An overview of Spark

Amogh Raghunath

Suhas Srinivasan

A class presentation for CS 561 (3/17/2016)

Worcester Polytechnic Institute

Outline

1. Introduction
2. Motivation
3. Spark Programming Model
4. Resilient Distributed Datasets
5. Spark Programming Interface
6. Representing Resilient Distributed Datasets
7. Implementation
8. Evaluation
9. Conclusion

Introduction

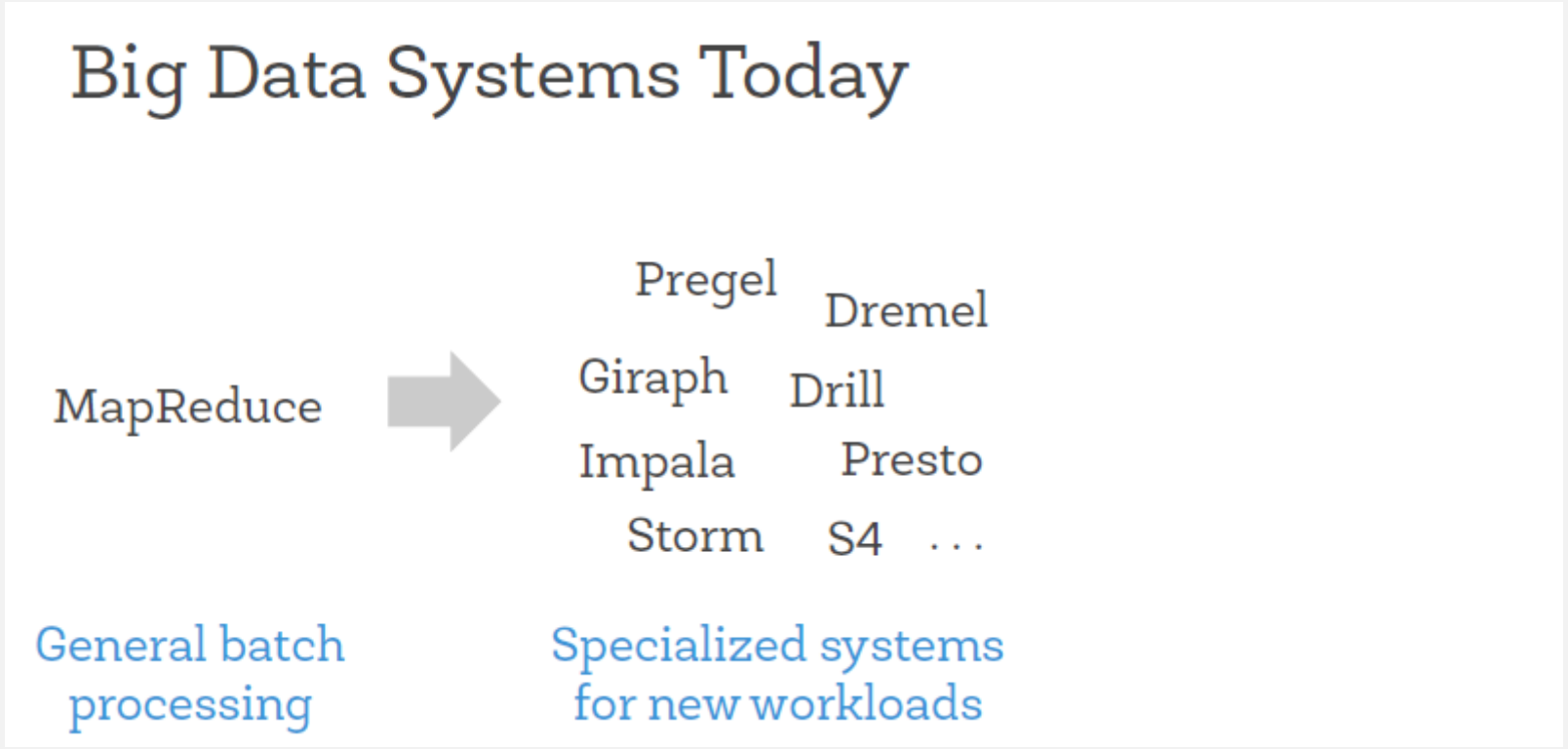
- Apache Spark is an open source cluster computing framework.
- Originally developed at the University of California, Berkeley's AMPLab by Matei Zaharia.
- Spark codebase was later donated to the Apache Software Foundation that has maintained it since.
- Fast & general engine for big data processing.
- Generalizes MapReduce model to support more types of processing.

Motivation

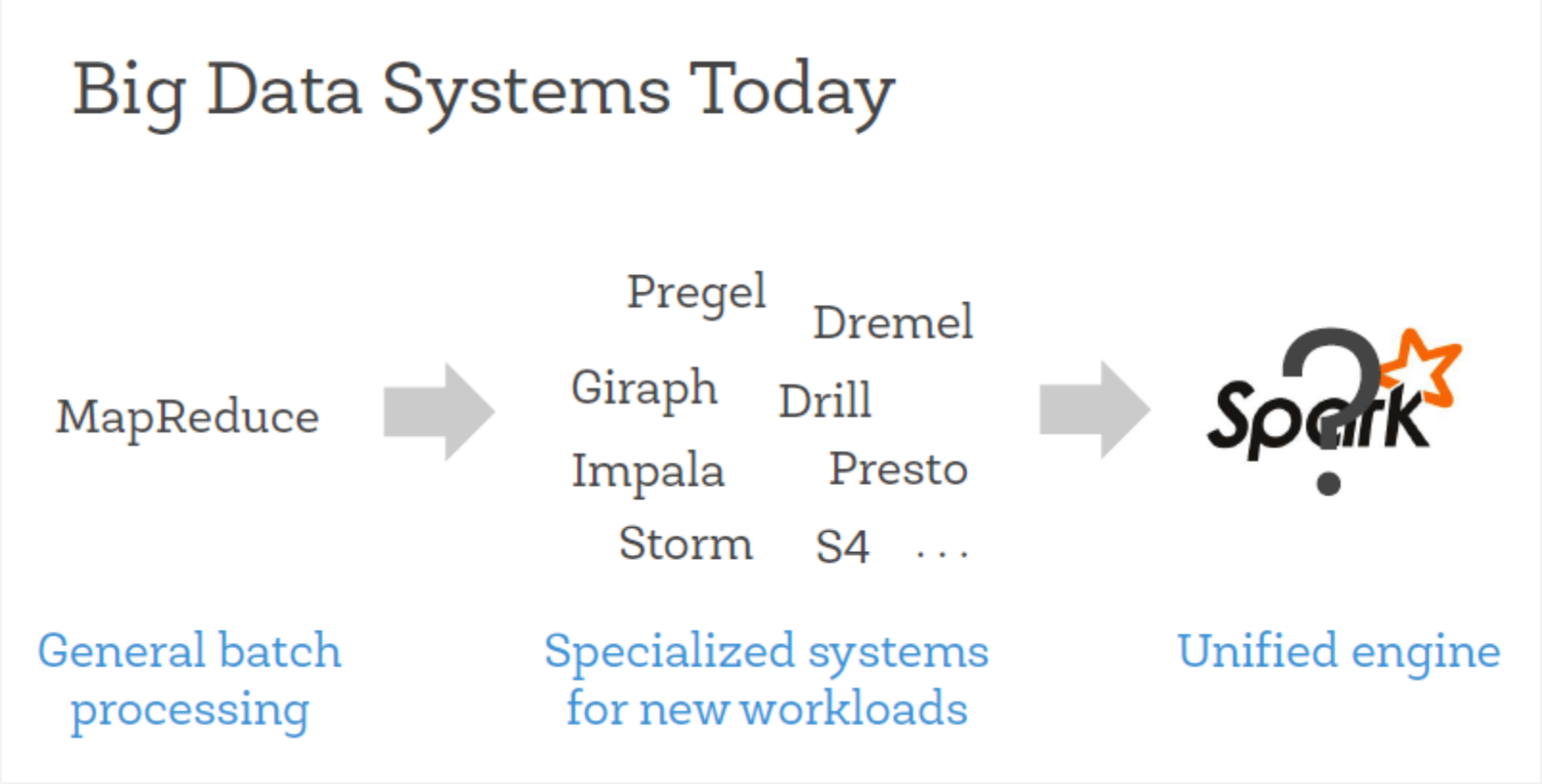
- MapReduce was great for batch processing, but users quickly needed to do more:
  - More complex, multi-pass algorithms
  - More interactive ad-hoc queries
  - More real-time stream processing

- Result: many specialized systems for these workloads

Motivation


Big Data Systems Today
MapReduce
Pregel
Dremel
Giraph
Drill
Impala
Presto
Storm
S4
. . .
General batch processing
Specialized systems for new workloads

Motivation


Big Data Systems Today
MapReduce
Pregel
Dremel
Giraph
Drill
Impala
Presto
Storm
S4
. . .
Spark
General batch processing
Specialized systems for new workloads
Unified engine

