

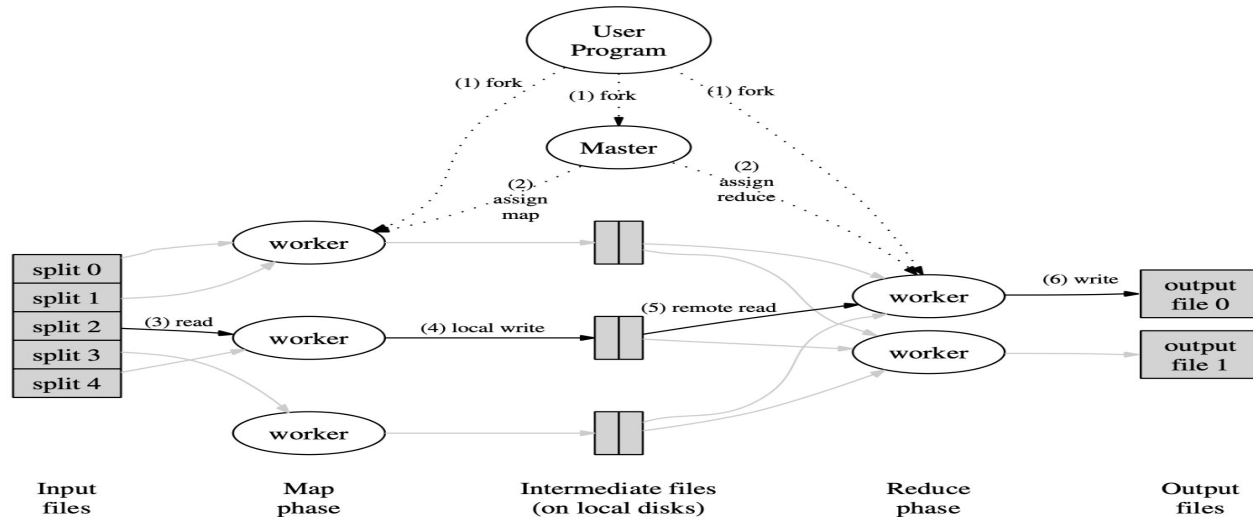
Resilient Distributed Dataset

Nayan Bhatia
CSE 215 Presentation

MapReduce:Advantages and drawbacks
Apache Sparks(Mainly focusing on RDD)
Motivation for RDD
Challenges
Recovery
Sparks operations
Scala
Conclusion
What has changed after the RDD paper?
Alternatives?
Discussion

MapReduce

- MapReduce is a programming model and software framework for large-scale data processing
- How it works?



MapReduce

- Key features
- Use case

Advantage of MapReduce

- Batch Processing
- Scalability
- Simple Programming Model
- Maturity
- Cost Effective solution
- Good for large-scale data processing
- Can run on commodity hardware.

Limitations of MapReduce

- Despite its popularity, MapReduce has some limitations.
- One of its major limitations is its slow performance when processing large amounts of data.
- Additionally, MapReduce has difficulties handling iterative algorithms and machine learning.
- Real-time data processing is also a challenge for MapReduce.

<https://datawhatnow.com/mapreduce-shuffle-sort/>

Apache Spark

- To address the limitations of MapReduce, Apache Spark was created.
- Spark is a distributed computing framework that provides a fast and easy-to-use platform for big data processing.

Features of Spark

- Spark has several key features that make it a better option than MapReduce.
- One of its most significant features is its in-memory processing capability, which makes it much faster than MapReduce.
- Spark also supports iterative algorithms, real-time processing, and offers APIs in multiple programming languages.

Resilient Distributed Dataset(RDD)

- Fundamentals data structure in Apache Spark
- Designed to support large-scale data processing
- Partitioned across multiple nodes for parallel processing and scalability

Resilient Distributed Dataset(RDD)

- Resilient, ensuring data is not lost in case of node failures
- Uses lazy evaluation for efficient processing
- Can be cached in memory for improved performance

Resilient Distributed Dataset(RDD)

- Widely used in Apache Hadoop ecosystem
- Used for a range of data processing tasks (batch processing, real-time processing, graph processing)
- Simple API makes it accessible to a wide range of developers and data scientists
- Improves productivity and makes data processing more accessible.

Motivation

MapReduce greatly simplified “big data” analysis on large, unreliable clusters

But as soon as it got popular, users wanted more:

- » More **complex**, multi-stage applications
(e.g. iterative machine learning & graph processing)
- » More **interactive** ad-hoc queries

Response: *specialized* frameworks for some of these apps (e.g. Pregel for graph processing)

Motivation

Complex apps and interactive queries both need one thing that MapReduce lacks:

Efficient primitives for **data sharing**

In MapReduce, the only way to share data across jobs is stable storage → slow!

