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
University of California, Berkeley

Presenter: Yejia Liu

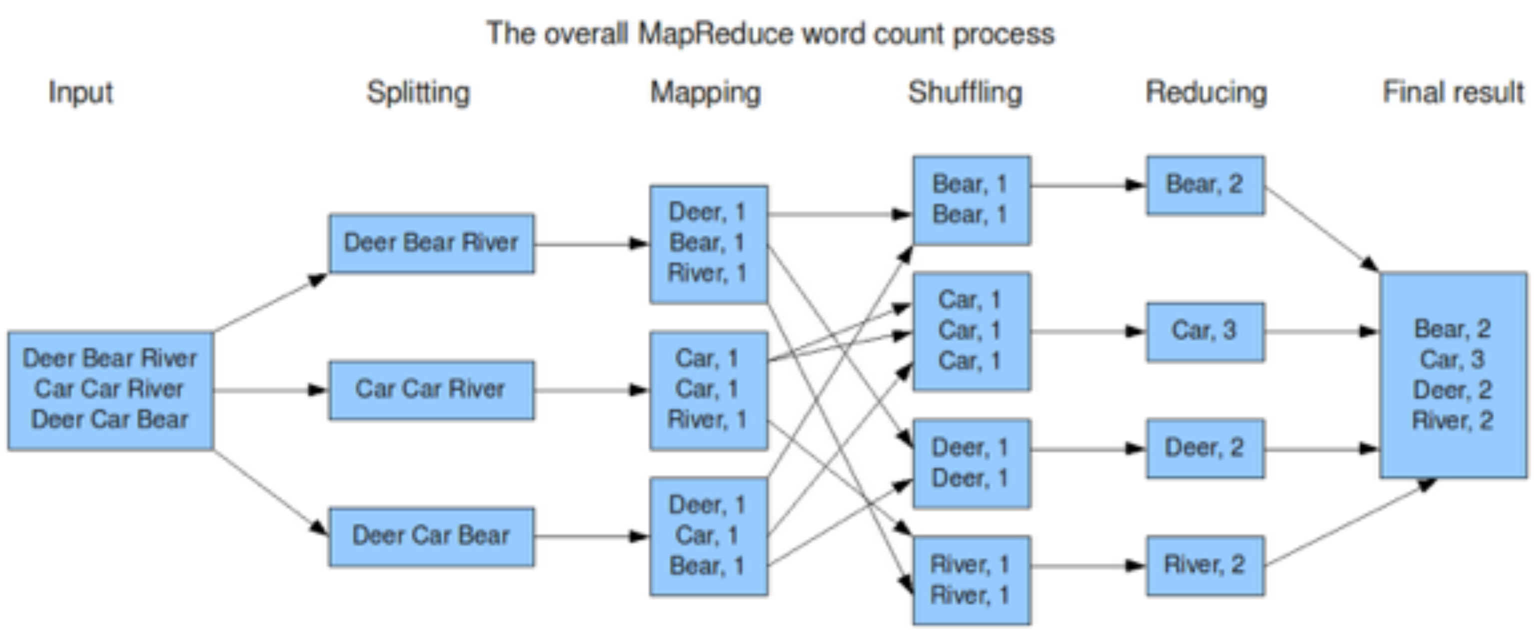
outline

- Why RDD
- RDD Overview
- Spark
- Representing RDD
- Implementation
- Evaluation
- Q&A

Why RDD?

Already have,

- MapReduce: Map(sub-divide& conquer), Reduce(combine& reduce cardinality)



- Dryad: for data parallel applications' execution
- Pregel: for large-scale graph processing

Why RDD?

However,

- MapReduce: huge memory consumption, batch processing orientation(interactive problem)
- Pregel: only support for specific computation patterns
- Inefficient for **interactive** data mining tools and **iterative** machine learning algorithms

