Massive Online Analysis - Storm,Spark

presentation by

R. Kishore Kumar

Research Scholar

Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur
Kharagpur-721302, India

Overview

Introduction to Big Data - Examples

- Weather Forcasting management
  - Raytheon - working with NASA and NOAA.
  - Every day, the system receives about 60GB worth of data from weather satellites.
  - Using sophisticated algorithms, they were able to process that data into environmental data records, such as cloud coverage and height. They produce sea ice concentrations, surface temperatures, as well as atmospheric pressure.
- Flight takeoff and landing
  - It produces 5 to 10 GB of data.
- Facebook
  - It produces 500+ TB of data per day.
  - User Behaviour

Introduction

- What is Big Data?
  - Big Data is more data and more varieties of data that must be handled by a conventional database.
  - The term also refers to the many tools and techniques that have emerged to help users mine valuable information from these massive data.

Introduction cont..

- Characteristics of Big Data
  - Huge volume of data
  - Complexity of datatypes and structures
  - Speed or Velocity of new data creation
- Criteria for Big Data Projects
  - Speed of decision making
  - Analysis flexibility
  - Throughput

Different types of Data Types

- **Structured** Eg. Transaction data
- **Semi Structured** Eg.XML data files that are self describing and defined by XML Schema
- **Quasi Structured** Eg.Webclickstream data that many contain some inconsistent in data values and formats
- **Unstructured** Eg.Text document,PDFs, Images, Audios, Videos

1 TB (1024 Gigabytes)	Oracle	RDBMS
1 PB (1024 Terabytes)	PDF, Excel, Word, ppt	Content Management
1 EB (1024 Petabytes)	Youtube, Tweet, FB, Wiki	No-SQL,Hadoop

HDFS Architecture

- Hadoop Distributed File System(HDFS)
  - HDFS is a file system designed for storing very large files with streaming data access, patterns, running clusters on commodity hardware.
  - HDFS is the primary storage system used by the Hadoop application.

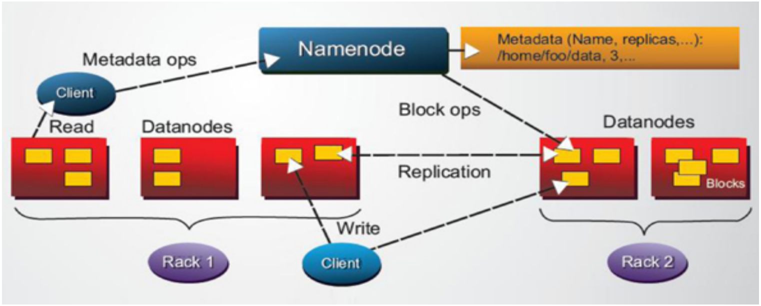


Figure : HDFS Architecture

HDFS Features

- HDFS Features
  - High fault Tolerant (replication of data in minimum 3 different nodes).
  - Suitable for application with large data sets.
  - Can be built out of commodity hardware.
- HDFS Drawback
  - Each time it should write into disk.

Apache Spark

- **Spark** uses a restricted abstraction of distributed shared memory: the Resilient Distributed Datasets (RDDs).
- An RDD is a collection of elements partitioned across the nodes of the cluster that can be operated in parallel.
- RDDs are partitioned to be distributed across nodes.
- RDDs are created by starting with a file in the Hadoop file system.
- Invoke Spark operations on each RDD to do computation.
- Offers efficient fault tolerance.