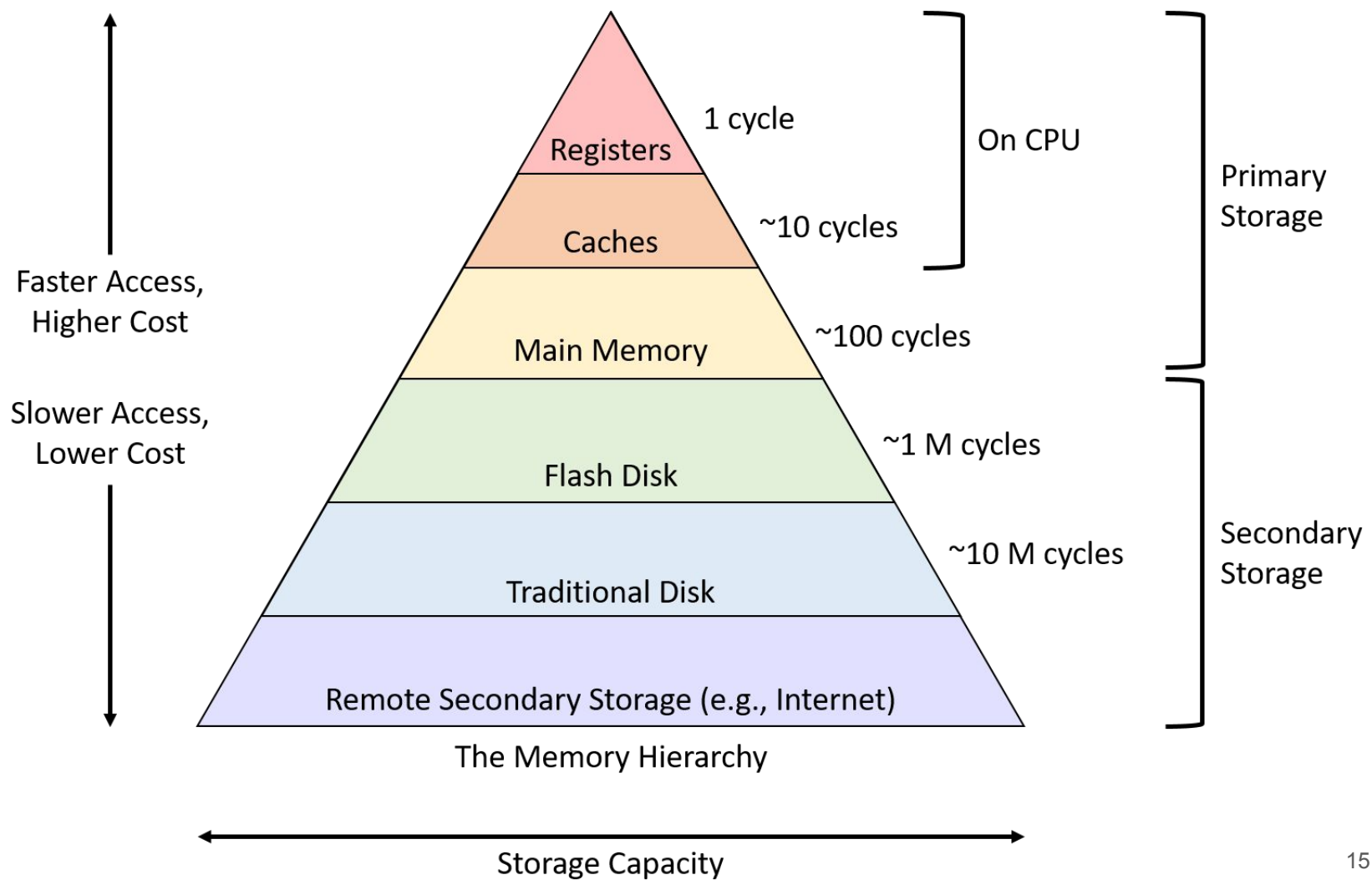
Motivation

MapReduce greatly simplified “big data” analysis on large, unreliable clusters

But as soon as it got popular, users wanted more:

- >> More **complex**, multi-stage applications
(e.g. iterative machine learning & graph processing)
- >> More **interactive** ad-hoc queries

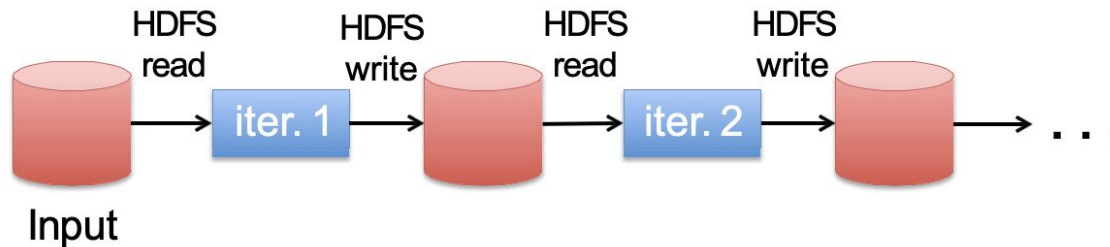
Response: *specialized* frameworks for some of these apps (e.g. Pregel for graph processing)



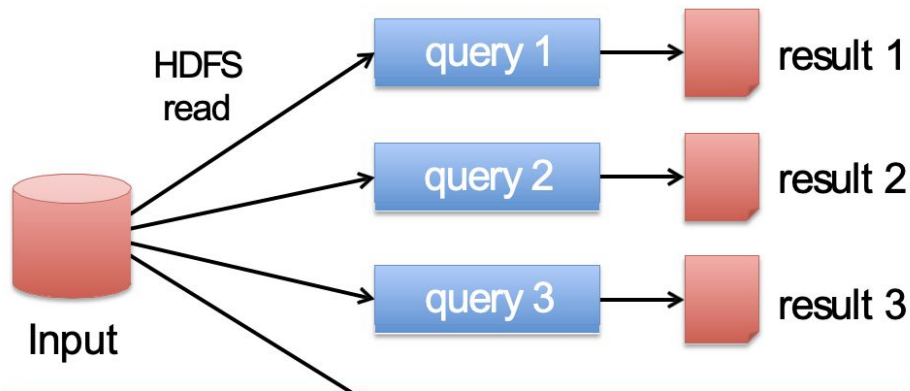
Examples

Blue boxes- map reduce stages
Red cylinders-stable storage(it also has to replicate across three machines to support Fault tolerance!)

Iterative queries

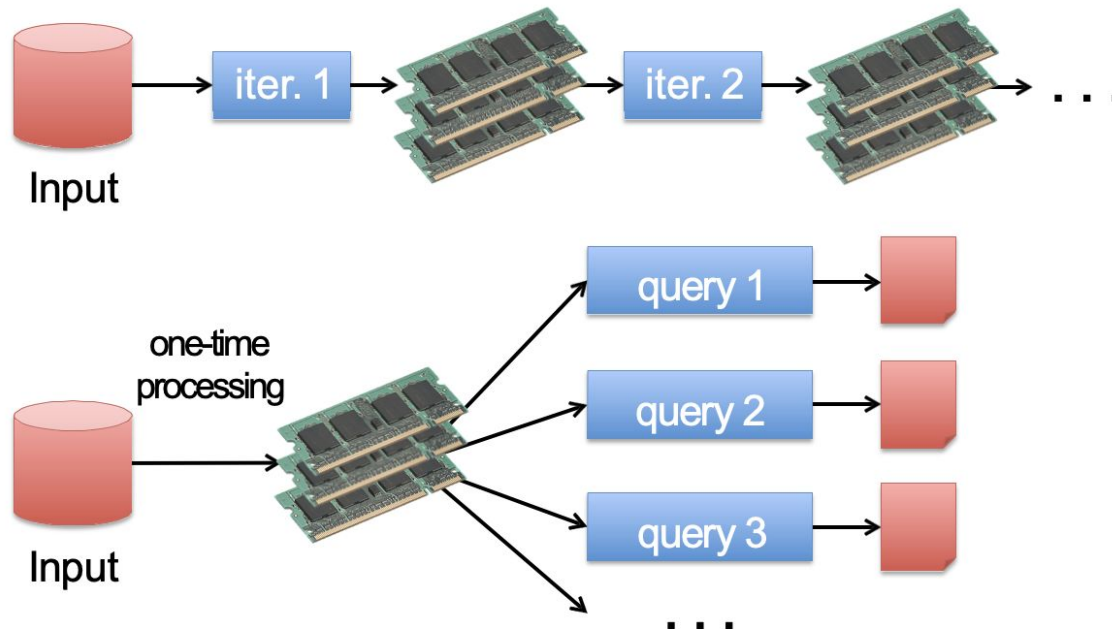


interactive queries



Slow due to replication and disk I/O,
but necessary for fault tolerance

Goal: In-Memory Data Sharing



10–100× faster than network/disk, but how to get FT?