We hope,

Find an efficient way to generally ***share data*** across multiple computations

RDD Overview

- **In-Memory data sharing**

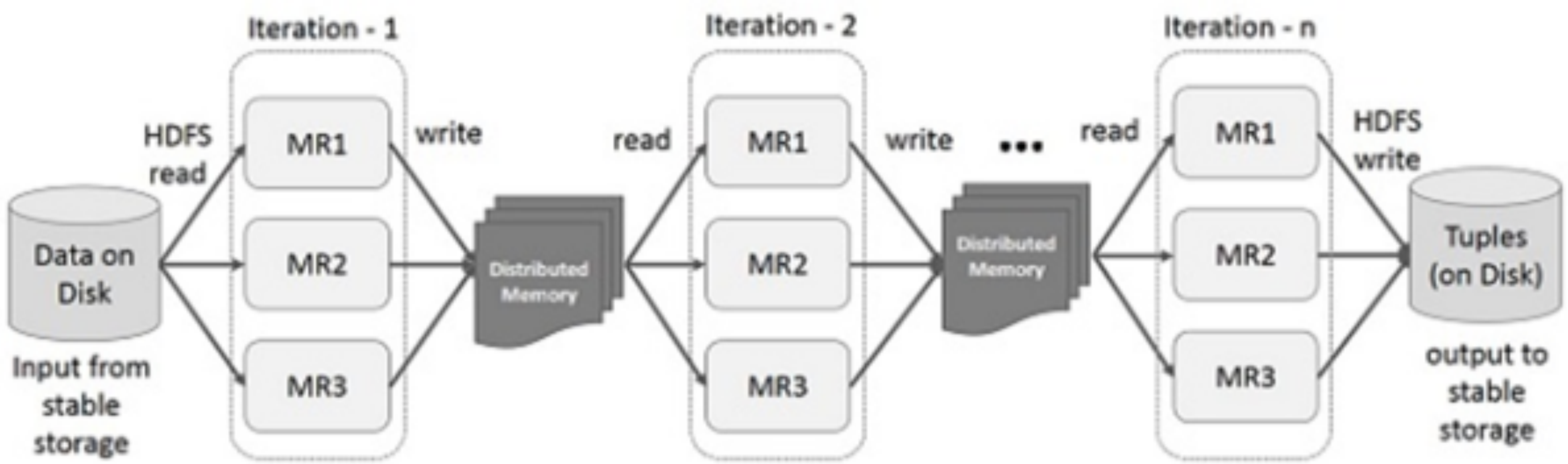
- **Fault tolerant**, parallel data structure
  — achieved by **coarse-grained** transformation
  — achieved by lineage graph of transformations
    * log one operation applied to many data items
    * recompute lost partitions on failure
    * no cost if no failure

- Read-only
  — *easy to checkpoint for RDD with long lineage graphs*

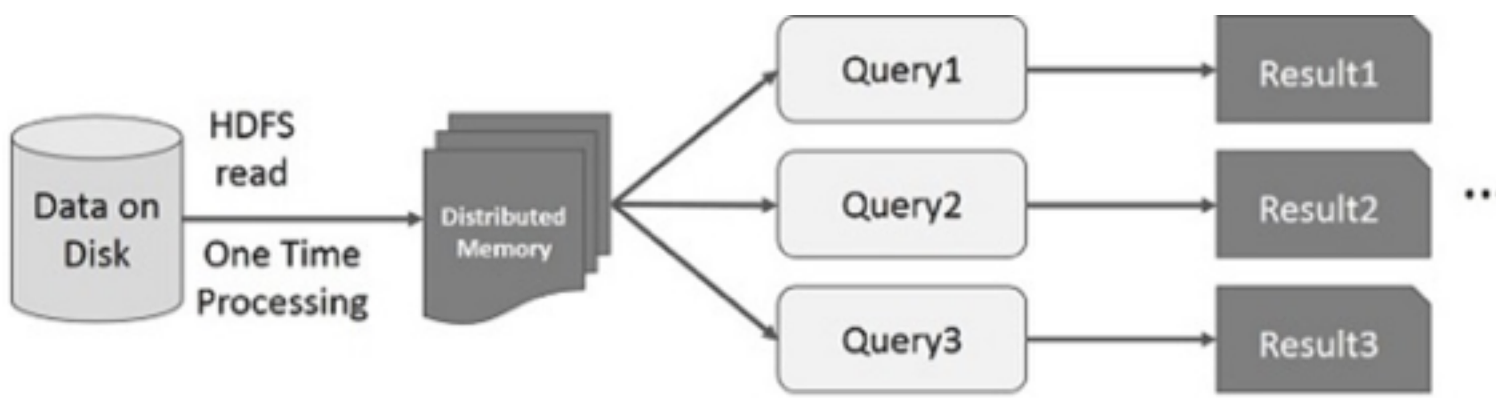
- Users can control partitioning
  — *useful for placement optimization by partitioning elements across machines based on the key*