Resilient Distributed Datasets

Motivation

- A Fault-Tolerant Abstraction for In-Memory Cluster Computing
- 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)

Resilient Distributed Datasets

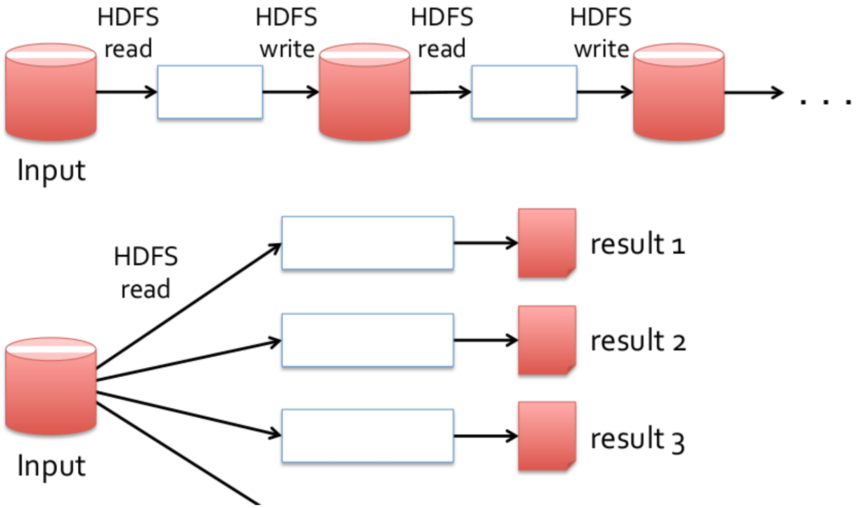
Motivation

- Complex apps and interactive queries both need one thing that
- MapReduce lacks:Efficient primitives for data sharing

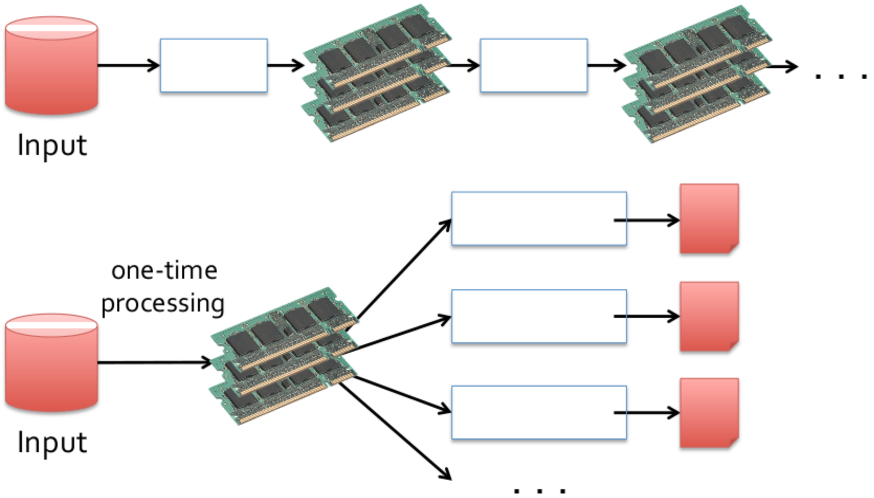
MapReduce

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

Resilient Distributed Datasets



Resilient Distributed Datasets



Challenge

Challenge

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

Challenge

Challenge

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

- RAMCloud, databases, distributed mem, Piccolo

Challenge

Challenge

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

- RAMCloud, databases, distributed mem, Piccolo

Challenge

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

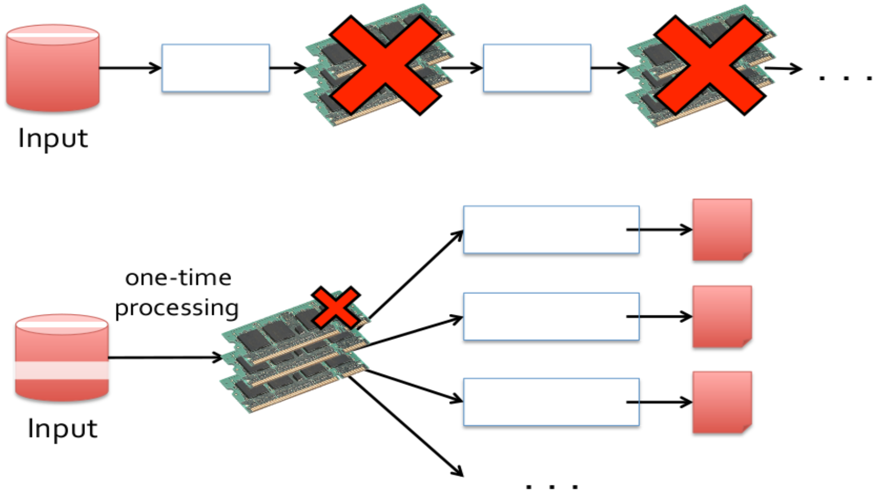
- Immutable, partitioned collections of records
- Can only be built through coarse-grained deterministic transformations (map, filter, join, ...)