Challenge

How to design a distributed memory abstraction that is both **fault-tolerant** and **efficient**?

Challenge

Existing storage abstractions have interfaces based on *fine-grained* updates to mutable state

- >> RAMCloud, databases, distributed mem, Piccolo

Requires replicating data or logs across nodes for fault tolerance

- >> Costly for data-intensive apps
- >> 10--100x slower than memory write

Solution: Resilient Distributed Datasets (RDDs)

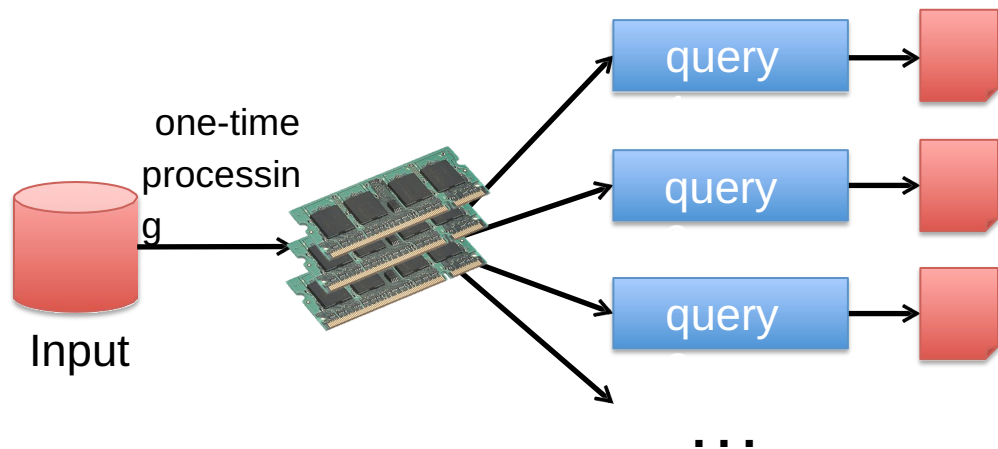
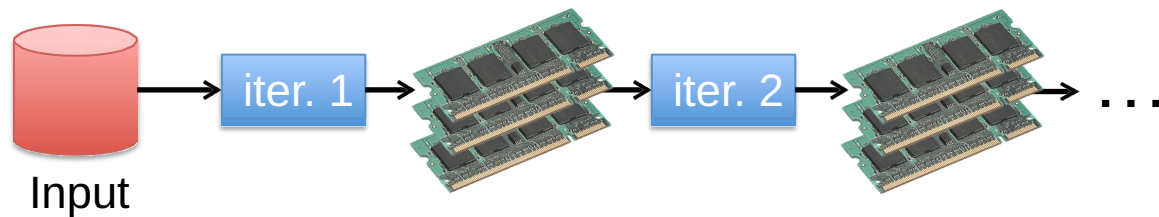
Restricted form of distributed shared memory

- >> Immutable, partitioned collections of records
- >> Can only be built through *coarse-grained* deterministic transformations (map, filter, join, ...)

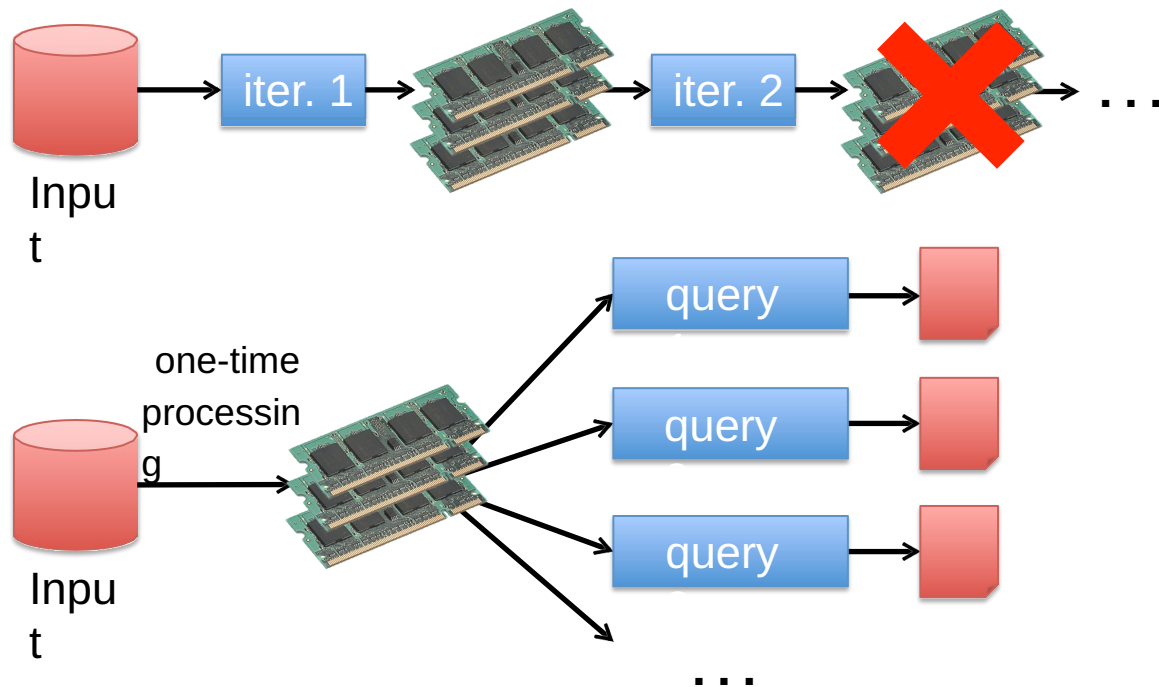
Efficient fault recovery using *lineage*