- Rich set of operators to manipulate data

RDD Overview



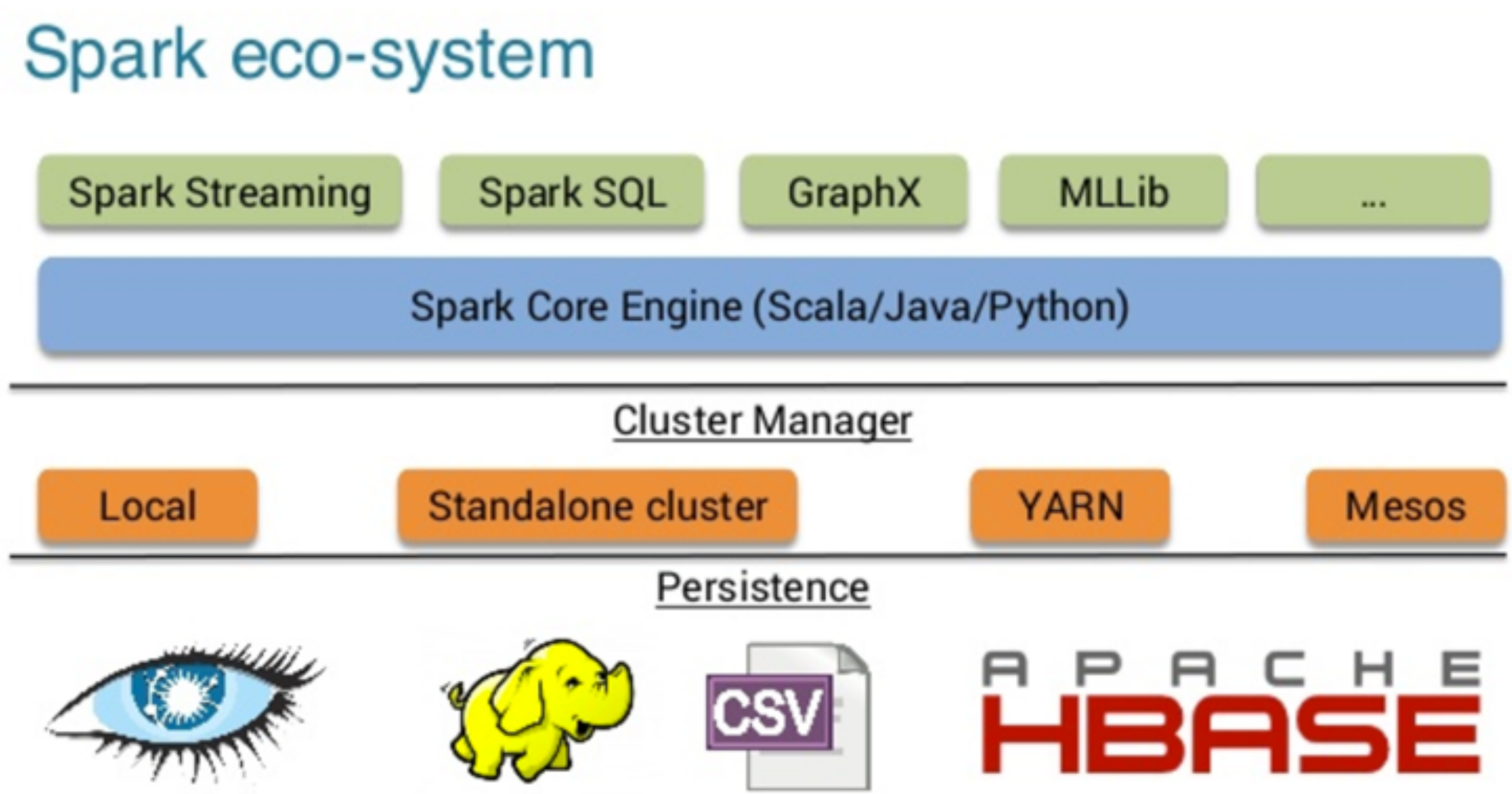
Iterative operations on Spark RDDs



Interactive operations on Spark RDDs

Spark Overview

A *fast* and *general* engine for large-scale data processing



Spark Programming Interface

- Transformation
  - define one or more RDDs from data on stable storage or other RDDs