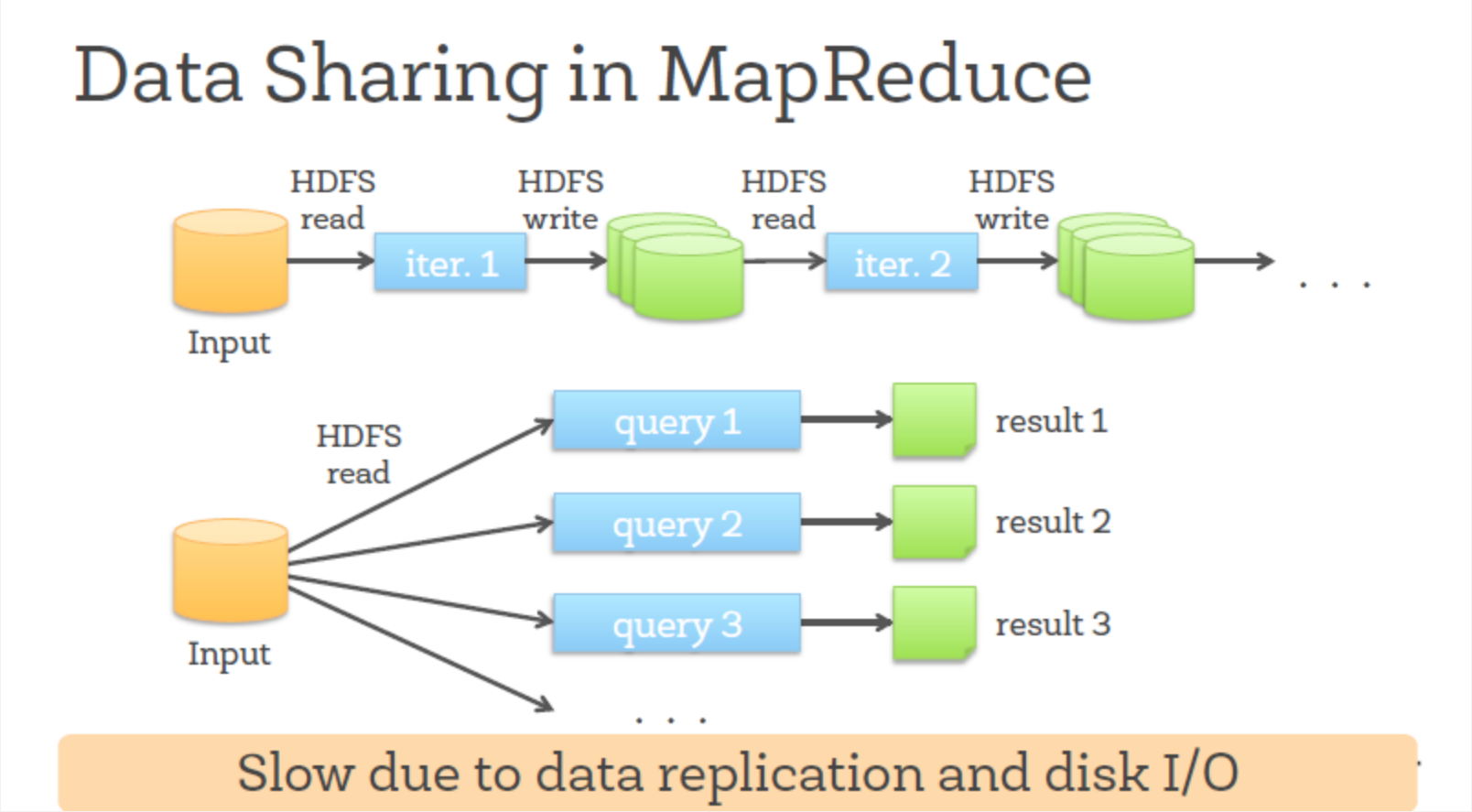
Motivation

- Problems with Specialized Systems
  - More systems to manage, tune and deploy.
- Can't combine processing types in one application
  - Even though many pipelines need to do this.
  - e.g. load data with SQL, then run machine learning.
- In many pipelines, data exchange between engines is the dominant cost.

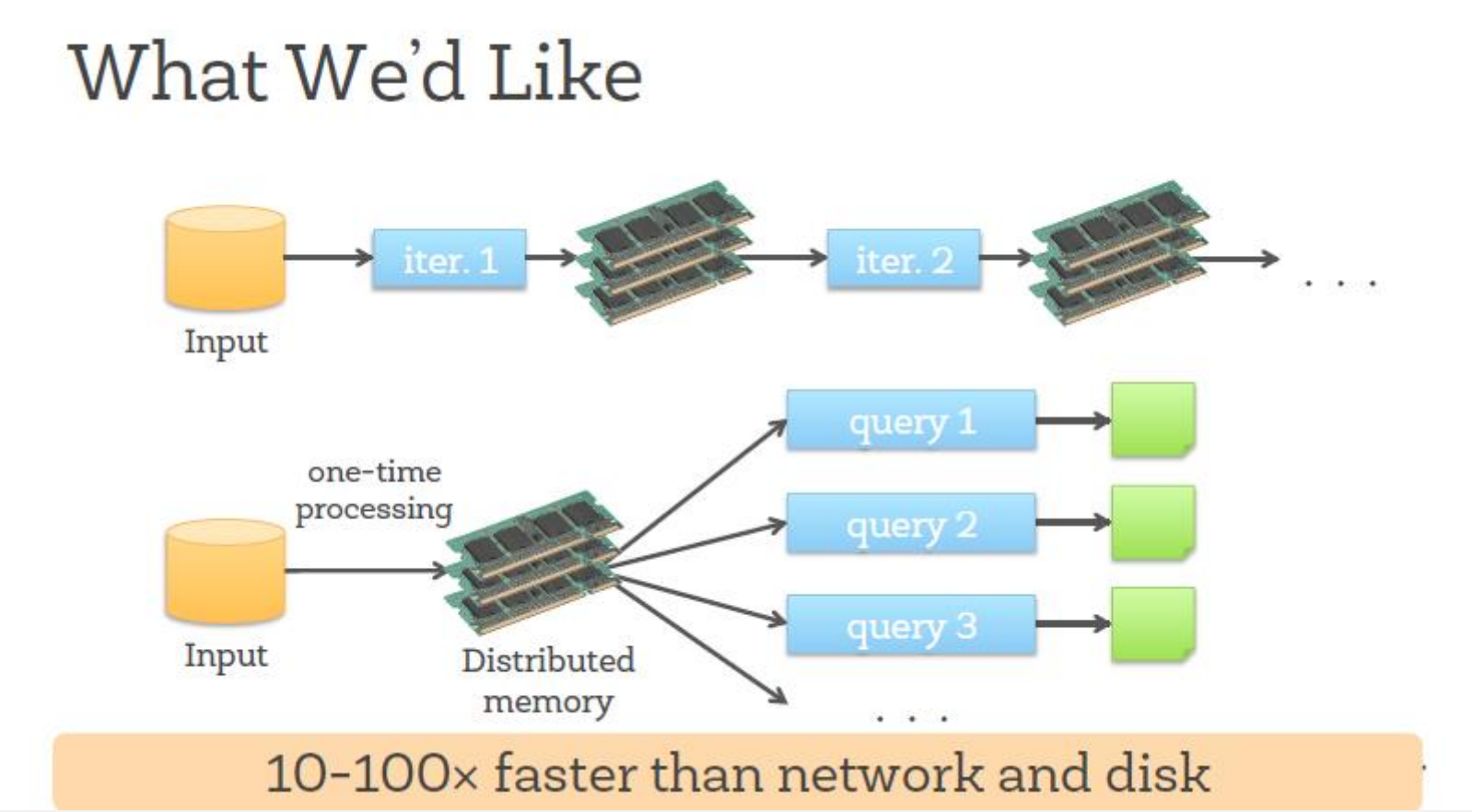
Motivation

- Recall 3 workloads were issues for MapReduce:
  - More complex, multi-pass algorithms
  - More interactive ad-hoc queries
  - More real-time stream processing
- While these look different, all 3 need one thing that MapReduce lacks: efficient data sharing

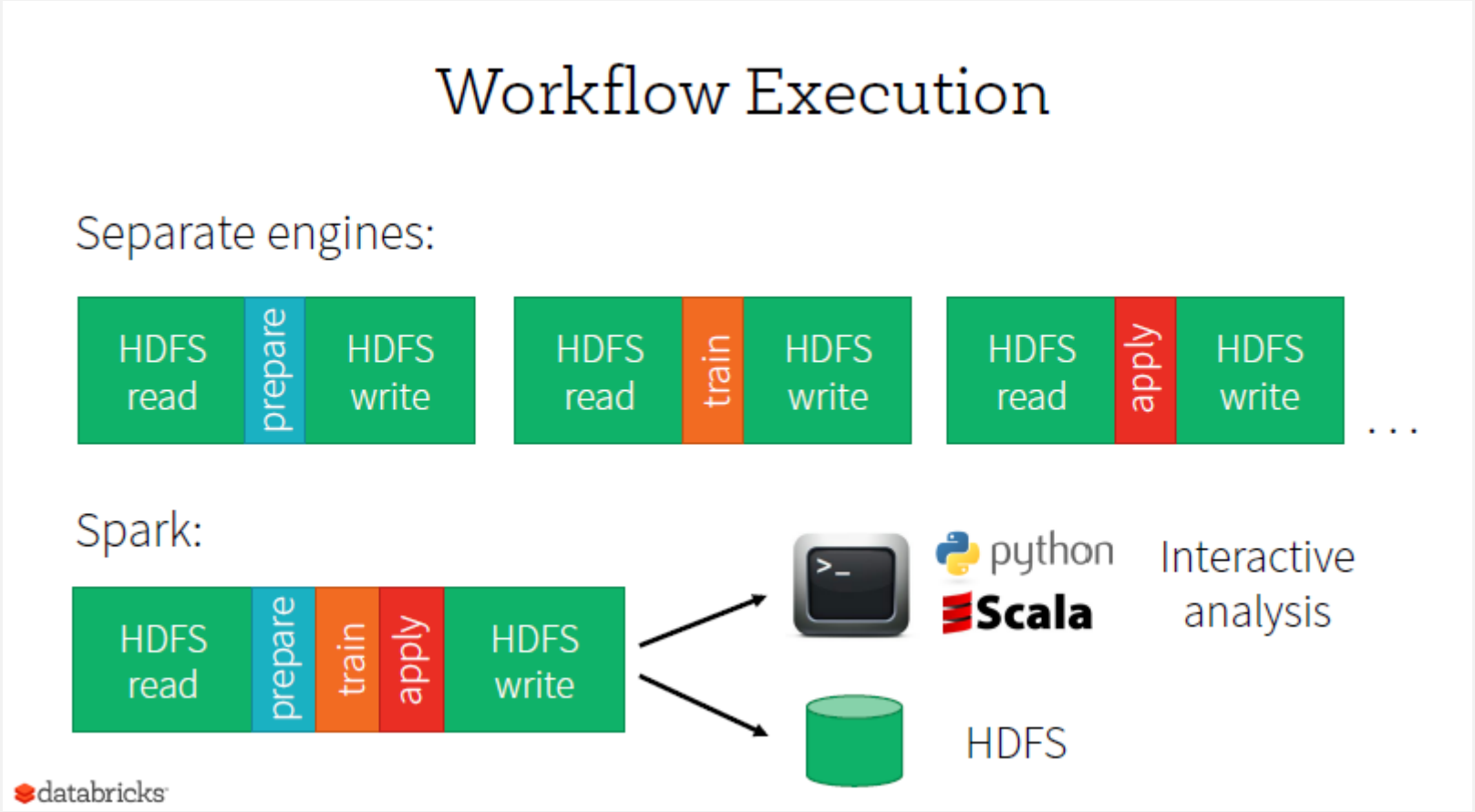
Motivation


Data Sharing in MapReduce
HDFS read
HDFS write
HDFS read
HDFS write
iter. 1
iter. 2
Input
HDFS read
query 1
query 2
query 3
result 1
result 2
result 3
Input
Slow due to data replication and disk I/O

Motivation


What We'd Like
iter. 1
iter. 2
Input
one-time
processing
query 1
query 2
query 3
Input
Distributed
memory
10-100× faster than network and disk

