Big Data Integration and Processing

05/2023
Thanh-Chung Dao Ph.D.

Lecture Agenda

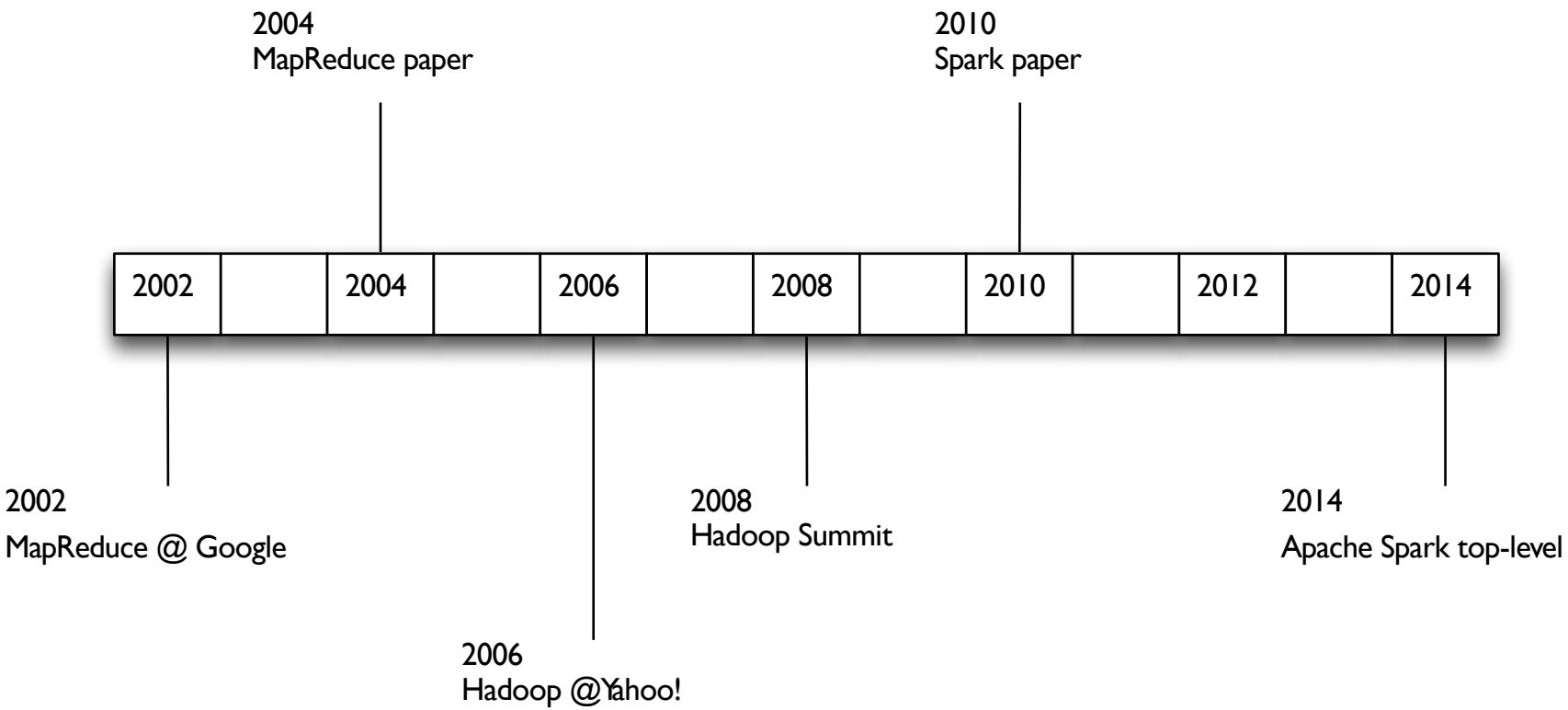
- P1: Spark introduction + Lab
- P2: Spark RDD + Lab
- P3: Spark Machine Learning + Lab
- P4: Spark Streaming, GraphX

Today's Agenda

- History of Spark
- Introduction
- Components of Stack
- Resilient Distributed Dataset – RDD

HISTORY OF SPARK

History of Spark



History of Spark

circa 1979 – **Stanford**, **MIT**, **CMU**, etc.
set/list operations in LISP, Prolog, etc., for parallel processing
www-formal.stanford.edu/ jmc/ history / lisp/ lisp.htm

circa 2004 – **Google**
MapReduce: Simplified Data Processing on Large Clusters Jeffrey Dean and Sanjay Ghemawat
research.google.com/ archive/ mapreduce.html

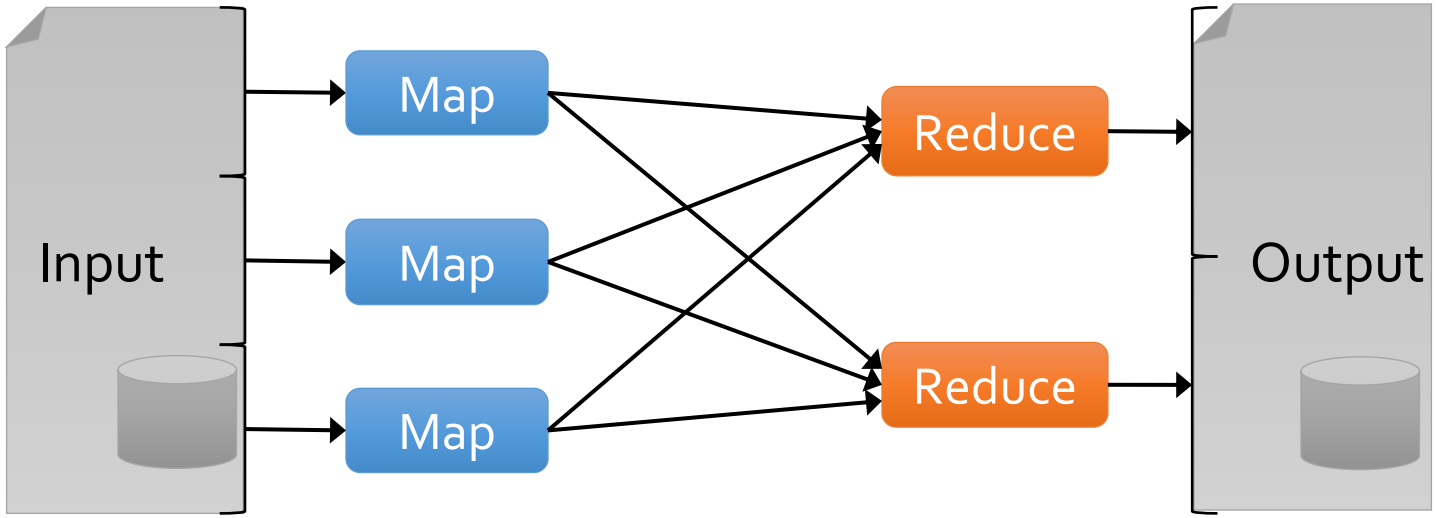
circa 2006 – **Apache**
Hadoop, originating from the Nutch Project Doug Cutting
research.yahoo.com/ files/ cutting.pdf

circa 2008 – **Yahoo**
web scale search indexing Hadoop Submit, HUG, etc.
developer.yahoo.com/hadoop/

circa 2009 – **Amazon AWS**
Elastic MapReduce
Hadoop modified for EC2/S3, plus support for Hive, Pig, Cascading, etc.
aws.amazon.com/ elasticmapreduce/

MapReduce

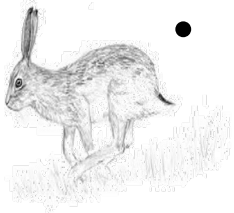
Most current cluster programming models are based on *acyclic data flow* from stable storage to stable storage



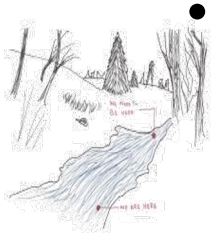
MapReduce

- Acyclic data flow is inefficient for applications that repeatedly reuse a *working set* of data:
  - **Iterative** algorithms (machine learning, graphs)
  - **Interactive** data mining tools (R, Excel, Python)

Data Processing Goals



- **Low latency (interactive) queries on historical data:** enable faster decisions
  - E.g., identify why a site is slow and fix it

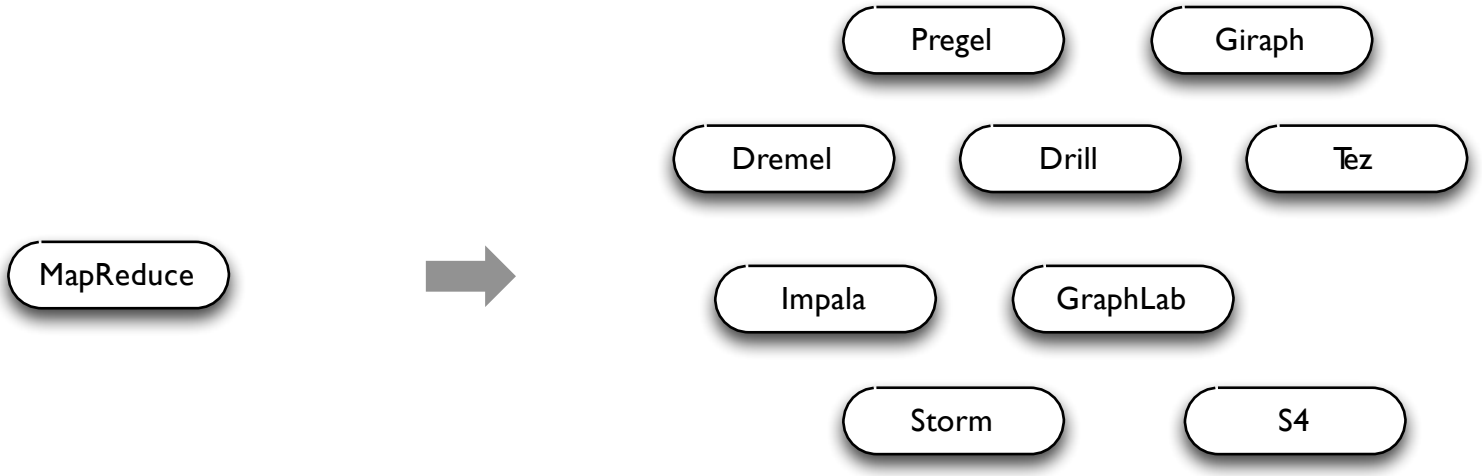


- **Low latency queries on live data (streaming):** enable decisions on real-time data
  - E.g., detect & block worms in real-time (a worm may infect **1mil** hosts in **1.3sec**)

- **Sophisticated data processing:** enable "better" decisions
  - E.g., anomaly detection, trend analysis

Therefore, people built specialized systems as workarounds…

Specialized Systems



General Batch Processing

Specialized Systems:
iterative, interactive, streaming, graph, etc.

The State of Spark, and Where We're Going Next
Matei Zaharia
Spark Summit (2013)
youtu.be/ nU6vO2EJAb4

Storage vs Processing Wars

NoSQL battles

Relational vs NoSQL

HBase vs Cassanrdra

Redis vs Memcached vs Riak

MongoDB vs CouchDB vs Couchbase

Neo4j vs Titan vs Giraph vs OrientDB

Solr vs Elasticsearch

Compute battles

MapReduce vs Spark

Spark Streaming vs Storm

Hive vs Spark SQL vs Impala

Mahout vs MLlib vs H20

Storage vs Processing Wars

NoSQL battles

Relational vs NoSQL

HBase vs Cassanrdra

Redis vs Memcached vs Riak

MongoDB vs CouchDB vs Couchbase

Neo4j vs Titan vs Giraph vs OrientDB

Solr vs Elasticsearch

Compute battles

MapReduce vs Spark

Spark Streaming vs Storm

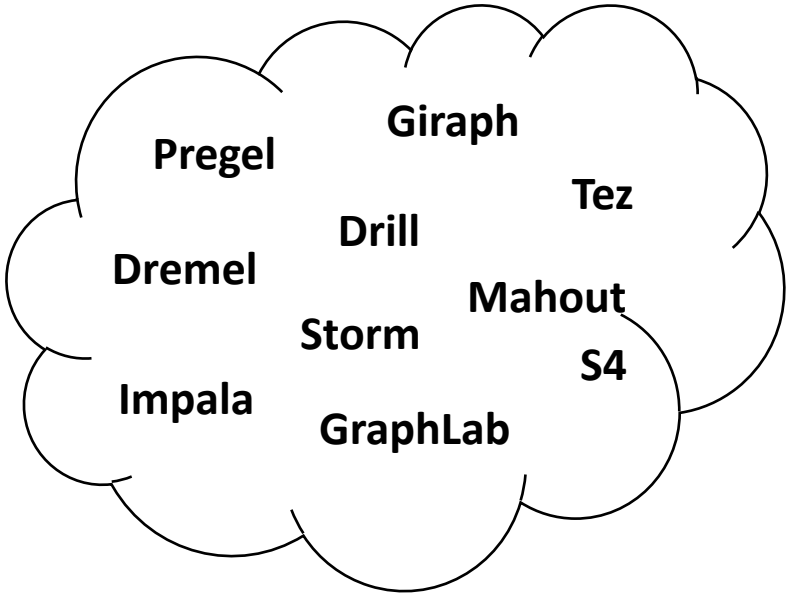
Hive vs Spark SQL vs Impala

Mahout vs MLlib vs H20

Specialized Systems

(2004 – 2013)

(2007 – 2015?)



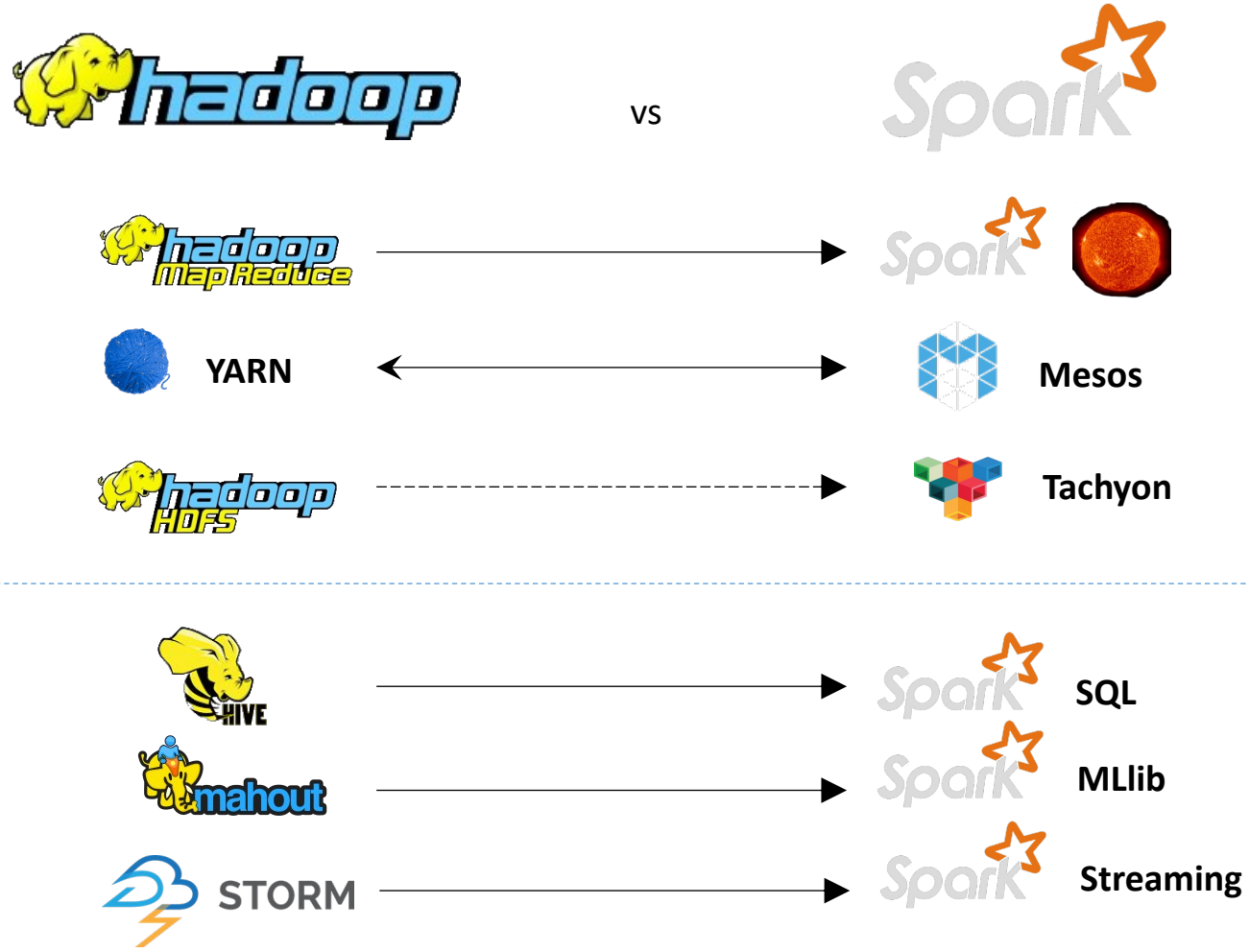
(2014 – ?)

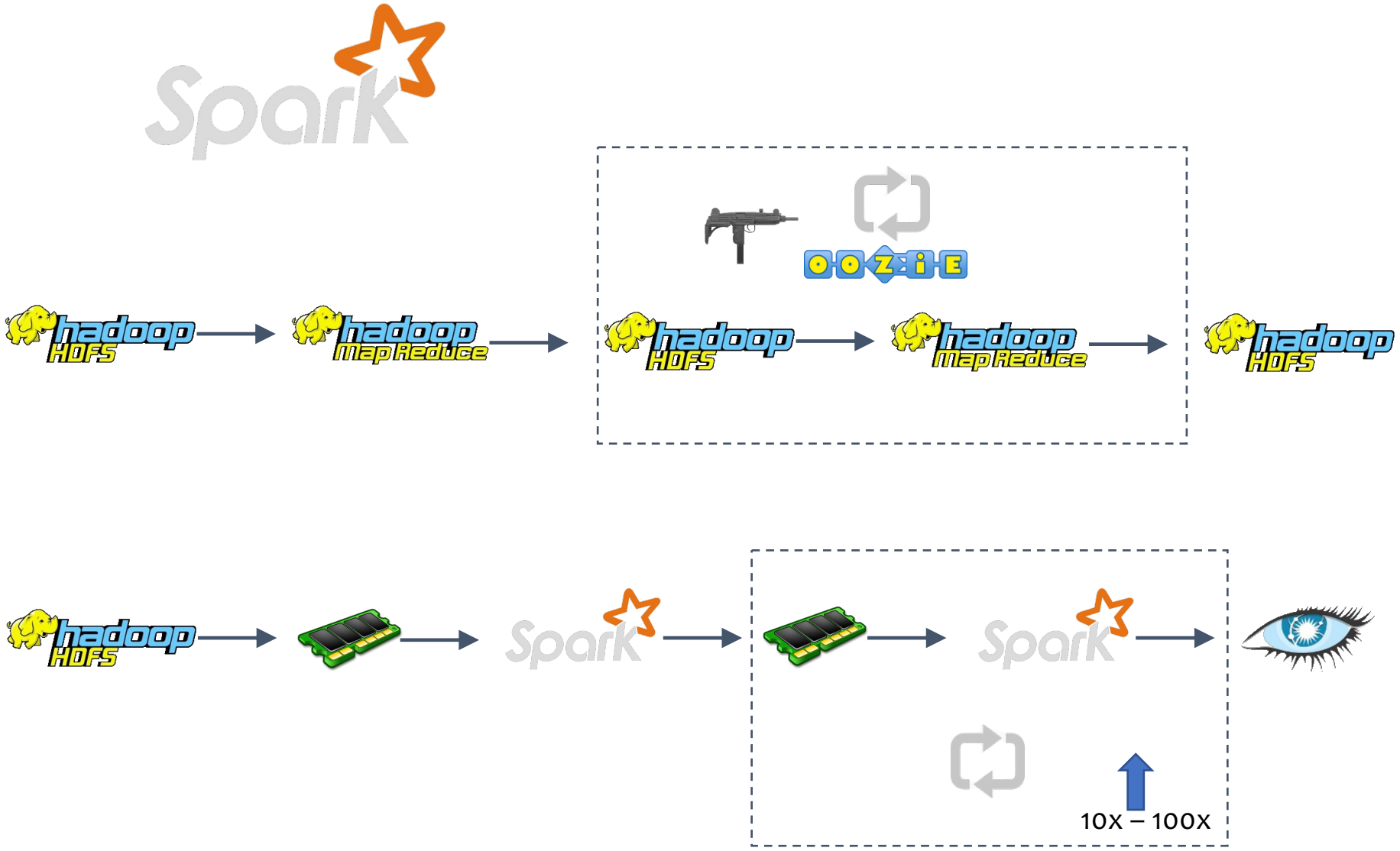
General Batch Processing

Specialized Systems

(iterative, interactive, ML, streaming, graph, SQL, etc)

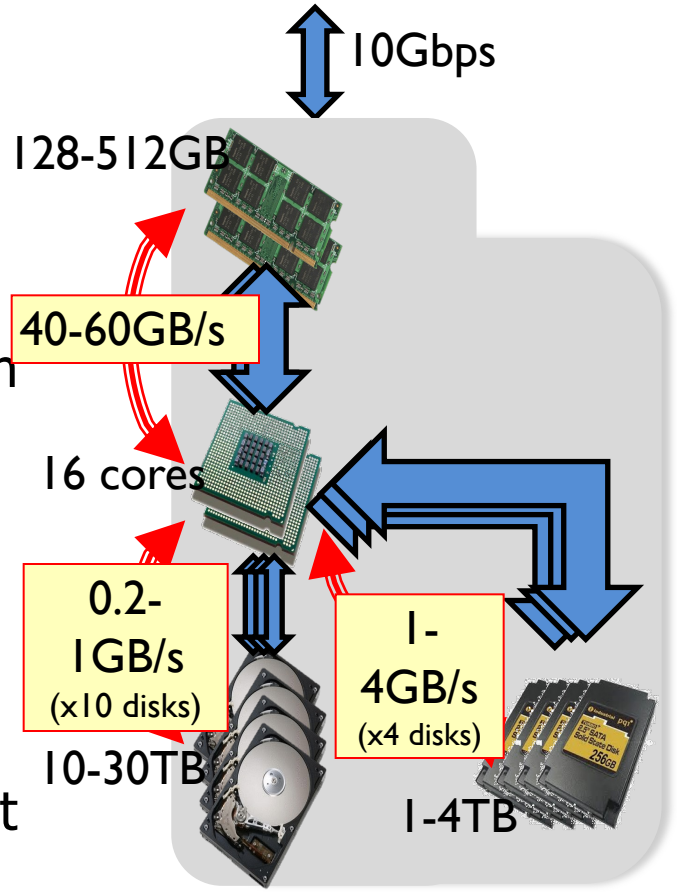
General Unified Engine


hadoop
vs
Spark
hadoop MapReduce
Spark
YARN
Mesos
hadoop HDFS
Tachyon
HIVE
SQL
mahout
MLlib
STORM
Streaming


Spark
hadoop HDFS
hadoop MapReduce
OOZIE
hadoop HDFS
hadoop MapReduce
hadoop HDFS
hadoop HDFS
Spark
Spark
10x – 100x

Support Interactive and Streaming Comp.

