distributed File System)

First iterations of their experiments

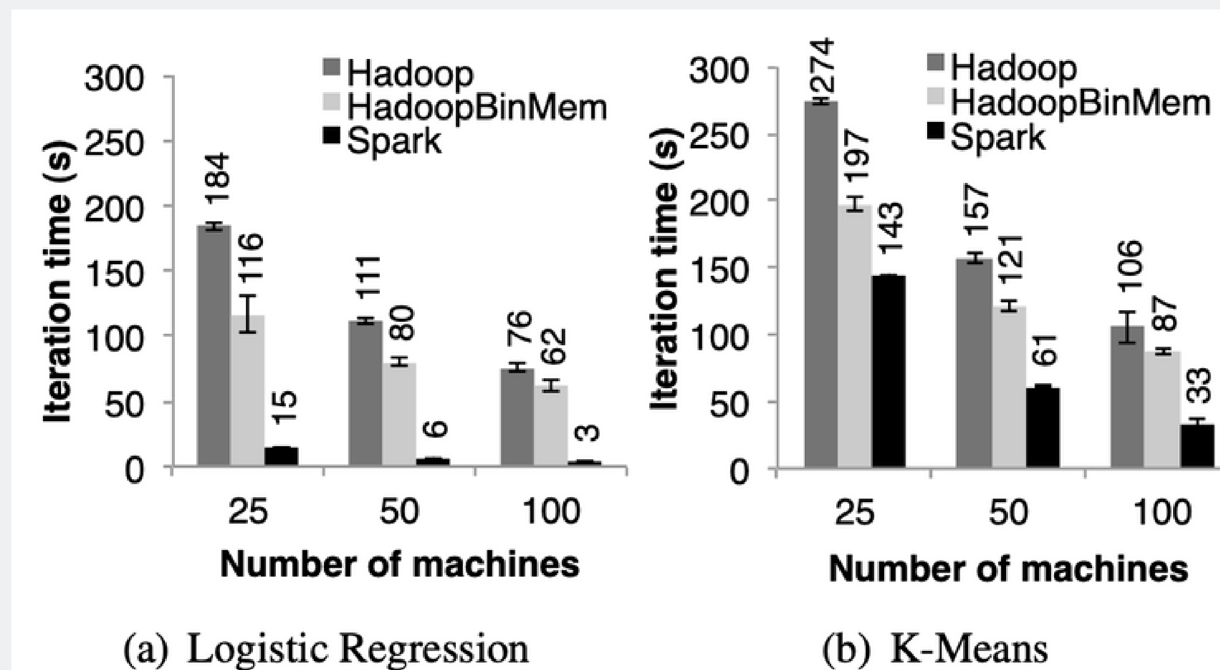
- HadoopBinMem slowest b/c convert data to a binary format

In subsequent iterations,

- logistic regression application
  - 25.3× faster than Hadoop
  - 20.7× faster than HadoopBinMem.
- k-means application
  - speedups ranging from 1.9× - 3.2×

Spark's efficiency in handling subsequent iterations and scaling with cluster size compared to traditional Hadoop-based approaches

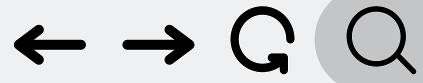
Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.




(a) Logistic Regression (b) K-Means

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

Title Page Background Motivation Overview Spark Program Evaluation



# Evaluation

Task: performance comparison between Spark and Hadoop for PageRank using a 54 GB Wikipedia dataset

10 iterations of the PageRank algorithm on a link graph, ~4 million articles.

- in-memory storage: 2.4× speedup w/t 30 nodes
- controlling the partitioning: 7.4 speedup w/t 30 nodes
    - ensure consistency across iterations
- linear scaling up to 60 nodes

Spark's efficiency in handling iterative algorithms like PageRank, leveraging in-memory storage and optimized RDD partitioning to achieve substantial performance gains over Hadoop, with scalable performance observed across a growing number of nodes.