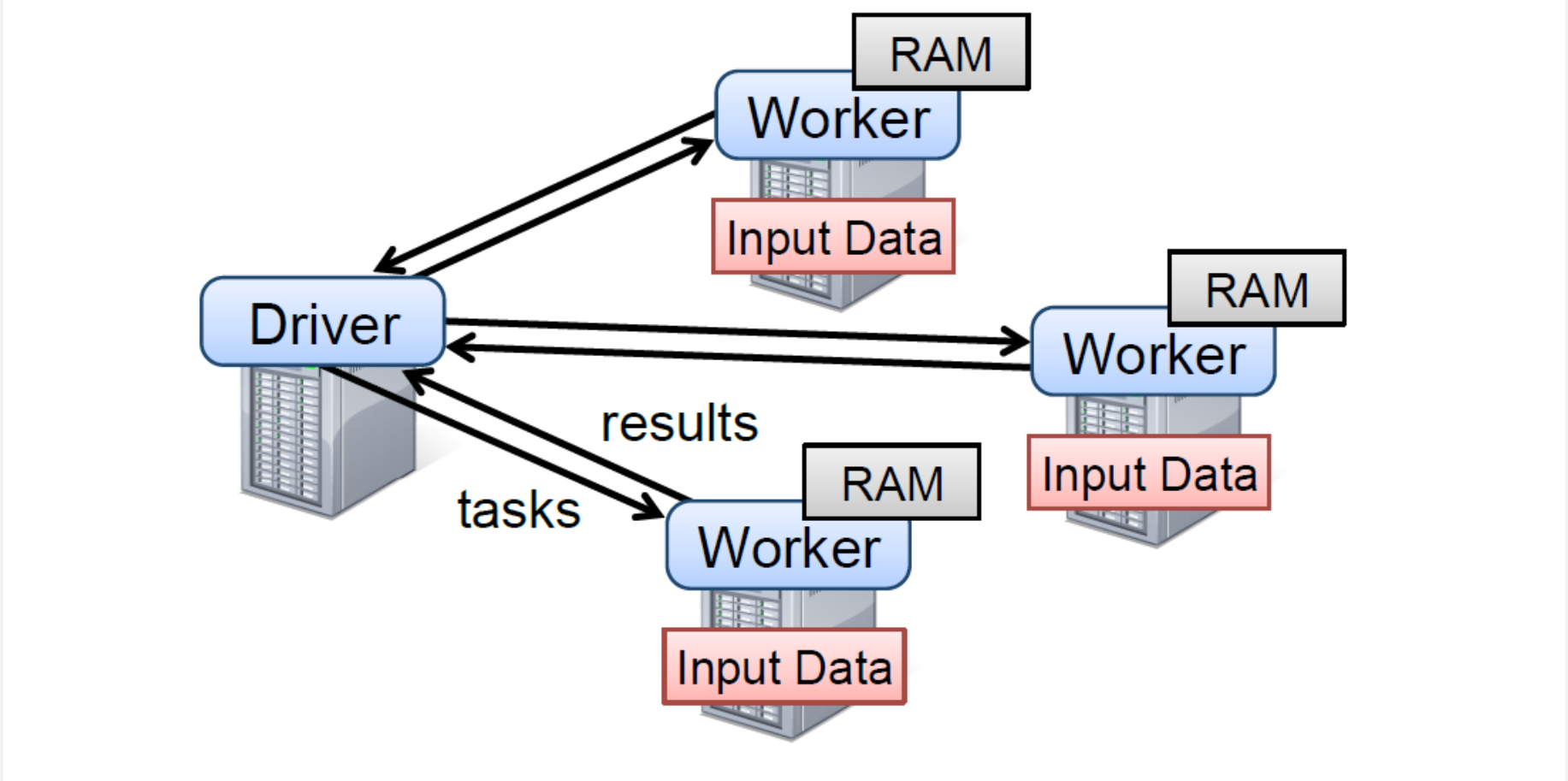
Motivation


Workflow Execution
Separate engines:
HDFS read
prepare
HDFS write
HDFS read
train
HDFS write
HDFS read
apply
HDFS write
. . .
Spark:
HDFS read
prepare
train
apply
HDFS write
python
Scala
Interactive analysis
HDFS
databricks

Spark Programming Model

1. Developers write a driver program that implements the high-level control flow of their application and launches various operations in parallel.
2. Spark provides two main abstractions for parallel programming: **resilient distributed datasets** and **parallel operations** on these datasets.
3. Spark supports two restricted types of **shared variables** that can be used in functions running on the cluster.

Spark Programming Model



Spark Programming Model

- A resilient distributed dataset (RDD) is a collection of objects that can be stored in memory or disk across a cluster.
- Built via parallel operations and are fault-tolerant without replication.

Spark Programming Model

Several parallel operations can be performed on RDDs:

- *reduce*: Combines dataset elements using an associative function to produce a result at the driver program.
- *collect*: Sends all elements of the dataset to the driver program.
- *foreach*: Passes each element through a user provided function.

Spark Programming Model

Developers can create two restricted types of shared variables to support two simple but common usage patterns:

- *Broadcast variables*: If a large read-only piece of data is used in multiple parallel operations, it is preferable to distribute it to the workers only once.
- *Accumulators*: These are variables that workers can only "add" to using an associative operation, and that only the driver can read. Useful for parallel sums and are fault tolerant.

Spark Programming Model

Language Support

Python

```
lines = sc.textFile(...)
lines.filter(lambda s: "ERROR" in s).count()
```

Scala

```
val lines = sc.textFile(...)
lines.filter(x => x.contains("ERROR")).count()
```

Java

```
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
  Boolean call(String s) {
    return s.contains("error");
  }
}).count();
```

Standalone Programs

- Python, Scala, & Java

Interactive Shells

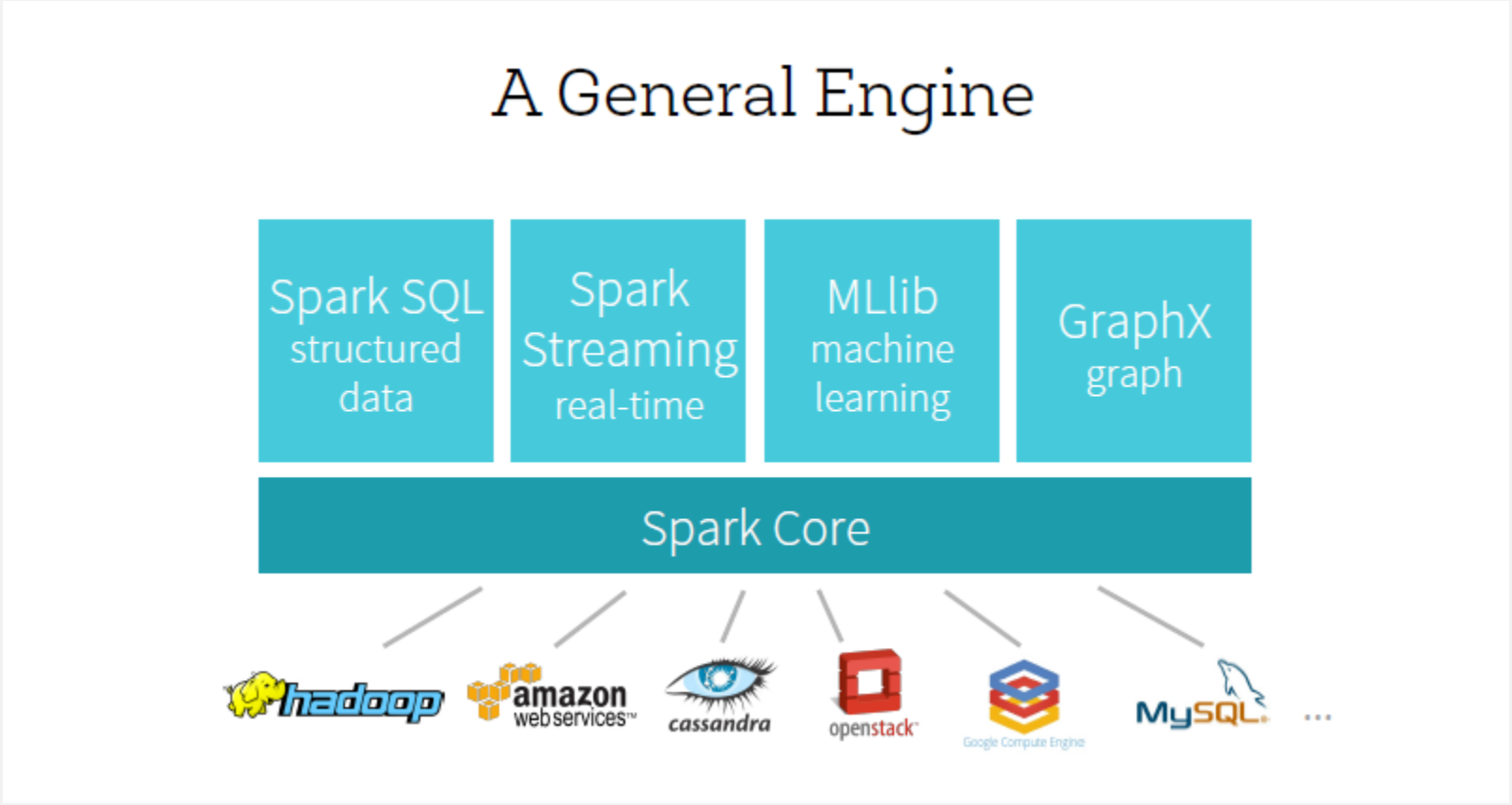
- Python & Scala

Performance

- Java & Scala are faster due to static typing
- …but Python is often fine

Spark

Spark Programming Model



Resilient Distributed Datasets

- Existing abstractions for in-memory storage on clusters offer an interface based on fine-grained updates.
- With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines.
- Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network.

Resilient Distributed Datasets

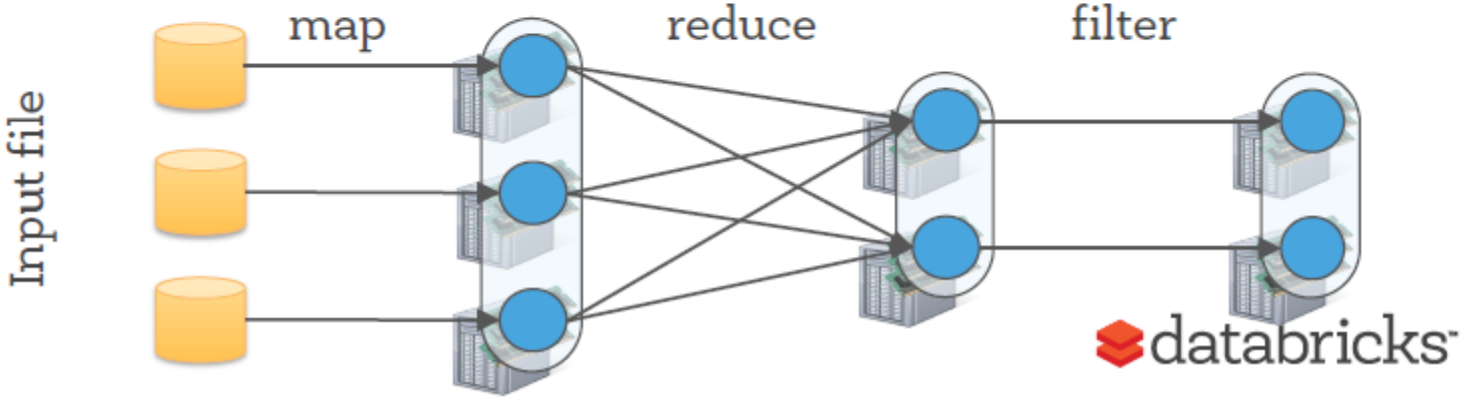
- A resilient distributed dataset (RDD) is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost.
- The elements of an RDD need not exist in physical storage; instead, a handle to an RDD contains enough information to compute the RDD starting from data in reliable storage.
- Users can control two other aspects of RDDs: *persistence* and *partitioning*.
- Users can indicate which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage).
- They can also ask that an RDD be partitioned across machines this is useful for placement optimizations.