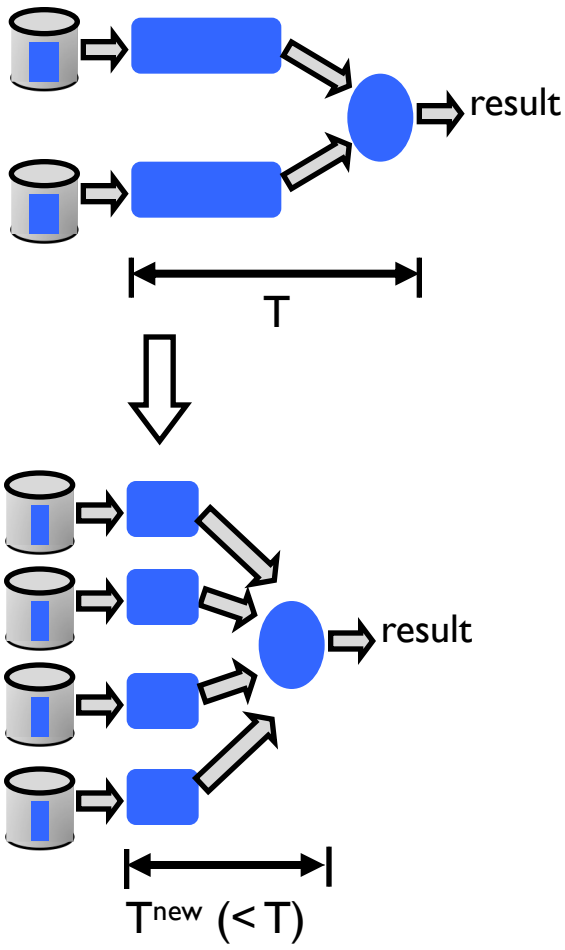
- Aggressive use of ***memory***
- Why?
  1. Memory transfer rates >> disk or SSDs
  2. Many datasets already fit into memory
     - Inputs of over 90% of jobs in Facebook, Yahoo!, and Bing clusters fit into memory
     - e.g., 1TB = 1 billion records @ 1KB each
  3. Memory density (still) grows with Moore's law
     - RAM/SSD hybrid memories at horizon



High end datacenter node

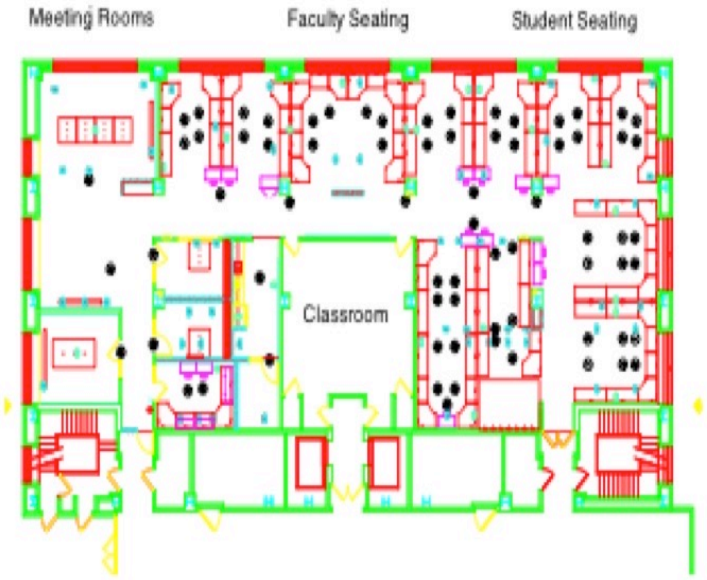
Support Interactive and Streaming Comp.

- Increase ***parallelism***
- Why?
  - Reduce work per node → improve latency
- Techniques:
  - Low latency parallel scheduler that achieve high locality
  - Optimized parallel communication patterns (e.g., shuffle, broadcast)
  - Efficient recovery from failures and straggler mitigation



Berkeley AMPLab

- "Launched" January 2011: 6 Year Plan
- 8 CS Faculty
- ~40 students
- 3 software engineers
- Organized for collaboration:



Berkeley AMPLab

- Funding:
  - 

XData,


CISE Expedition Grant
  - Industrial, founding sponsors
  - 18 other sponsors, including

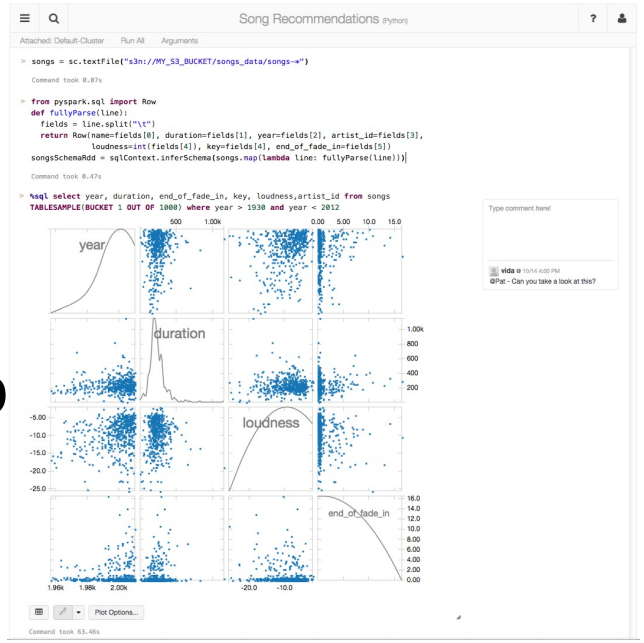
Goal: Next Generation of Analytics Data Stack for Industry & Research:

- Berkeley Data Analytics Stack (BDAS)
- Release as Open Source

Databricks

making big data simple

- Founded in late 2013
- by the creators of Apache Spark
- Original team from UC Berkeley AMPLab
- Raised $47 Million in 2 rounds

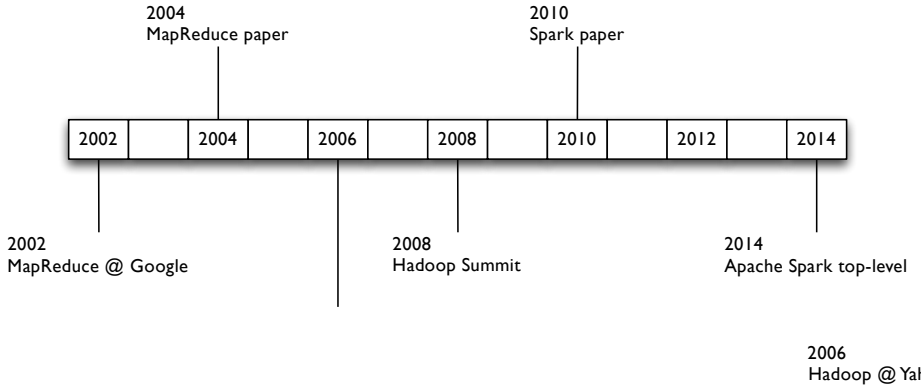


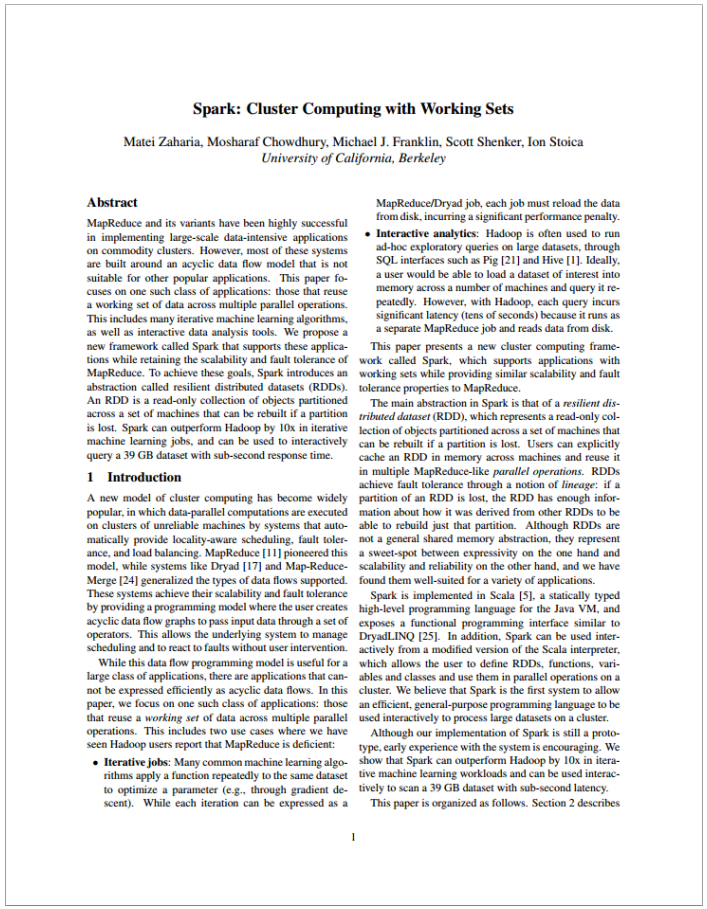
Databricks Cloud:

"A unified platform for building Big Data pipelines – from ETL to Exploration and Dashboards, to Advanced Analytics and Data Products."

The Databricks team contributed more than **75%** of the code added to Spark in the 2014

History of Spark





Spark: Cluster Computing with Working Sets

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters. However, most of these systems are built around an acyclic data flow model that is not suitable for other popular applications. This paper focuses on one such class of applications: those that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. We propose a new framework called Spark that supports these applications while retaining the scalability and fault tolerance of MapReduce. To achieve these goals, Spark introduces an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time.

1 Introduction

A new model of cluster computing has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing. MapReduce [11] pioneered this model, while systems like Dryad [17] and Map-Reduce-Merge [24] generalized the types of data flows supported. These systems achieve their scalability and fault tolerance by providing a programming model where the user creates acyclic data flow graphs to pass input data through a set of operators. This allows the underlying system to manage scheduling and to react to faults without user intervention.

While this data flow programming model is useful for a large class of applications, there are applications that cannot be expressed efficiently as acyclic data flows. In this paper, we focus on one such class of applications: those that reuse a *working set* of data across multiple parallel operations. This includes two use cases where we have seen Hadoop users report that MapReduce is deficient:

- **Iterative jobs**: Many common machine learning algorithms apply a function repeatedly to the same dataset to optimize a parameter (e.g., through gradient descent). While each iteration can be expressed as a MapReduce/Dryad job, each job must reload the data from disk, incurring a significant performance penalty.
- **Interactive analytics**: Hadoop is often used to run ad-hoc exploratory queries on large datasets, through SQL interfaces such as Pig [21] and Hive [1]. Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk.

This paper presents a new cluster computing framework called Spark, which supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce.

The main abstraction in Spark is that of a *resilient distributed dataset* (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like *parallel operations*. RDDs achieve fault tolerance through a notion of *lineage*: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity on the one hand and scalability and reliability on the other hand, and we have found them well-suited for a variety of applications.

Spark is implemented in Scala [5], a statically typed high-level programming language for the Java VM, and exposes a functional programming interface similar to DryadLINQ [25]. In addition, Spark can be used interactively from a modified version of the Scala interpreter, which allows the user to define RDDs, functions, variables and classes and use them in parallel operations on a cluster. We believe that Spark is the first system to allow an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster.

Although our implementation of Spark is still a prototype, early experience with the system is encouraging. We show that Spark can outperform Hadoop by 10x in iterative machine learning workloads and can be used interactively to scan a 39 GB dataset with sub-second latency.

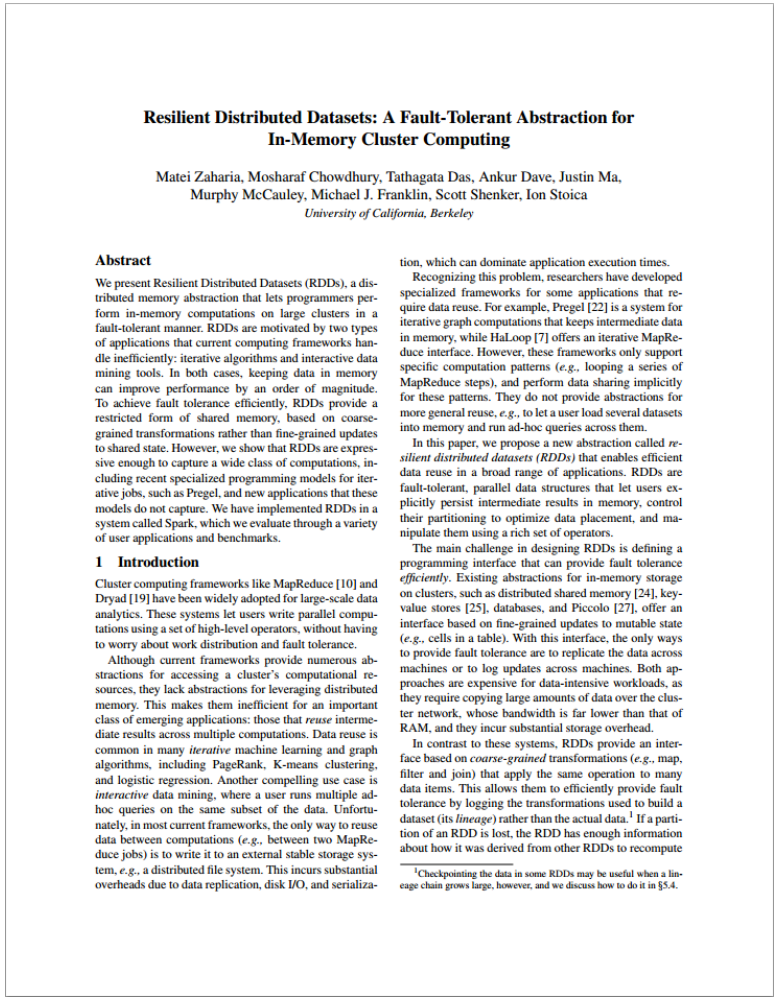
This paper is organized as follows. Section 2 describes

1

Spark: Cluster Computing with Working Sets
Matei Zaharia, Mosharaf Chowdhury,
Michael J. Franklin, Scott Shenker, Ion Stoica
USENIX HotCloud (2010)
people.csail.mit.edu/ matei/ papers/ 2010/ hotcloud_spark.pdf

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 NSDI (2012)
usenix.org/ system/ files/ conference/ nsdi12/ nsdi12-final138.pdf

History of Spark



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

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (*e.g.*, between two MapReduce jobs) is to write it to an external stable storage system, *e.g.*, a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.*, looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.*, to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.*, cells in a table). 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, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.*, map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

[1]Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4.

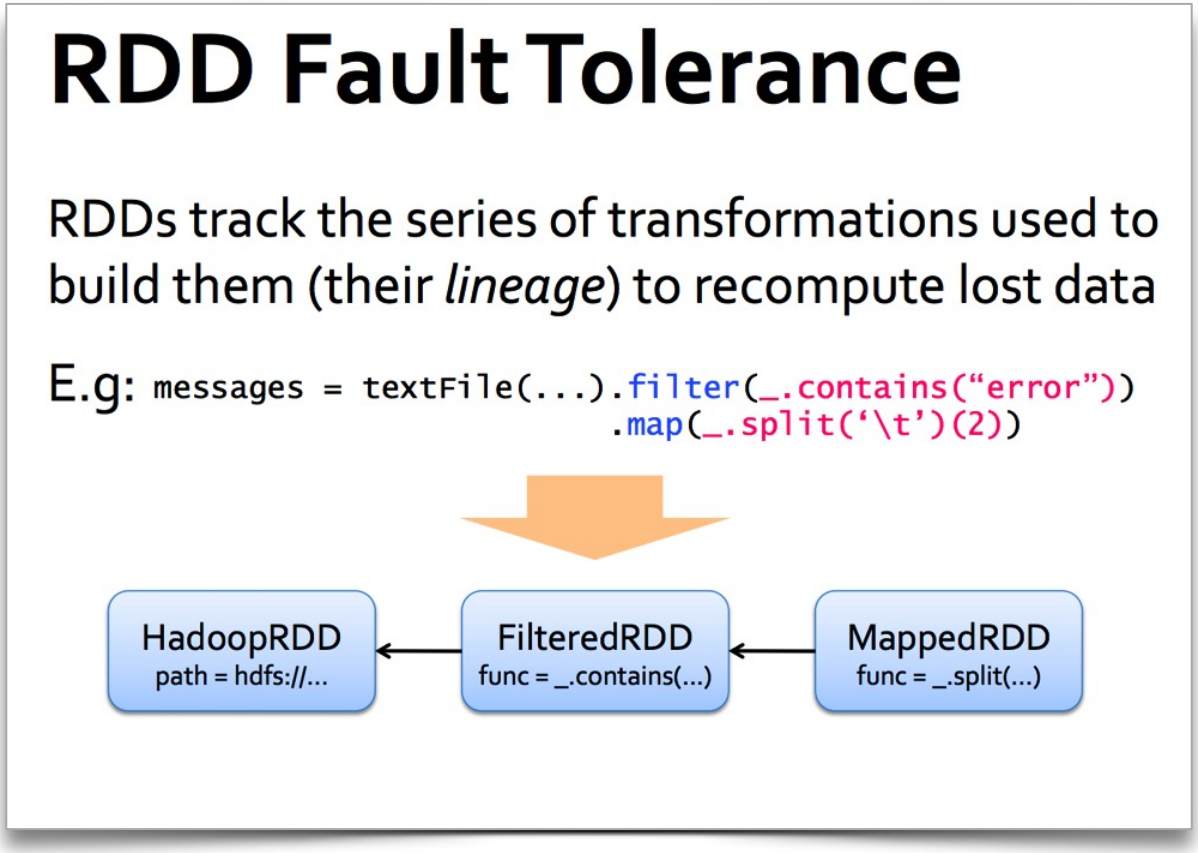
April 2012

"We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.

RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools.

In both cases, keeping data in memory can improve performance by an order of magnitude."

History of Spark

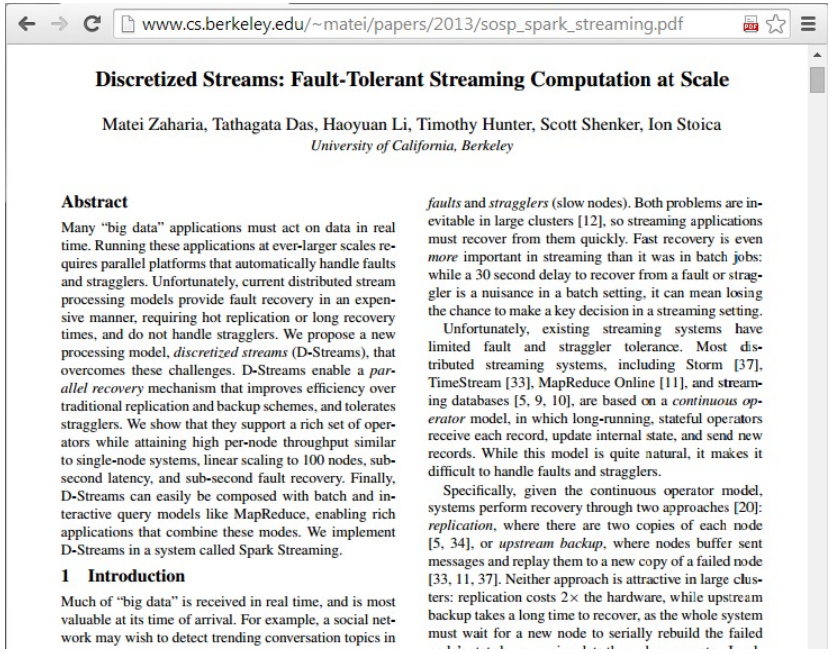


The State of Spark, and Where We're Going Next
Matei Zaharia
Spark Summit (2013)
youtu.be/ nU6vO2EJAb4

History of Spark

STREAMING

Analyze real time streams of data in ½ second intervals



www.cs.berkeley.edu/~matei/papers/2013/sosp_spark_streaming.pdf

Discretized Streams: Fault-Tolerant Streaming Computation at Scale

Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

Many "big data" applications must act on data in real time. Running these applications at ever-larger scales requires parallel platforms that automatically handle faults and stragglers. Unfortunately, current distributed stream processing models provide fault recovery in an expensive manner, requiring hot replication or long recovery times, and do not handle stragglers. We propose a new processing model, *discretized streams* (D-Streams), that overcomes these challenges. D-Streams enable a *parallel recovery* mechanism that improves efficiency over traditional replication and backup schemes, and tolerates stragglers. We show that they support a rich set of operators while attaining high per-node throughput similar to single-node systems, linear scaling to 100 nodes, sub-second latency, and sub-second fault recovery. Finally, D-Streams can easily be composed with batch and interactive query models like MapReduce, enabling rich applications that combine these modes. We implement D-Streams in a system called Spark Streaming.

1 Introduction

Much of "big data" is received in real time, and is most valuable at its time of arrival. For example, a social network may wish to detect trending conversation topics in *faults* and *stragglers* (slow nodes). Both problems are inevitable in large clusters [12], so streaming applications must recover from them quickly. Fast recovery is even *more* important in streaming than it was in batch jobs: while a 30 second delay to recover from a fault or straggler is a nuisance in a batch setting, it can mean losing the chance to make a key decision in a streaming setting.

Unfortunately, existing streaming systems have limited fault and straggler tolerance. Most distributed streaming systems, including Storm [37], TimeStream [33], MapReduce Online [11], and streaming databases [5, 9, 10], are based on a *continuous operator* model, in which long-running, stateful operators receive each record, update internal state, and send new records. While this model is quite natural, it makes it difficult to handle faults and stragglers.

Specifically, given the continuous operator model, systems perform recovery through two approaches [20]: *replication*, where there are two copies of each node [5, 34], or *upstream backup*, where nodes buffer sent messages and replay them to a new copy of a failed node [33, 11, 37]. Neither approach is attractive in large clusters: replication costs 2× the hardware, while upstream backup takes a long time to recover, as the whole system must wait for a new node to serially rebuild the failed

```
TwitterUtils.createStream(...)
    .filter(_.getText.contains("Spark"))
    .countByWindow(Seconds(5))
```

History of Spark

SQL

Seemlessly mix SQL queries with Spark programs.

Spark SQL: Relational Data Processing in Spark

Michael Armbrust[†], Reynold S. Xin[†], Cheng Lian[†], Yin Huai[†], Davies Liu[†], Joseph K. Bradley[†], Xiangrui Meng[†], Tomer Kaftan[‡], Michael J. Franklin[†‡], Ali Ghodsi[†], Matei Zaharia[†*]

[†]Databricks Inc. *MIT CSAIL [‡]AMPLab, UC Berkeley

ABSTRACT

Spark SQL is a new module in Apache Spark that integrates relational processing with Spark's functional programming API. Built on our experience with Shark, Spark SQL lets Spark programmers leverage the benefits of relational processing (*e.g.,* declarative queries and optimized storage), and lets SQL users call complex analytics libraries in Spark (*e.g.,* machine learning). Compared to previous systems, Spark SQL makes two main additions. First, it offers much tighter integration between relational and procedural processing, through a declarative DataFrame API that integrates with procedural Spark code. Second, it includes a highly extensible optimizer, Catalyst, built using features of the Scala programming language, that makes it easy to add composable rules, control code generation, and define extension points. Using Catalyst, we have built a variety of features (*e.g.,* schema inference for JSON, machine learning types, and query federation to external databases) tailored for the complex needs of modern data analysis. We see Spark SQL as an evolution of both SQL-on-Spark and of Spark itself, offering richer APIs and optimizations while keeping the benefits of the Spark programming model.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

Keywords

Databases; Data Warehouse; Machine Learning; Spark; Hadoop

1 Introduction

Big data applications require a mix of processing techniques, data sources and storage formats. The earliest systems designed for these workloads, such as MapReduce, gave users a powerful, but

While the popularity of relational systems shows that users often prefer writing declarative queries, the relational approach is insufficient for many big data applications. First, users want to perform ETL to and from various data sources that might be semi- or unstructured, requiring custom code. Second, users want to perform advanced analytics, such as machine learning and graph processing, that are challenging to express in relational systems. In practice, we have observed that most data pipelines would ideally be expressed with a combination of both relational queries and complex procedural algorithms. Unfortunately, these two classes of systems—relational and procedural—have until now remained largely disjoint, forcing users to choose one paradigm or the other.

This paper describes our effort to combine both models in Spark SQL, a major new component in Apache Spark [39]. Spark SQL builds on our earlier SQL-on-Spark effort, called Shark. Rather than forcing users to pick between a relational or a procedural API, however, Spark SQL lets users seamlessly intermix the two.

Spark SQL bridges the gap between the two models through two contributions. First, Spark SQL provides a *DataFrame API* that can perform relational operations on both external data sources and Spark's built-in distributed collections. This API is similar to the widely used data frame concept in R [32], but evaluates operations lazily so that it can perform relational optimizations. Second, to support the wide range of data sources and algorithms in big data, Spark SQL introduces a novel extensible optimizer called *Catalyst*. Catalyst makes it easy to add data sources, optimization rules, and data types for domains such as machine learning.