Resilient Distributed Datasets (RDDs)

Efficient fault recovery using lineage

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

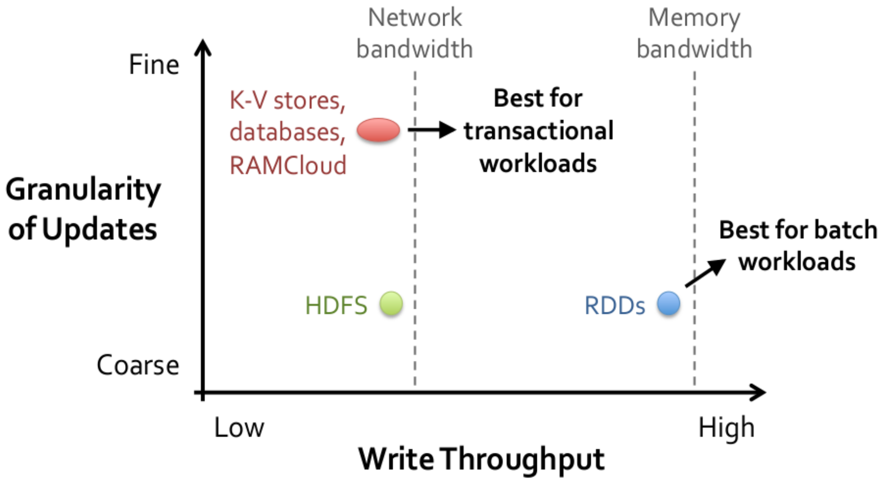
Resilient Distributed Datasets



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 dont

Tradeoff Space



Spark Programming Interface

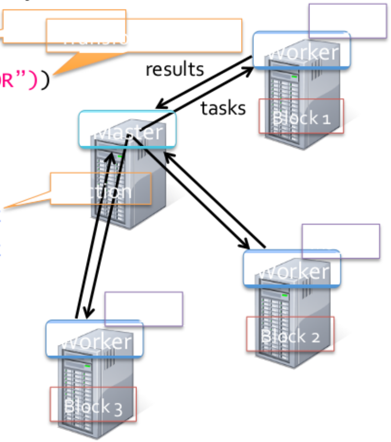
- DryadLINQ-like API in the Scala language
- Usable interactively from Scala interpreter
- Provides:
  - Resilient distributed datasets (RDDs)
  - Operations on RDDs: **transformations** (build new RDDs),**actions** (compute and output results)
  - Control of each RDDs **partitioning** (layout across nodes) and **persistence** (storage in RAM, on disk, etc)