Resilient Distributed Datasets

Fault Tolerance

RDDs track *lineage* info to rebuild lost data

```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```

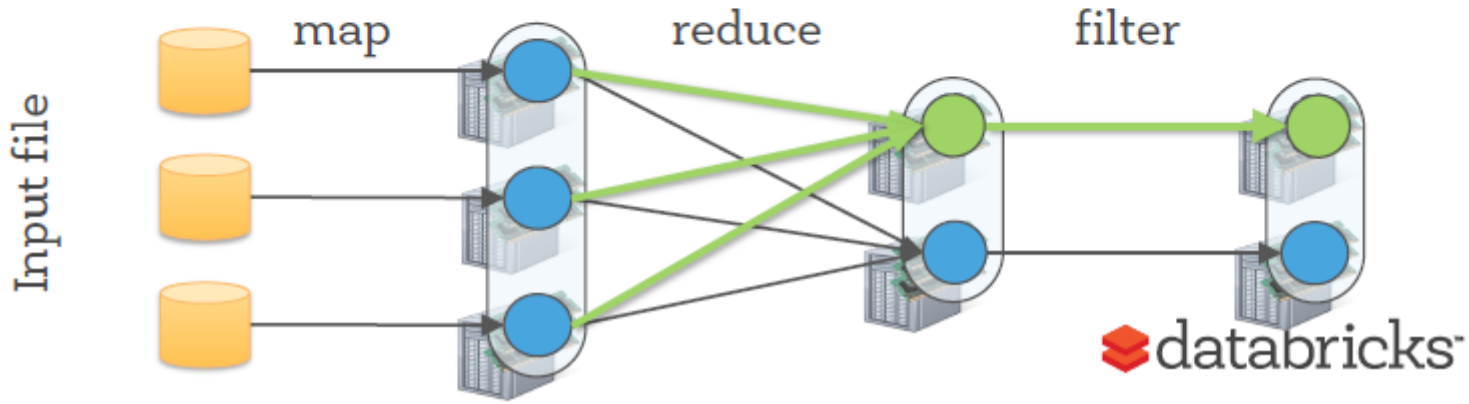


Resilient Distributed Datasets

Fault Tolerance

RDDs track *lineage* info to rebuild lost data

```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



Resilient Distributed Datasets

Advantages

- Existing frameworks (like MapReduce) access the computational power of the cluster, but not distributed memory.
  - Time consuming and inefficient for applications that reuse intermediate results.

- RDDs allow **in-memory** storage of intermediate results, enabling efficient reuse of data.

Resilient Distributed Datasets

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?)

Comparison of RDDs with distributed shared memory

Resilient Distributed Datasets

Creating RDDs

- Two methods:
  1. Loading an external dataset.
  2. Creating an RDD from an existing RDD.

Resilient Distributed Datasets

1. Loading an external dataset

- Most common method for creating RDDs
- Data can be located in any storage system like HDFS, Hbase , Cassandra etc.
- Example:

```
lines = spark.textFile("hdfs://...")
```

Resilient Distributed Datasets

2. Creating an RDD from an Existing RDD

- An existing RDD can be used to create a new RDD.
- The Parent RDD remains intact and is not modified.
- The parent RDD can be used for further operations.
- Example

```
errors = lines.filter(_.startsWith("ERROR"))
```

Resilient Distributed Datasets

Operations

- Transformations and Actions are two main types of operations that can be performed on a RDD.

- Concept similar to MapReduce:
  - Transformations are like the *map()* function.
  - Actions are like the *reduce()* function.

Resilient Distributed Datasets

Transformations

- Operations on existing RDDs that can return a new RDD.
- Transformation examples: *map*, *filter*, *join*.
  - Example: running a filter on one RDD to produce another RDD.

Resilient Distributed Datasets

Transformations

```
lines = spark.textFile("hdfs://...")

errors = lines.filter(_.startsWith("ERROR"))
```

- Original parent RDD is left intact and can be used in future transformations.
- No action takes place, just metadata of `errors` RDD are created.

Resilient Distributed Datasets

Actions

- Perform a computation on existing RDDs producing a result.
- Result is either:
  - Returned to the Driver Program.
  - Stored in a files system (like HDFS).
- Examples:
  - *count()*
  - *collect()*
  - *reduce()*
  - *save()*

Resilient Distributed Datasets

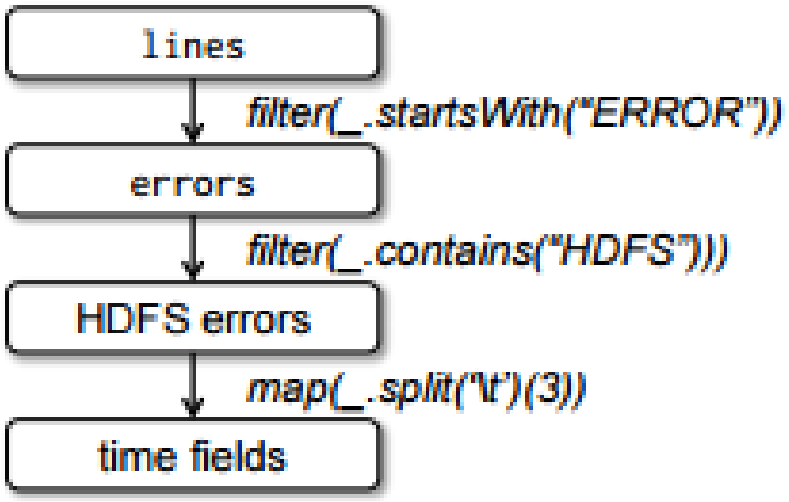
Fault Tolerance

- In event of node failure, operations can proceed.
- Spark uses an approach called the **Lineage Graph** or **Directed Acyclic Graph (DAG).**
- Critical to maintain dependencies between RDDs.
- Linage Graph are maintained by the DAGScheduler.

Resilient Distributed Datasets

Fault Tolerance

- Model that describes steps required and business logic needed to create the end result of the transformation process.
- Does not store the actual data.
- Example:



Resilient Distributed Datasets

Lazy Evaluation

- Transformation operations in RDD are referred to as being lazy.
  - Results are not physically computed right away.
  - Metadata regarding the transformations is recorded.
  - Transformations are implemented only when an action is invoked.
- Example:

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

  - RDD errors are not returned to Driver program.
  - Instead, the transformations are implemented only when a action on *errors* RDD is invoked (like `errors.persist()`).

Resilient Distributed Datasets

Example:

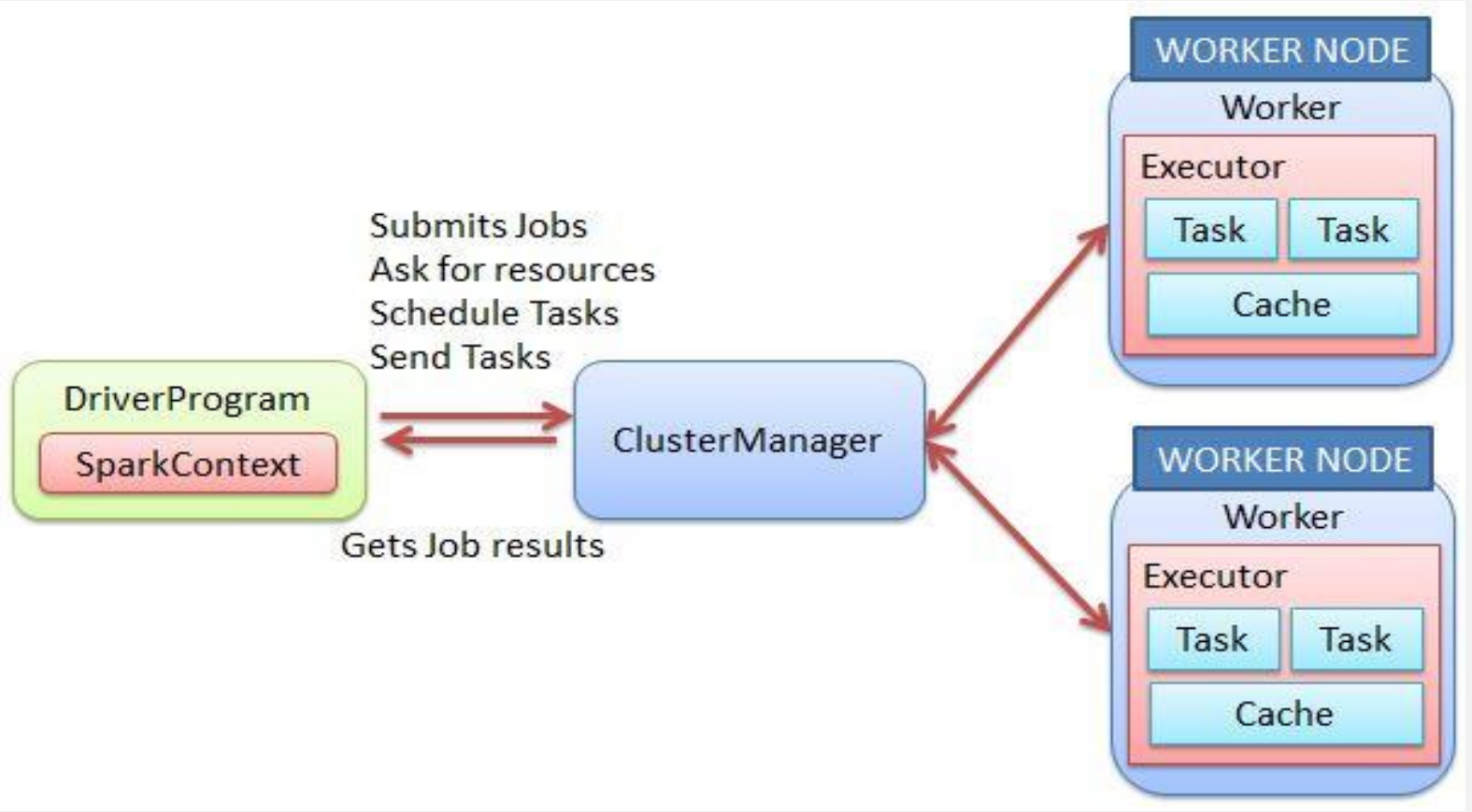
```
lines = spark.textFile("hdfs://...")


errors = lines.filter(_.startsWith("ERROR"))


errors.count()
```