- >> Log one operation to apply to many elements
- >> Recompute lost partitions on failure
- >> No cost if nothing fails

RDD Recovery



RDD Recovery

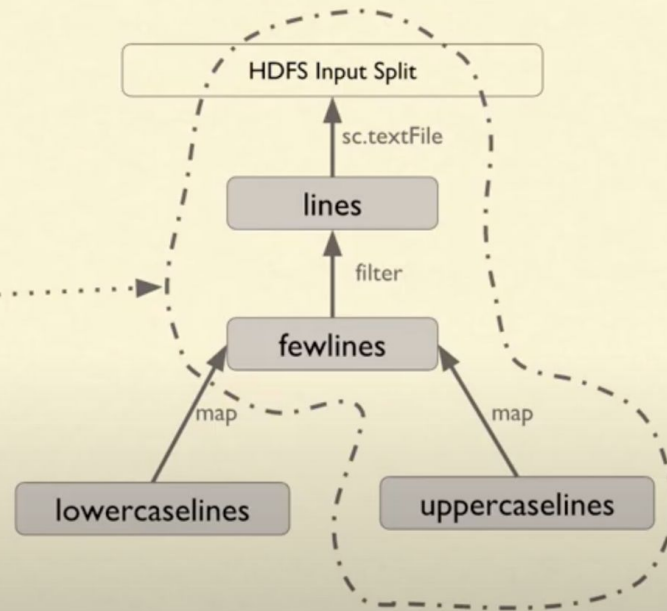


Lineage Graph

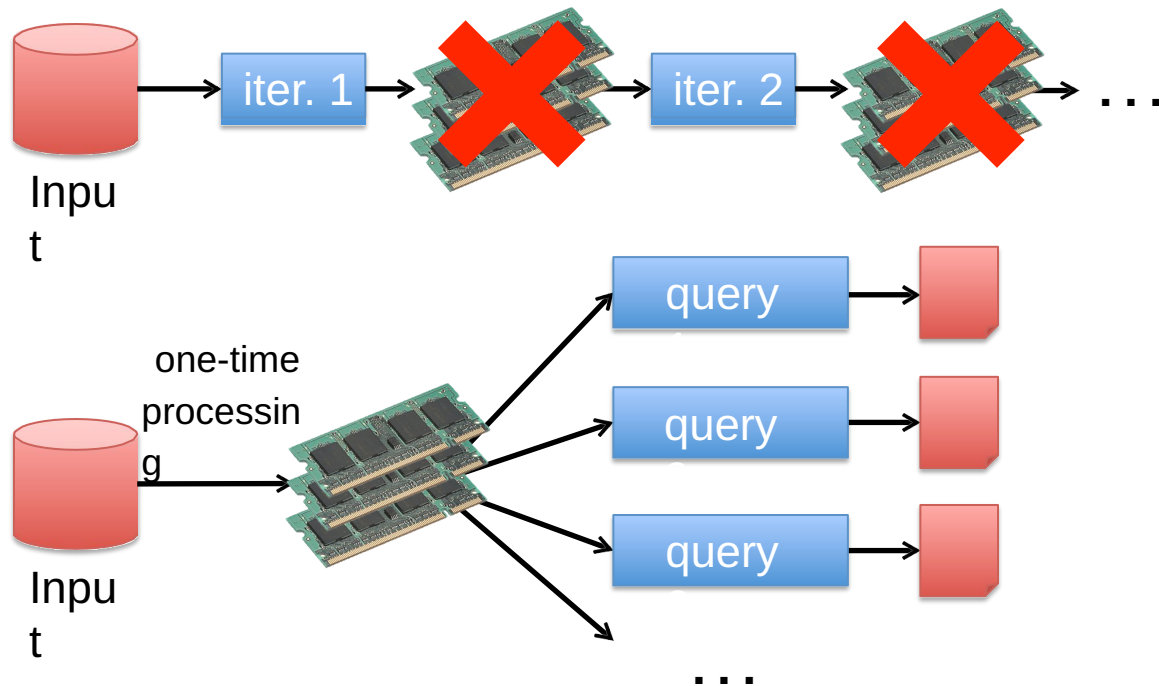
Spark Code

```
lines = sc.textFile("myfile");  
fewlines = lines.filter(...)  
uppercaselines = fewlines.map(...)  
lowercaselines = fewlines.map(...)  
  
uppercaselines.count()
```

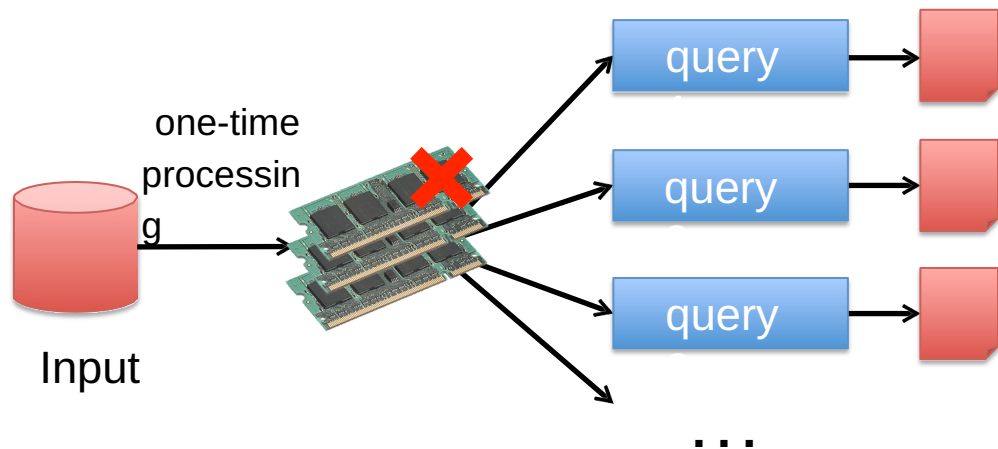
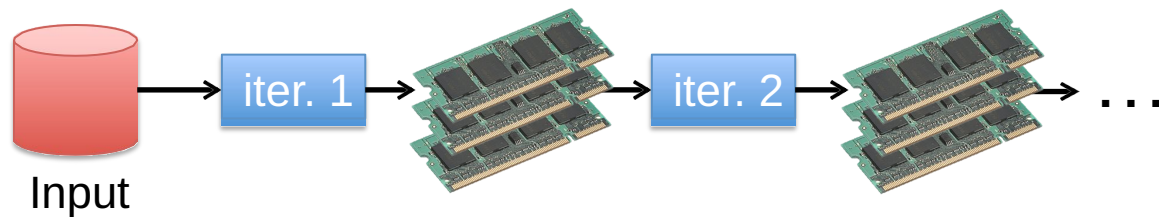
Lineage Graph