Example:Log Mining

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

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

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

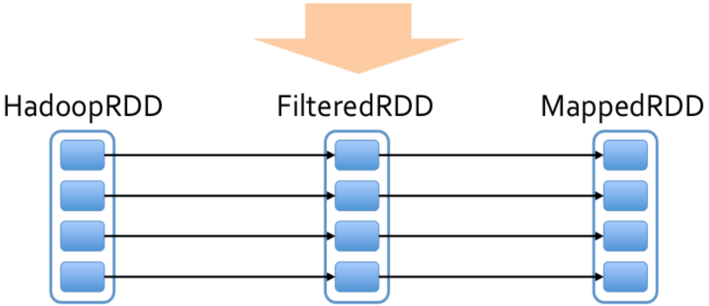


Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

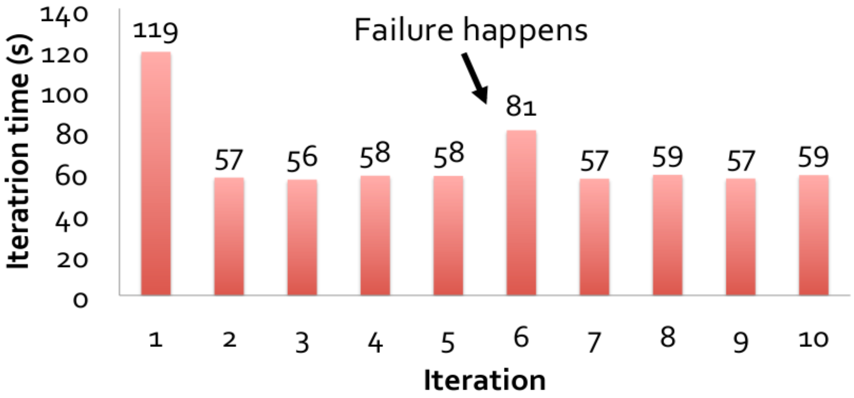
Fault Recovery

- 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))
```



Fault Recovery Results



Example: Page Rank

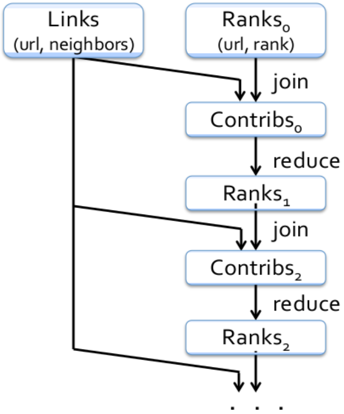
1. Start each page with a rank of 1
2. On each iteration, update each page's rank to

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