Resilient Distributed Datasets

Applications not suited for RDDs

- RDDs are best suited for batch applications that apply the same operation to all elements of a dataset.
- Less suitable for applications that make asynchronous fine grained updates to shared state
  - Storage system for web application
  - Incremental Web crawler

Spark Programming Interface



Spark Programming Interface

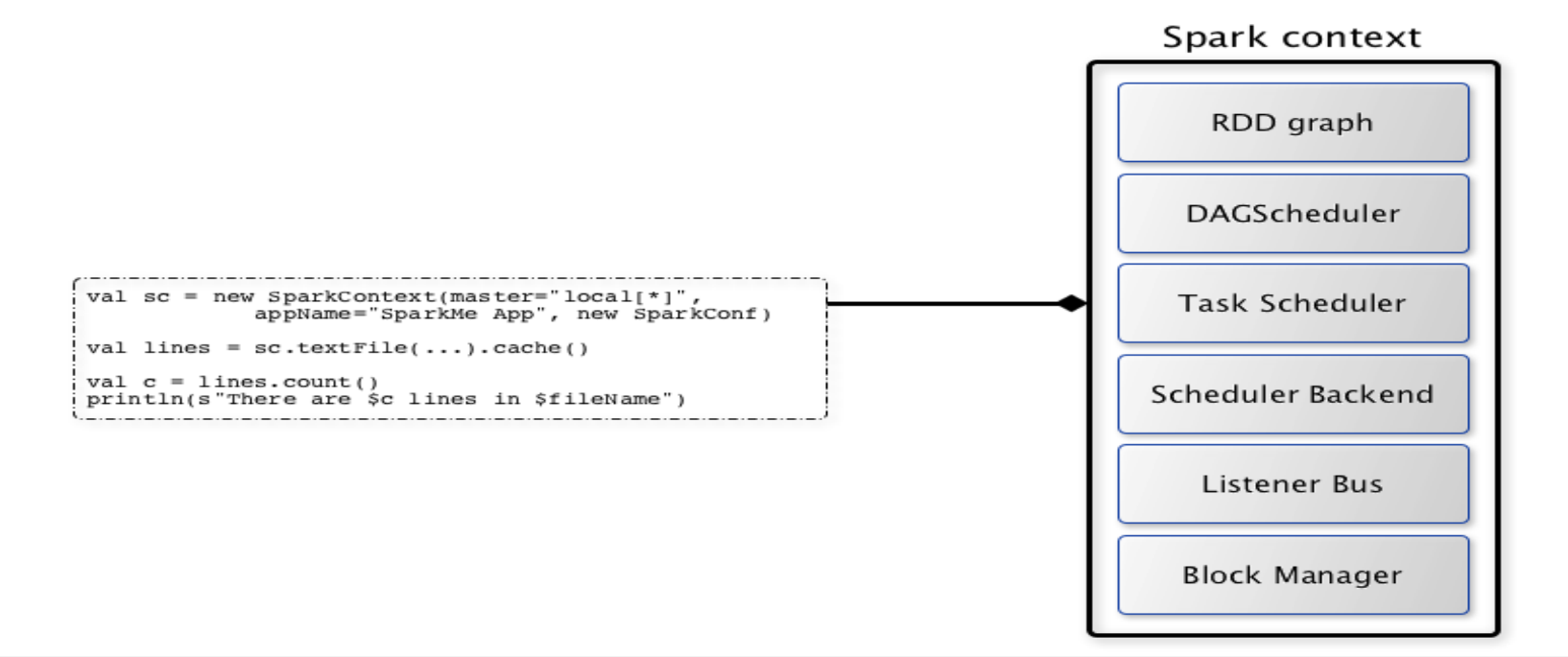
Driver Program

- Every spark application consists of a “driver program”.
  - Responsible for launching parallel tasks on various cluster nodes.
  - Encapsulates the *main()* function of the code.
  - Defines distributed datasets across the nodes.
  - Applies required operations across the distributed datasets.

Spark Programming Interface

SparkContext (sc)

- Means of connecting the driver program to the cluster.
- Once SparkContext is ready, it can be used to built an RDD.



Spark Programming Interface

Executers

- Driver Program manages nodes called executers.
  - Used to run distributed operations.
  - Each executor performs part of operation.

- Example: running *count()* function
  - Different partition of the data sent to each executor.
  - Each executor counts the number of lines in its data partition only.

Representing Resilient Distributed Datasets

- RDDs are broken down into:
  - Partitions
  - Dependencies on parent RDDs
- How to represent dependencies between RDDs?
  - Narrow Dependency

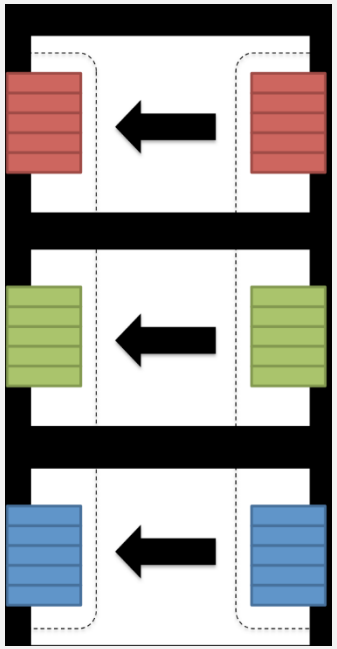
    Example: Map
  - Wide Dependency

    Example : Join

Representing Resilient Distributed Datasets

Narrow Dependency

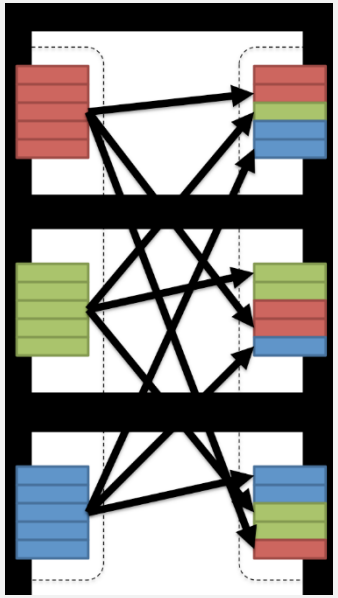
- All the partitions of the RDD will be consumed by a single child RDD.
- Example:
  - Filter
  - Map



Representing Resilient Distributed Datasets

Wide Dependency

- Multiple child RDDs may depend on a parent RDD.
- Example:
  - Join
  - Group By



Representing Resilient Distributed Datasets

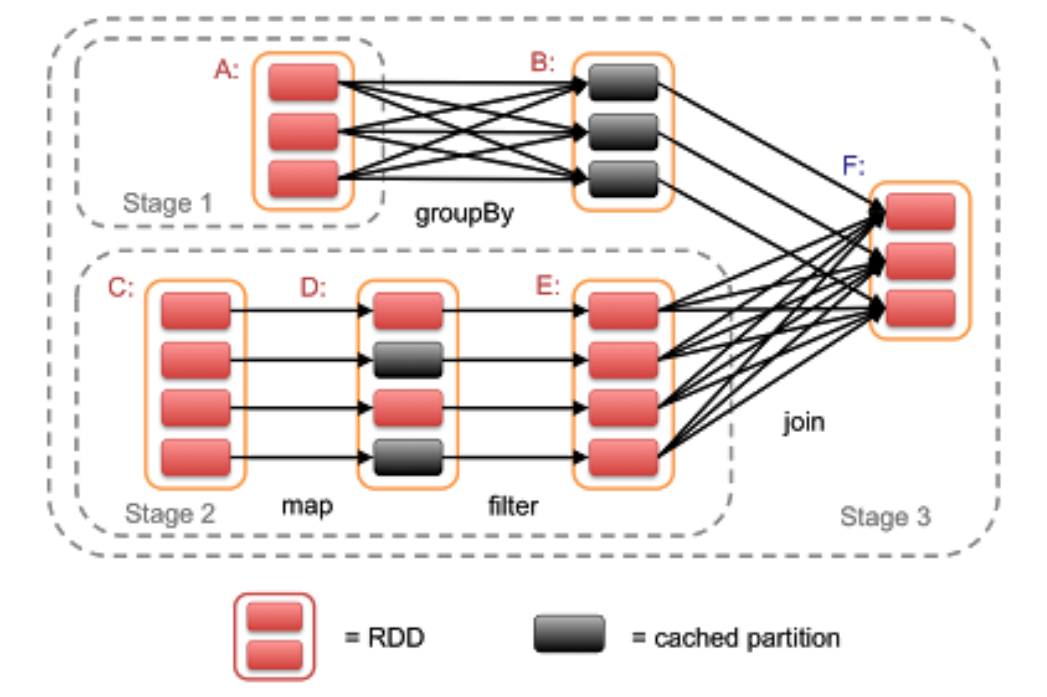
Operation	Meaning
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p*, *parentIters*)	Compute the elements of partition *p* given iterators for its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

Interface used to represent RDD in Spark

Implementation

Job Scheduling

- When an action is invoked on an RDD, the scheduler checks the lineage graph to be executed.



Implementation

Memory Management

Three options for storage of persistent RDDs:

1. In-memory storage as deserialized Java objects (fastest performance)
2. In-memory storage as serialized data (Memory efficient but lower performance)
3. On-disk storage (RDD is too large to fit in memory, highest cost)

Implementation

Memory Management

- LRU eviction policy at the level of RDDs is used.
- When a new RDD partition is computed but there is not enough space, a partition from the LRU RDD is evicted.
- Unless this is the same RDD as the one with the new partition keep the old partition to prevent cycling partitions.
- Users get further control via a “persistence priority” for each RDD.