The DataFrame API offers rich relational/procedural integration within Spark programs. DataFrames are collections of structured records that can be manipulated using Spark's procedural API, or using new relational APIs that allow richer optimizations. They can

```
sqlCtx = new HiveContext(sc)
results = sqlCtx.sql(
  "SELECT * FROM people")
names = results.map(lambda p:
p.name)
```

History of Spark

GRAPHX

Analyze networks of nodes and edges using graph processing

GraphX: A Resilient Distributed Graph System on Spark

Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica

AMPLab, EECS, UC Berkeley
{rxin, jegonzal, franklin, istoica}@cs.berkeley.edu

ABSTRACT

From social networks to targeted advertising, big graphs capture the structure in data and are central to recent advances in machine learning and data mining. Unfortunately, directly applying existing data-parallel tools to graph computation tasks can be cumbersome and inefficient. The need for intuitive, scalable tools for graph computation has lead to the development of new *graph-parallel* systems (*e.g.*, Pregel, PowerGraph) which are designed to efficiently execute graph algorithms. Unfortunately, these new graph-parallel systems do not address the challenges of graph construction and transformation which are often just as problematic as the subsequent computation. Furthermore, existing graph-parallel systems provide limited fault-tolerance and support for interactive data mining.

We introduce GraphX, which combines the advantages of both data-parallel and graph-parallel systems by efficiently expressing graph computation within the Spark data-parallel framework. We leverage new ideas in distributed graph representation to efficiently distribute graphs as tabular data-structures. Similarly, we leverage advances in data-flow systems to exploit in-memory computation and fault-tolerance. We provide powerful new operations to simplify graph construction and transformation. Using these primitives we implement the PowerGraph and Pregel abstractions in less than 20 lines of code. Finally, by exploiting the Scala foundation of Spark, we enable users to interactively load, transform, and compute on massive graphs.

1. INTRODUCTION

From social networks to advertising and the web, big graphs can be found in a wide range of important applications. By modeling the and distributed systems. By abstracting away the challenges of large-scale distributed system design, these frameworks simplify the design, implementation, and application of new sophisticated graph algorithms to large-scale real-world graph problems.

While existing graph-parallel frameworks share many common properties, each presents a slightly different view of graph computation tailored to either the originating domain or a specific family of graph algorithms and applications. Unfortunately, because each framework relies on a separate runtime, it is difficult to compose these abstractions. Furthermore, while these frameworks address the challenges of graph computation, they do not address the challenges of data ETL (preprocessing and construction) or the process of interpreting and applying the results of computation. Finally, few frameworks have built-in support for interactive graph computation.

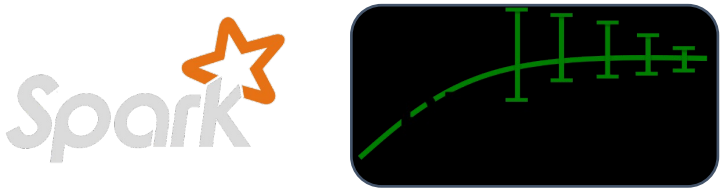
Alternatively *data-parallel* systems like MapReduce and Spark [12] are designed for scalable data processing and are well suited to the task of graph construction (ETL). By exploiting data-parallelism, these systems are highly scalable and support a range of fault-tolerance strategies. More recent systems like Spark even enable interactive data processing. However, naively expressing graph computation and graph algorithms in these data-parallel abstractions can be challenging and typically leads to complex joins and excessive data movement that does not exploit the graph structure.

To address these challenges we introduce GraphX, a graph computation system which runs in the Spark data-parallel framework. GraphX extends Spark's Resilient Distributed Dataset (RDD) abstraction to introduce the Resilient Distributed Graph (RDG), which associates records with vertices and edges in a graph and provides a collection of expressive computational primitives. Using these

```
graph = Graph(vertices, edges)
messages =
spark.textFile("hdfs://...")
graph2 =
graph.joinVertices(messages) {
  (id, vertex, msg) => ...
}
```

https://amplab.cs.berkeley.edu/wp-content/uploads/2013/05/grades-graphx_with_fonts.pdf

History of Spark



SQL queries with Bounded Errors and Bounded Response Times

BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data

Sameer Agarwal†, Barzan Mozafari°, Aurojit Panda†, Henry Milner†, Samuel Madden°, Ion Stoica*†

†*University of California, Berkeley* °*Massachusetts Institute of Technology* **Conviva Inc.*
{sameerag, apanda, henrym, istoica}@cs.berkeley.edu, {barzan, madden}@csail.mit.edu

Abstract

In this paper, we present BlinkDB, a massively parallel, approximate query engine for running interactive SQL queries on large volumes of data. BlinkDB allows users to trade-off query accuracy for response time, enabling interactive queries over massive data by running queries on data samples and presenting results annotated with meaningful error bars. To achieve this, BlinkDB uses two key ideas: (1) an adaptive optimization framework that builds and maintains a set of multi-dimensional stratified samples from original data over time, and (2) a dynamic sample selection strategy that selects an appropriately sized sample based on a query's accuracy or response time requirements. We evaluate BlinkDB against the well-known TPC-H benchmarks and a real-world analytic workload derived from Conviva Inc., a company that manages video distribution over the Internet. Our experiments on a 100 node cluster show that BlinkDB can answer queries on up to 17 TBs of data in less than 2 seconds (over 200× faster than Hive), within an error of 2-10%.

1. Introduction

Modern data analytics applications involve computing aggregates over a large number of records to *roll-up* web clicks, cessing of large amounts of data by trading result accuracy for response time and space. These techniques include sampling [10, 14], sketches [12], and on-line aggregation [15]. To illustrate the utility of such techniques, consider the following simple query that computes the average `SessionTime` over all users originating in `New York`:

```
SELECT AVG(SessionTime)
FROM Sessions
WHERE City = 'New York'
```

Suppose the `Sessions` table contains 100 million tuples for `New York`, and cannot fit in memory. In that case, the above query may take a long time to execute, since disk reads are expensive, and such a query would need multiple disk accesses to stream through all the tuples. Suppose we instead executed the same query on a sample containing only 10,000 `New York` tuples, such that the entire sample fits in memory. This would be orders of magnitude faster, while still providing an approximate result within a few percent of the actual value, an accuracy good enough for many practical purposes. Using sampling theory we could even provide confidence bounds on the accuracy of the answer [16].

Previously described approximation techniques make different trade-offs between efficiency and the generality of the

SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
WITHIN 2 SECONDS

Queries with Time Bounds

SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
ERROR 0.1 **CONFIDENCE** 95.0%

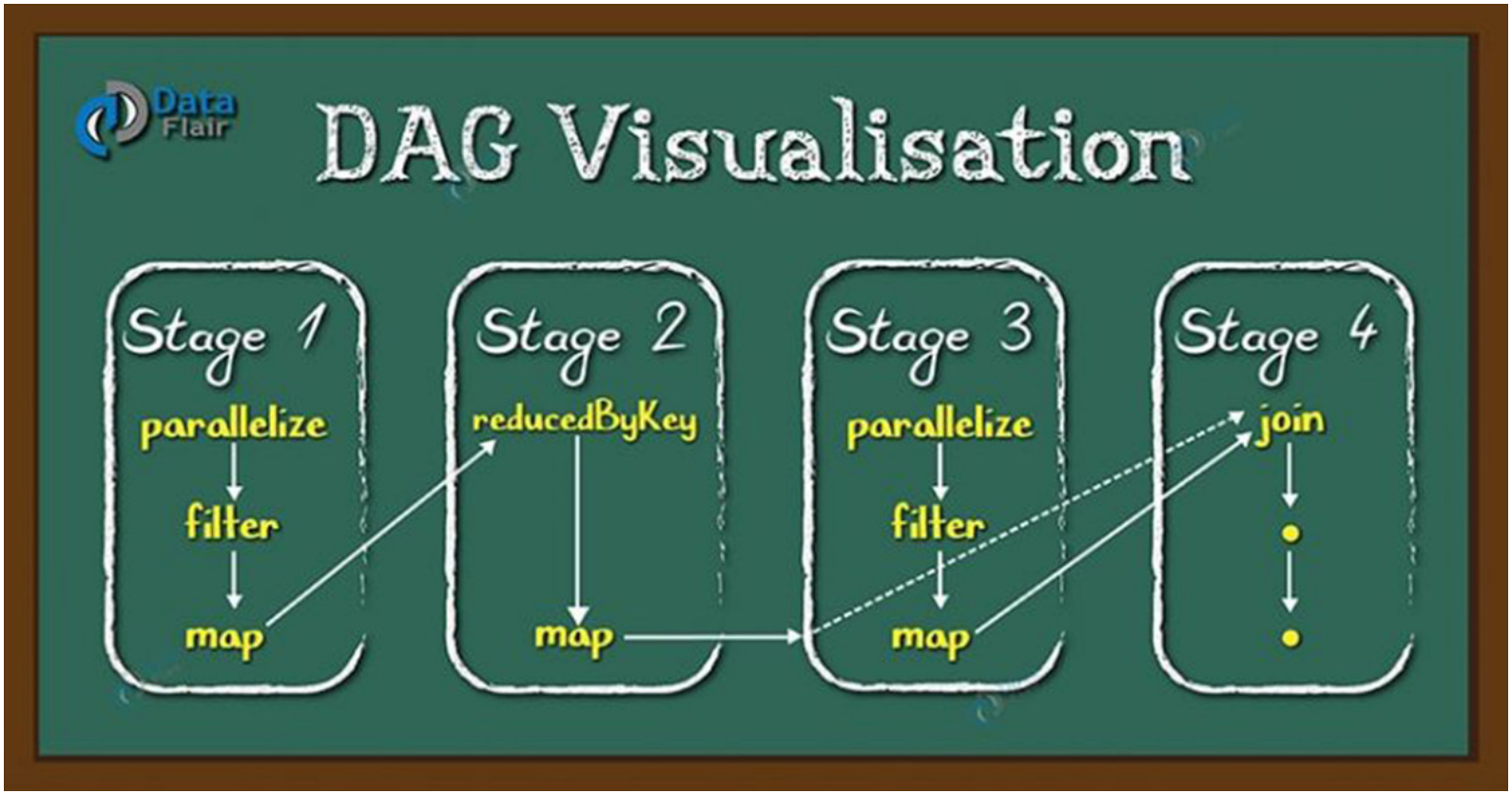
Queries with Error Bounds

https://www.cs.berkeley.edu/~sameerag/blinkdb_eurosys13.pdf

History of Spark

- Unlike the various specialized systems, Spark's goal was to *generalize* MapReduce to support new apps within same engine
- Two reasonably small additions are enough to express the previous models:
  - *fast data sharing*
  - *general DAGs*
- This allows for an approach which is more efficient for the engine, and much simpler for the end users

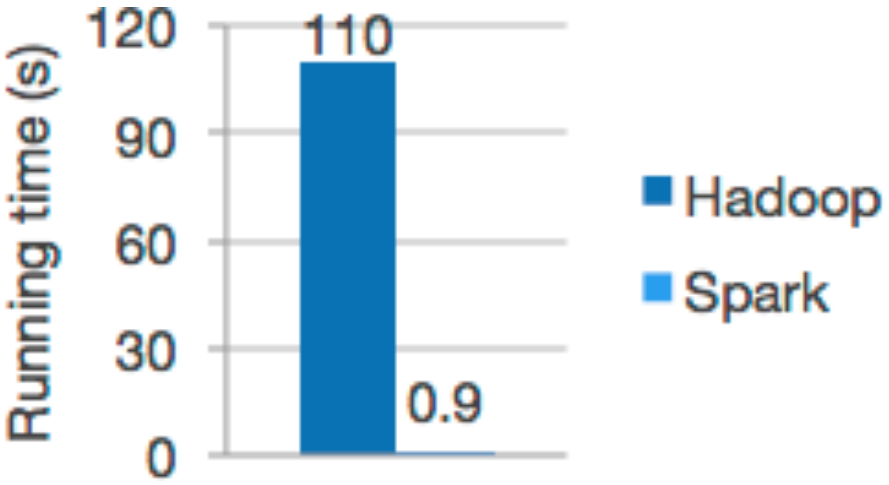
Directed Acyclic Graph - DAG



SPARK INTRODUCTION

What is Apache Spark

- Spark is a unified analytics engine for large-scale data processing
- Speed: run workloads 100x faster
  - High performance for both batch and streaming data
  - Computations run in memory



Logistic regression in Hadoop and Spark

What is Apache Spark

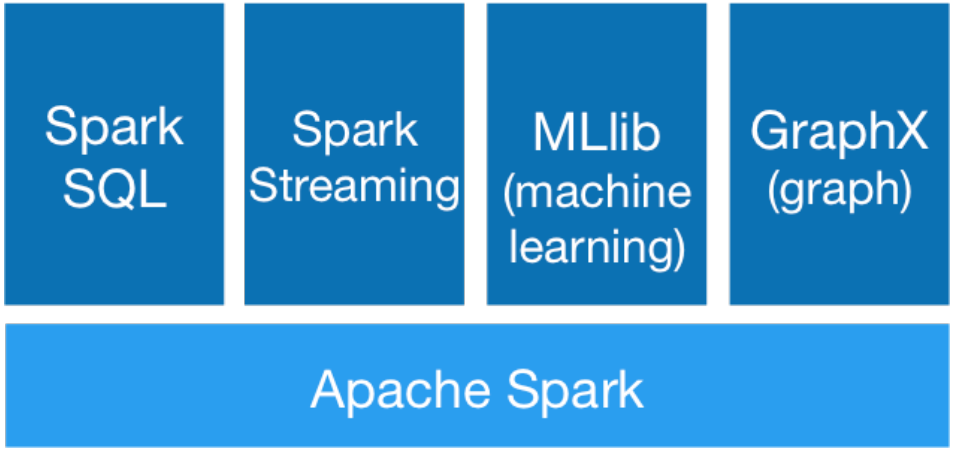
- Ease of Use: write applications quickly in Java, Scala, Python, R, SQL
  - Offer over 80 high-level operators
  - Use them interactively form Scala, Python, R, and SQL

```
df = spark.read.json("logs.json")df.
where("age > 21")
select("name.first").show()
```

Spark's Python DataFrame API
Read JSON files with automatic schema inference

What is Apache Spark

- Generality: combine SQL, Streaming, and complex analytics
  - Provide libraries including SQL and DataFrames, Spark Streaming, MLib, GraphX,
  - Wide range of workloads e.g., batch applications, interactive algorithms, interactive queries, streaming



What is Apache Spark

- Run Everywhere:
  - run on Hadoop, Apache Mesos, Kubernetes, standalone or in the cloud.
  - access data in HDFS, Aluxio, Apache Cassandra, Apache Hbase, Apache Hive, etc.

Comparison between Hadoop and Spark

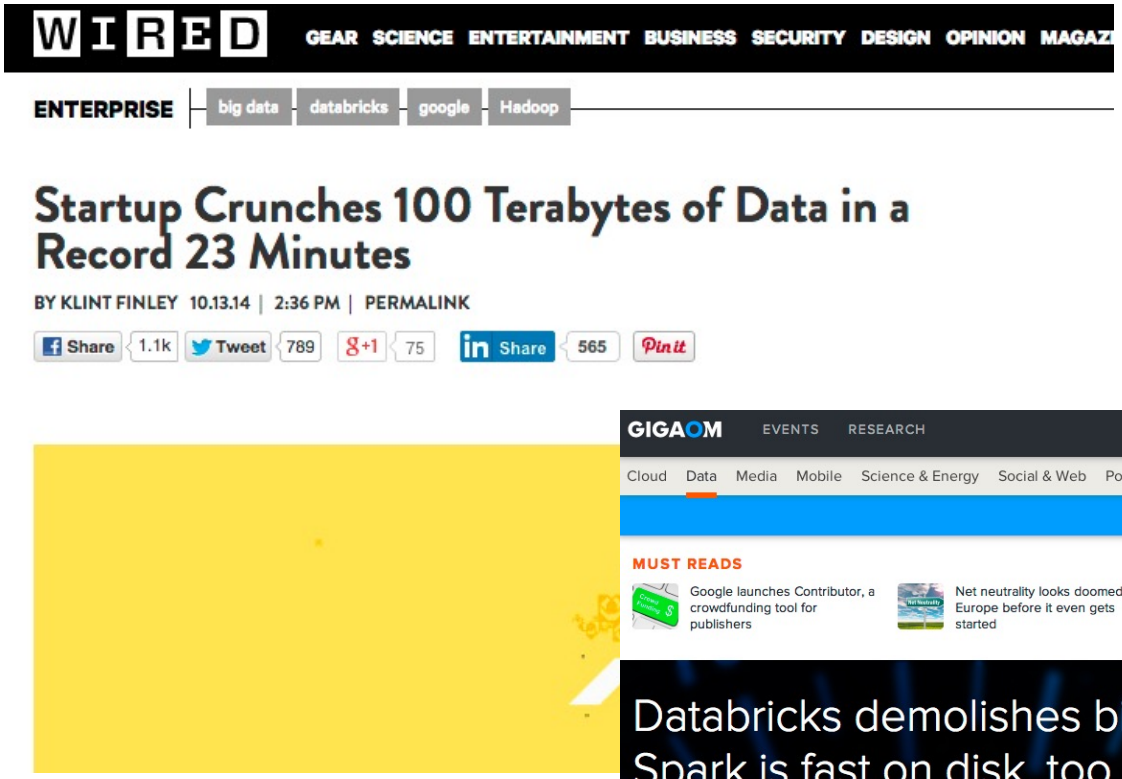
		
Strengths	▪ Can collect any data ▪ Limitless in size	▪ Can work off any Hadoop collection ▪ Runs on Hadoop, or other clusters ▪ In-memory processing makes it very fast ▪ Supports Java, Scala, Python, and R*, and can be used with SQL.
Used for	▪ Initial data ingestion ▪ Data curation ▪ Large-scale "boil the ocean" analytics ▪ Data archiving	▪ Complex query processing of large amounts of data quickly ▪ Can handle ad hoc queries
Limitations	▪ MapReduce is hard to program ▪ Disk-based batch nature limits speed, agility.	▪ Limited only by processor speed, available memory, cores, and cluster size.

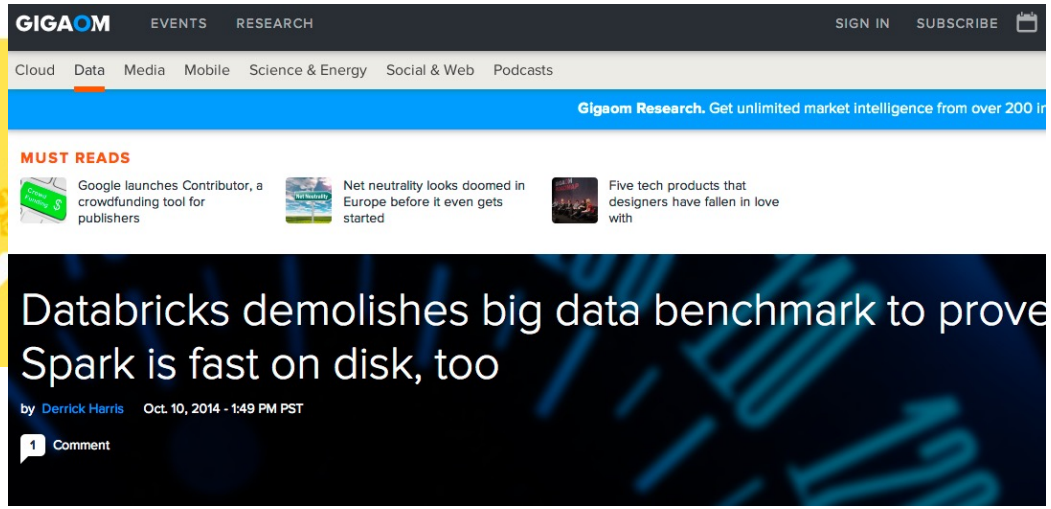
100TB Daytona Sort Competition

	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**

All the sorting took place on disk (HDFS) without using Spark's in-memory cache!

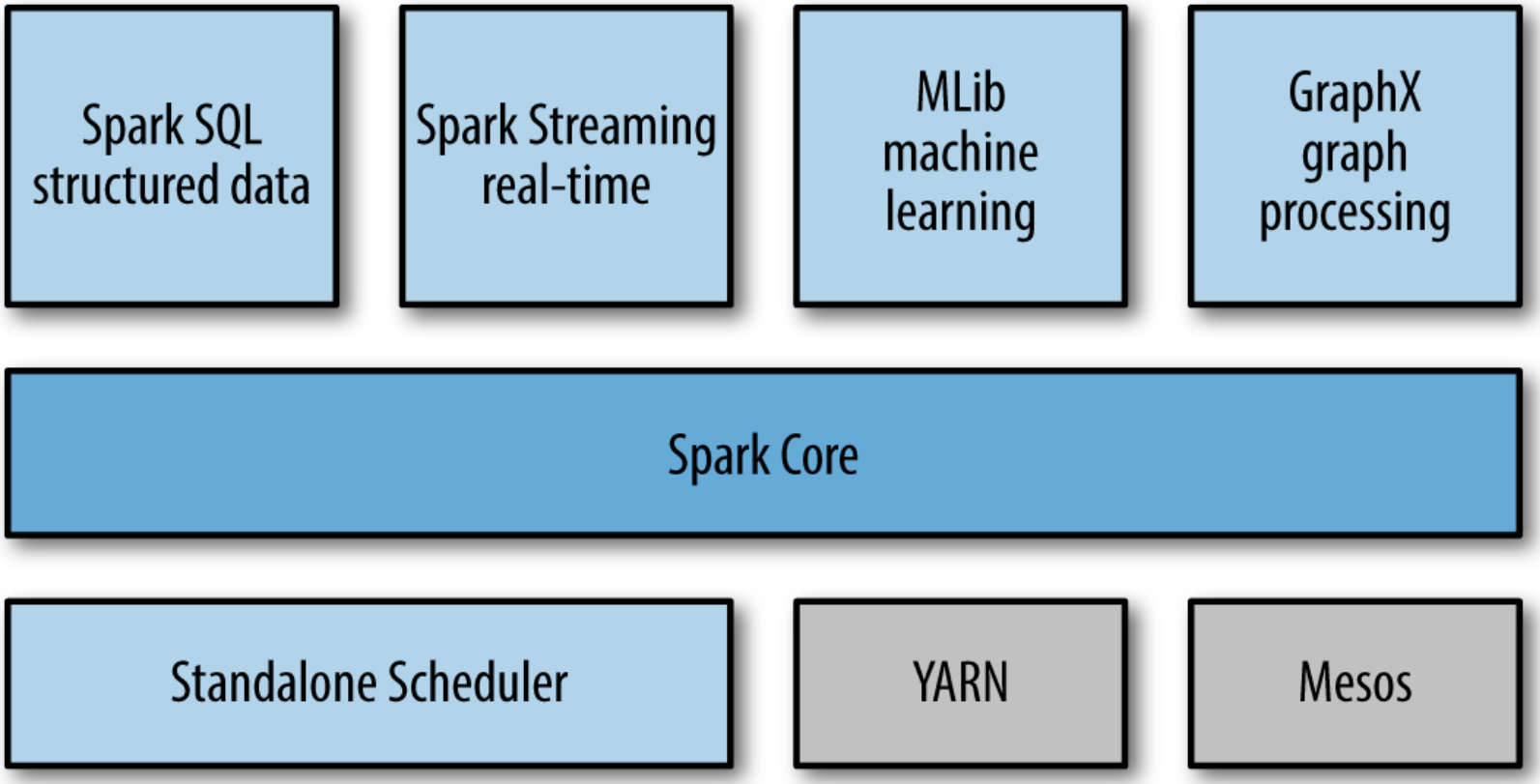
Spark sorted the same data **3X faster** using **10X fewer machines** than Hadoop MR in 2013.


WIRED
GEAR SCIENCE ENTERTAINMENT BUSINESS SECURITY DESIGN OPINION MAGAZ
ENTERPRISE
big data
databricks
google
Hadoop
Startup Crunches 100 Terabytes of Data in a Record 23 Minutes
BY KLINT FINLEY 10.13.14 | 2:36 PM | PERMALINK
Share 1.1k
Tweet 789
+1 75
Share 565
Pin it


GIGAOM
EVENTS
RESEARCH
SIGN IN
SUBSCRIBE
Cloud
Data
Media
Mobile
Science & Energy
Social & Web
Podcasts
Gigaom Research. Get unlimited market intelligence from over 200 ir
MUST READS
Google launches Contributor, a crowdfunding tool for publishers
Net neutrality looks doomed in Europe before it even gets started
Five tech products that designers have fallen in love with
Databricks demolishes big data benchmark to prove Spark is fast on disk, too
by Derrick Harris
Oct. 10, 2014 - 1:49 PM PST
1 Comment

Components of Stack

The Spark stack



The Spark stack

- Spark Core:
  - contain basic functionality of Spark including task scheduling, memory management, fault recovery, etc.
  - provide APIs for building and manipulating RDDs
- SparkSQL
  - allow querying structured data via SQL, Hive Query Language
  - allow combining SQL queries and data manipulations in Python, Java, Scala

The Spark stack

- Spark Streaming: enables processing of live streams of data via APIs
- Mlib:
  - contain common machine language functionality
  - provide multiple types of algorithms: classification, regression, clustering, etc.
- GraphX:
  - library for manipulating graphs and performing graph-parallel computations
  - extend Spark RDD API

The Spark stack

- Cluster Managers
  - Hadoop Yarn
  - Apache Mesos, and
  - Standalone Schedular (simple manager in Spark).