```
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 Spark



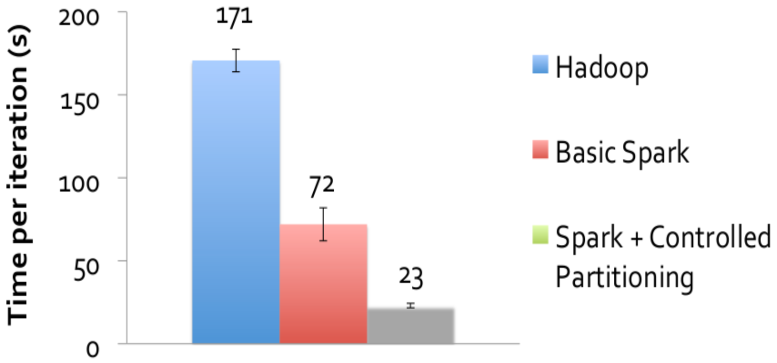
`links` & `ranks` repeatedly joined

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())
```

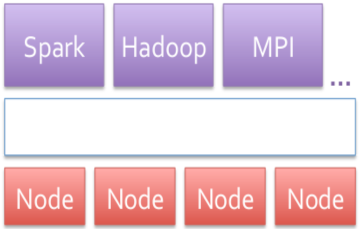
PageRank Performance



Implementation Spark

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

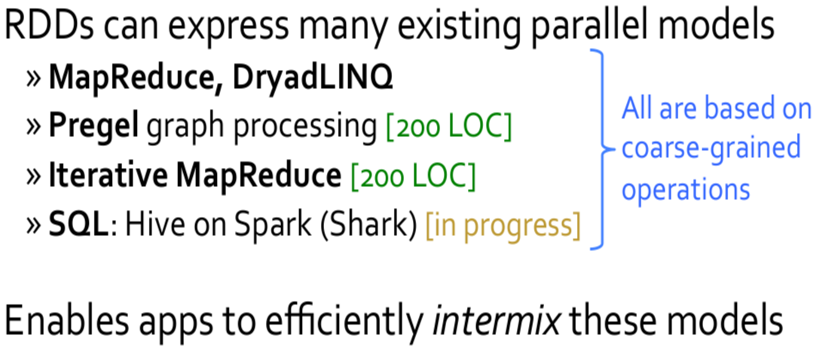
Can read from any Hadoop input source (HDFS, S3, ...)



No changes to Scala language or compiler

» Reflection + bytecode analysis to correctly ship code

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

Open Source Community

15 contributors, **5+** companies using Spark,
3+ applications projects at Berkeley

User applications:

» Data mining 40x faster than Hadoop (Conviva)
» Exploratory log analysis (Foursquare)
» Traffic prediction via EM (Mobile Millennium)
» Twitter spam classification (Monarch)
» DNA sequence analysis (SNAP)
» . . .

Related works

RAMCloud, Piccolo, GraphLab, parallel DBs

» Fine-grained writes requiring replication for resilience

Pregel, iterative MapReduce

» Specialized models; can't run arbitrary / ad-hoc queries

DryadLINQ, FlumeJava

» Language-integrated "distributed dataset" API, but cannot share datasets efficiently *across* queries