  *** map(), filter(), groupByKey(), reduceByKey(), union(), join(), sort(), partitonBy()

Lazy Evaluation

- Action
  - return a value to the application or export data to stable storage

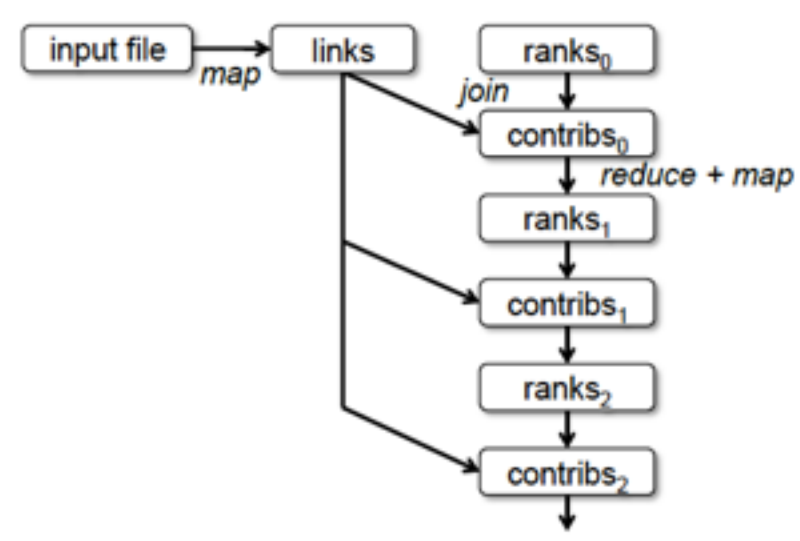
  *** count(), collect(), reduce(), lookup(), save()

Example: Pagerank

```
//Page Rank
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
```



Representing RDD

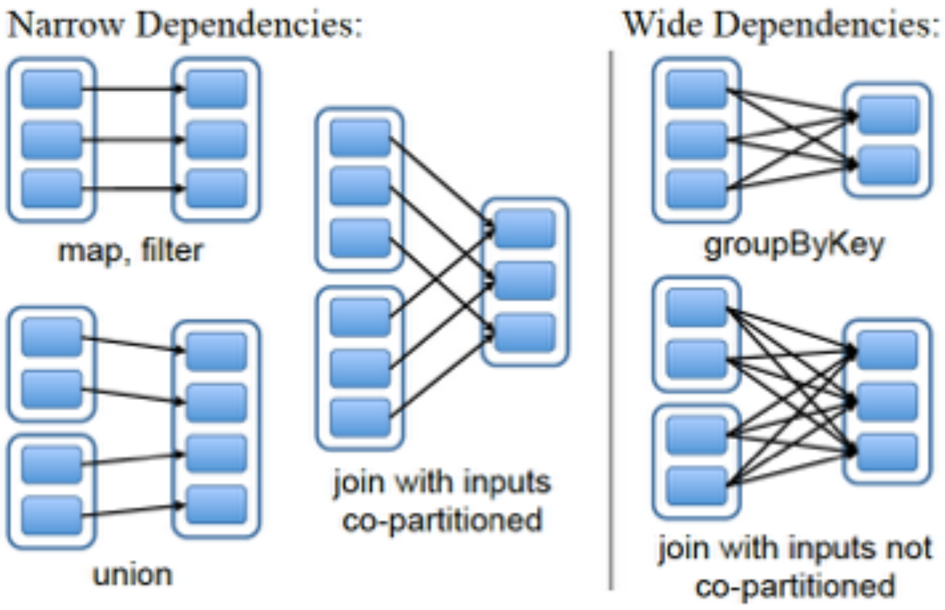
Goal

- To track lineage of RDDs through a series of transformations
- To enable system and user manipulate RDDs via a rich set of operators