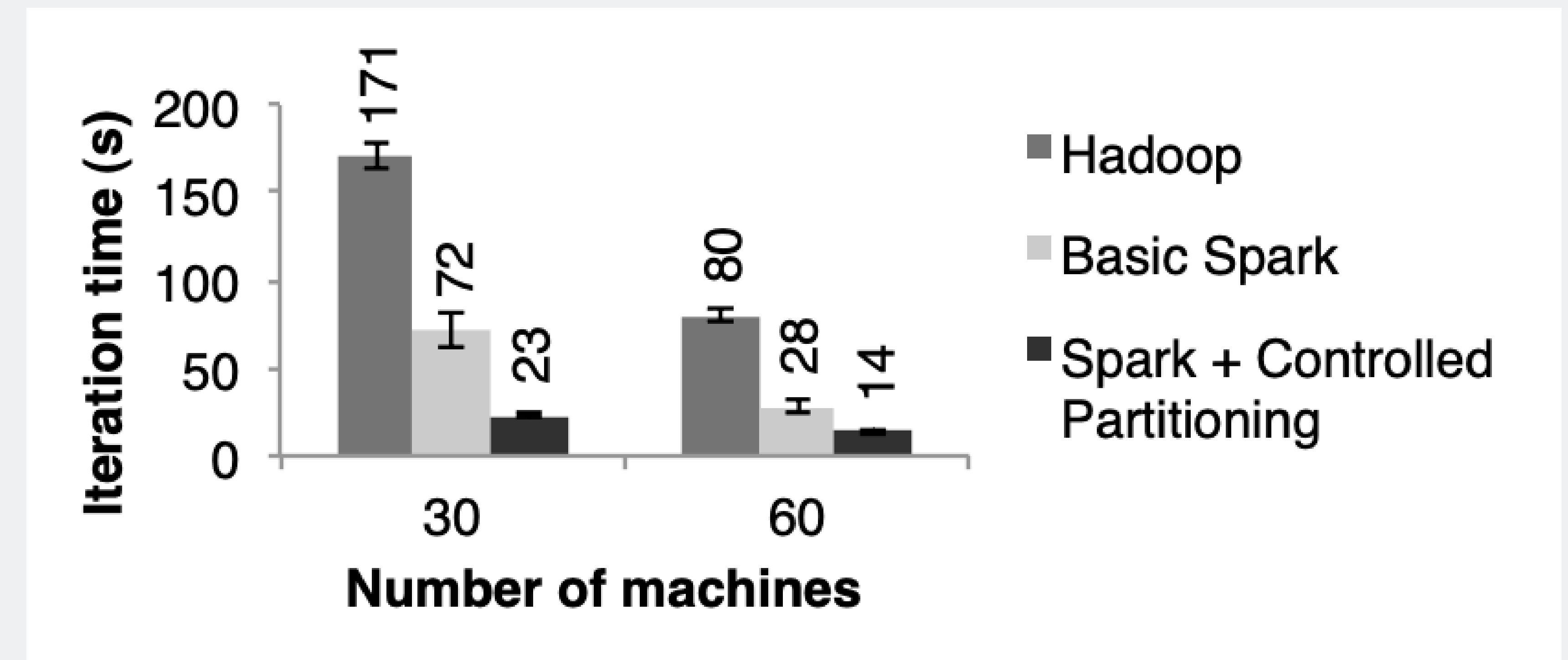
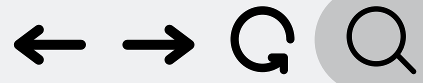



Figure 10: Performance of PageRank on Hadoop and Spark.



# Evaluation

Task: assess the cost of reconstructing RDD partitions using lineage after a node failure during the execution of a k-means application

- 10 iterations of k-means algorithm on a 75-node cluster under normal conditions -vs- when a node failed at the start of the 6th iteration
- ~58 seconds until the end of the 5th iteration.
- 6th iteration: node fail
  - tasks and partitions were lost
  - re-executed these tasks in parallel (re-reading and reconstructing RDDs)
  - 80 seconds.
- returned to 58 seconds.