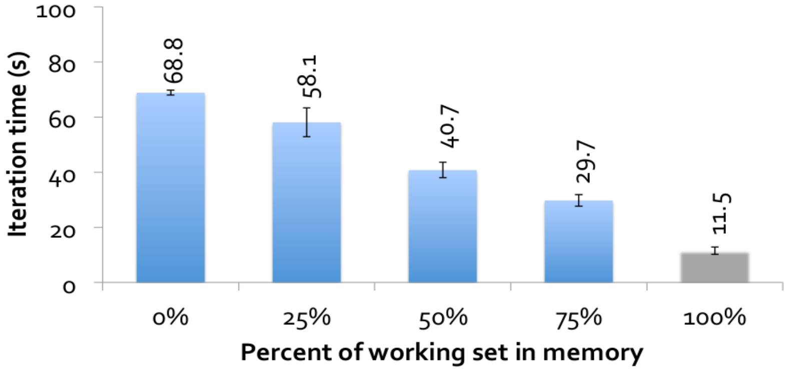
Nectar [OSDI 10]

» Automatic expression caching, but over distributed FS

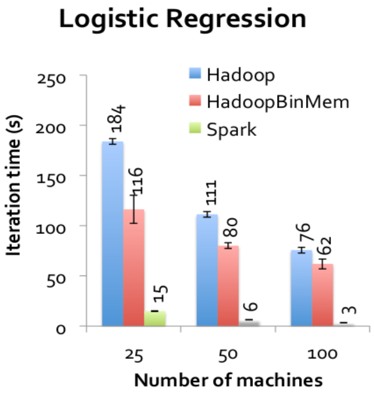
PacMan [NSDI 12]

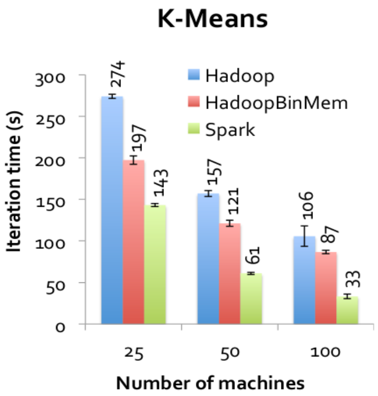
» Memory cache for HDFS, but writes still go to network/disk

Behavior with Insufficient RAM

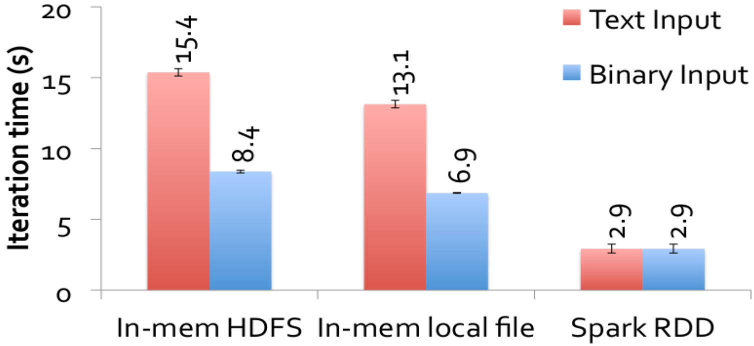


Scalability

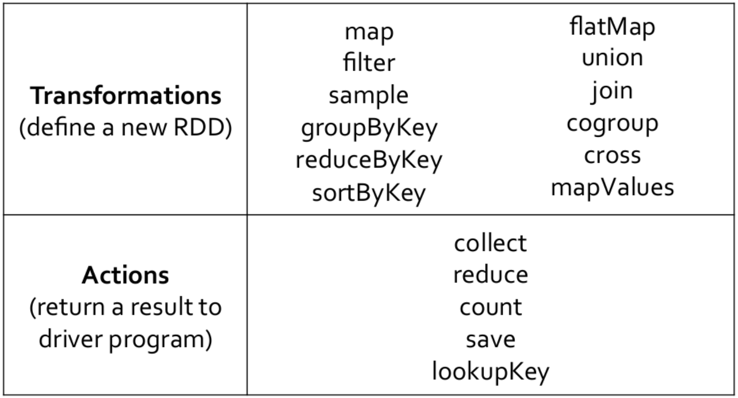




Breaking Down the Speedup



Spark Operations



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

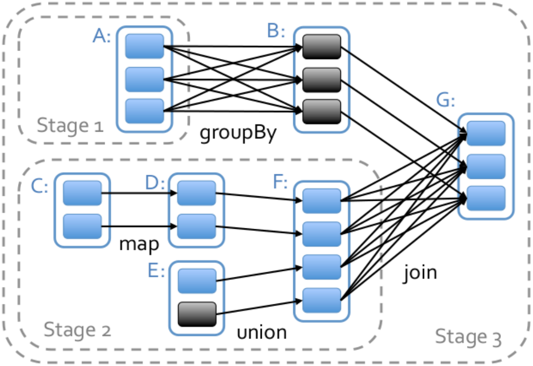
Task Scheduler

Dryad-like DAGs

Pipelines functions within a stage

Locality & data reuse aware

Partitioning-aware to avoid shuffles



■ = cached data partition

Apache Spark

- RDDs can only be created through reading data from stable stroage or Spark operations over existing RDDs.
- Spark defines two kinds of RDD operations:
  - **transformations** which apply the same operation on every record of the RDD to generate a separate, new RDD. Map()
  - **actions** which aggregate the RDD to generate computation result. Reduce()(runs on the all nodes parallel)
- In our implementation, each iteration consists of two map operations and two reduce operations.