Solution

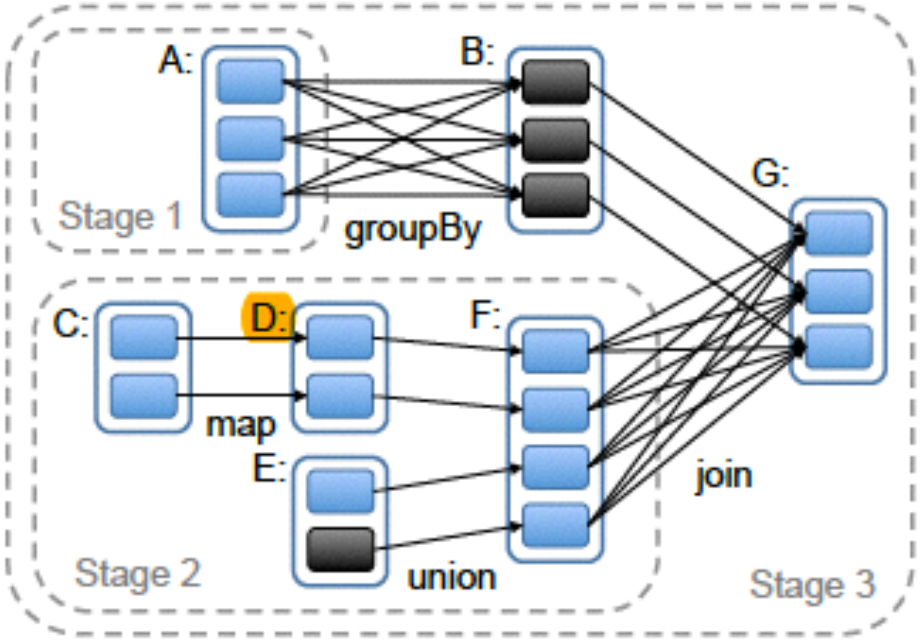
Graph-Based representation

- A set of **partitions**
- A set of **dependencies** on parent
  - wide dependency: one- to- many
  - narrow dependency: one- to- one
- **Metadata** about partitioning schema and data
- **Function** to compute dataset based on parent



Implementation

1.Job Scheduling



- Build DAG of **stages** when action happens
  - based on RDD's lineage graph
  - for each stage, pipelined transformations with narrow dependencies
  - boundaries between RDDs requiring wide dependencies
- Launch tasks to compute until get target RDD

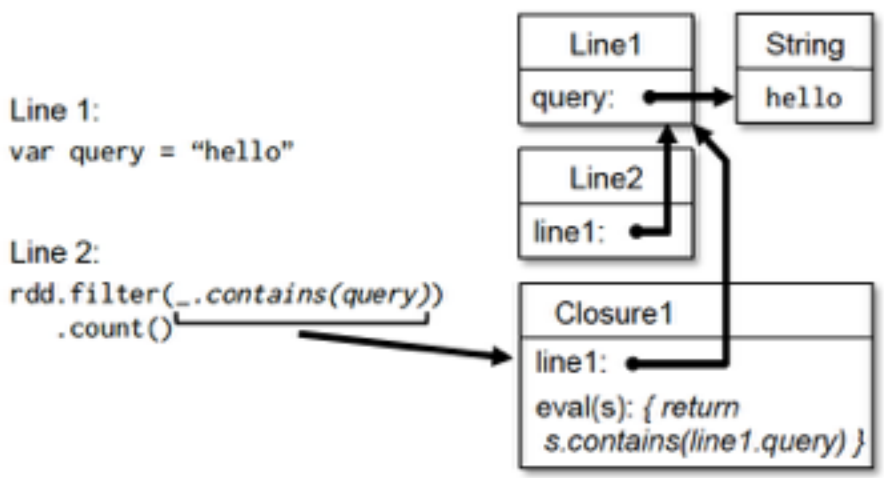
** Delay Scheduling*

*For **data locality**, distribute a task to the node containing its requiring data. If the node not available, the wait for a short time.*

Implementation

2. Interpreter Integration

- Interactive shell
- Two changes
  - Class Shipping: work node serves classes through HTTP
  - Modified Code Generation: objects are referred directly