RDD Recovery



RDD Recovery



Generality of RDDs

Despite their restrictions, RDDs can express surprisingly many parallel algorithms

- » These naturally *apply the same operation to many items*

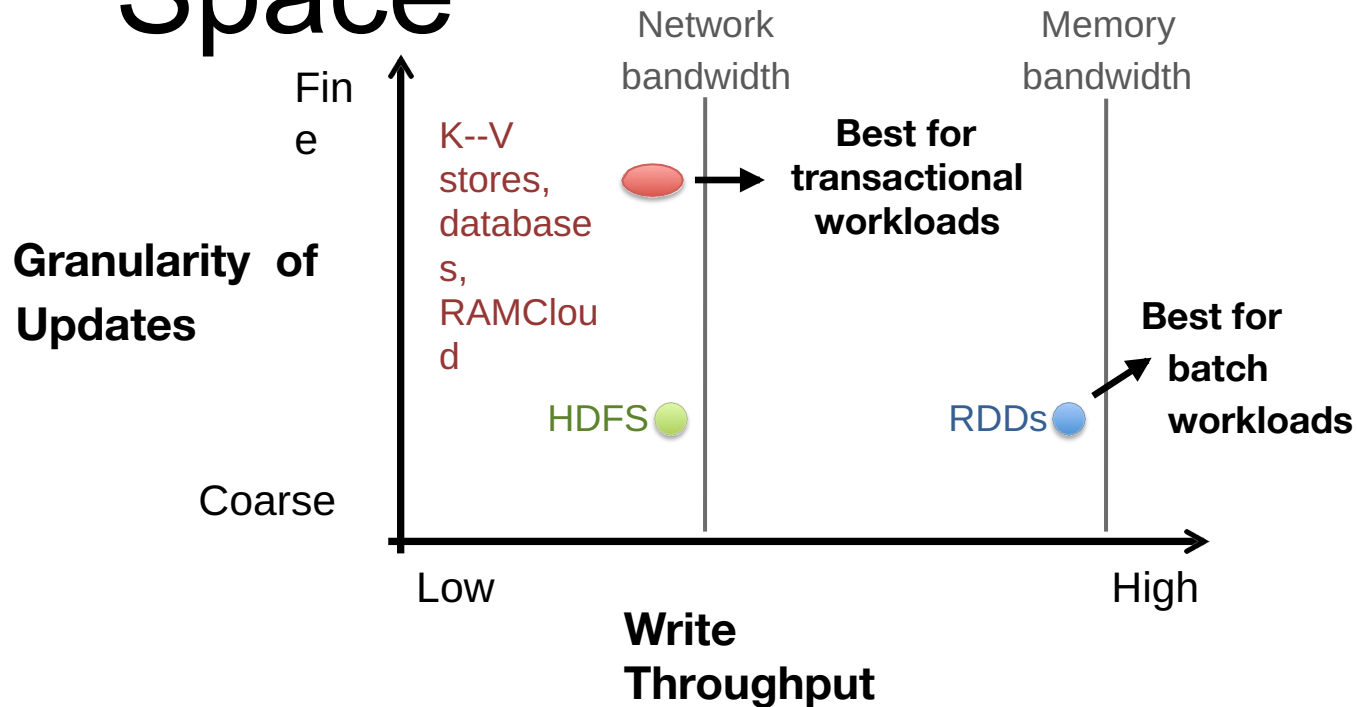
Unify many current programming models

- » *Data flow models*: MapReduce, Dryad, SQL, ...

- » *Specialized models* for iterative apps: BSP (Pregel), iterative MapReduce (Haloop), bulk incremental, ...

Support *new apps* that these models don't

Tradeoff Space



Transformations and Actions

| | |
|--|--|
| Transformations (define a new RDD) | map flatMap filter union sample join groupByKey cogroup reduceByKey cross sortByKey mapValues |
| Actions (return a result to driver program) | collect reduce count save lookupKey |



Here are examples for each of the RDD operations:



1. Transformations:

- ``map``: This transformation returns a new RDD by applying a function to each element in the original RDD. For example, given an RDD of numbers, you can use ``map`` to create a new RDD that contains the squares of each number:

python

Copy code

```
rdd = sc.parallelize([1, 2, 3, 4])
squareRDD = rdd.map(lambda x: x**2)
# squareRDD = [1, 4, 9, 16]
```

- ``filter``: This transformation returns a new RDD that contains only the elements that satisfy a given condition. For example, given an RDD of numbers, you can use ``filter`` to create a new RDD that contains only even numbers:

python

Copy code

```
rdd = sc.parallelize([1, 2, 3, 4])
evenRDD = rdd.filter(lambda x: x % 2 == 0)
# evenRDD = [2, 4]
```

Note:
Transformations are lazily evaluated!

- `flatMap`: This transformation returns a new RDD by applying a function to each element in the original RDD and flattening the results. For example, given an RDD of sentences, you can use `flatMap` to create a new RDD that contains all the words in each sentence:

less


 Copy code

```
rdd = sc.parallelize(["Hello world", "How are you?"])
wordRDD = rdd.flatMap(lambda sentence: sentence.split(" "))
# wordRDD = ["Hello", "world", "How", "are", "you?"]
```

2. Actions:

- **`count`**: This action returns the number of elements in the RDD. For example, you can use **`count`** to count the number of elements in an RDD:


makefile

 Copy code

```
rdd = sc.parallelize([1, 2, 3, 4])
count = rdd.count()
# count = 4
```

- **`first`**: This action returns the first element in the RDD. For example, you can use **`first`** to get the first element in an RDD:


makefile

 Copy code

```
rdd = sc.parallelize([1, 2, 3, 4])
first = rdd.first()
# first = 1
```

- **`reduce`**: This action returns the result of reducing the elements of the RDD using a given binary operator. For example, you can use **`reduce`** to sum the elements in an RDD:

python

 Copy code

```
rdd = sc.parallelize([1, 2, 3, 4])
sum = rdd.reduce(lambda x, y: x + y)
# sum = 10
```

Type Inference

```
val df = spark.read.csv("data.csv")
df.printSchema()
root
 |-- col1: string (nullable = true)
 |-- col2: integer (nullable = true)
 |-- col3: double (nullable = true)
```

```
val df = spark.read.csv("data.csv")
df.filter(col("col2") > 10).show()
df.filter(col("col1") > 10).show()
```