Implementation

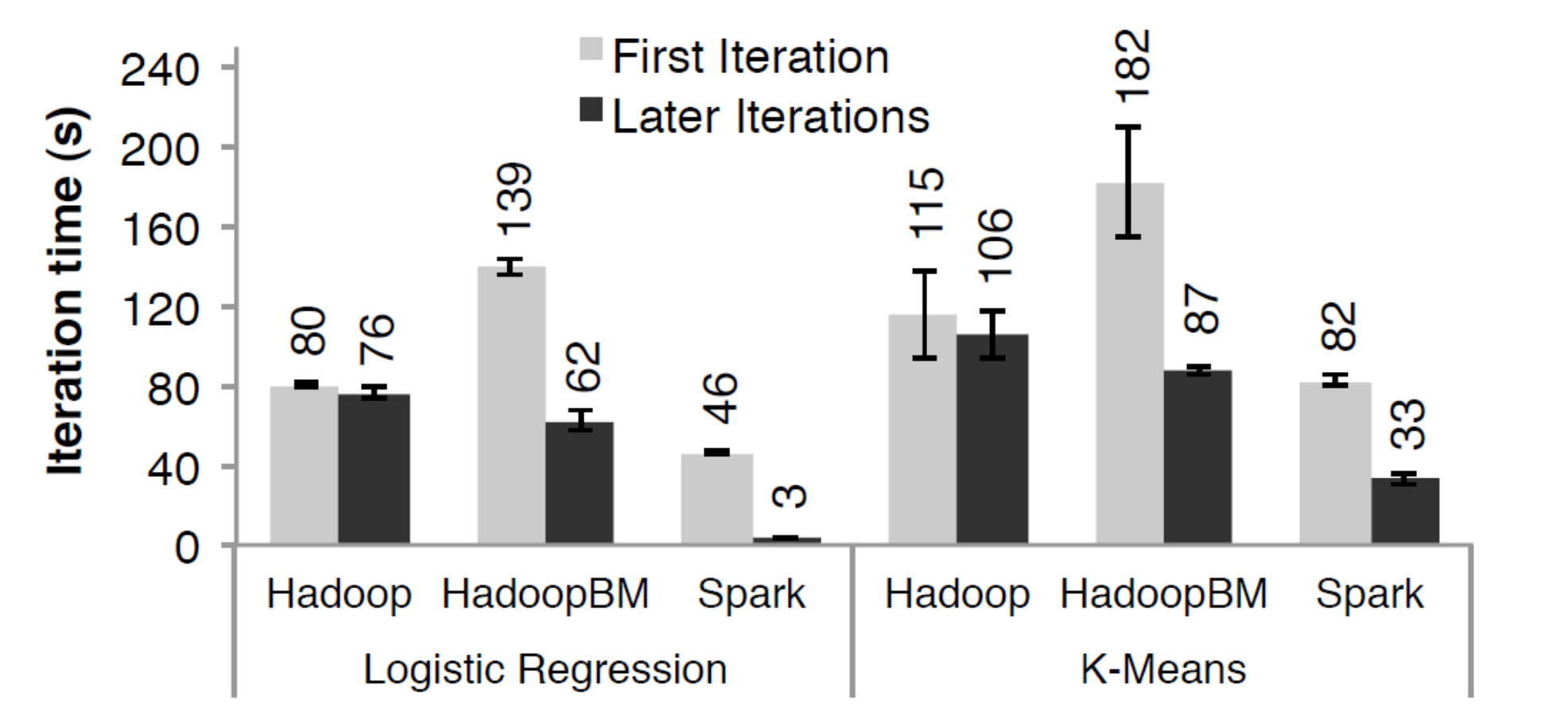
Checkpointing

- Recovery may be time-consuming for RDDs with long lineage chains.
- Spark provides an API for checkpointing (a REPLICATE flag to *persist*).
- Automatic checkpointing – the scheduler knows the size of each dataset and the time it took to first compute it, it should be able to select an optimal set of RDDs to checkpoint to minimize recovery time.
- Metadata can also be checkpointed to account for Driver node failure.

Evaluation

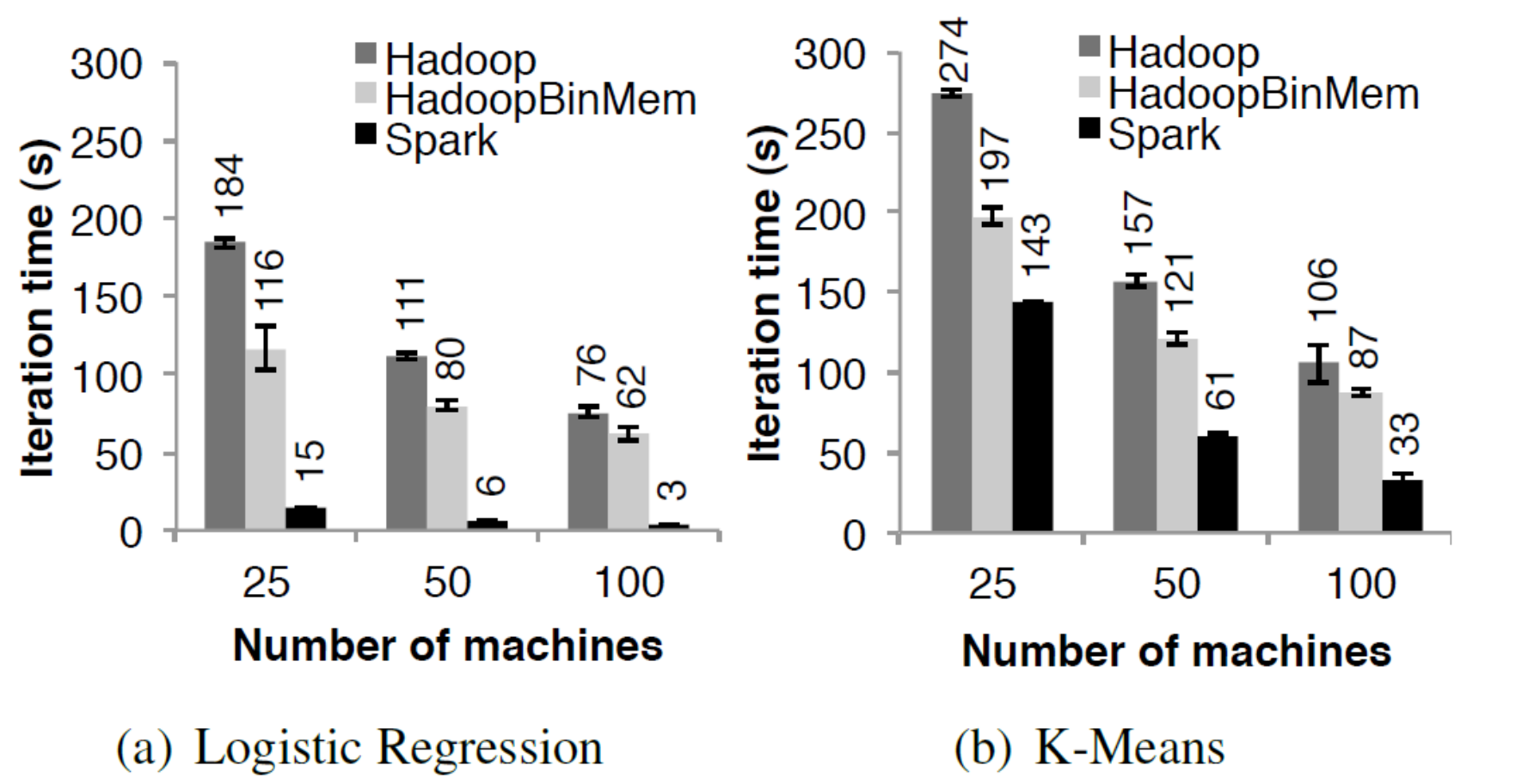
- Spark outperforms Hadoop by up to 20x in iterative machine learning and graph applications.
- The speedup comes from avoiding I/O and deserialization costs by storing data in memory as Java objects.
- When nodes fail, Spark can recover quickly by rebuilding only the lost RDD partitions.
- Spark can be used to query a 1 TB dataset interactively with latencies of 5–7 seconds.

Evaluation



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.

Evaluation



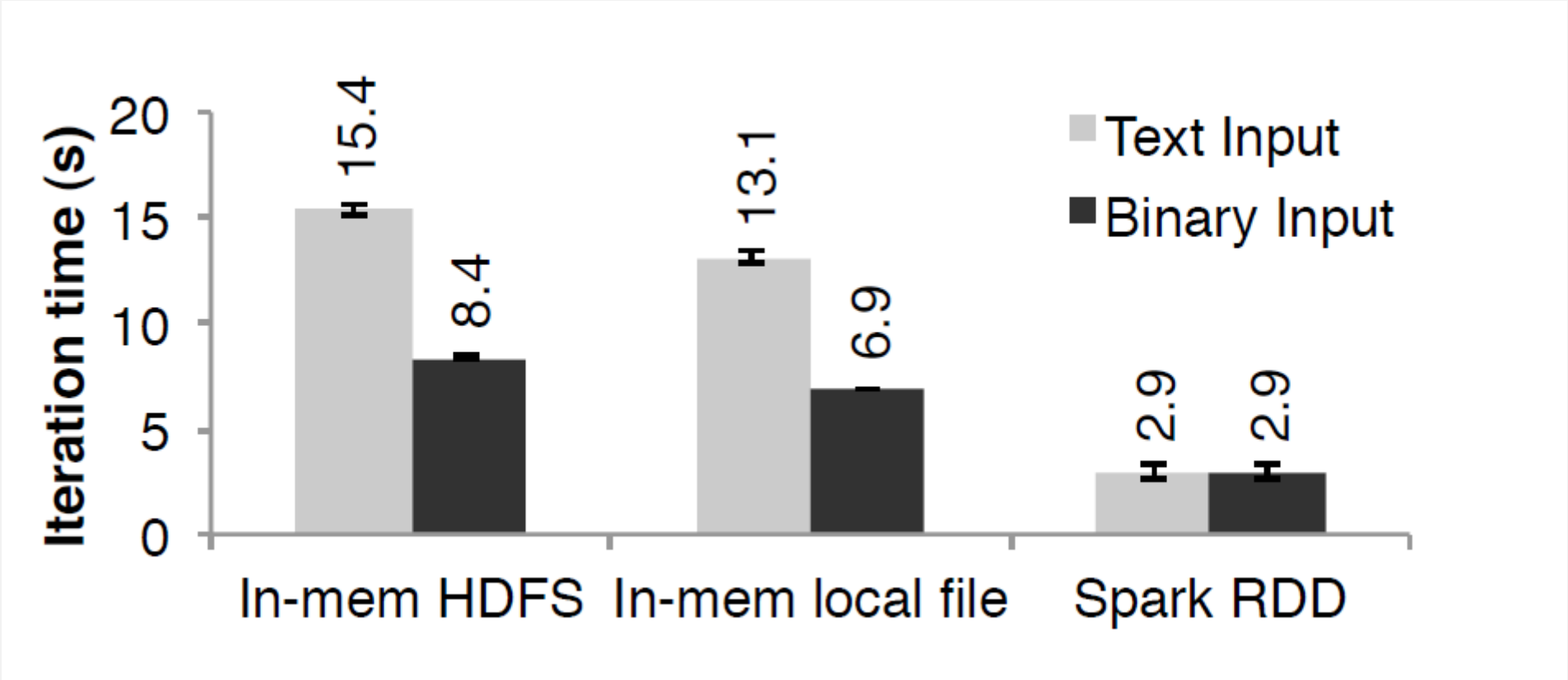
Running times for iterations after the first in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB.