a) Lines typed by user b) Resulting object graph

Implementation

3. Memory Management

- In-memory storage
- On-disk storage
- LRU policy

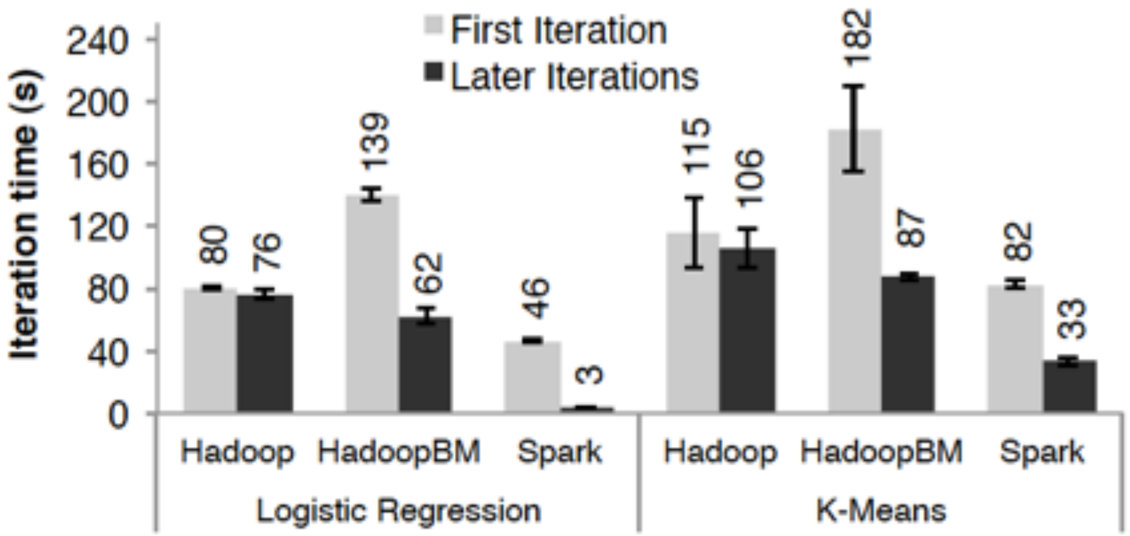
*** User can decide which RDD to cache, automatic GC

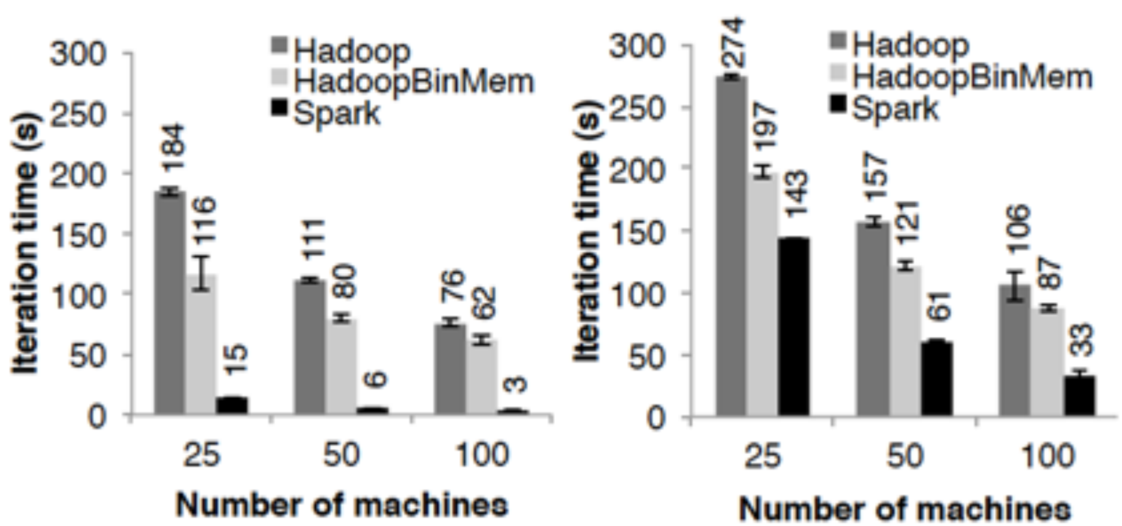
4. Support for Checkpointing

- Useful for RDD with long lineage graph with wide dependencies
- User can decide which data to checkpoint
- Easy to checkpoint with RDD's read-only nature