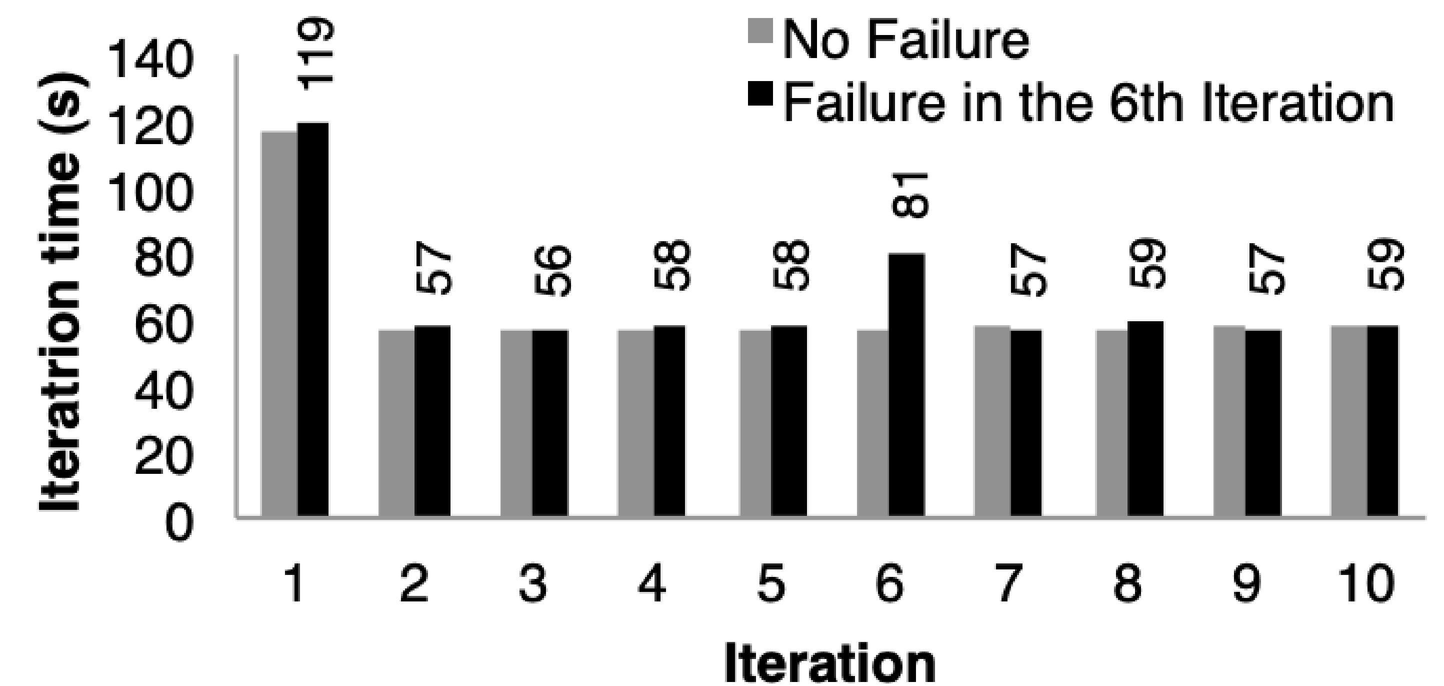



Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

Spark has a lightweight and efficient approach to fault tolerance and recovery in distributed data processing

# Evaluation

Task: investigated the behavior of Spark when there is insufficient memory to store all RDDs across iterations

- configure to limit the amount of memory used to store RDDs on each machine
- performance of logistic regression tasks degraded as the available storage space for RDDs decreased.

Spark is capable of adapting to memory constraints by gracefully degrading performance as memory availability decreases, highlighting its ability to handle scenarios where job data exceeds available memory resources in a distributed computing environment.