Spark Programming Interface

DryadLINQ-like API in the Scala language

Usable interactively from Scala interpreter

Provides:

- >> Resilient distributed datasets (RDDs)
- >> Operations on RDDs: *transformations* (build new RDDs),
actions (compute and output results)
- >> Control of each RDD's *partitioning* (layout across nodes)
and *persistence* (storage in RAM, on disk, etc)

Example: Log Mining

The masters contain the lineage information

Load error messages from a log into memory, then interactively search for various patterns

Base RDD

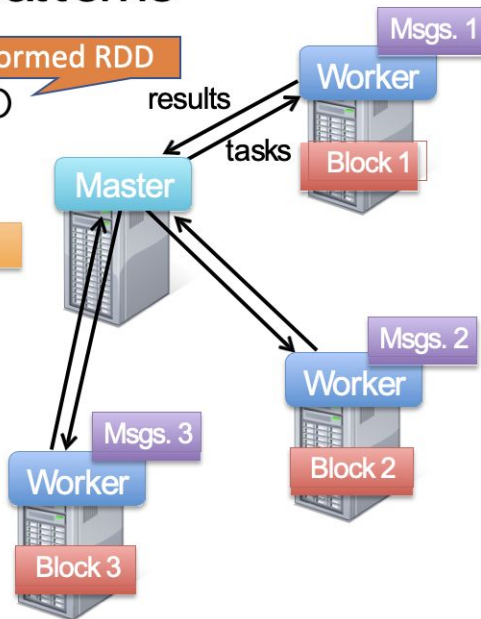
```
lines = spark.textFile("hdfs://...")  
errors = lines.filter(_.startsWith("ERROR"))  
messages = errors.map(_.split('\t')(2))  
messages.persist()
```

Transformed RDD

```
messages.filter(_.contains("foo")).count  
messages.filter(_.contains("bar")).count
```

Action

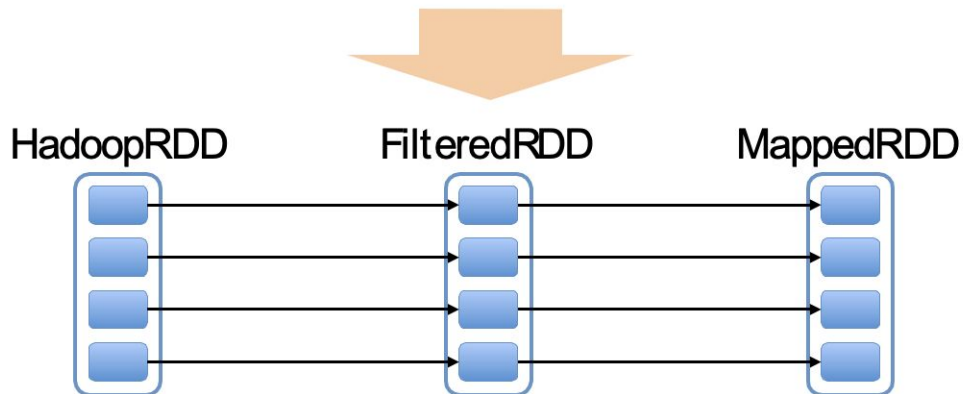
Result: scaled to 1TB data in 5-7s
(vs 170s for on-disk data)



Fault Recovery

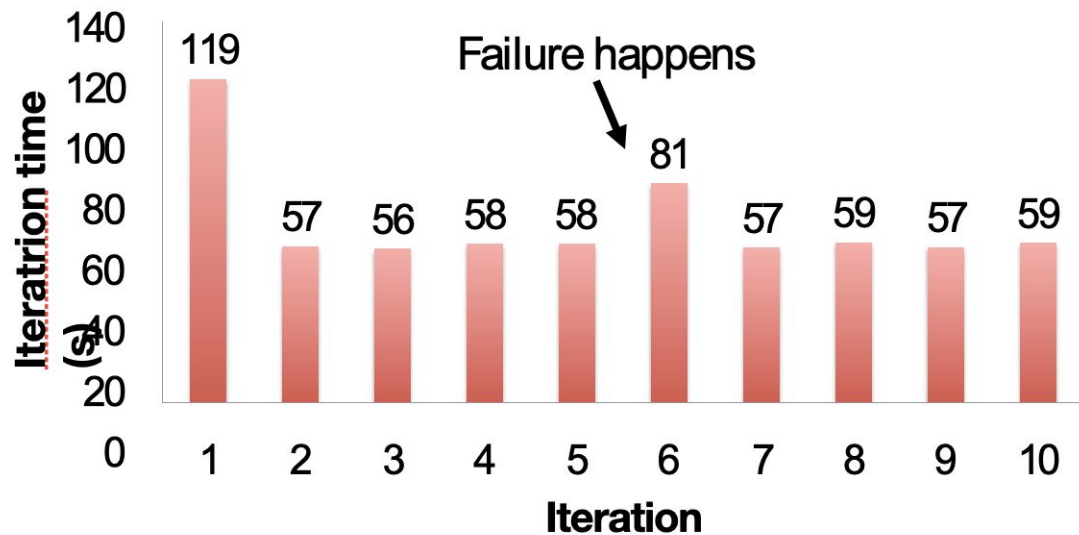
RDDs track the graph of transformations that built them (their *lineage*) to rebuild lost data

E.g.: `messages = textFile(...).filter(_.contains("error")).map(_.split('\t')(2))`



Either lineage or checkpointing for fault recovery.