Resilient Distributed Dataset – RDD

- RDD Basics
- Creating RDDs
- RDD Operations
- Common Transformation and Actions
- Persistence (Caching)

RDD Basics

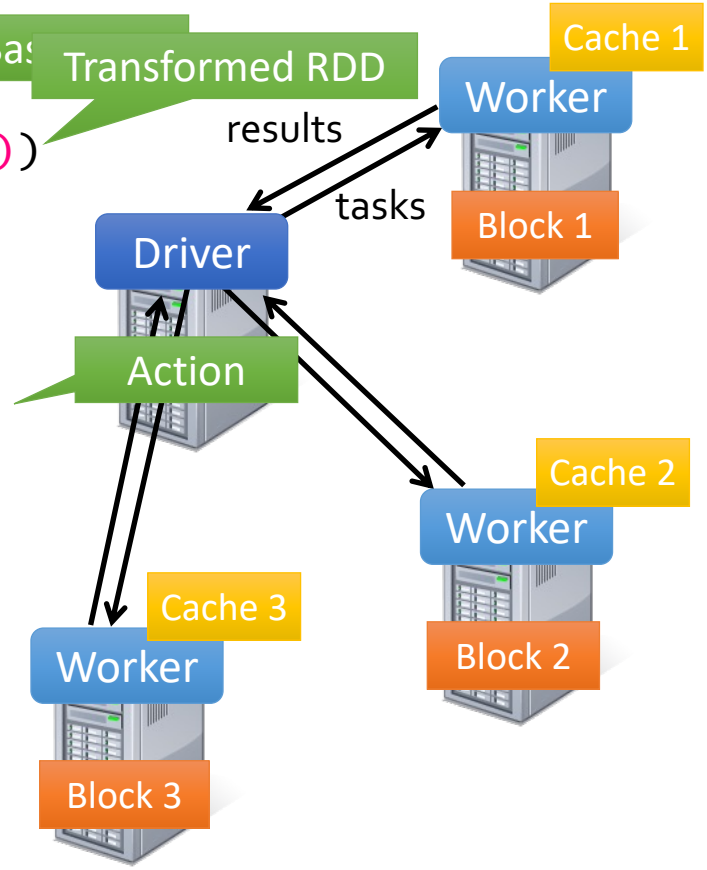
- RDD:
  - Immutable distributed collection of objects
  - Split into multiple partitions => can be computed on different nodes
- All work in Spark is expressed as
  - creating new RDDs
  - transforming existing RDDs
  - calling actions on RDDs

Example

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

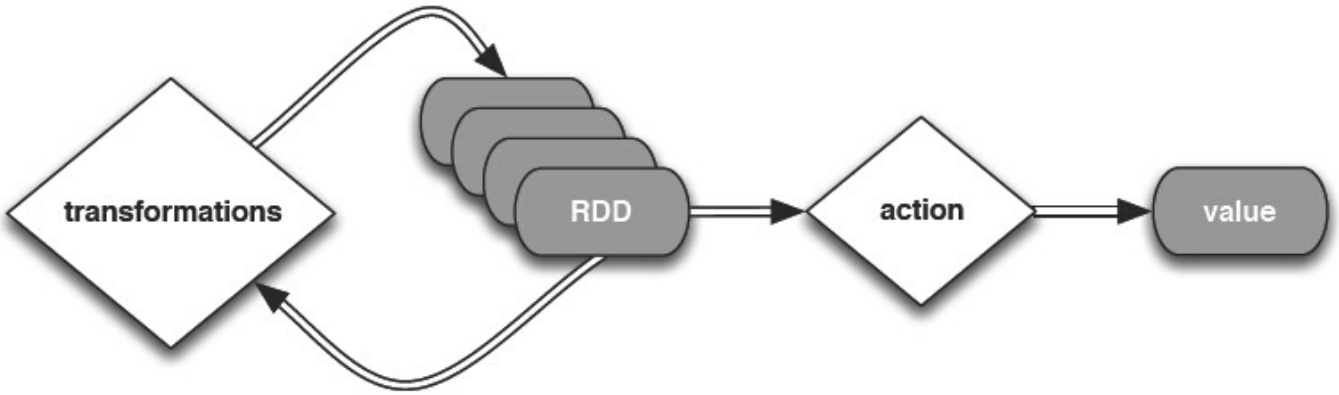
```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(_.startsWith(“ERROR”))
messages = errors.map(_.split(‘\t’)(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains(“foo”)).count
cachedMsgs.filter(_.contains(“bar”)).count
. . .
```



RDD Basics

- Two types of operations: transformations and actions
- Transformations: construct a new RDD from a previous one e.g., filter data
- Actions: compute a result base on an RDD e.g., count elements, get first element



Transformations

- Create new RDDs from existing RDDs
- Lazy evaluation
  - See the whole chain of transformations
  - Compute just the data needed
- Persist contents:
  - persist an RDD in memory to reuse it in future
  - persist RDDs on disk is possible

Typical works of a Spark program

1. Create some input RDDs form external data
2. Transform them to define new RDDs using transformations like filter()
3. Ask Spark to persist() any intermediate RDDs that will need to be reused
4. Launch actions such as count(), first() to kick off a parallel computation

Resilient Distributed Dataset – RDD

- RDD Basics
- Creating RDDs
- RDD Operations
- Common Transformation and Actions
- Persistence (Caching)

Two ways to create RDDs

1. Parallelizing a collection: uses parallelize()

- Python

```
lines = sc.parallelize(["pandas", "i like pandas"])
```

- Scala

```
val lines = sc.parallelize(List("pandas", "i like pandas"))
```

- Java

```
JavaRDD<String> lines = sc.parallelize(Arrays.asList("pandas", "i like pandas"));
```

Two ways to create RDDs

2. Loading data from external storage

- Python

```
lines =
sc.textFile("/path/to/README.md")
```

- Scala

```
val lines =
sc.textFile("/path/to/README.md")
```

- Java

```
JavaRDD<String> lines =
sc.textFile("/path/to/README.md");
```

Resilient Distributed Dataset – RDD

- RDD Basics
- Creating RDDs
- RDD Operations
- Common Transformation and Actions
- Persistence (Caching)

RDD Operations

- Two types of operations
  - Transformations: operations that return a new RDDs e.g., map(), filter()
  - Actions: operations that return a result to the driver program or write it to storage such as count(), first()
- Treated differently by Spark
  - Transformation: lazy evaluation
  - Action: execution at any time

Transformation

- Example 1. Use filter()
- Python

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
```

- Scala

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line =>
line.contains("error"))
```

- Java

```
JavaRDD<String> inputRDD = sc.textFile("log.txt");
JavaRDD<String> errorsRDD = inputRDD.filter(
new Function<String, Boolean>() {
public Boolean call(String x) {
        return x.contains("error"); }}
  });
```

Transformation

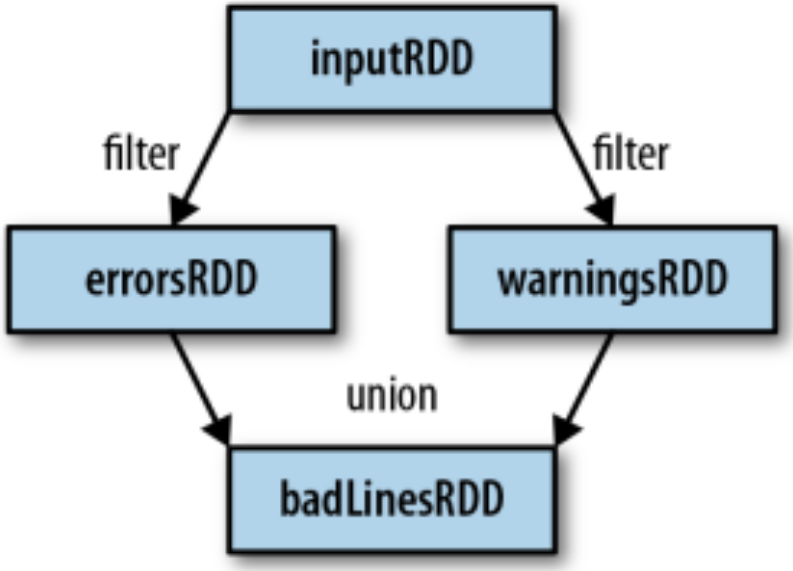
- filter()
  - does not change the existing *inputRDD*
  - returns a pointer to an entirely new RDD
  - *inputRDD* still can be reused
- union()

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD=inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

- transformations can operate on any number of input RDDs

Transformation

- Spark keeps track dependencies between RDDs, called the lineage graph
- Allow recovering lost data



Actions

- Example. count the number of errors
- Python

```
print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
print line
```

- Scala

```
println("Input had " + badLinesRDD.count() + " concerning
lines")
println("Here are 10 examples:")
badLinesRDD.take(10).foreach(println)
```

- Java

```
System.out.println("Input had " + badLinesRDD.count() + "
concerning lines")
System.out.println("Here are 10 examples:")
for (String line: badLinesRDD.take(10)) {
System.out.println(line);
}
```

Resilient Distributed Dataset – RDD

- RDD Basics
- Creating RDDs
- RDD Operations
- Common Transformation and Actions
- Persistence (Caching)

RDD Basics

Transformations	Actions
map flatMap filter sample union groupByKey reduceByKey join cache …	reduce collect count save lookupKey …

Transformations

transformation	*description*
map(*func***)**	return a new distributed dataset formed by passing each element of the source through a function *func*
filter(*func***)**	return a new dataset formed by selecting those elements of the source on which *func* returns true
flatMap(*func***)**	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)
sample(*withReplacement, fraction, seed***)**	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator *seed*
union(*otherDataset***)**	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks***]))**	return a new dataset that contains the distinct elements of the source dataset

Transformations

transformation	*description*
groupByKey([*numTasks*])	when called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs
reduceByKey(*func*, [*numTasks*])	when called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function
sortByKey([*ascending*], [*numTasks*])	when called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
join(*otherDataset*, [*numTasks*])	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key
cogroup(*otherDataset*, [*numTasks*])	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[W]) tuples – also called groupWith
cartesian(*otherDataset*)	when called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)

Actions

action	*description*
reduce(*func*)	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect()	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
count()	return the number of elements in the dataset
first()	return the first element of the dataset – similar to *take(1)*
take(*n*)	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
takeSample(*withReplacement, fraction, seed*)	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

Actions

action	*description*
`saveAsTextFile(`*`path`***`)`**	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
`saveAsSequenceFile(`*`path`***`)`**	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
`countByKey()`	only available on RDDs of type `(K, V)`. Returns a `Map` of `(K, Int)` pairs with the count of each key
`foreach(`*`func`***`)`**	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

Resilient Distributed Dataset – RDD

- RDD Basics
- Creating RDDs
- RDD Operations
- Common Transformation and Actions
- Persistence (Caching)

Persistence levels

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

Persistence

- Example

```
val result = input.map(x => x * x)
result.persist(StorageLevel.DISK_ONLY)
println(result.count())
println(result.collect().mkString(","))
```

Acknowledgement and References

IT4931
Lưu trữ và phân tích dữ liệu lớn

05/2023
Thanh-Chung Dao Ph.D.

Agenda

Zeppelin notebook

What and why we need it?

Installation using Docker

Usage

Load, inspect, and save data

Loading data from difference sources

Simple inspecting commands

Saving data

Zeppelin notebook

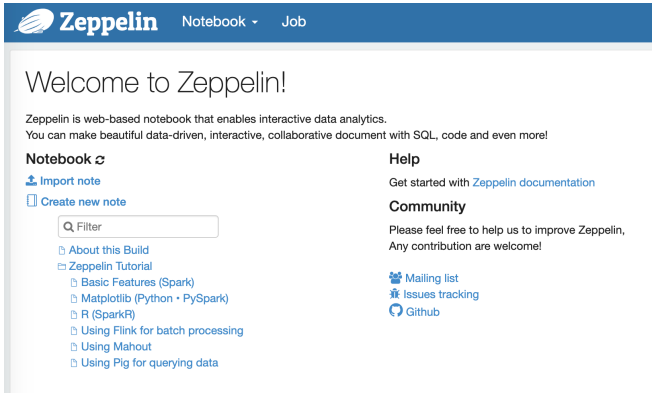
- A web-based interface for interactive data analytics
  - Easy to write and access your code
  - Support many programming languages
    - Scala (with Apache Spark), Python (with Apache Spark), SparkSQL, Hive, Markdown, Angular, and Shell
  - Data visualization
- Monitoring Spark jobs

Installation using Docker

- Install Docker and login
  - https://docs.docker.com/docker-for-windows/install/
  - https://docs.docker.com/docker-for-mac/install/
- Download lecture's git repository
  - https://github.com/bk-blockchain/big-data-class
- Run Zeppelin using docker-composer
  - docker-compose up -d --build spark_master
  - http://localhost

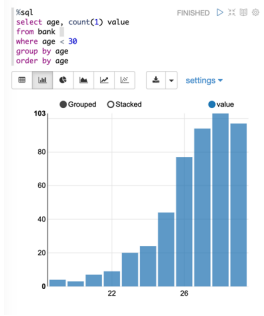
Zeppelin usage

- Run the first node: “About this Build”
  - Check Spark version



- Check Spark running mode
  - http://localhost:4040
  - Need to start Spark first by running the first note

- Run the second node: “Tutorial/Basic Features (Spark)”
  - Load data into table
  - SQL example



Useful Docker commands

- Login to a container
  - docker ps (get any container id)
  - docker exec -it container_id bash

- List all containers: docker ps -a
- Stop a container: docker stop container_id
- Start a stopped container: docker start container_id

Load, inspect, and save data

- Data is always huge that does not fit on a single machine
  - Data is distributed on many storage nodes
- Data scientists can likely focus on the format that their data is already in
  - Engineers may wish to explore more output formats
- Spark supports a wide range of input and output sources

Data sources

- File formats and filesystems
  - Local or distributed filesystem, such as NFS, HDFS, or Amazon S3
  - File formats including text, JSON, SequenceFiles, and protocol buffers
- Structured data sources through Spark SQL
  - Apache Hive
  - Parquet
  - JSON
  - From RDDs
- Databases and key/value stores
  - Cassandra, HBase, Elasticsearch, and JDBC dbs

File Formats

- Formats range from unstructured, like text, to semistructured, like JSON, to structured, like SequenceFiles

Table 5-1. Common supported file formats

Format name	Structured	Comments
Text files	No	Plain old text files. Records are assumed to be one per line.
JSON	Semi	Common text-based format, semistructured; most libraries require one record per line.
CSV	Yes	Very common text-based format, often used with spreadsheet applications.
SequenceFiles	Yes	A common Hadoop file format used for key/value data.
Protocol buffers	Yes	A fast, space-efficient multilanguage format.
Object files	Yes	Useful for saving data from a Spark job to be consumed by shared code. Breaks if you change your classes, as it relies on Java Serialization.

From Learning Spark [1]

Lab: loading, inspecting, and saving data

- On the Zeppelin notebook
  - http://localhost:8080/#/notebook/2EAMFFAH7

References

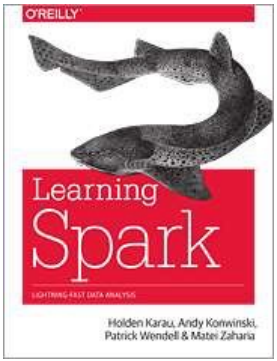
- [1] Karau, Holden, et al. *Learning spark: lightning-fast big data analysis*. " O'Reilly Media, Inc.", 2015.

IT4931
Tích hợp và xử lý dữ liệu lớn

05/2023
Thanh-Chung Dao Ph.D.

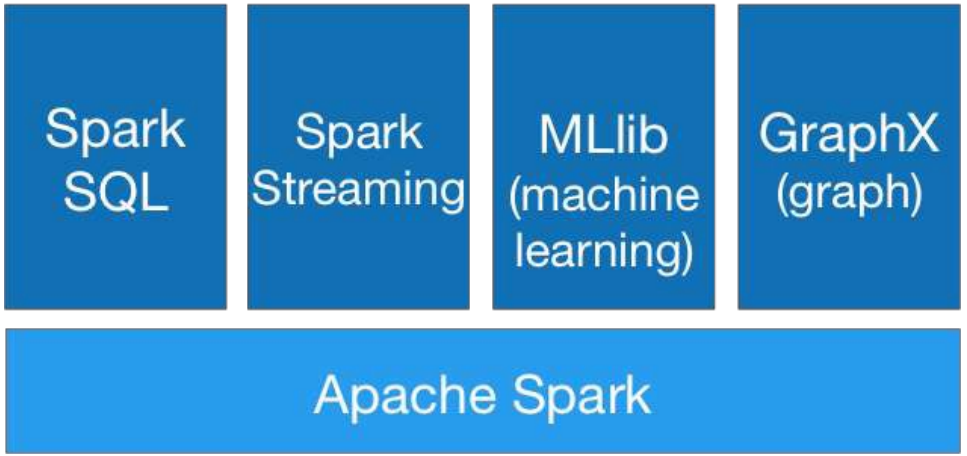
From where to learn Spark ?

http://spark.apache.org/



http://shop.oreilly.com/product/0636920028512.do

Spark architecture



Easy ways to run Spark ?

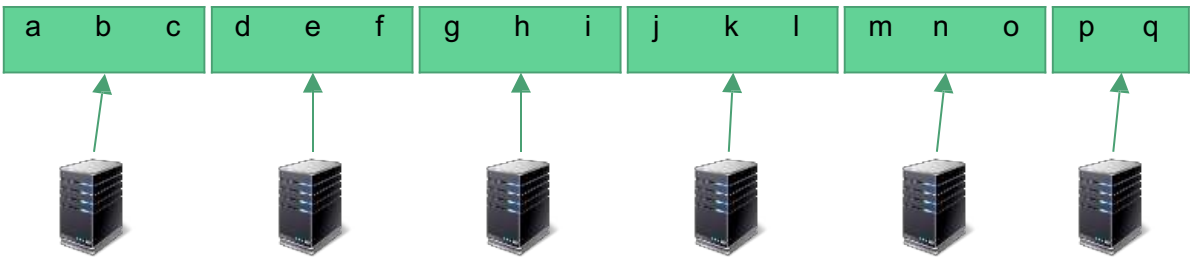
- ★ your IDE (ex. Eclipse or IDEA)
- ★ Standalone Deploy Mode: simplest way to deploy Spark on a single machine
- ★ Docker & Zeppelin
- ★ EMR
- ★ Hadoop vendors (Cloudera, Hortonworks)

Digital Ocean (Kuberneste cluster)

Supported languages

RDD

An RDD is simply an immutable distributed collection of objects!



Spark

RDD (Resilient Distributed Dataset)

RDD (Resilient Distributed Dataset)

- Resilient: If data in memory is lost, it can be recreated
- Distributed: Processed across the cluster
- Dataset: Initial data can come from a source such as a file, or it can be created programmatically

- **RDDs are the fundamental unit of data in Spark**
- **Most Spark programming consists of performing operations on RDDs**

Creating RDD (I)

Python
```
lines = sc.parallelize([“workshop”, “spark”])
```

Scala
```
val lines = sc.parallelize(List(“workshop”, “spark”))
```

Java
```
JavaRDD<String> lines = sc.parallelize(Arrays.asList(“workshop”, “spark”))
```

Creating RDD (II)

Python
lines = sc.textFile("/*path/to/file.txt*")

Scala
val lines = sc.textFile("/*path/to/file.txt*")

Java
JavaRDD<String> lines = sc.textFile("/*path/to/file.txt*")

RDD persistence

```
MEMORY_ONLY
MEMORY_AND_DISK
MEMORY_ONLY_SER
MEMORY_AND_DISK_SER
DISK_ONLY
MEMORY_ONLY_2
MEMORY_AND_DISK_2
OFF_HEAP
```

Working with RDDs

RDDs

RDDs can hold any serializable type of element

- –Primitive types such as integers, characters, and booleans
- –Sequence types such as strings, lists, arrays, tuples, and dicts (including nested data types)
- –Scala/Java Objects (if serializable)
- –Mixed types

§ **Some RDDs are specialized and have additional functionality**

- –Pair RDDs
- –RDDs consisting of key-value pairs
- –Double RDDs
- –RDDs consisting of numeric data

Creating RDDs from Collections

You can create RDDs from collections instead of files
–sc.parallelize(*collection*)

```
myData = ["Alice","Carlos","Frank","Barbara"]
> myRdd = sc.parallelize(myData)
> myRdd.take(2) ['Alice', 'Carlos']
```

Creating RDDs from Text Files (1)

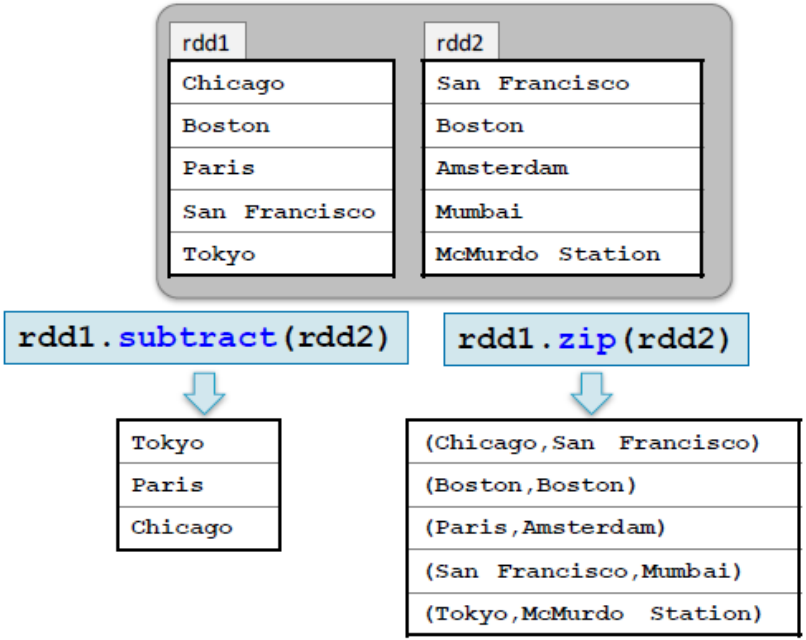
For file-based RDDs, use SparkContext.textFile

- Accepts a single file, a directory of files, a wildcard list of files, or a comma-separated list of files. Examples:
- **sc.textFile("myfile.txt")**
- **sc.textFile("mydata/")**
- **sc.textFile("mydata/*.log")**
- **sc.textFile("myfile1.txt,myfile2.txt")**
- Each line in each file is a separate record in the RDD

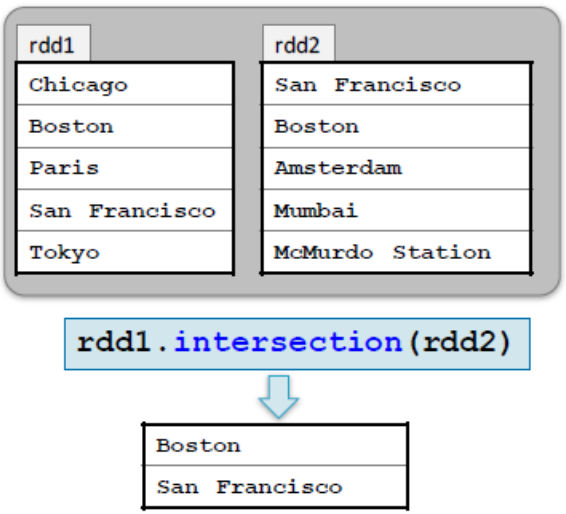
Files are referenced by absolute or relative URI

- Absolute URI:
- **file:/home/training/myfile.txt**
- **hdfs://nnhost/loudacre/myfile.txt**

Examples: Multi-RDD Transformations (1)



Examples: Multi-RDD Transformations (2)



Some Other General RDD Operations

Other RDD operations

- *first* returns the first element of the RDD
- *foreach* applies a function to each element in an RDD
- *top(n)* returns the largest *n* elements using natural ordering

Sampling operations

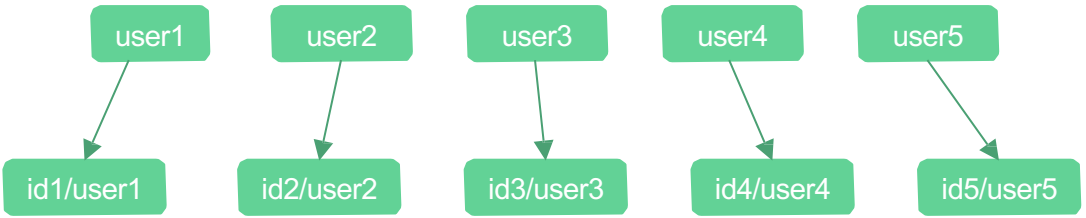
- sample creates a new RDD with a sampling of elements
- *take* Sample returns an array of sampled elements

Other data structures in Spark

- ★ Paired RDD
- ★ DataFrame
- ★ DataSet

Paired RDD

Paired RDD = an RDD of key/value pairs

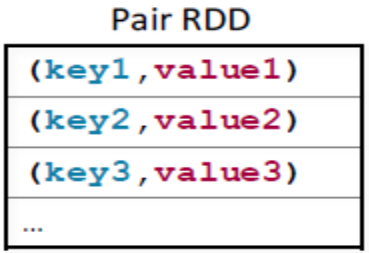


Pair RDDs

Pair RDDs

§ **Pair RDDs are a special form of RDD**
- –Each element must be a key-value pair (a two-element *tuple*)
- –Keys and values can be any type

§ **Why?**
- –Use with map-reduce algorithms
- –Many additional functions are available for common data processing needs
- –Such as sorting, joining, grouping, and counting



Creating Pair RDDs

The first step in most workflows is to get the data into key/value form

- What should the RDD should be keyed on?
- What is the value?