Evaluation

HadoopBinMem ran slower due to several factors:

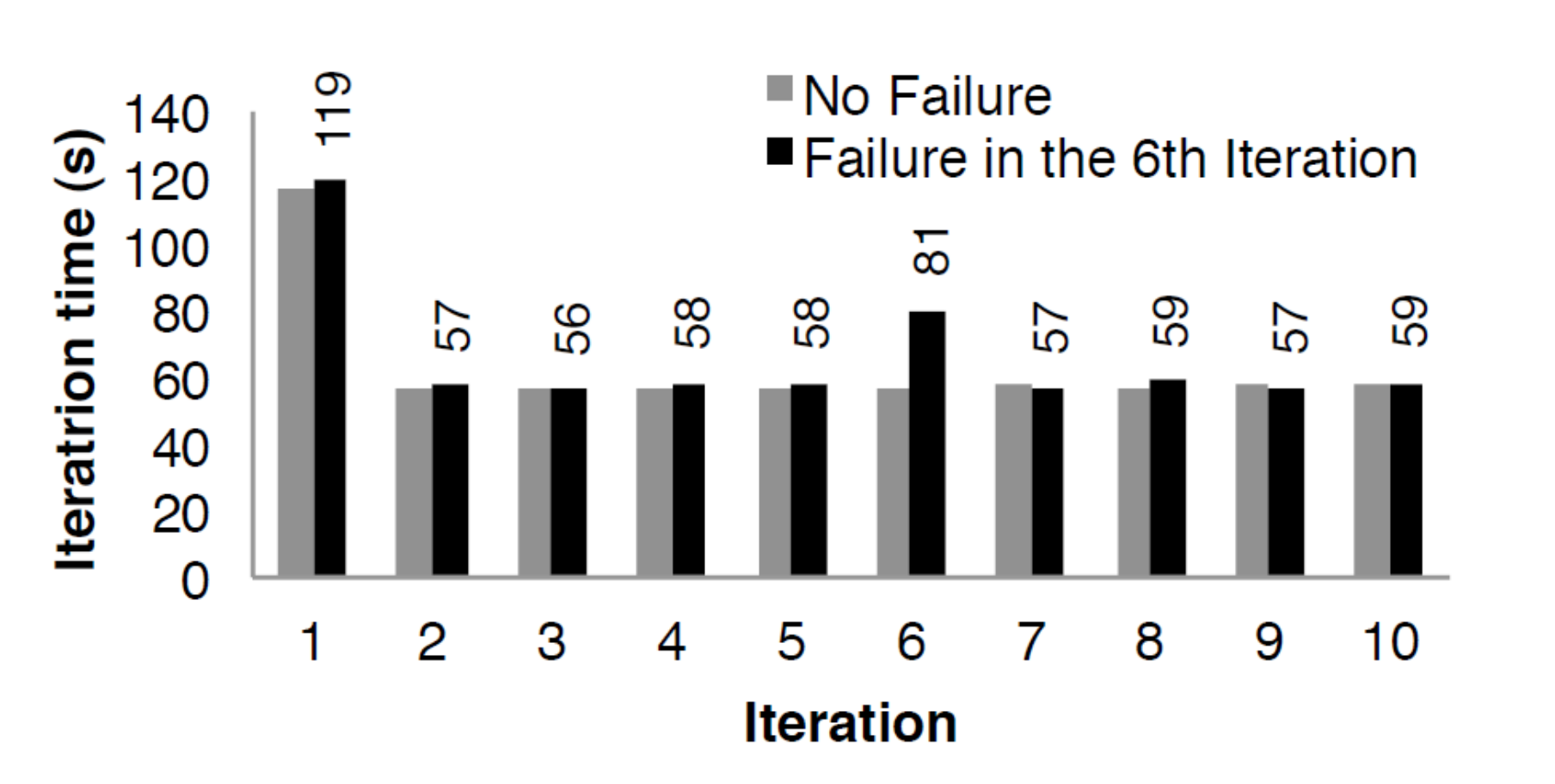
1. Minimum overhead of the Hadoop software stack.
2. Overhead of HDFS while serving data.
3. Deserialization cost to convert binary records to usable in-memory Java objects.

Evaluation



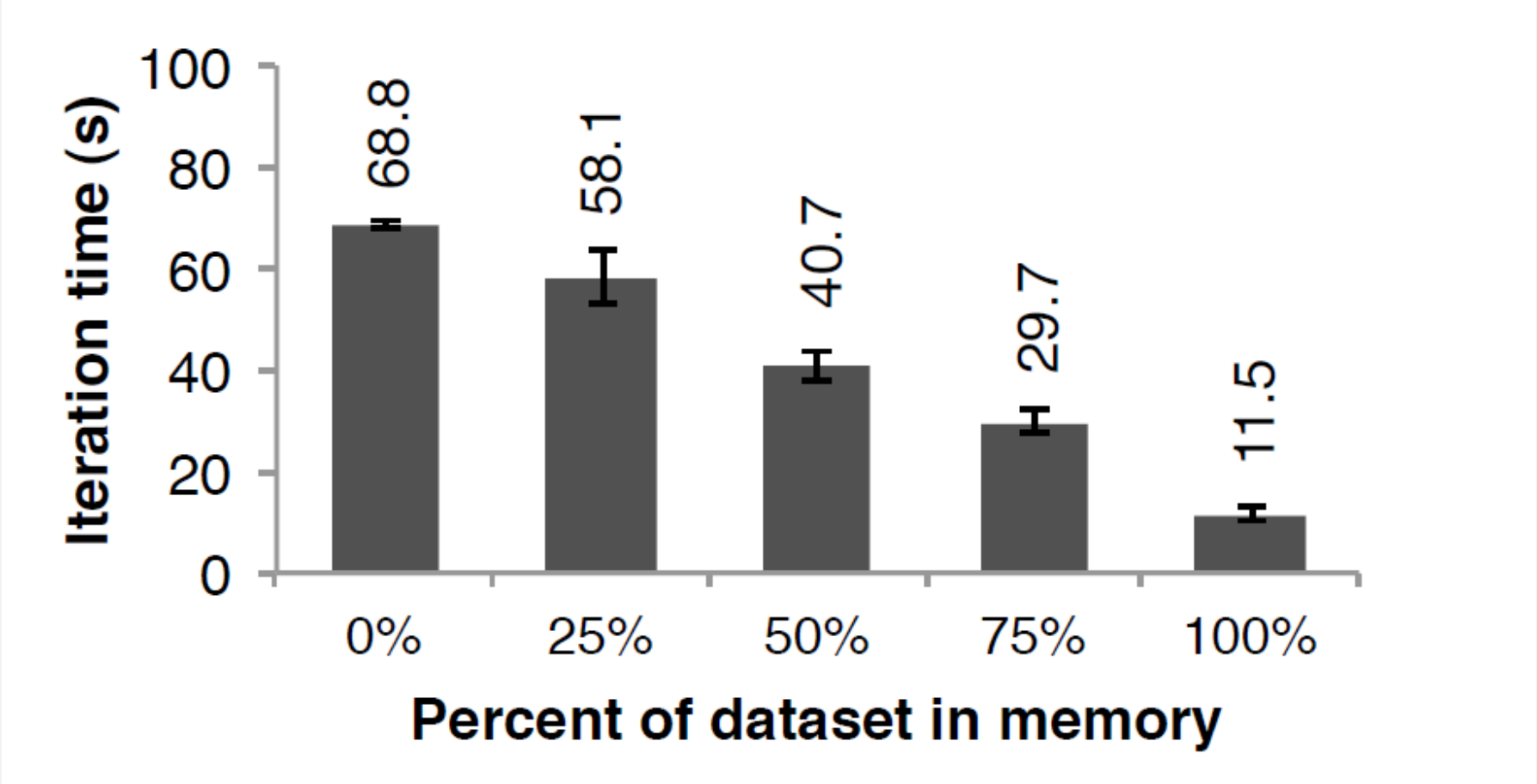
Iteration times for logistic regression using 256 MB data on a single machine for different sources of input.

Evaluation



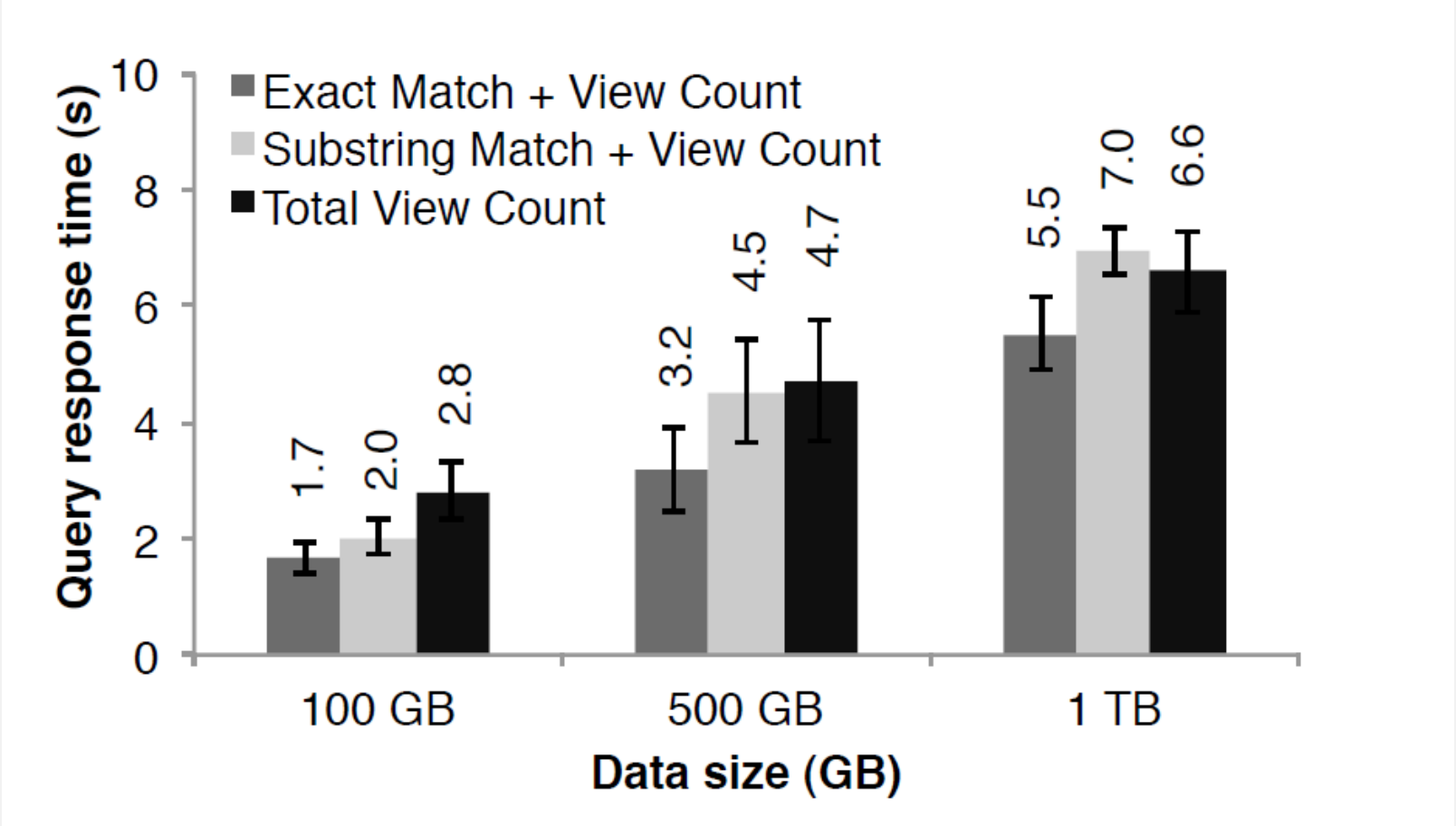
Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

Evaluation



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

Evaluation



Response times for interactive queries on Spark, scanning increasingly larger input datasets on 100 machines. Querying the 1 TB file from disk took 170s.