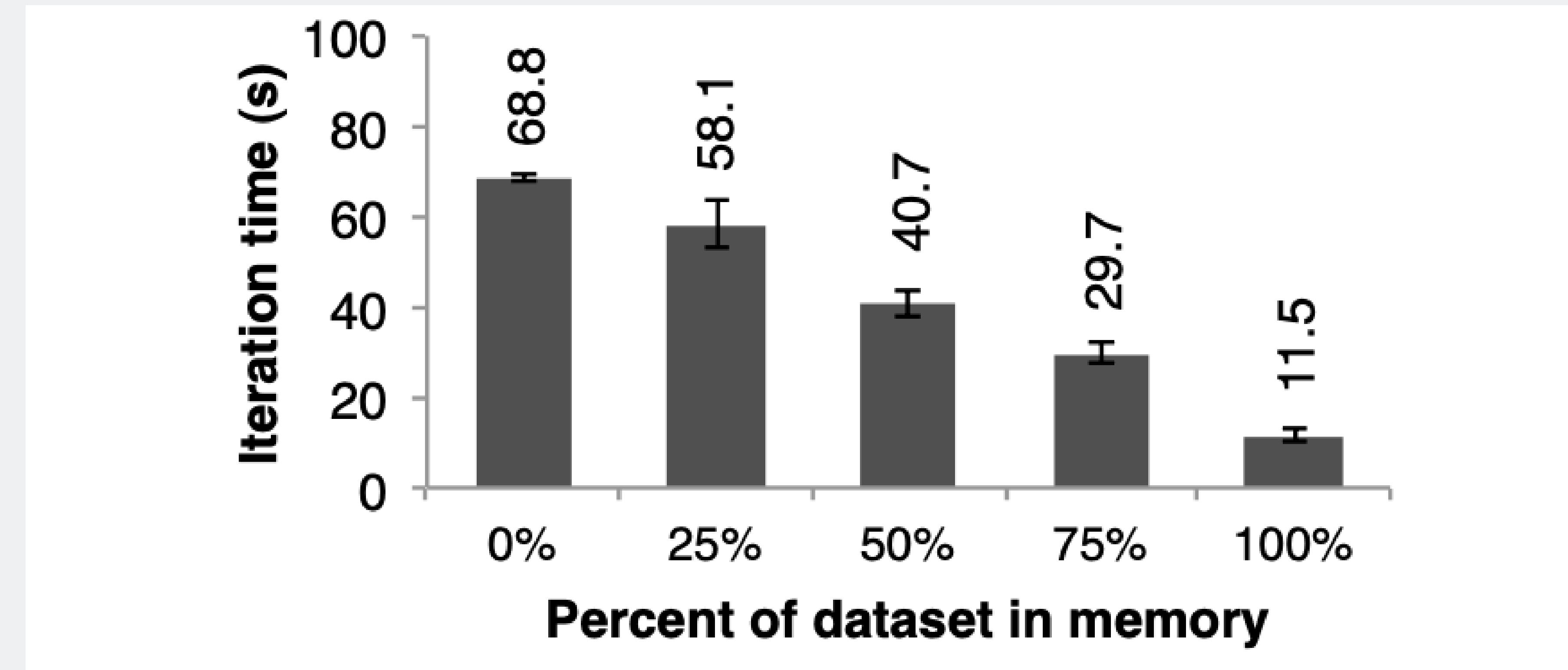
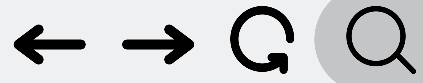



Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

Title Page | Background | Motivation | Overview | Spark Program | Evaluation



# Evaluation

Task: showcase Spark's capability for interactive querying of large datasets using 1TB of Wikipedia page view logs

- Queries were designed to find total views of all pages, pages with titles exactly matching a specified word, ...
- Response times for these queries were analyzed and compared using different subsets of the dataset

- Spark took 5–7 seconds.
- querying from disk took 170 seconds.

The findings illustrate that RDDs make Spark an efficient and powerful tool for interactive data mining, providing substantial performance improvements over traditional on-disk data querying methods.