Fault Recovery Results



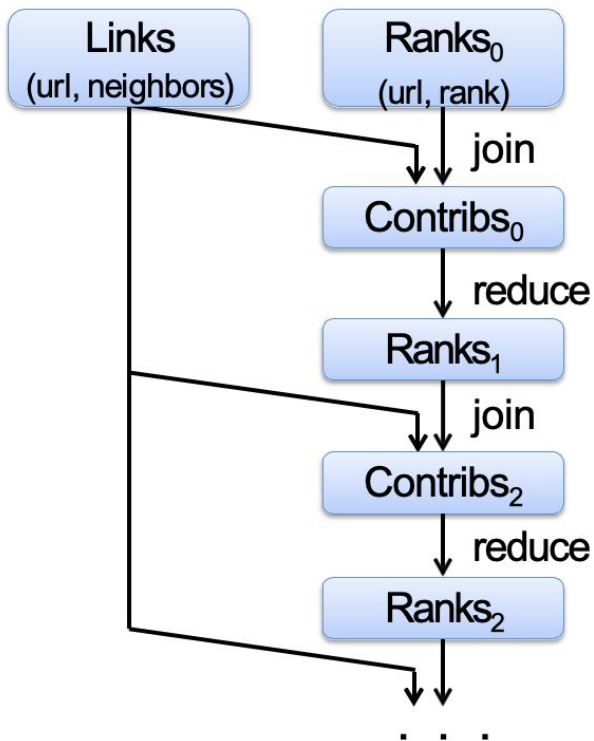
Example: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each page's rank to

$$\sum_{i \in \text{neighbors}} \text{rank}_i / |\text{neighbors}_i|$$

```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs
for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
}
```

Optimizing Placement



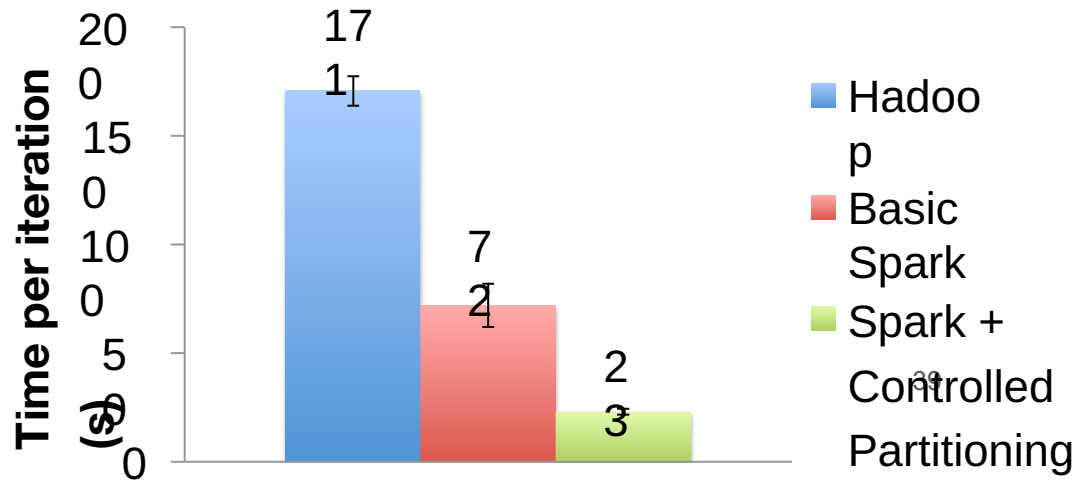
links & ranks repeatedly joined

Can *co-partition* them (e.g. hash both on URL) to avoid shuffles

Can also use app knowledge, e.g., hash on DNS name

```
links = links.partitionBy(  
    new URLPartitioner())
```

PageRank Performance

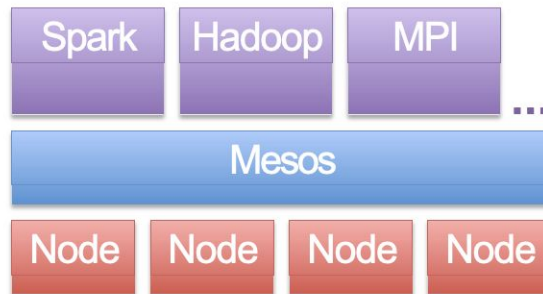


Implementation

Runs on Mesos [NSDI 11]
to share clusters w/ Hadoop

Can read from any Hadoop
input source (HDFS, S3, ...)

No changes to Scala language or compiler
» Reflection + bytecode analysis to correctly ship code



www.spark-project.org

Programming Models Implemented on Spark

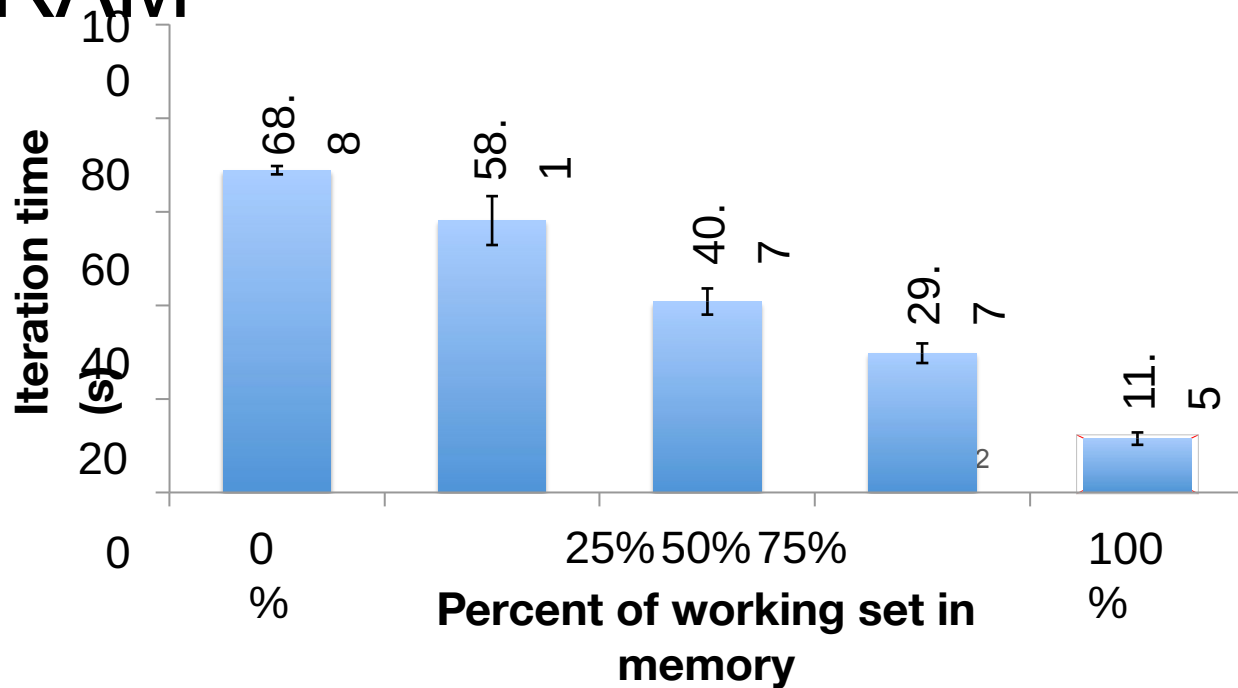
RDDs can express many existing parallel models