- Like Hadoop, the efficiency of Spark highly depends on the parallelism of the algorithm itself.

Apache Spark

- A second abstraction in Spark is **shared variables** that can be used in parallel operations.
- When Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task
- Spark supports two types of shared variables: **broadcast variables**, which can be used to cache a value in memory on all nodes, and **accumulators**, which are variables that are only added to, such as counters and sums.

Apache Spark

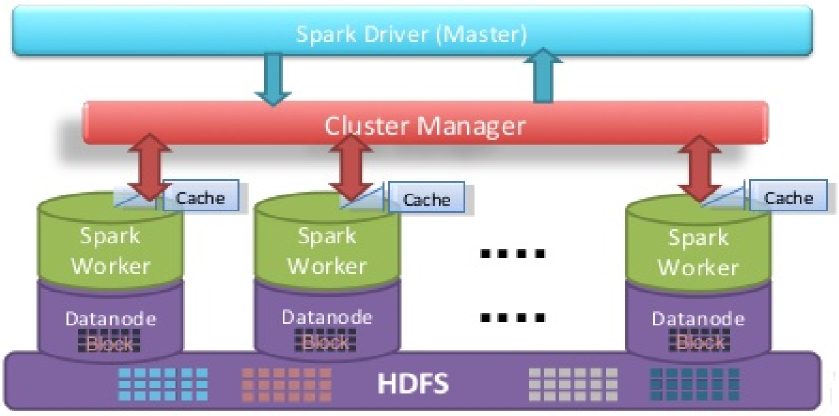


Figure : Spark Architecture

Spark Engine

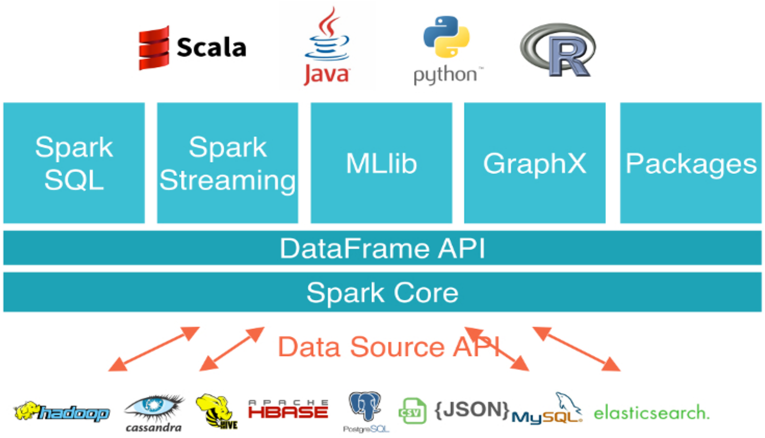
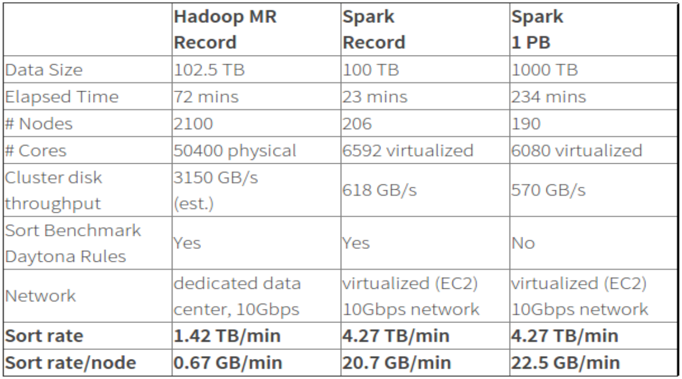


Figure : Spark Engine

Approaching with Apache Spark

- Approaching with Apache Spark



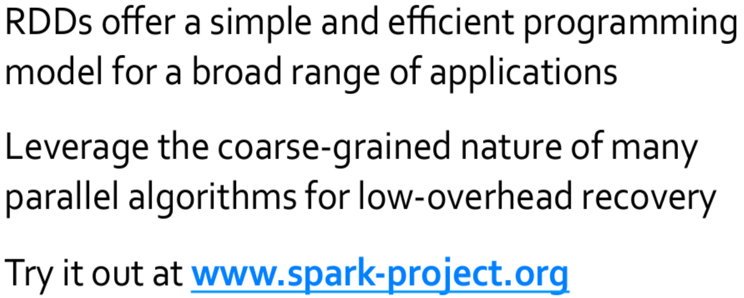
	Hadoop MR Record	**Spark Record**	**Spark 1 PB**
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

- http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html

Approaching with Apache Spark

- val input = sc.textFile("log.txt")
- val splitedLines = input.map(line => line.split(" ")).map(words => (words(0), 1)).reduceByKey(a, b) => $a + b$

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

Try it out at www.spark-project.org

Thank You