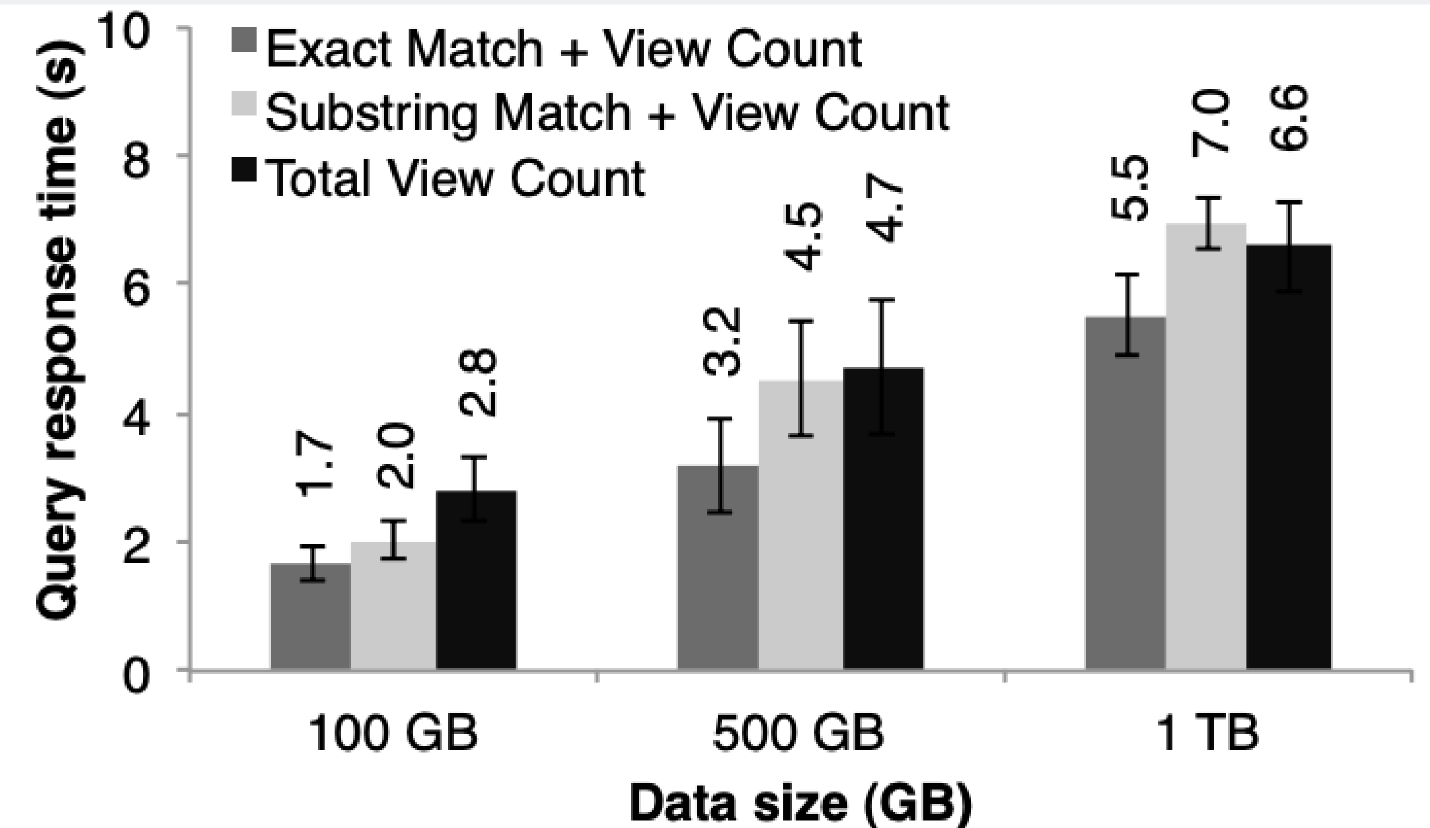
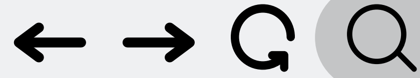



Figure 14: Response times for interactive queries on Spark, scanning increasingly larger input datasets on 100 machines.

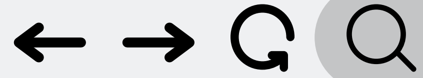


# Discussion

**Expressing Existing Programming Models**

- MapReduce
  - This model can be expressed using the *flatMap* and *groupByKey* operations in Spark, or *reduceByKey* if there is a combiner.
- DryadLINQ
  - The DryadLINQ system provides a wider range of operators than MapReduce over the more general Dryad runtime, but these are all bulk operators that correspond directly to RDD transformations available in Spark (*map, groupByKey, join*, etc).
- SQL
  - Like DryadLINQ expressions, SQL queries perform data-parallel operations on sets of records.
- Pregel
  - Pregel applies the same user function to all the vertices on each iteration, so we can implement Pregel with RDDs.
  - Spark can store the vertex states for each iteration in an RDD and perform a bulk transformation (flatMap) to apply this function and generate an RDD of messages. It can then join this RDD with the vertex states to perform the message exchange.
  - Pregel is implemented as a 200-line library in Spark.

Title Page | Background | Motivation | Overview | Spark Program | Discussion



# Discussion

**Leveraging RDDs for Debugging**

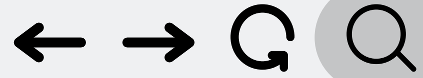
RDDs were orginally designed to be deterministically recomputable for fault tolerance purposes, but this property also facilitates debugging.