- » **MapReduce, DryadLINQ**
- » **Pregel** graph processing [200 LOC]
- » **Iterative MapReduce** [200 LOC]
- » **SQL**: Hive on Spark (Shark) [in progress]

All are based on
coarse-grained
operations

Enables apps to efficiently *intermix* these models

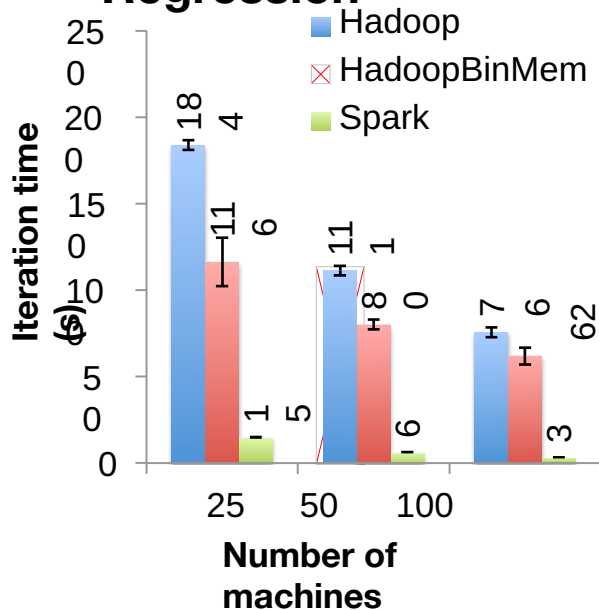
Behavior with Insufficient RAM



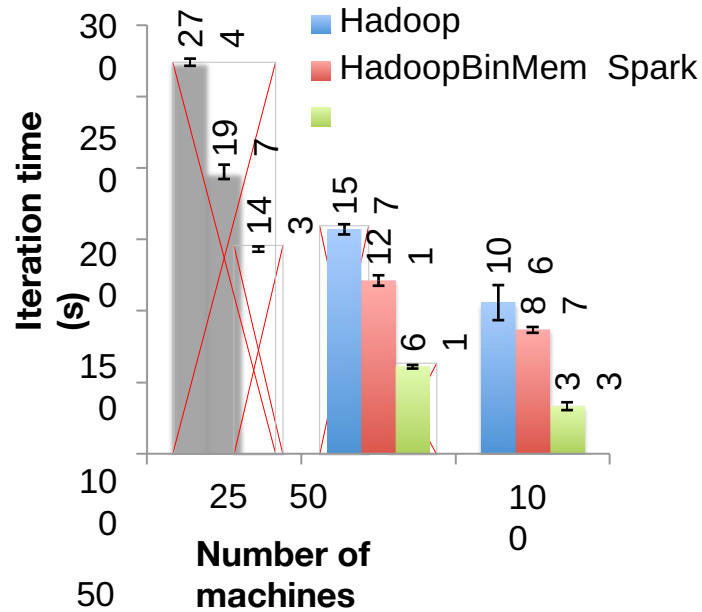
Scalability

HadoopBinMem is an optimized version of Hadoop that uses a binary format and writes intermediate data to an in-memory file system.

Logistic Regression



K--Means



Conclusion

RDDs offer a simple and efficient programming model for a broad range of applications

Leverage the coarse-grained nature of many parallel algorithms for low-overhead recovery

44

Try it out at www.spark-project.org

What has changed?

DataFrame API: Spark introduced the DataFrame API, which provides a more user-friendly interface for working with structured data.

Structured Streaming: Spark introduced Structured Streaming, a high-level API for building real-time data pipelines.

Catalyst Optimizer: Spark introduced the Catalyst Optimizer, a highly optimized query optimizer that provides performance improvements for Spark SQL and DataFrame operations.

MLlib: Spark's machine learning library, MLlib, has been improved and expanded to include a wider range of algorithms and tools for big data machine learning.

Cluster Manager Integration: Spark has **improved its integration** with various cluster managers, including standalone, Apache Mesos, and Apache YARN, making it easier to deploy and manage Spark clusters.

SparkR: Spark introduced SparkR, an R interface to Spark, allowing R users to take advantage of Spark's distributed computing capabilities.

Graph Processing: Spark introduced GraphX, a graph processing library, and later introduced GraphFrames, a graph processing library built on top of DataFrames.

Issues with Sparks?

Interactive queries: RDDs are optimized for batch processing and are not well-suited for interactive queries, which require low latency.

Fine-grained updates: RDDs are designed for large-scale data processing and are not optimized for fine-grained updates to small portions of the data.

Relationships between data: RDDs are designed for unstructured data and do not have the capability to handle complex relationships between data.

Graph processing: While RDDs can be used for graph processing, it is not their primary strength and there are other frameworks, such as Apache Giraph, which are better suited for graph processing tasks.

Real-time stream processing: RDDs are designed for batch processing and are not well-suited for real-time stream processing, which requires low latency and the ability to process data in real-time.

In these cases, alternative solutions, such as Apache Hive, Apache Cassandra, or Apache Flink, may be more suitable.

Alternatives

Apache Flink: An open-source, distributed stream processing framework.

Apache Storm: A distributed real-time processing system for processing large volumes of data.

Apache Samza: A real-time stream processing framework for processing high-volume data streams.

Apache Beam: An open-source, unified programming model for both batch and streaming data processing.

Apache Hive: A data warehousing and SQL-like query language for big data processing.

Google Cloud Dataflow: A fully managed, cloud-native data processing service for both batch and streaming data.

Apache NiFi: A data ingestion and distribution system for managing and processing data in real-time.

Comments/Questions

Intelligent decision to use Scala

Feasibility of SIMD?

Feasibility of something like Intel optane?

A lot of design decisions like checkpointing and persist are left out to user. If not used correctly, it can be costly.

Addition of Dataframes and Dataset Apis for wide dependencies.

Depending on the workloads, different accelerator integration can provide significant performance gains and enable Spark to scale to larger datasets and more complex processing requirements. Eg. Nvidia RAPIDS library for GPU acceleration, and Spark-Xilinx for FPGA acceleration.

References

https://www.usenix.org/sites/default/files/conference/protected-files/nsdi_zaharia.pdf

[Spark RDD: A Fault-Tolerant Abstraction for In-Memory Cluster Computing](#)

[Wikipedia](#)

[ChatGPT](#)