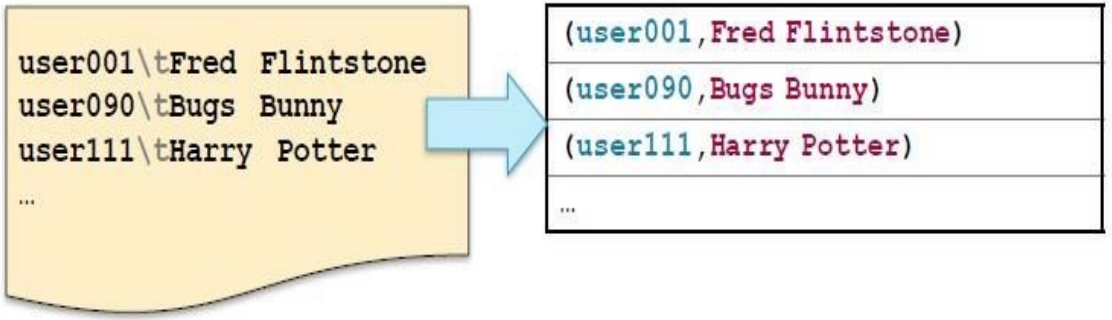
Commonly used functions to create pair RDDs

- **map**
- **flatMap / flatMapValues**
- **keyBy**

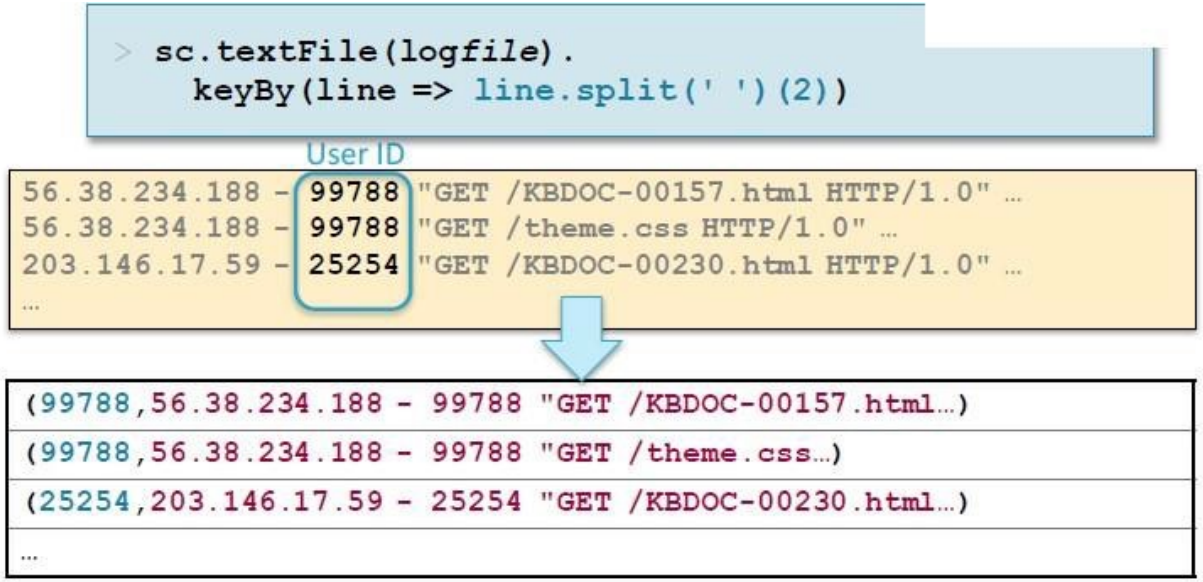
Example: A Simple Pair RDD

Example: Create a pair RDD from a tab-separated file

```
> val users = sc.textFile(file).
   map(line => line.split('\t').
   map(fields => (fields(0),fields(1)))
```

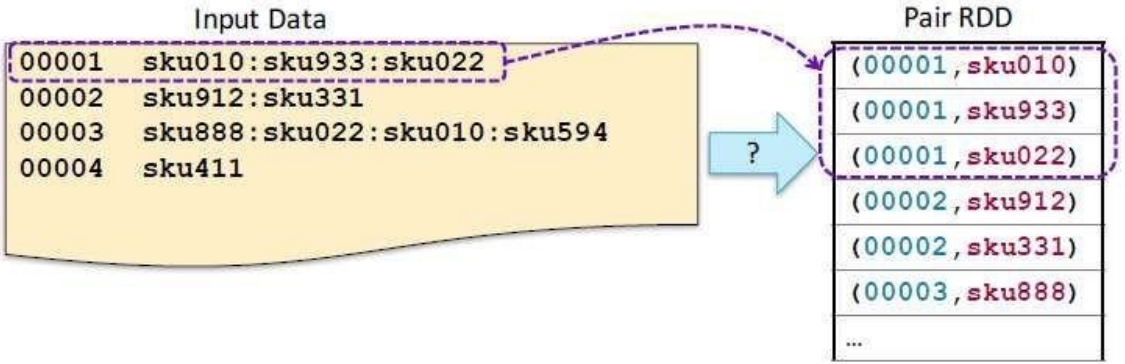


Example: Keying Web Logs by User ID



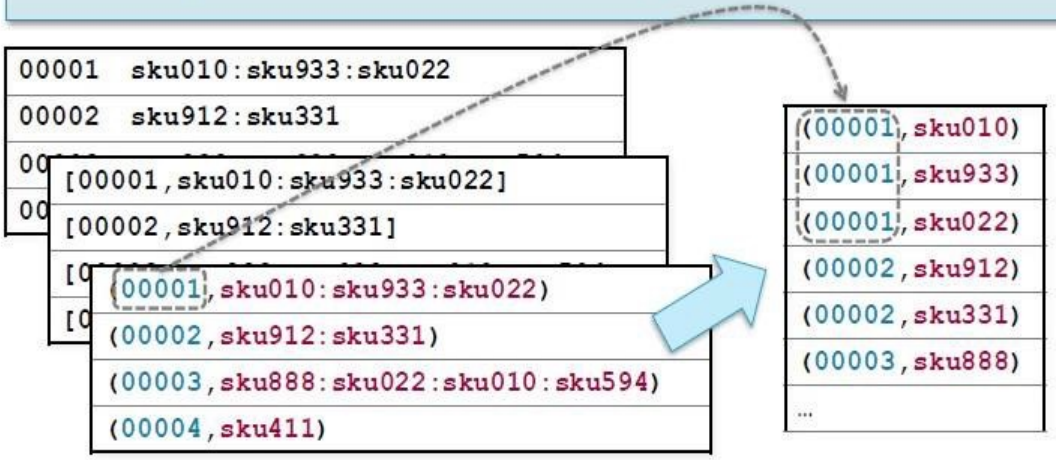
Mapping Single Rows to Multiple Pairs

- **How would you do this?**
  - Input: order numbers with a list of SKUs in the order
  - Output: order (key) and sku (value)



Answer : Mapping Single Rows to Multiple Pairs

```
> sc.textFile(file) \
  .map(lambda line: line.split('\t')) \
  .map(lambda fields: (fields[0],fields[1]))
  .flatMapValues(lambda skus: skus.split(':'))
```



Map-Reduce

§ **Map-reduce is a common programming model**
  – Easily applicable to distributed processing of large data sets

§ **Hadoop MapReduce is the major implementation**
  – Somewhat limited
  – Each job has one map phase, one reduce phase
  – Job output is saved to files

§ **Spark implements map-reduce with much greater flexibility**
  – Map and reduce functions can be interspersed
  – Results can be stored in memory
  – Operations can easily be chained

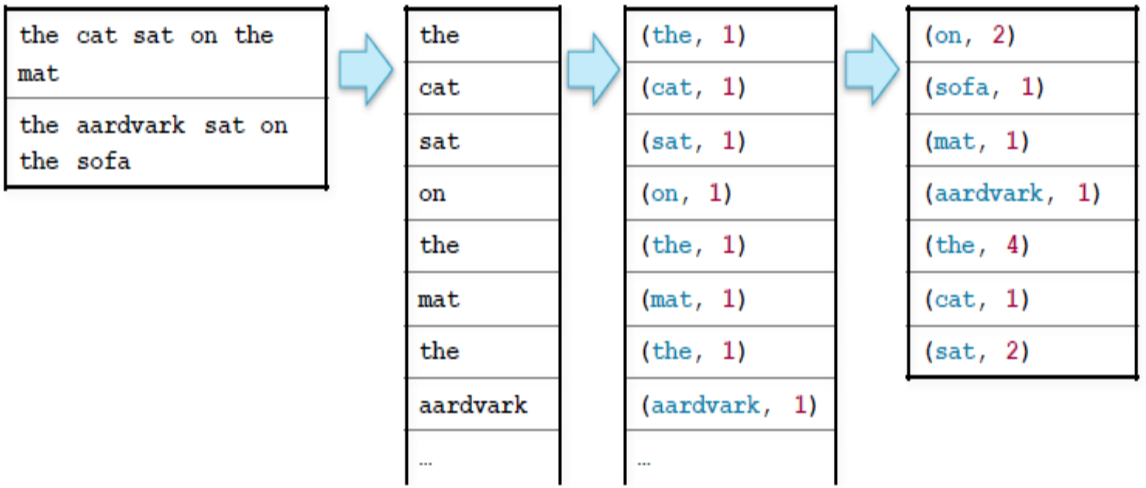
Map-Reduce in Spark

§ **Map-reduce in Spark works on pair RDDs**

§ **Map phase**
- –Operates on one record at a time
- –"Maps" each record to zero or more new records
- –Examples: **map, flatMap, filter, keyBy**

§ **Reduce phase**
- –Works on map output
- –Consolidates multiple records
- –Examples: **reduceByKey, sortByKey, mean**

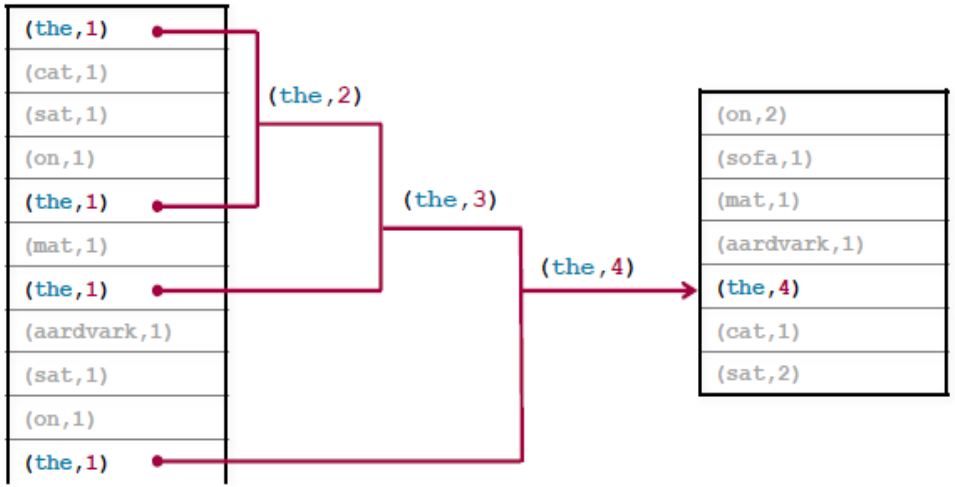
Example: Word Count



reduceByKey

The function passed to reduceByKey combines values from two keys

– Function must be binary



```
> val counts = sc.textFile(file).flatMap
  (line => line.split(' ')).map(word => (word
  ,1)).reduceByKey((v1,v2) => v1+v2)
```

OR

```
> val counts = sc.textFile(file).flatMap
  (_.split(' ')).
  map((_,1)).
  reduceByKey(_+_)
```

Pair RDD Operations

§ **In addition to map and reduceByKey operations, Spark has several operations specific to pair RDDs**

§ **Examples**

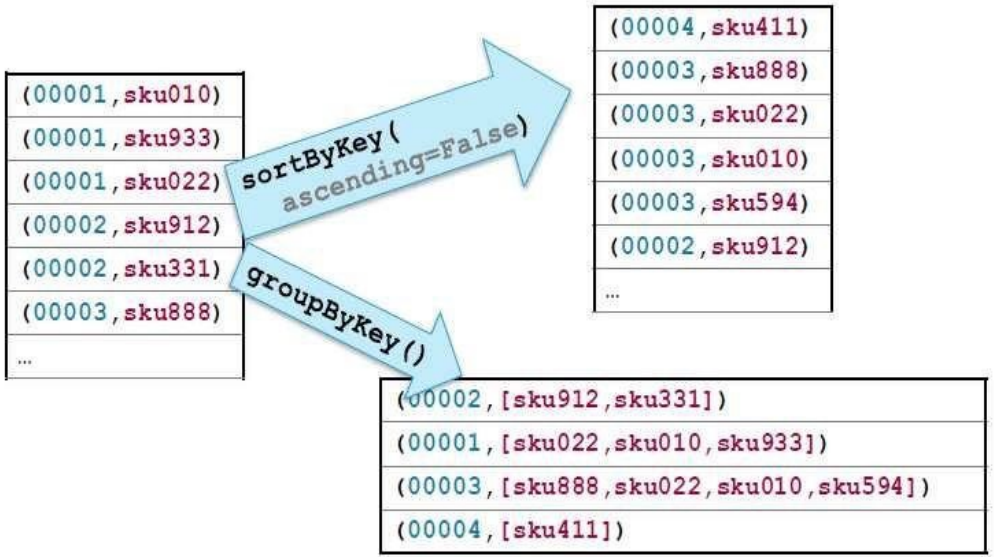
–**countByKey** returns a map with the count of occurrences
of each key

–**groupByKey** groups all the values for each key in an RDD

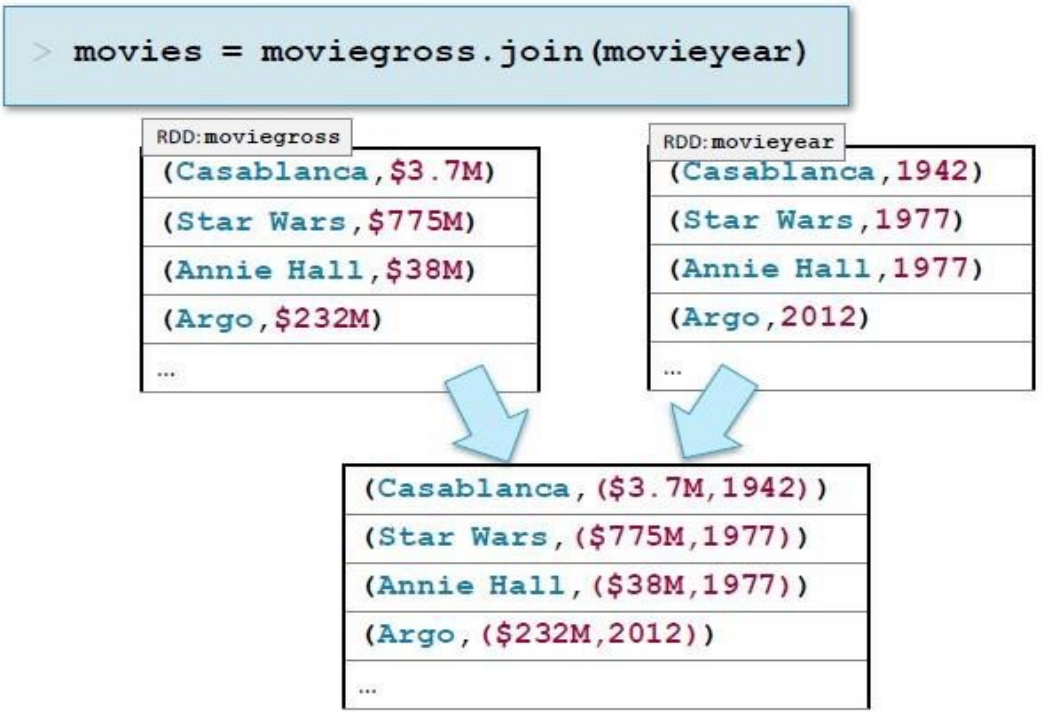
–**sortByKey** sorts in ascending or descending order

–**join** returns an RDD containing all pairs with matching keys from two RDD

Example: Pair RDD Operations



Example: Joining by Key



Other Pair Operations

§ **Some other pair operations**

–**keys** returns an RDD of just the keys, without the values

–**values** returns an RDD of just the values, without keys

–**lookup(*key*)** returns the value(s) for a key

–**leftOuterJoin**, **rightOuterJoin** , **fullOuterJoin** join two RDDs, including keys defined in the left, right or either RDD respectively

–**mapValues**, **flatMapValues** execute a function on just the values,
keeping the key the same

DataFrames and Apache Spark SQL

What is Spark SQL?

§ **What is Spark SQL?**
- –Spark module for structured data processing
- –Replaces Shark (a prior Spark module, now deprecated)
- –Built on top of core Spark

§ **What does Spark SQL provide?**
- –The DataFrame API—a library for working with data as tables
- –Defines DataFrames containing rows and columns
- –DataFrames are the focus of this chapter!
- –Catalyst Optimizer—an extensible optimization framework
- –A SQL engine and command line interface

SQL Context

§ **The main Spark SQL entry point is a SQL context object**
  – Requires a **SparkContext** object
  – The SQL context in Spark SQL is similar to Spark context in core Spark
§ **There are two implementations**
  – **SQLContext**
  – Basic implementation
  – **HiveContext**
  – Reads and writes Hive/HCatalog tables directly
  – Supports full HiveQL language
  – Requires the Spark application be linked with Hive libraries
  – Cloudera recommends using **HiveContext**

Creating a SQL Context

- **The Spark shell creates a HiveContext instance automatically**
  - Call **sqlContext**
  - You will need to create one when writing a Spark application
  - Having multiple SQL context objects *is* allowed

- **A SQL context object is created based on the Spark context**

Language: Scala

```
import org.apache.spark.sql.hive.HiveContext
val sqlContext = new HiveContext(sc)
import sqlContext.implicits._
```

DataFrames

- **DataFrames are the main abstraction in Spark SQL**
  - Analogous to RDDs in core Spark
  - A distributed collection of structured data organized into Named columns
  - Built on a *base RDD* containing **Row** objects

Creating a DataFrame from a Data Source

- **sqlContext.read returns a DataFrameReader object**

- **DataFrameReader provides the functionality to load data into a DataFrame**

- **Convenience functions**
  - **json(*filename*)**
  - **parquet(*filename*)**
  - **orc(*filename*)**
  - **table(*hive-tablename*)**
  - **jdbc(*url*,*table*,*options*)**

Example: Creating a DataFrame from a JSON File

Language: Scala

```
val sqlContext = new HiveContext(sc)
import sqlContext.implicits._
val peopleDF = sqlContext.read.json("people.json")
```

File: people.json

```
{"name":"Alice", "pcode":"94304"}
{"name":"Brayden", "age":30, "pcode":"94304"}
{"name":"Carla", "age":19, "pcode":"10036"}
{"name":"Diana", "age":46}
{"name":"Étienne", "pcode":"94104"}
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

Example: Creating a DataFrame from a Hive/Impala Table

Language: Scala

```
val sqlContext = new HiveContext(sc)
import sqlContext.implicits._
val customerDF = sqlContext.read.table("customers")
```

Table: customers

cust_id	name	country
001	Ani	us
002	Bob	ca
003	Carlos	mx
...	...	...

cust_id	name	country
001	Ani	us
002	Bob	ca
003	Carlos	mx
...	...	...

Loading from a Data Source Manually

§ **You can specify settings for the DataFrameReader**
–**format**: Specify a data source type
–**option**: A key/value setting for the underlying data source
–**schema**: Specify a schema instead of inferring from the data source
§ **Then call the generic base function load**

```
sqlContext.read.
   format("com.databricks.spark.avro").
   load("/loudacre/accounts_avro")
```

```
sqlContext.read.
   format("jdbc").
   option("url","jdbc:mysql://localhost/loudacre").
   option("dbtable","accounts").
   option("user","training").
   option("password","training").
   load()
```

Data Sources

- **Spark SQL 1.6 built-in data source types**
  - **table**
  - **json**
  - **parquet**
  - **jdbc**
  - **orc**
- **You can also use third party data source libraries, such as**
  - Avro (included in CDH)
  - HBase
  - CSV
  - MySQL
  - and more being added all the time

DataFrame Basic Operations

- § Basic operations deal with DataFrame metadata (rather than its data)
- § Some examples
  - –schema returns a schema object describing the data
  - –printSchema displays the schema as a visual tree
  - –cache / persist persists the DataFrame to disk or memory
  - –columns returns an array containing the names of the columns
  - –dtypes returns an array of (column name,type) pairs
  - –explain prints debug information about the DataFrame to the console

DataFrame Basic Operations

Language: Scala

```
> val peopleDF = sqlContext.read.json("people.json")
> peopleDF.dtypes.foreach(println)
(age,LongType)
(name,StringType)
(pcode,StringType)
```

DataFrame Actions

- Some DataFrame actions
  - –collect returns all rows as an array of Row objects
  - –take(*n*) returns the first n rows as an array of Row objects
  - –count returns the number of rows
  - –show(*n*)displays the first n rows (default=20)

Language: Scala

```
> peopleDF.count()
res7: Long = 5

> peopleDF.show(3)
age  name    pcode
null Alice   94304
30   Brayden 94304
19   Carla   10036
```

DataFrame Queries

- **DataFrame query methods return new DataFrames**
  - Queries can be chained like transformations
- **Some query methods**
  - **distinct** returns a new DataFrame with distinct elements of this DF
  - **join** joins this DataFrame with a second DataFrame
  - Variants for inside, outside, left, and right joins
  - **limit** returns a new DataFrame with the first **n** rows of this DF
  - **select** returns a new DataFrame with data from one or more columns of the base DataFrame
  - **where** returns a new DataFrame with rows meeting specified query criteria (alias for **filter**)

DataFrame Query Strings

- **Some query operations take strings containing simple query expressions**
  - Such as `select` and `where`
- **Example: `select`**

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