Evaluation

- In iterative machine learning applications,



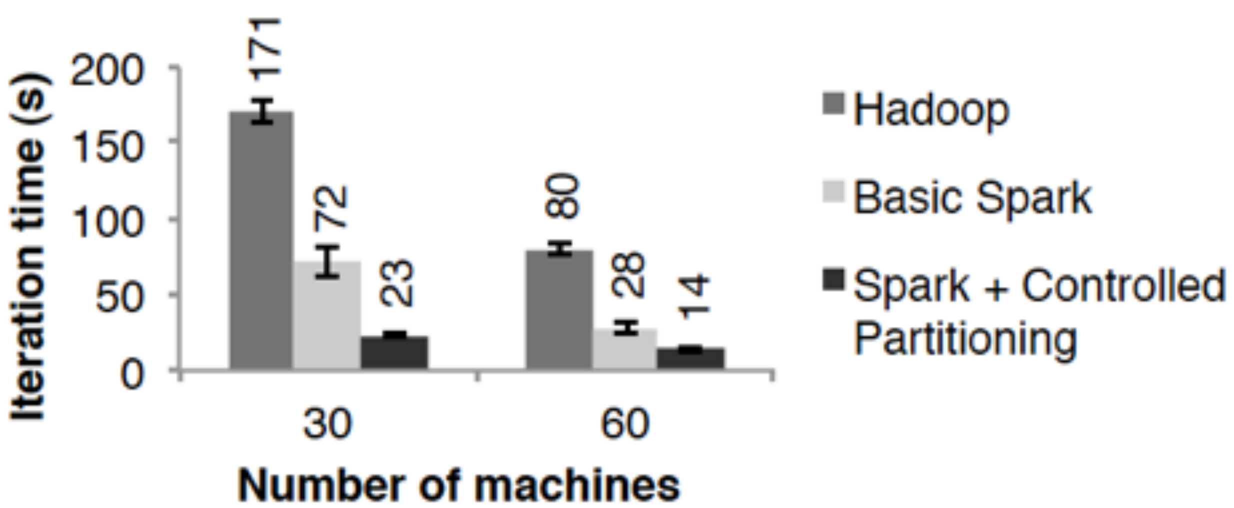


(a) Logistic Regression (b) K-Means

outperforms Hadoop by up to **20x**

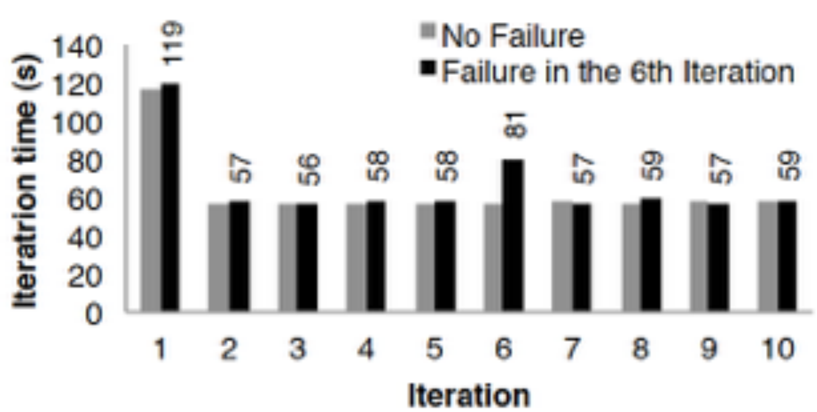
Evaluation

- In PageRank,



2.4x speedup over Hadoop on 30 nodes

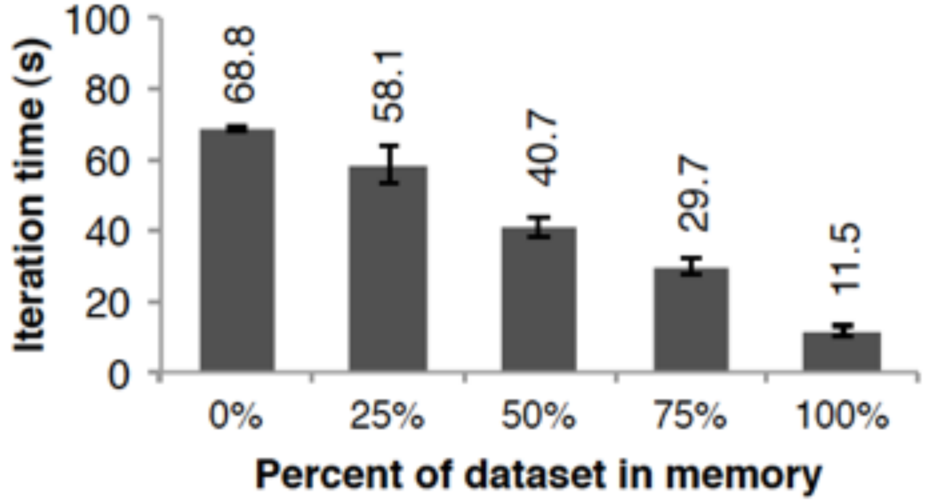
- Fault Tolerance