By logging the lineage of RDDs created during a job, one can:

- Reconstruct these RDDs later and let the user query them interactively
- Re-run any task from the job in a single-process debugger, by recomputing the RDD partitions it depends on

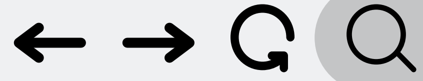
This approach adds virtually zero recording overhead because only the RDD lineage graph needs to be logged.

Title Page | Background | Motivation | Overview | Spark Program | Related Work



# Related Works

- Cluster Programming Models
  - Data Flow Models (MapReduce, Dryad, Ciel)
  - High-level Programming Interfaces (DryadLINQ, FlumeJava)
  - Specialized Systems (Pregel, Twister, HaLoop)
- Caching Systems
  - Systems with Caching Capabilities (Nectar, Ciel, FlumeJava)
  - Proposals for In-memory Caching (Ananthanarayanan et al.)
- Lineage
- Relational Databases



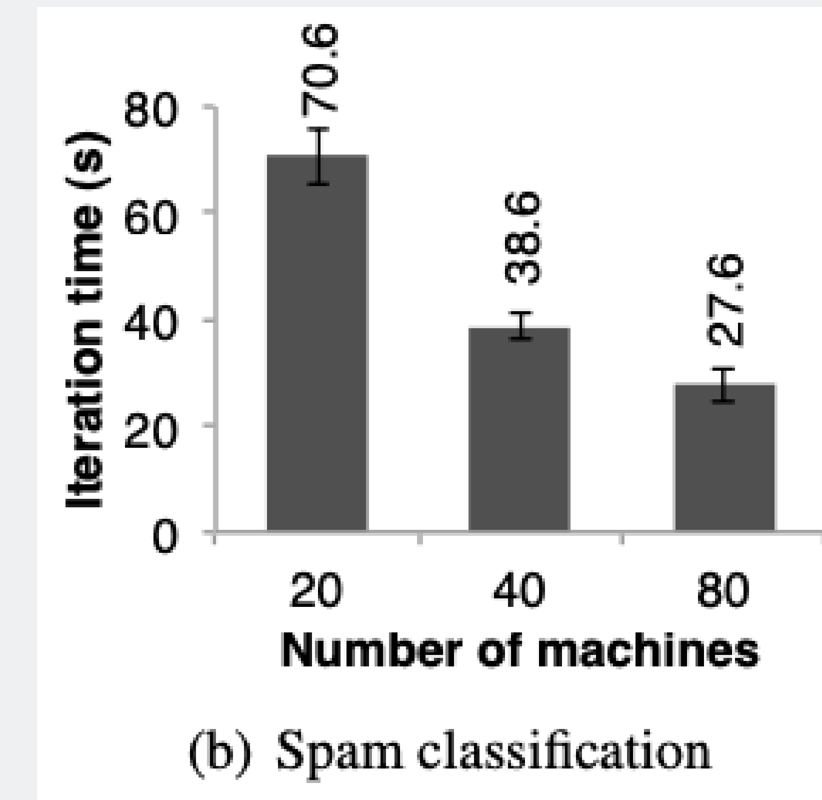
# Conclusion

Future plans / Shortcomings

- investigate sharing RDDs across instances of Spark through a unified memory manager

Companies using RDDs

- Monarch: Twitter spam classification
- SNAP: DNA sequence analysis
- Mobile Millennium: traffic prediction



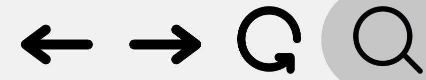

(b) Spam classification

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

# Study Questions

1. How do Resilient Distributed Datasets (RDDs) facilitate fault tolerance in distributed computing? Discuss the implications of using lineage for recovery in terms of computational overhead and system performance.

2. Explain the impact of narrow and wide dependencies on the performance of Spark applications. Provide an example of a computation that would benefit from narrow dependencies and discuss how each dependency affects the fault recovery process.



# Citations

Lee, I. (n.d.). Fault tolerance. Wallarm. Retrieved April 10, 2024, from https://www.wallarm.com/what/what-is-fault-tolerance.

Interactive Operations on MapReduce. (n.d.). Tutorials point. Retrieved April 10, 2024, from https://www.tutorialspoint.com/apache_spark/apache_spark_quick_guide.htm.

Iterative Operations on MapReduce. (n.d.). Tutorials point. Retrieved April 10, 2024, from https://www.tutorialspoint.com/apache_spark/apache_spark_quick_guide.htm.