```
peopleDF.
  select("age")
```

age
null
30
19
46
null

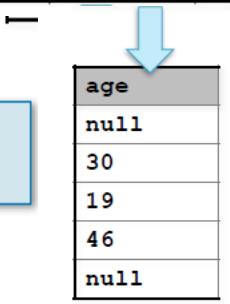
```
peopleDF.
  select("name","age")
```

name	age
Alice	null
Brayden	30
Carla	19
Diana	46
Étienne	null

Querying DataFrames using Columns

§ **Columns can be referenced in multiple ways**

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104



age
null
30
19
46
null

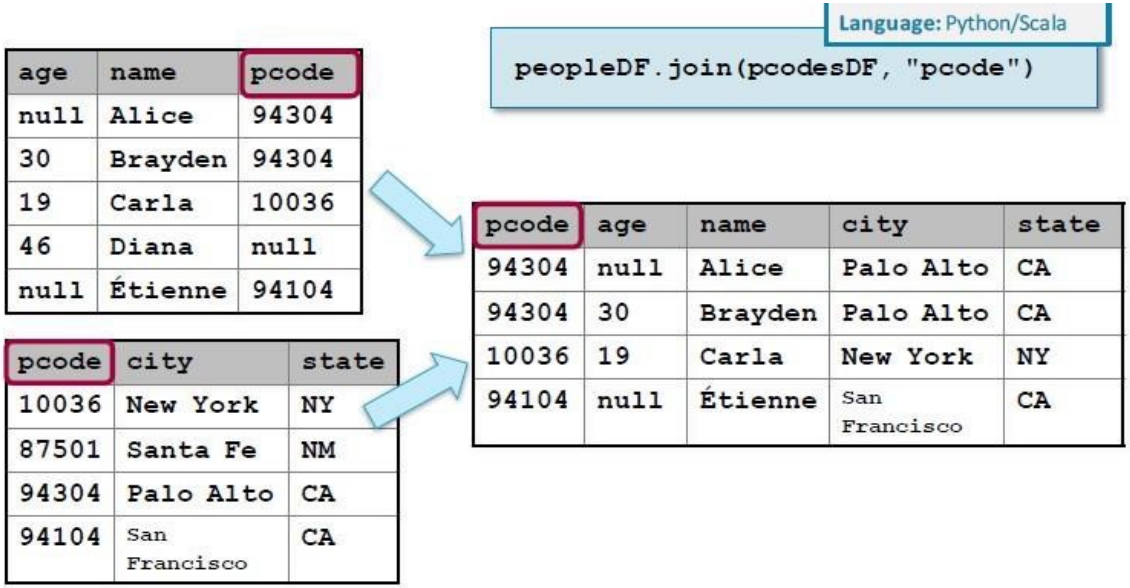
- **Scala**

```
val ageDF = peopleDF.select(peopleDF("age"))
```

```
val ageDF = peopleDF.select($"age")
```

Joining DataFrames

§ **A basic inner join when join column is in both DataFrames**



Joining DataFrames

- **Specify type of join as `inner` (default), `outer`, `left_outer`, `right_outer`, or `leftsemi`**

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

pcode	city	state
10036	New York	NY
87501	Santa Fe	NM
94304	Palo Alto	CA
94104	San Francisco	CA

Language: Python

```
peopleDF.join(pcodesDF, "pcode",
"left_outer")
```

Language: Scala

```
peopleDF.join(pcodesDF,
Array("pcode"), "left_outer")
```

pcode	age	name	city	state
94304	null	Alice	Palo Alto	CA
94304	30	Brayden	Palo Alto	CA
10036	19	Carla	New York	NY
null	46	Diana	null	null
94104	null	Étienne	San Francisco	CA

SQL Queries

§ **When using HiveContext, you can query Hive/Impala tables using HiveQL**

– Returns a DataFrame

Language: Python/Scala

```
sqlContext.
  sql("""SELECT * FROM customers WHERE name LIKE "A%" """)
```

Table: customers

cust_id	name	country
001	Ani	us
002	Bob	ca
003	Carlos	mx
…	…	…

cust_id	name	country
001	Ani	us

Saving DataFrames

§ **Data in DataFrames can be saved to a data source**

§ **Use DataFrame.write to create a DataFrameWriter**

§ **DataFrameWriter provides convenience functions to externally save the data represented by a DataFrame**

–**jdbc** inserts into a new or existing table in a database

–**json** saves as a JSON file

–**parquet** saves as a Parquet file

–**orc** saves as an ORC file

–**text** saves as a text file (string data in a single column only)

–**saveAsTable** saves as a Hive/Impala table (**HiveContext** only)

Language: Python/Scala

```
peopleDF.write.saveAsTable("people")
```

Options for Saving DataFrames

§ DataFrameWriter option methods
- –format specifies a data source type
- –mode determines the behavior if file or table already exists:
  overwrite, append, ignore or error (default is error)
- –partitionBy stores data in partitioned directories in the form
  column=value (as with Hive/Impala partitioning)
- –options specifies properties for the target data source
- –save is the generic base function to write the data

Language: Python/Scala

```
peopleDF.write.
  format("parquet").
  mode("append").
  partitionBy("age").
  saveAsTable("people")
```

DataFrames and RDDs

§ **DataFrames are built on RDDs**

–Base RDDs contain **Row** objects

–Use **rdd** to get the underlying RDD

```
peopleRDD = peopleDF.rdd
```

peopleDF

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

peopleRDD

```
Row[null,Alice,94304]
Row[30,Brayden,94304]
Row[19,Carla,10036]
Row[46,Diana,null]
Row[null,Étienne,94104]
```

DataFrames and RDDs

§ **Row RDDs have all the standard Spark actions and transformations**
–Actions: **collect**, **take**, **count**, and so on
–Transformations: **map**, **flatMap**, **filter**, and so on

§ **Row RDDs can be transformed into pair RDDs to use map-reduce methods**

§ **DataFrames also provide convenience methods (such as map, flatMap,and foreach)for converting to RDDs**

Working with **Row** Objects

- Use **Array**-like syntax to return values with type **Any**
- **row(*n*)** returns element in the *n*th column
- **row.fieldIndex("age")**returns index of the **age** column
- Use methods to get correctly typed values
- **row.getAs[Long]("age")**
- Use type-specific **get** methods to return typed values
- **row.getString(*n*)** returns *n*th column as a string
- **row.getInt(*n*)** returns *n*th column as an integer
- And so on

Example: Extracting Data from **Row** Objects

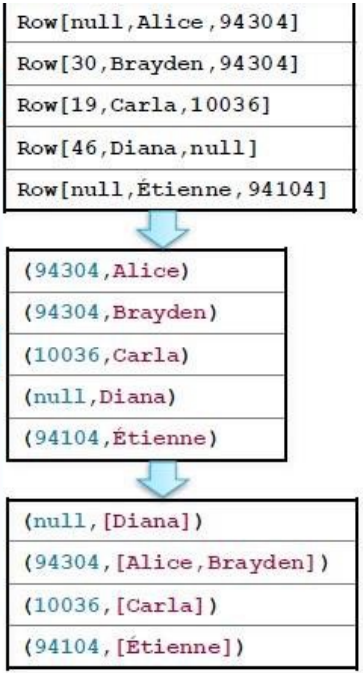
- **Extract data from Row objects**

Language: Python

```
peopleRDD = peopleDF \
  .map(lambda row: (row.pcode,row.name))
peopleByPCode = peopleRDD \
  .groupByKey()
```

Language: Scala

```
val peopleRDD = peopleDF.
  map(row =>
        (row(row.fieldIndex("pcode")),
         row(row.fieldIndex("name"))))
val peopleByPCode = peopleRDD.
  groupByKey()
```



Converting RDDs to DataFrames

§ **You can also create a DF from an RDD using createDataFrame**

Language: Scala

```
import org.apache.spark.sql.types._
import org.apache.spark.sql.Row
val schema = StructType(Array(
  StructField("age", IntegerType, true),
  StructField("name", StringType, true),
  StructField("pcode", StringType, true)))
val rowrdd = sc.parallelize(Array(Row(40,"Abram","01601"),
                                  Row(16,"Lucia","87501")))
val mydf = sqlContext.createDataFrame(rowrdd,schema)
```

Working with Spark RDDs, Pair-RDDs

RDD Operations

Transformations

map()
flatMap()
filter()
union()
intersection()
distinct()
groupByKey()
reduceByKey()
sortByKey()
join()
…

Actions

count()
collect()
first(), top(n)
take(n), takeOrdered(n)
countByValue()
reduce()
foreach()
…

Lambda Expression

PySpark WordCount example:

```
input_file = sc.textFile("/path/to/text/file")
map = input_file.flatMap(lambda line: line.split(" ")) \
                    .map(lambda word: (word, 1))
counts = map.reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("/path/to/output/")
```

lambda arguments: expression

PySpark RDD API

https://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD

map(*f*, *preservesPartitioning=False*) [source]

Return a new RDD by applying a function to each element of this RDD.

```
>>> rdd = sc.parallelize(["b", "a", "c"])
>>> sorted(rdd.map(lambda x: (x, 1)).collect())
[('a', 1), ('b', 1), ('c', 1)]
```

flatMap(*f*, *preservesPartitioning=False*) [source]

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

```
>>> rdd = sc.parallelize([2, 3, 4])
>>> sorted(rdd.flatMap(lambda x: range(1, x)).collect())
[1, 1, 1, 2, 2, 3]
>>> sorted(rdd.flatMap(lambda x: [(x, x), (x, x)]).collect())
[(2, 2), (2, 2), (3, 3), (3, 3), (4, 4), (4, 4)]
```

Practice with flight data (1)

Data: **airports.dat** (https://openflights.org/data.html)

[Airport ID, Name, City, Country, IATA, ICAO, Latitude, Longitude, Altitude, Timezone, DST, Tz database, Type, Source]

Try to do somethings:

- Create RDD from textfile
- Count the number of airports
- Filter by country
- Group by country
- Count the number of airports in each country

Practice with flight data (2)

- Data: **airports.dat** (https://openflights.org/data.html)

[Airport ID, Name, City, Country, IATA, ICAO, Latitude, Longitude, Altitude, Timezone, DST, Tz database, Type, Source]

- Data: **routes.dat**

[Airline, Airline ID, Source airport, Source airport ID, Destination airport, Destination airport ID, Codeshare, Stops, Equipment]

Try to do somethings:

- Join 2 RDD
- Count the number of flights arriving in each country

Working with DataFrame and Spark SQL

Creating a DataFrame(1)

```
%pyspark
from pyspark.sql import *

Employee = Row("firstName", "lastName", "email", "salary")

employee1 = Employee('Basher', 'armbrust', 'bash@edureka.co', 100000)
employee2 = Employee('Daniel', 'meng', 'daniel@stanford.edu', 120000 )
employee3 = Employee('Muriel', None, 'muriel@waterloo.edu', 140000 )
employee4 = Employee('Rachel', 'wendell', 'rach_3@edureka.co', 160000 )
employee5 = Employee('Zach', 'galifianakis', 'zach_g@edureka.co', 160000 )

employees = [employee1,employee2,employee3,employee4,employee5]

print(Employee[0])

print(employees)

dframe = spark.createDataFrame(employees)
dframe.show()
```

Creating a DataFrame

From CSV file:

```
%pyspark
flightData2015 = spark\
  .read\
  .option("inferSchema", "true")\
  .option("header", "true")\
  .csv("/usr/zeppelin/module9/2015-summary.csv")

flightData2015.show()
```

From RDD:

```
%pyspark
from pyspark.sql import *
list = [('Ankit',25),('Jalfaizy',22),('saurabh',20),('Bala',26)]
rdd = sc.parallelize(list)
people = rdd.map(lambda x: Row(name=x[0], age=int(x[1])))
df = spark.createDataFrame(people)

df.show()
```

DataFrame APIs

- **DataFrame**: show(), collect(), createOrReplaceTempView(), distinct(), filter(), select(), count(), groupBy(), join()…
- **Column**: like()
- **Row**: row.key, row[key]
- **GroupedData**: count(), max(), min(), sum(), …

https://spark.apache.org/docs/latest/api/python/pyspark.sql.html

Spark SQL

- Create a temporary view
- Query using SQL syntax

```
%pyspark
flightData2015.createOrReplaceTempView("flight_data_2015")

maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")

maxSql.show()
```

IT4931
Tích hợp và xử lý dữ liệu lớn

06/2023
Thanh-Chung Dao Ph.D.

Spark running mode

- Local
- Clustered
  - Spark Standalone
  - Spark on Apache Mesos
  - Spark on Hadoop YARN

Hello World: Word-Count

```
1  import sys
2  from pyspark import SparkContext
3  sc = SparkContext(appName="WordCountExample")
4  lines = sc.textFile(sys.argv[1])
5  counts = lines.flatMap(lambda x: x.split(' ')) \
6                    .map(lambda x: (x, 1)) \
7                    .reduceByKey(lambda x,y:x+y)
8  output = counts.collect()
9  for (word, count) in output:
10     print "%s: %i" % (word, count)
11 sc.stop()
```

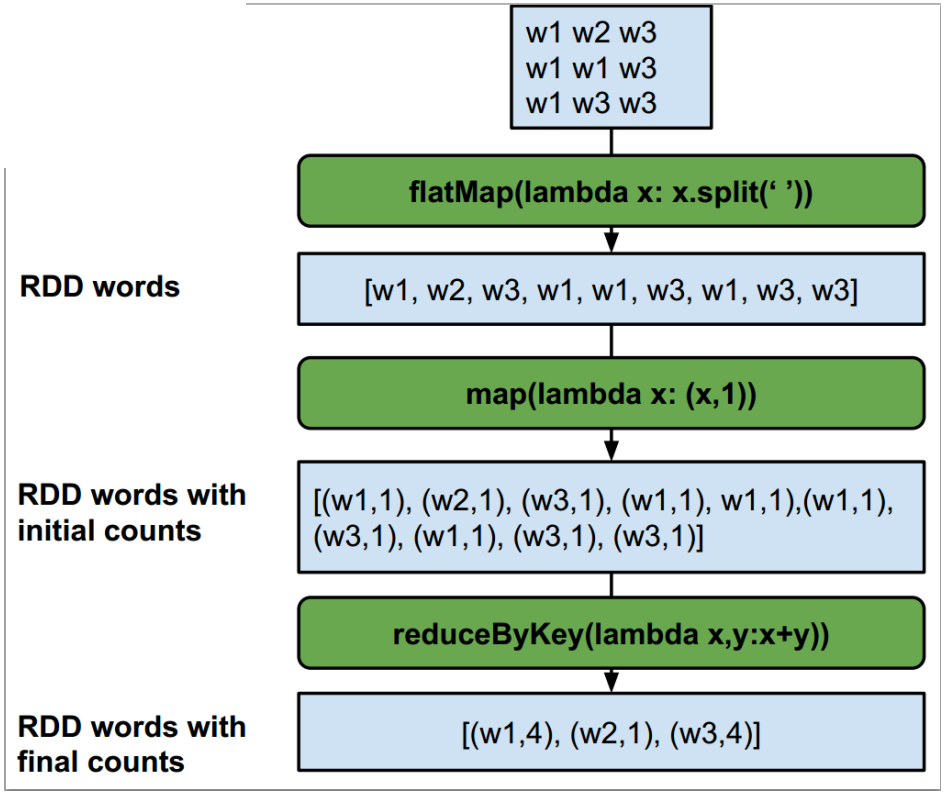


Figure from [1]

Run using command line

- Turn on docker bash
- spark-submit wordcount.py README.md
- Result will be shown as follows

```
19/05/19 07:56:51 INFO scheduler.TaskSchedulerImpl: Removed
pool
19/05/19 07:56:51 INFO scheduler.DAGScheduler: ResultStage 1
) finished in 0.150 s
19/05/19 07:56:51 INFO scheduler.DAGScheduler: Job 0 finishe
0, took 2.050066 s
Turks: 1
States,294: 1
Algeria,United: 1
States,2025: 1
States,955: 1
States,Czech: 1
Colombia,United: 1
States,588: 1
States,Dominican: 1
```

Lab: Word-Count

- Lab on the Zeppelin notebook
- Github source code
  - https://github.com/bk-blockchain/big-data-class

Flight data:

- Analyzing flight data from the United States Bureau of Transportation statistics

- Lab on the Zeppelin notebook
- Github source code
  - https://github.com/bk-blockchain/big-data-class

References

- [1] https://datamize.wordpress.com/2015/02/08/visualizing-basic-rdd-operations-through-wordcount-in-pyspark/

IT4931
Tích hợp và xử lý dữ liệu lớn

06/2023
Thanh-Chung Dao Ph.D.

Agenda

- What is Spark Streaming
- Operation on DStreams

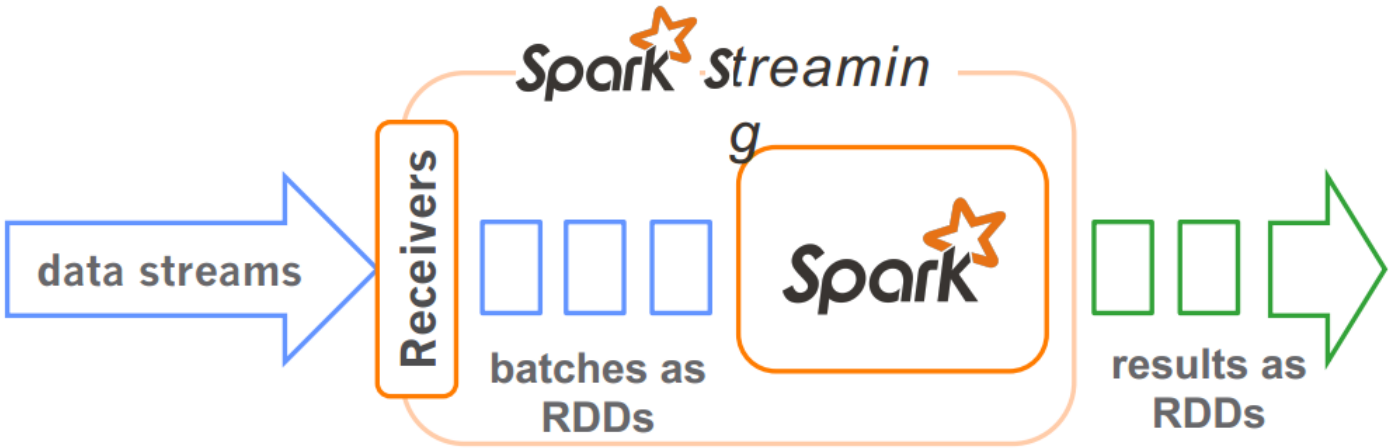
What is Spark Streaming

Spark Streaming

- Scalable, fault-tolerant stream processing system
- Receive data streams from input sources, process them in a cluster, push out to databases/dashboards

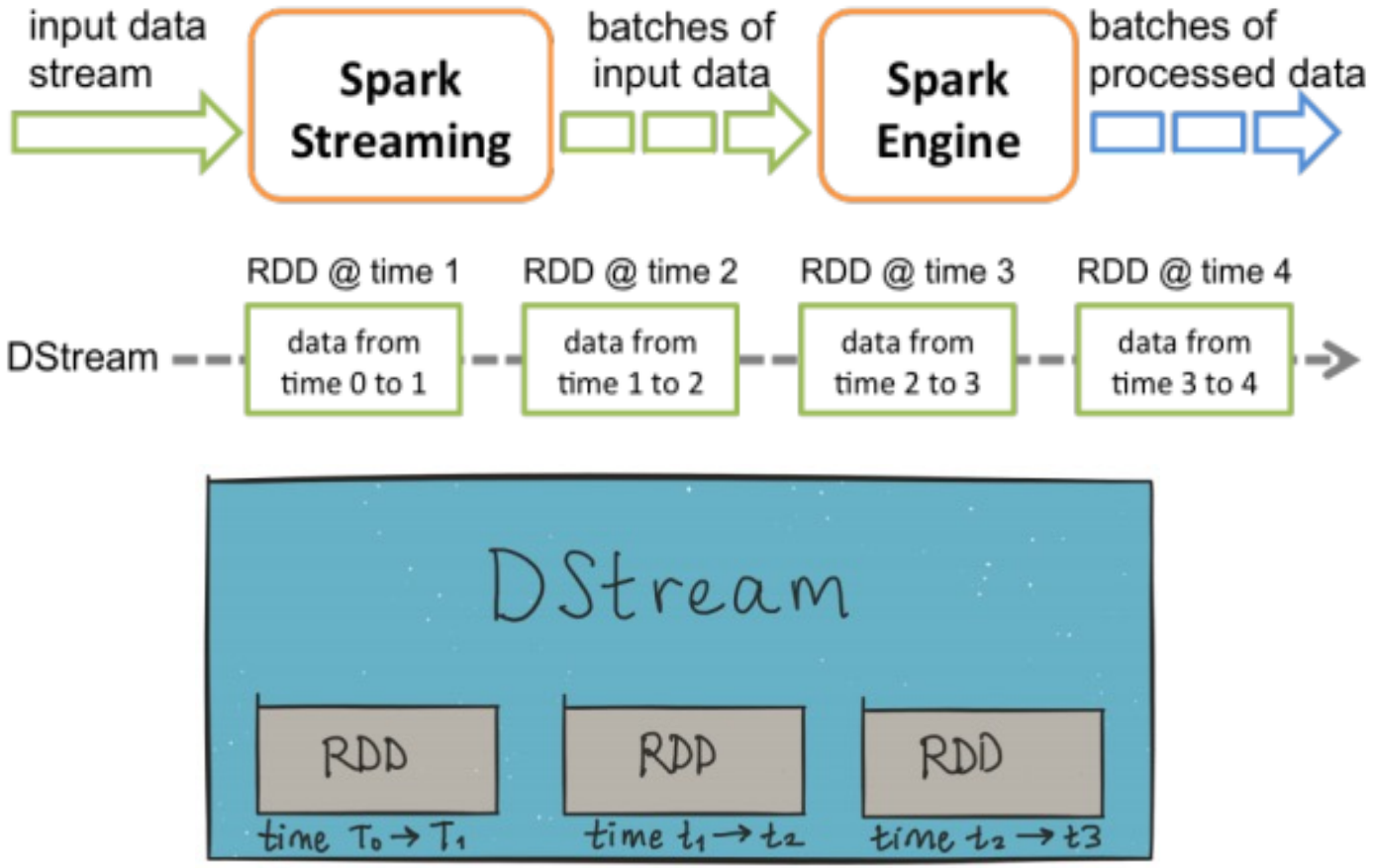
How does it work?

- The stream is treated as a series of very small, deterministic batches of data
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Processed results are pushed out in batches



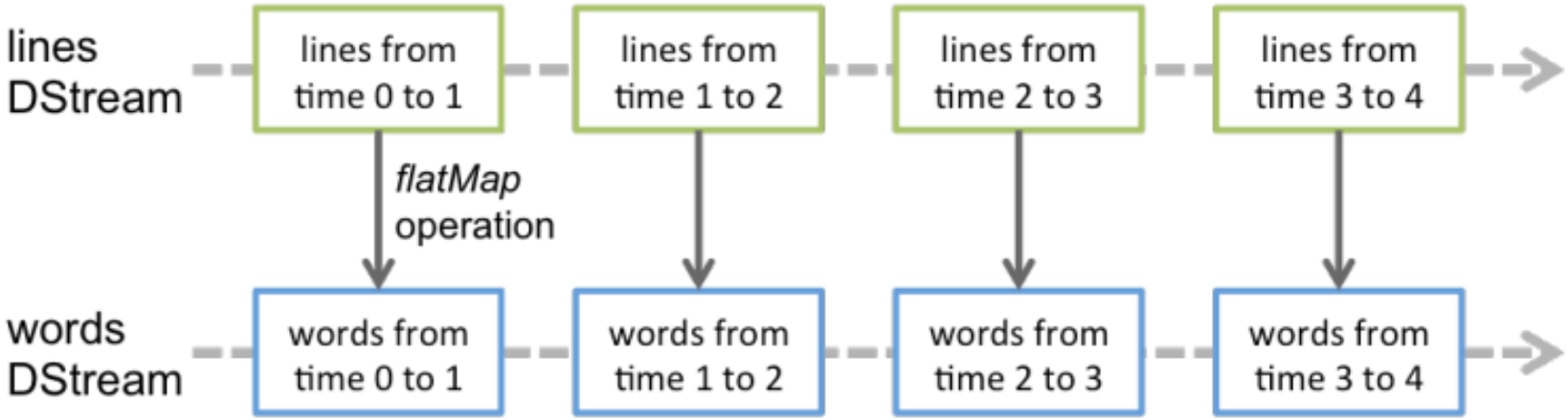
Discretized Stream (DStream)

- Sequence of RDDs representing a stream of data



Discretized Stream (DStream)

- Any operation applied on a DStream translates to operations on the underlying RDDs



StreamingContext

- The main entry point of all Spark Streaming functionality