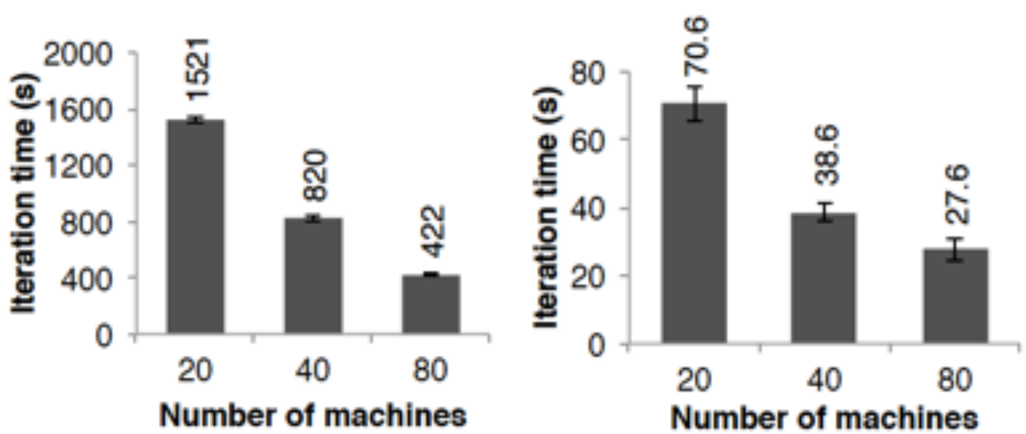
Iteration time: 58s-> at 6th iteration, a node failed-> 81s-> after recovery->58s

Evaluation

- Behaviour with insufficient memory



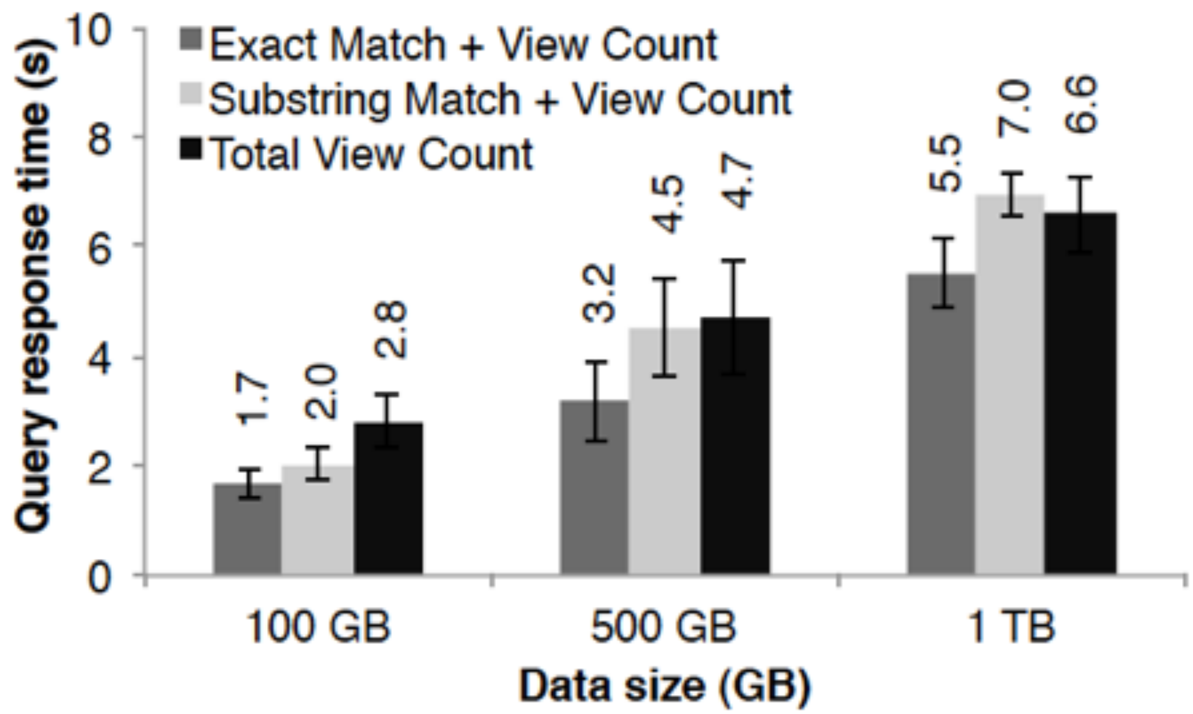
- In User Application



(a) Traffic modeling (b) Spam classification

Evaluation

- In interactive data mining(1TB of Wiki pages view log)



Q&A

THANKS!

Any question?