Conclusion

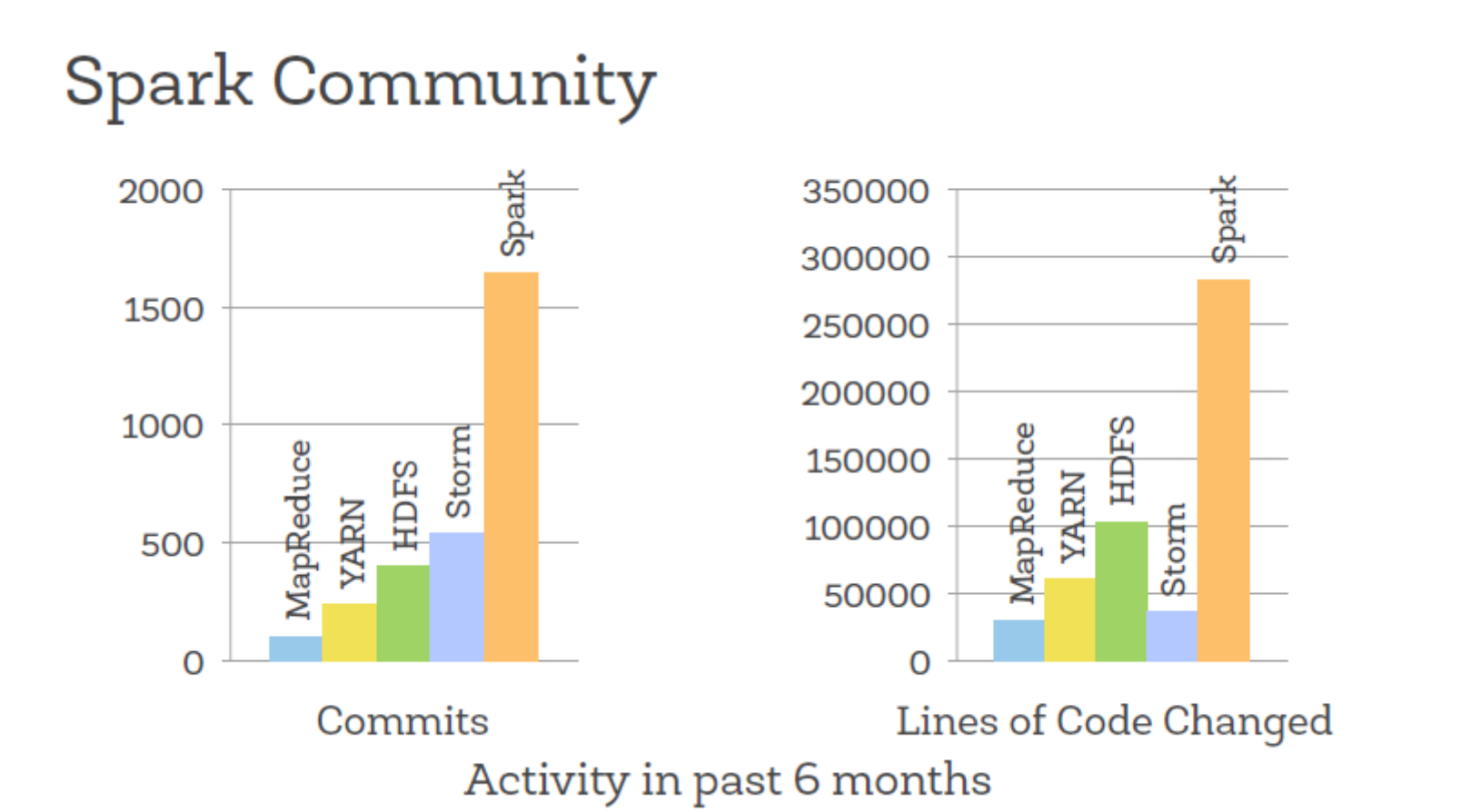
- How should we design computing platforms for the new era of massively parallel clusters?
- As we saw the answer can, in many cases, be quite simple: a single abstraction for computation, based on coarse-grained operations with efficient data sharing, can achieve state-of-the-art performance.

Conclusion

Lessons Learned

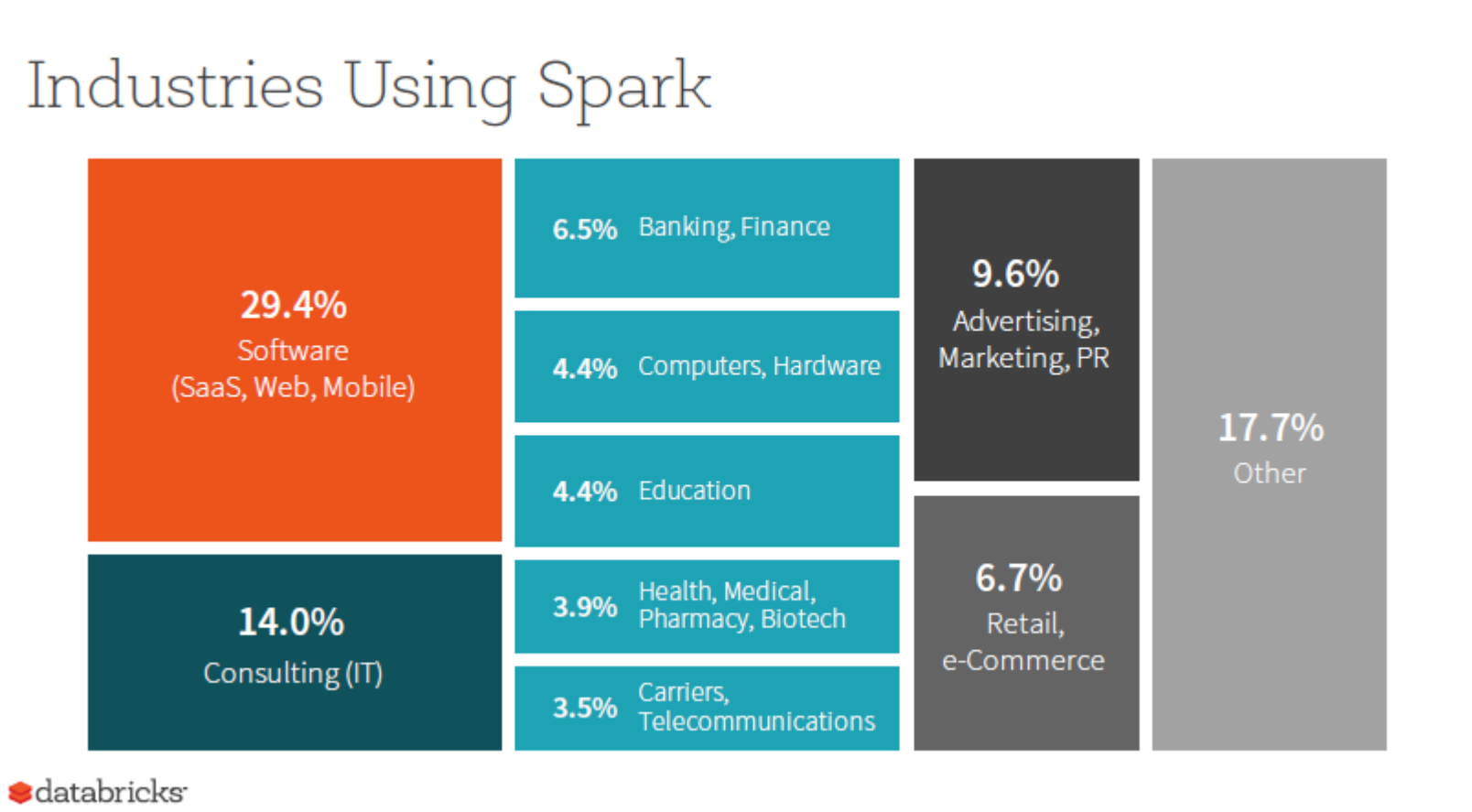
- The importance of data sharing.
- Value performance in a shared setting over single-application.
- Optimize the bottlenecks that matter.
- Simple designs compose.

Conclusion



Most active open source project in big data processing.

Conclusion


Industries Using Spark
29.4%
Software
(SaaS, Web, Mobile)
14.0%
Consulting (IT)
6.5% Banking, Finance
4.4% Computers, Hardware
4.4% Education
3.9% Health, Medical, Pharmacy, Biotech
3.5% Carriers, Telecommunications
9.6%
Advertising, Marketing, PR
6.7%
Retail, e-Commerce
17.7%
Other
databricks

Conclusion

Users

1000+ companies

. . .

Distributors + Apps

50+ companies

. . .

References

1. Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”.
2. Matei Zaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”.
3. Matei Zaharia’s dissertation, “An Architecture for Fast and General Data Processing on Large Clusters”.
4. Databricks resources (https://databricks.com/resources/slides).
5. Apache Spark programming guide (https://spark.apache.org/docs/1.6.0/programming-guide.html).

References

6. Learning Spark – O'reilly
7. https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-sparkcontext.html
8. http://blog.explainmydata.com/2014/05/spark-should-be-better-than-mapreduce.html
9. http://horicky.blogspot.com/2013/12/spark-low-latency-massively-parallel.html
10. http://horicky.blogspot.com/2013/12/spark-low-latency-massively-parallel.html

Thank you!