```
val conf = new
SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, batchinterval)
```

- appname: name of the application
- master: a Spark, Mesos, or YARN cluster URL
- batchinternval: time interval (in second) of each batch

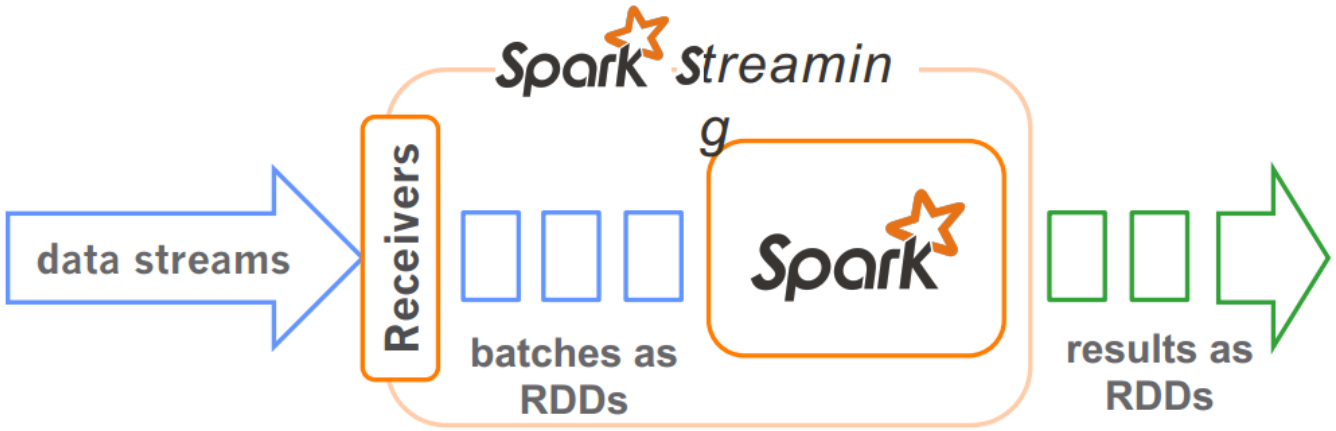
Operation on DStreams

Operation on DStreams

- Three categories
  - Input operation
  - Transformation operation
  - Output operation

Input Operations

- Every input DStream is associated with a Receiver object
- Two built-in categories of streaming sources:
  - Basic sources, e.g., file systems, socket connection
  - Advanced sources, e.g., Twitter, Kafka



Input Operations

- Basic sources
  - Socket connection

```
// Create a DStream that will connect to hostname:port
ssc.socketTextStream("localhost", 9999)
```

  - File stream

```
streamingContext.fileStream[…](dataDirectory)
```

- Advanced sources

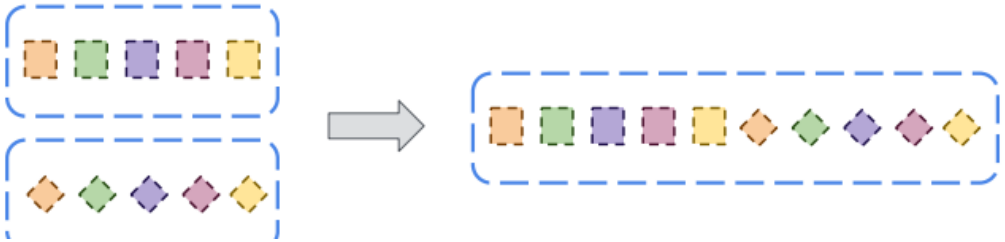
```
val ssc = new StreamingContext(sparkContext, Seconds(1))
val tweets = TwitterUtils.createStream(ssc, auth)
```

Transformation

map,
flatmap,
filter

count,
reduce,
countByValue,
reduceByKey

union,
join
cogroup



Transformation

Transformation	Meaning
map (func)	Return a new DStream by passing each element of the source DStream through a function func
flatmap(func)	Similar to map, but each input item can be mapped to 0 or more output items
filter(func)	Return a new DStream by selecting only the records of the source DStream on which func returns true

Transformation

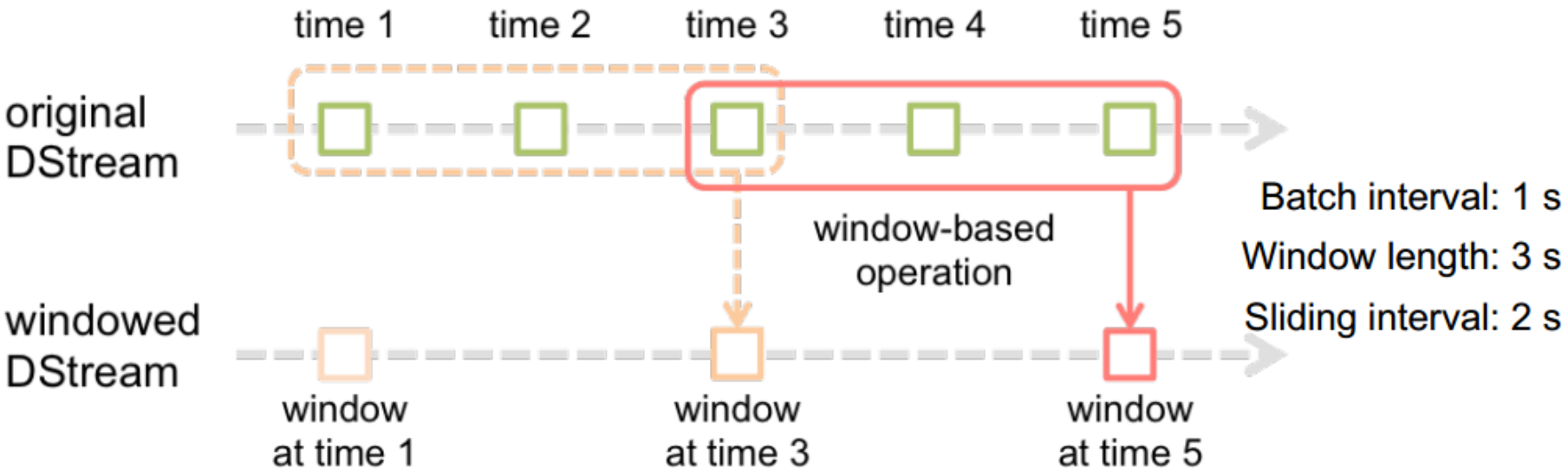
Transformation	Meaning
count	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream
countbyValue	Returns a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.
reduce(func)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function func (which takes two arguments and returns one).
reducebyKey(func)	When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function

Transformation

Transformation	Meaning
union(otherStream)	Return a new DStream that contains the union of the elements in the source DStream and otherDStream.
join(otherStream)	When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.

Window Operations

- Spark provides a set of transformations that apply to a sliding window of data
- A window is defined by: window length and siding interval



Window Operations

- *window(windowLength, slideInterval)*
  - Returns a new DStream which is computed based on windowed batches

- *countByWindow(windowLength, slideInterval)*
  - Returns a sliding window count of elements in the stream.

- *reduceByWindow(func, windowLength, slideInterval)*
  - Returns a new single-element DStream, created by aggregating elements in the stream over a sliding interval using func.

Output Operation

- Push out DStream's data to external systems, e.g., a database or a file system

Operation	Meaning
print	Prints the first ten elements of every batch of data in a DStream on the driver node running the application
saveAsTextFiles	Save this DStream's contents as text files
saveAsHadoopFiles	Save this DStream's contents as Hadoop files.
foreachRDD(func)	Applies a function, func, to each RDD generated from the stream

Example

Word Count

```
val context = new StreamingContext(conf, Seconds(1))

val lines = context.socketTextStream(...)

val words = lines.flatMap(_.split(" "))

val wordCounts = words.map(x => (x, 1)).reduceByKey(_+_)

wordCounts.print()

context.start()
```

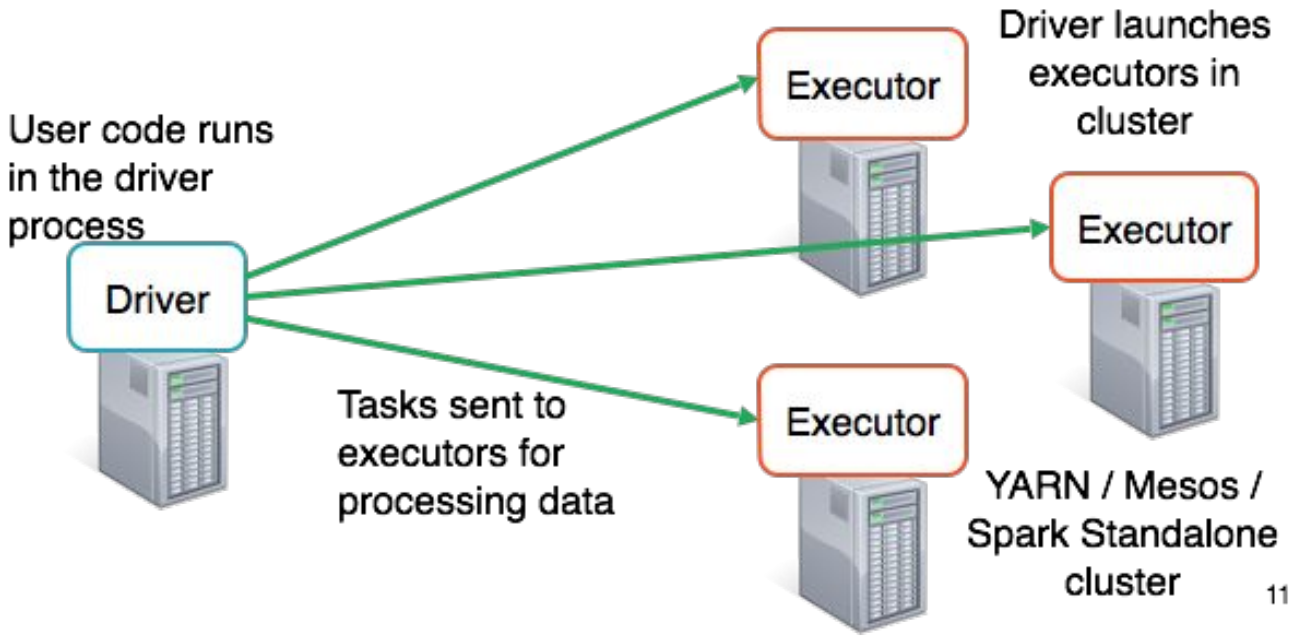
Print the DStream contents on screen

Start the streaming job

databricks

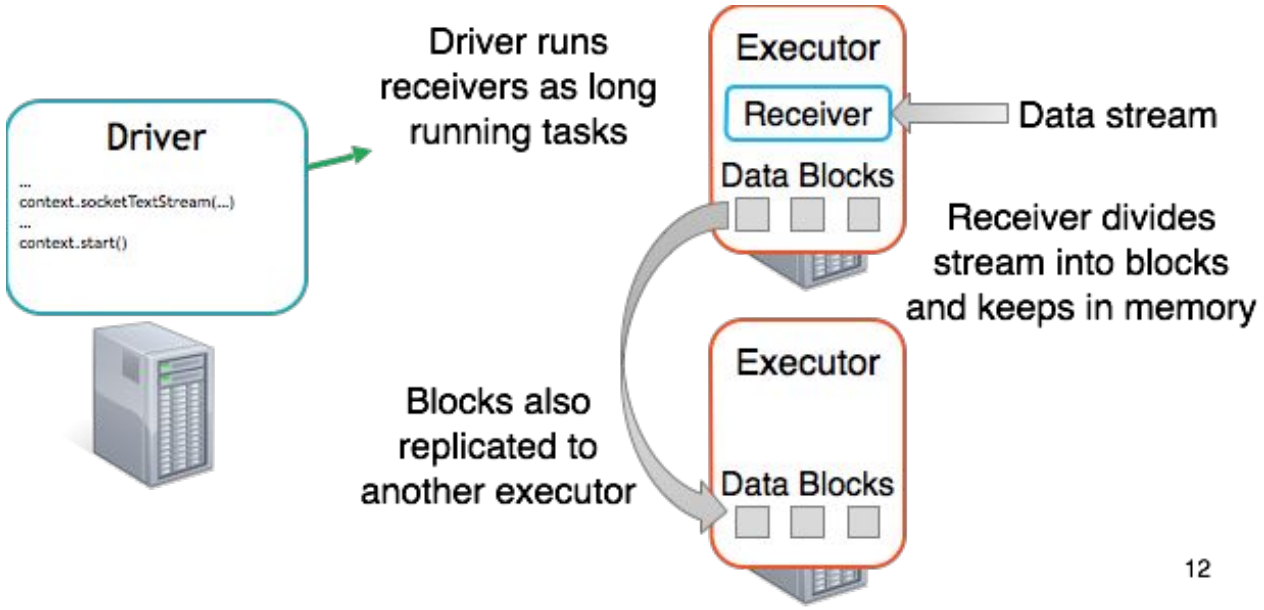
Lifecycle of a streaming app

Execution in any Spark Application



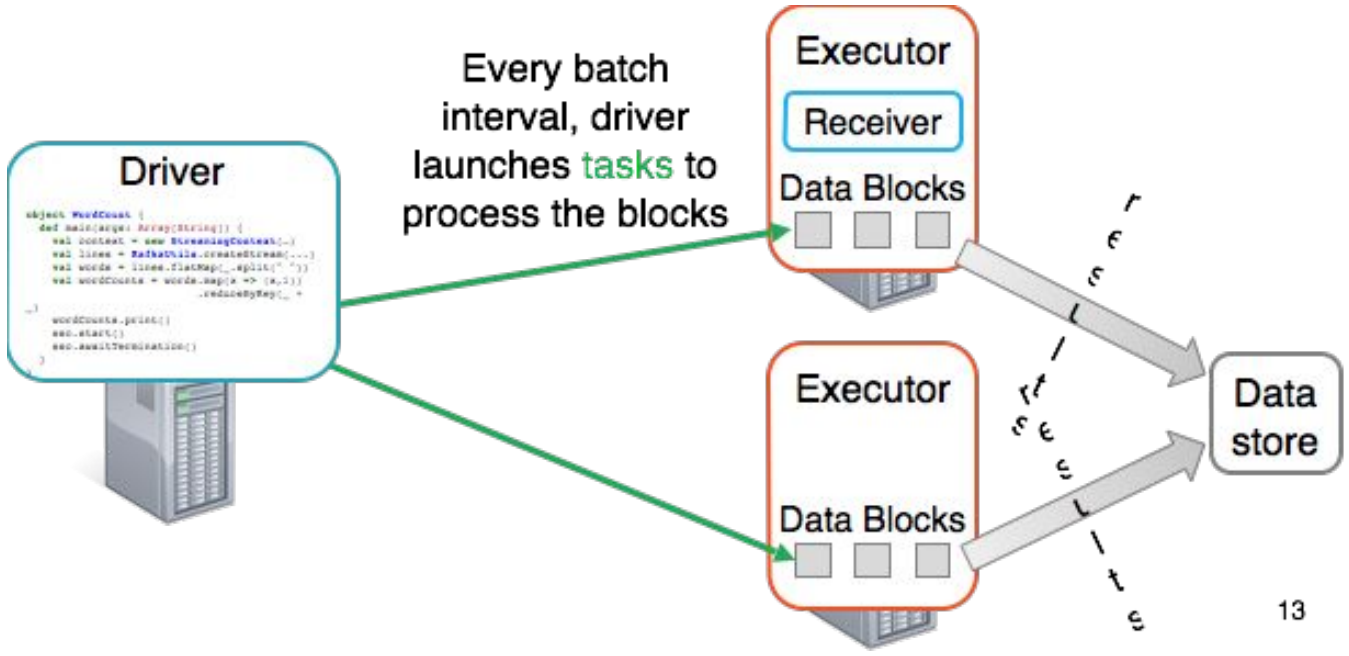
databricks

Execution in Spark Streaming: Receiving data



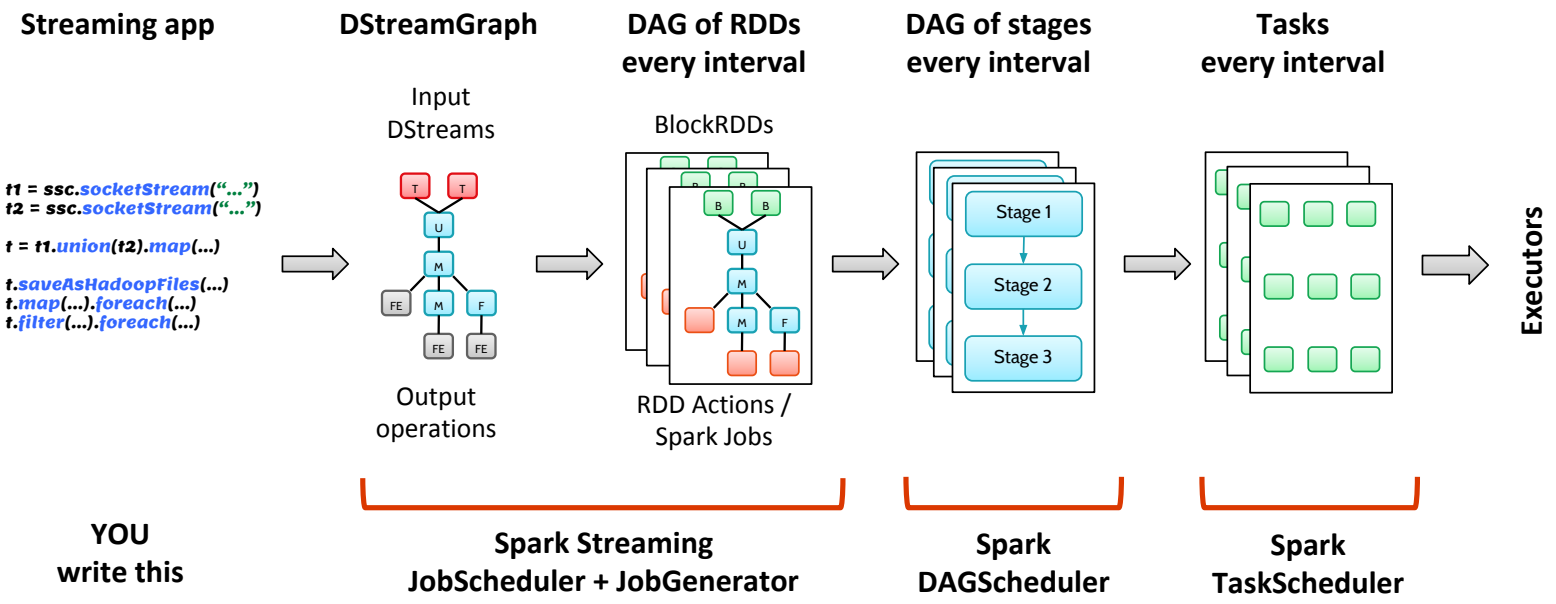
databricks

Execution in Spark Streaming: Processing data



databricks

End-to-end view



Dynamic Load Balancing

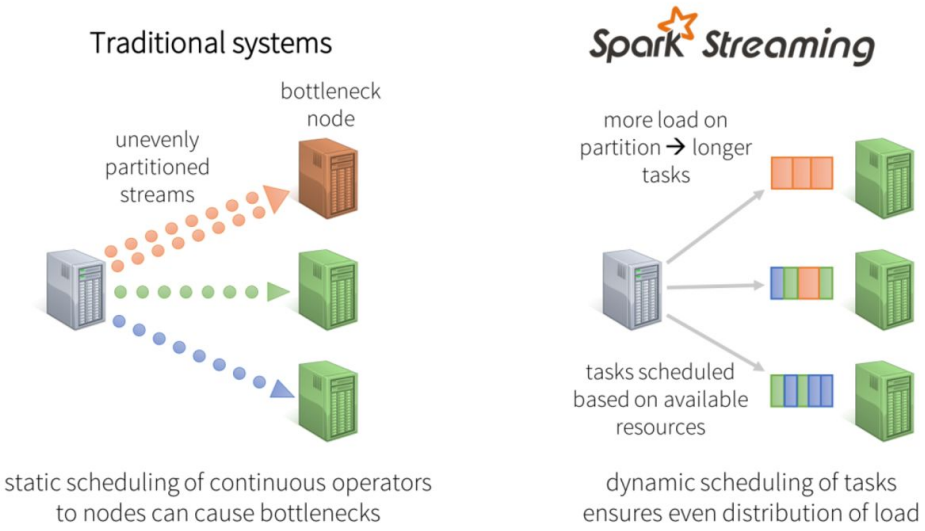


Figure 3: Dynamic load balancing

databricks

Fast failure and Straggler recovery

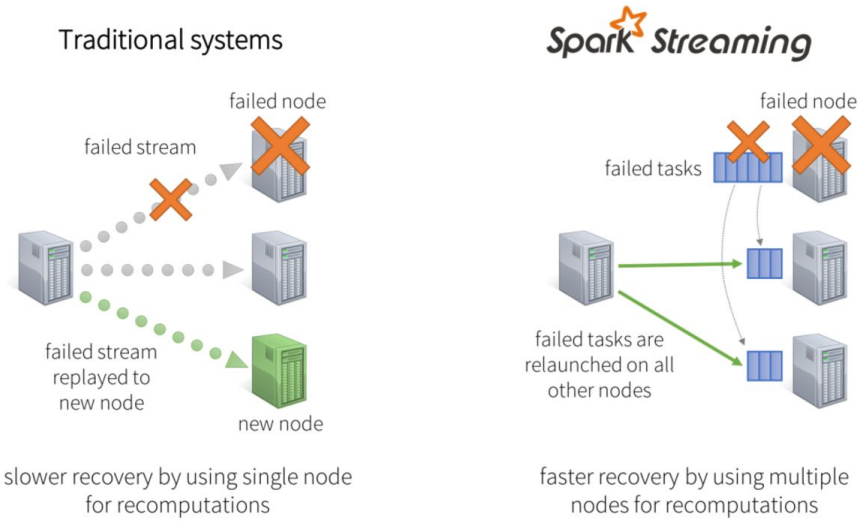


Figure 4: Faster failure recovery with redistribution of computation

databricks

Bài tập

- Ex1: Đưa ra danh sách các từ và số lần xuất hiện
- Ex2: Đưa ra danh sách các từ và số lần xuất hiện trong 30s gần nhất
- Ex3: Đưa ra danh sách các kí tự và số lần xuất hiện trong 30s gần nhất

Acknowledgement and References

Books:

- Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia. Learning Spark. Oreilly
- James A. Scott. Getting started with Apache Spark. MapR Technologies

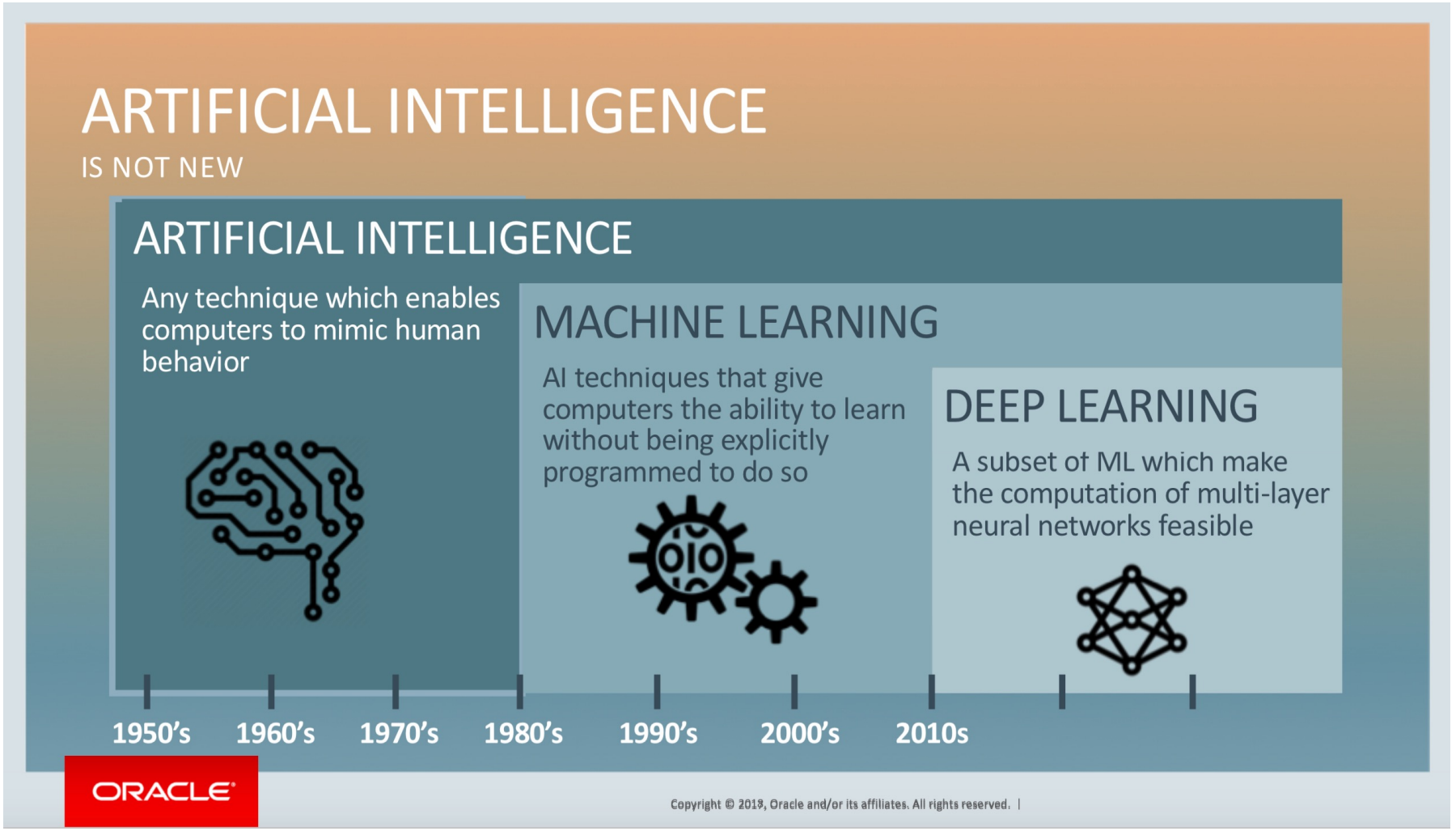
Slides:

- Amir H. Payberah. Scalable Stream Processing – Spark Streaming and Flink
- Matteo Nardelli. Spark Streaming: Hands on Session
- DataBricks. Spark Streaming
- DataBricks: Spark Streaming: Best Practices

IT4931
Tích hợp và xử lý dữ liệu lớn

06/2023
Thanh-Chung Dao Ph.D.

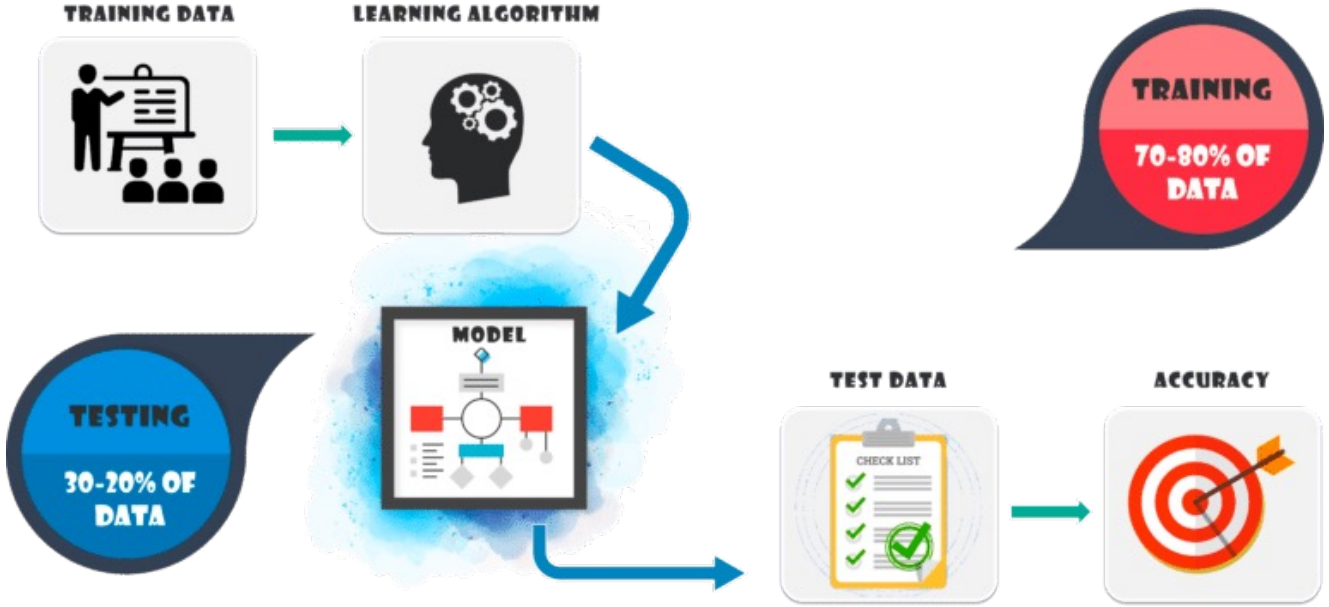
Machine learning



From [1]

Machine Learning Lifecycle

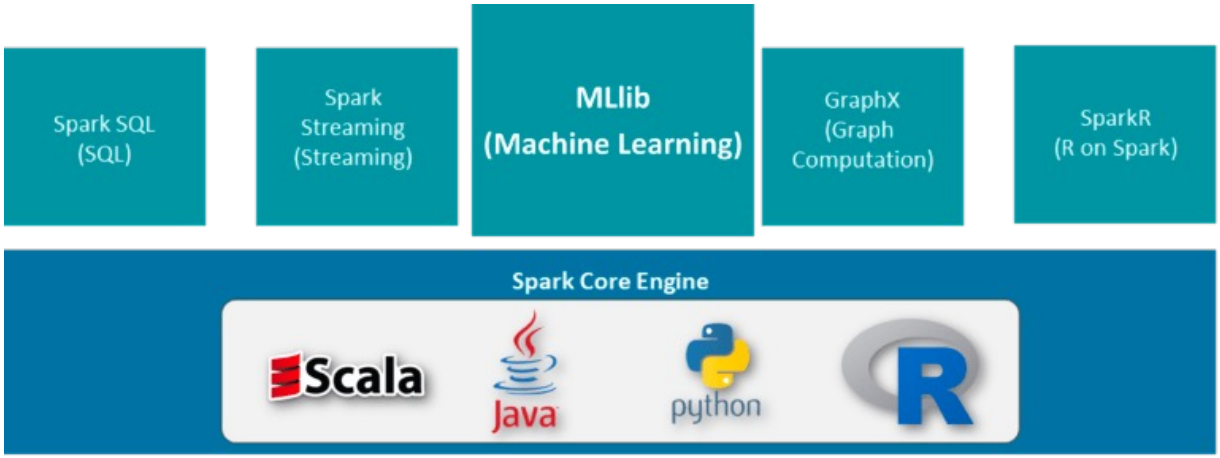
- Two major phases
  - **Training Set**
    - You have the complete training dataset
    - You can extract features and train to fit a model.
  - **Testing Set**
    - Once the model is obtained, you can predict using the model obtained on the training set



From [2]

Spark ML and PySpark

- Spark ML is a machine-learning library
  - Classification: logistic regression, naive Bayes
  - Regression: generalized linear regression, survival regression
  - Decision trees, random forests, and gradient-boosted trees
  - Recommendation: alternating least squares (ALS)
  - Clustering: K-means, Gaussian mixtures (GMMs)
  - Topic modeling: latent Dirichlet allocation (LDA)
  - Frequent item sets, association rules, and sequential pattern mining
- PySpark is an interface for using Python



From [2]

Binary Classification Example [3]

- **Binary Classification** is the task of predicting a binary label
  - Is an email spam or not spam?
  - Should I show this ad to this user or not?
  - Will it rain tomorrow or not?
- The Adult dataset
  - https://archive.ics.uci.edu/ml/datasets/Adult
  - 48842 individuals and their annual income
  - We will use this information to predict if an individual earns **<=50K or >50k** a year

Dataset Information

- Attribute Information:
  - age: continuous
  - workclass: Private,Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked
  - fnlwgt: continuous
  - education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc...
  - education-num: continuous
  - marital-status: Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent...
  - occupation: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners...
  - relationship: Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried
  - race: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black
  - sex: Female, Male
  - capital-gain: continuous
  - capital-loss: continuous
  - hours-per-week: continuous
  - native-country: United-States, Cambodia, England, Puerto-Rico, Canada, Germany...
- Target/Label: - <=50K, >50K

Analyzing Flow

- Load data
- Preprocess Data
- Fit and Evaluate Models
  - Logistic Regression
  - Decision Trees
  - Random Forest
- Make Classification

Lab: Running Binary Classification on Zeppelin

- Get the prepared notebook
- Run and try to understand algorithms

References

- [1] https://blogs.oracle.com/bigdata/difference-ai-machine-learning-deep-learning
- [2] https://www.edureka.co/blog/pyspark-mllib-tutorial/
- [3] https://docs.databricks.com/spark/latest/mllib/binary-classification-mllib-pipelines.html

IT4931
Tích hợp và xử lý dữ liệu lớn

06/2023
Thanh-Chung Dao Ph.D.

Spark ML

- Spark ML is a machine-learning library
  - Classification: logistic regression, naive Bayes
  - Regression: generalized linear regression, survival regression
  - Decision trees, random forests, and gradient-boosted trees
  - Recommendation: alternating least squares (ALS)
  - Clustering: K-means, Gaussian mixtures (GMMs)
  - Topic modeling: latent Dirichlet allocation (LDA)
  - Frequent item sets, association rules, and sequential pattern mining

Classification vs Prediction

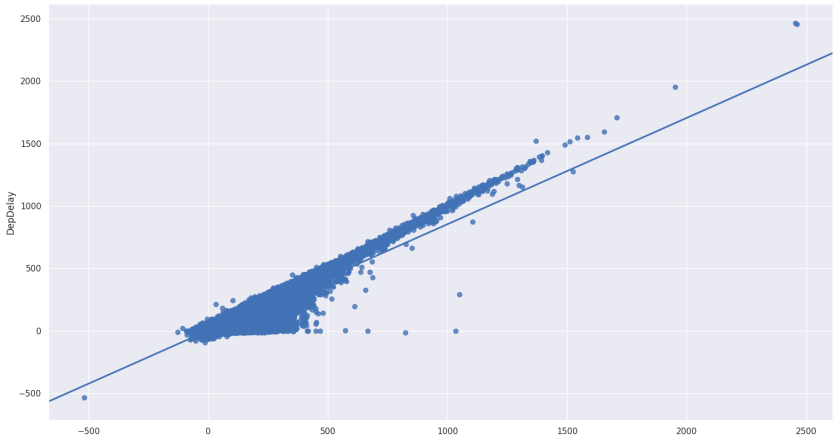
- Classification models predict categorical class labels [2]
  - Binary classification
- Prediction models predict continuous valued functions
  - Regression analysis is a statistical methodology that is most often used for numeric prediction

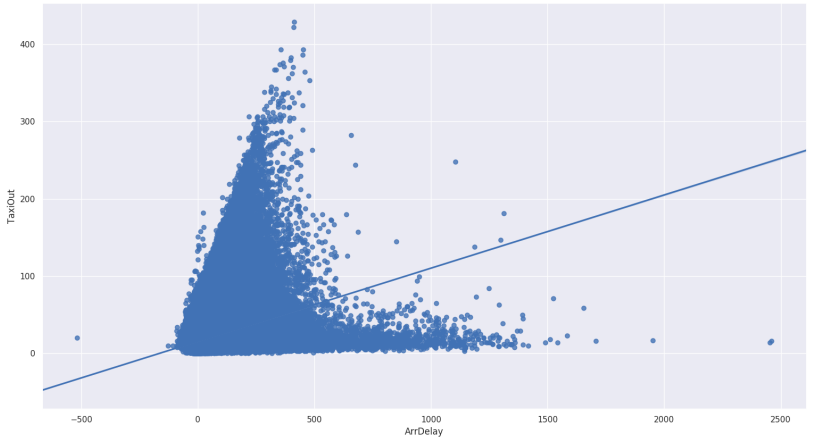
Predicting the arrival delay of commercial flights [1]

- Problem
  - We want to be able to predict, based on historical data
    - The arrival delay of a flight using only information available before the flight takes off
- Dataset
  - http://stat-computing.org/dataexpo/2009/the-data.html
  - The data used was published by the US Department of Transportation
  - It compromises almost 23 years worth of data
- Approach
  - Using a regression algorithm

Dataset Information

	Name	Description
1	Year	1987-2008
2	Month	1-12
3	DayofMonth	1-31
4	DayOfWeek	1 (Monday) - 7 (Sunday)
5	DepTime	actual departure time (local, hhmm)
6	CRSDepTime	scheduled departure time (local, hhmm)
7	ArrTime	actual arrival time (local, hhmm)
8	CRSArrTime	scheduled arrival time (local, hhmm)
9	UniqueCarrier	unique carrier code
10	FlightNum	flight number
11	TailNum	plane tail number
12	ActualElapsedTime	in minutes
13	CRSElapsedTime	in minutes
14	AirTime	in minutes
15	ArrDelay	arrival delay, in minutes
16	DepDelay	departure delay, in minutes





Analyzing Flow

- Load data
- Preprocess Data
- Train the data and obtain a model
- Evaluate the resulting model
- Make Predictions

Lab: Running Prediction of Flight Delay on Zeppelin

- Write code using PySpark
  - Get the prepared notebook
- Run and try to understand algorithms
- The original source code (in Scala)
  - https://github.com/pedroduartecosta/Spark-PredictFlightDelay

References

- [1] https://medium.com/@pedrodc/building-a-big-data-machine-learning-spark-application-for-flight-delay-prediction-4f9507cdb010
- [2] https://www.tutorialspoint.com/data_mining/dm_classification_prediction.htm

IT4931
Tích hợp và xử lý dữ liệu lớn

06/2023
Thanh-Chung Dao Ph.D.

GraphX

- Apache Spark's API for graphs and graph-parallel computation
- GraphX unifies ETL (Extract, Transform & Load) process
- Exploratory analysis and iterative graph computation within a single system

Use cases

- Facebook's friends, LinkedIn's connections
- Internet's routers
- Relationships between galaxies and stars in astrophysics and Google's Maps
- Disaster detection, banking, stock market

RDD on GraphX

- GraphX extends the Spark RDD with a Resilient Distributed Property Graph
- The property graph is a directed multigraph which can have multiple edges in parallel
- The parallel edges allow multiple relationships between the same vertices

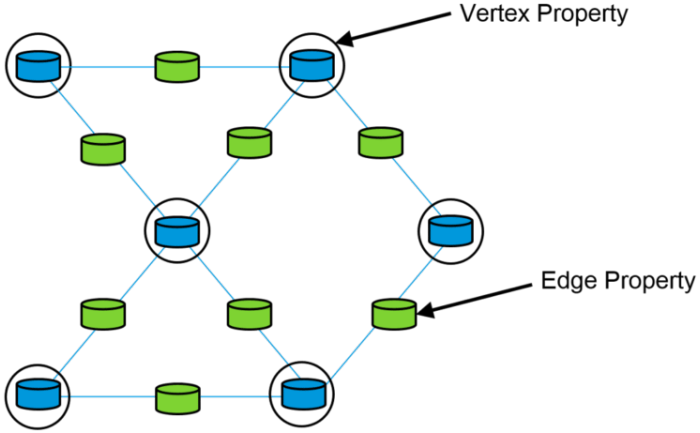


Figure: *Property Graph*

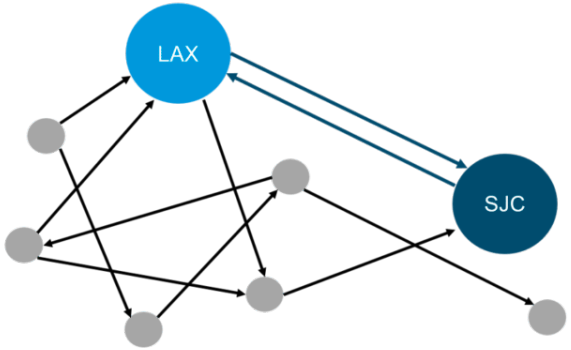


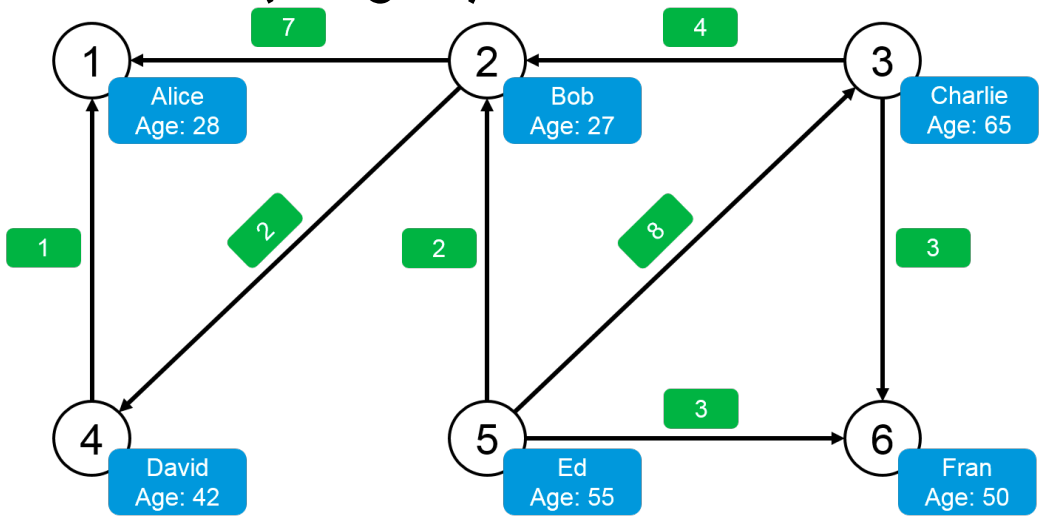
Figure: *An example of property graph*

Spark GraphX Features

- **Flexibility**
  - Spark GraphX works with both graphs and computations
  - GraphX unifies ETL (Extract, Transform & Load), exploratory analysis and iterative graph computation
- **Speed**
  - The fastest specialized graph processing systems
- **Growing Algorithm Library**
  - Page rank, connected components, label propagation, SVD++, strongly connected components and triangle count

GraphX with Examples

- The graph here represents the Twitter users and whom they follow on Twitter. For e.g. Bob follows Davide and Alice on Twitter
- Looking at the graph, we can extract information about the people (vertices) and the relations between them (edges)



Source code

```
1  //Importing the necessary classes
2  import org.apache.spark._
3  import org.apache.spark.rdd.RDD
4  import org.apache.spark.util.IntParam
5  import org.apache.spark.graphx._
6  import org.apache.spark.graphx.util.GraphGenerators
```

Displaying Vertices: Further, we will now display all the names and ages of the users (vertices).

```
1  val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)
2  val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)
3  val graph: Graph[(String, Int), Int] = Graph(vertexRDD, edgeRDD)
4  graph.vertices.filter { case (id, (name, age)) => age > 30 }
5  .collect.foreach { case (id, (name, age)) => println(s"$name is $age")}
```

The output for the above code is as below:

```
David is 42
Fran is 50
Ed is 55
Charlie is 65
```

More source code

Displaying Edges: Let us look at which person likes whom on Twitter.

```
1  for (triplet <- graph.triplets.collect)
2  {
3  println(s"${triplet.srcAttr._1} likes ${triplet.dstAttr._1}")
4  }
```

The output for the above code is as below:

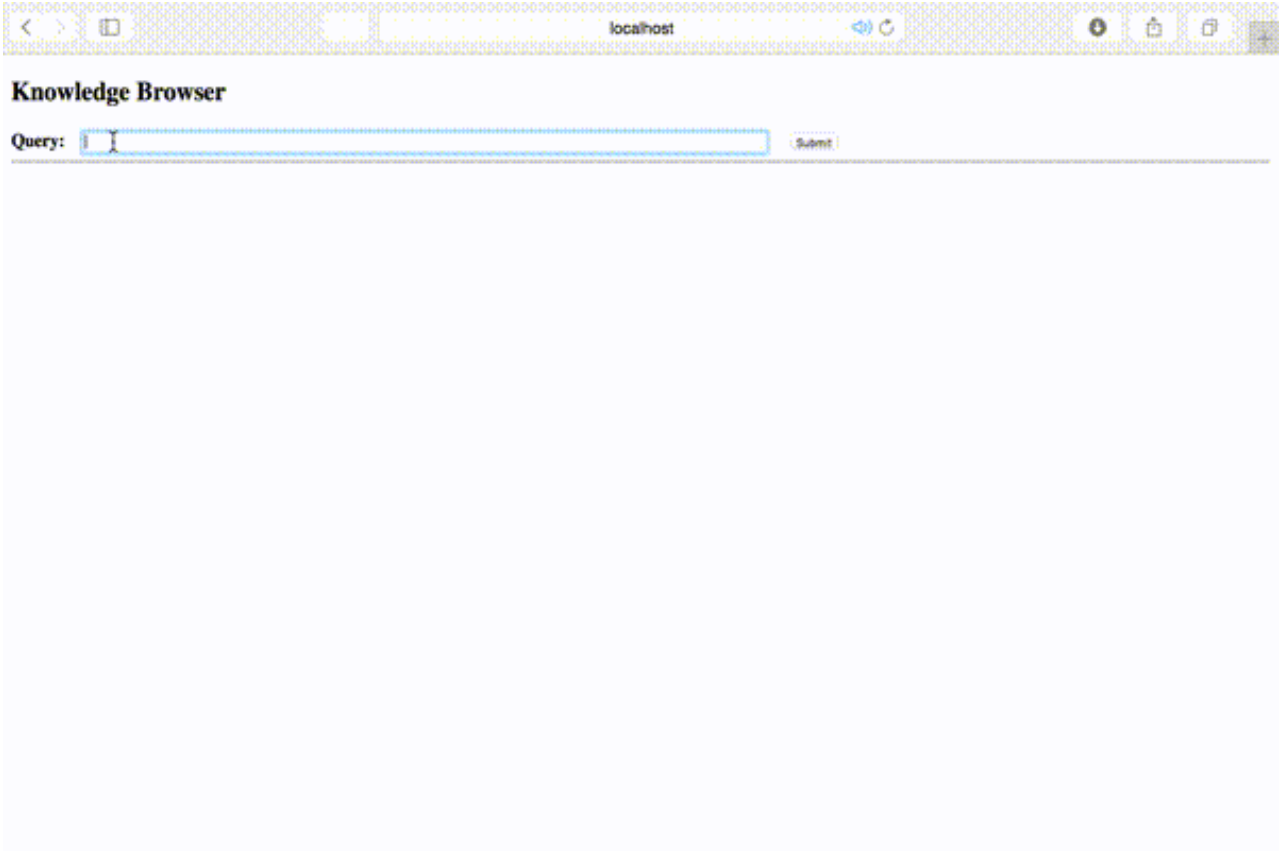
```
Bob likes Alice
Bob likes David
Charlie likes Bob
Charlie likes Fran
David likes Alice
Ed likes Bob
Ed likes Charlie
Ed likes Fran
```

Other example in PySpark

```
## pyspark --packages graphframes:graphframes:0.6.0-spark2.2-s_2.11
from graphframes import *
from pyspark import *
from pyspark.sql import *
spark = SparkSession.builder.appName('fun').getOrCreate()
vertices = spark.createDataFrame([('1', 'Carter', 'Derrick', 50),
                                  ('2', 'May', 'Derrick', 26),
                                 ('3', 'Mills', 'Jeff', 80),
                                  ('4', 'Hood', 'Robert', 65),
                                  ('5', 'Banks', 'Mike', 93),
                                 ('98', 'Berg', 'Tim', 28),
                                 ('99', 'Page', 'Allan', 16)],
                                 ['id', 'name', 'firstname', 'age'])
edges = spark.createDataFrame([('1', '2', 'friend'),
                               ('2', '1', 'friend'),
                              ('3', '1', 'friend'),
                              ('1', '3', 'friend'),
                               ('2', '3', 'follows'),
                               ('3', '4', 'friend'),
                               ('4', '3', 'friend'),
                               ('5', '3', 'friend'),
                               ('3', '5', 'friend'),
                               ('4', '5', 'follows'),
                              ('98', '99', 'friend'),
                              ('99', '98', 'friend')],
                              ['src', 'dst', 'type'])
g = GraphFrame(vertices, edges)
## Take a look at the DataFrames
g.vertices.show()
g.edges.show()
## Check the number of edges of each vertex
g.degrees.show()
```

Spark Knowledge Graph

- Example: https://github.com/spoddutur/graph-knowledge-browser



Acknowledgement and References

Books:

Slides:

- https://www.edureka.co/blog/spark-graphx/