MA-INF 4223 - Lab Distributed Big Data Analytics

Spark Fundamentals I

Dr. Hajira Jabeen, Gezim Sejdiu, Denis Lukovnikov

Summer Semester 2019

Lesson objectives

- After completing this lesson, you should be able to:
  - Justify the purpose of Spark
  - List and describe Spark Stack components
  - Understand the basics of Resilient Distributed Dataset (RDD) , Data Frames and GraphX
  - Download and configure Spark- standalone
  - Scala overview
  - Launch and use Spark’s Scala shell

Shortcomings of Mapreduce

- Force your pipeline into a Map and a Reduce steps
  - You might need to perform:
    - join
    - filter
    - map-reduce-map
- Read from disk for each Map/Reduce job
  - Expensive for some type of algorithm i.e:
    - Iterative algorithm : Machine learning

Shortcomings of Mapreduce

Solution?

- ❖ New framework: Support the same features of MapReduce and many more.
- ❖ Capable of reusing Hadoop ecosystem : e.g HDFS, YARN, etc.

- ❖ Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

Apache Spark

Spark

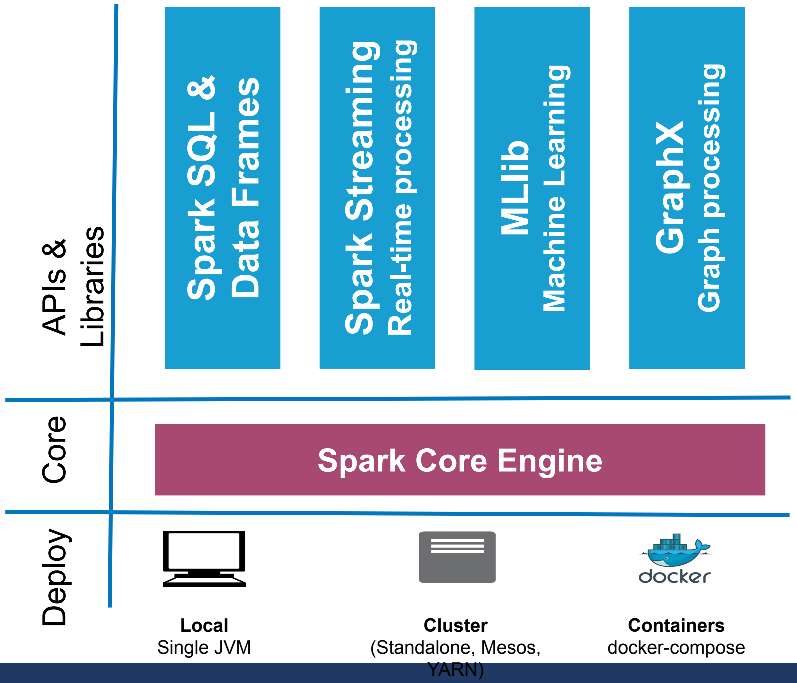
- Zaharia, Matei, et al. "**Spark: Cluster Computing with Working Sets**." HotCloud 10.10-10 (2010): 95.
- Zaharia, Matei, et al. "**Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing**." Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.
- Shi, Juwei, et al. "**Clash of the titans: Mapreduce vs. spark for large scale data analytics**." Proceedings of the VLDB Endowment 8.13 (2015): 2110-2121.

Introduction to Spark

- Apache Spark is an open-source **distributed** and highly scalable in-memory data processing and analytics system. It is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Scala, Java, Python and R which and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.

Introduction to Spark

Spark Stack – A unified analytics stack



Brief History of Spark

- Originally developed on UC Berkeley AMPLab in 2009.
- open-sourced in 2010.
- Spark paper came out.
- Spark Streaming was incorporated as a core component in 2011.
- In 2013, Spark was transferred to the Apache Software foundation.
- As of 2014, Spark is a top-level Apache project.
- July 26th, 2016 Spark 2.0 released with major improvements in every area over the 1.x branches

Who uses Spark and why?

Data Scientists:

- ➢ Analyze and model data.
- ➢ Data transformations and prototyping.
- ➢ Statistics and Machine Learning.

Software Engineers:

- ➢ Data processing systems.
- ➢ API for distributed processing dataflow.
- ➢ Reliable, high performance and easy to monitor platform.

Spark: Overview

- Spark provides parallel distributed processing, fault tolerance on commodity hardware,
- Focus : Applications that reuse a working set of data across multiple parallel operations
  - Iterative Machine learning
  - Interactive Data analysis
  - Systems like Pregel, did not provide ad hoc processing, or general reuse
  - General purpose programming interface
  - Resilient distributed datasets (RDDs)
    - Read only, persistent

RDDs - Resilient Distributed Dataset

- Enable efficient data reuse in a broad range of applications
- Fault tolerant, parallel Data Structures
- Users can
  - Persist the RDDs in Memory
  - Control partitioning
  - Manipulate using rich set of operations
- Coarse-grained transformations
  - Map, Filter, Join, applied to many data items concurrently
  - Keeps the lineage

RDDs

- RDD is represented by a Scala object
- Created in 4 ways
  - From a file in a shared file system
  - By “parallelizing” a Scala collection
  - Transforming an existing RDD
  - Changing the persistence of an existing RDD
    - Cache (hint, spill otherwise)
    - Save (to HDFS)

RDD

- RDDs can only be created through transformations
  - Immutable,
  - no need of checkpointing
  - only lost partitions need to be recomputed
- Stragglers can be mitigated by running backup copies
- Tasks scheduled based on data locality
- Degrade gracefully by spilling to disk
- Not suitable to asynchronous updates to shared state

Parallel Operations on RDDs

- Reduce
  - Combines dataset elements using an associative function to produce a result at the driver program.
- Collect
  - Sends all elements of the dataset to the driver program
- Foreach
  - Passes each element through a user provided function

Shared Variables

- Broadcast Variables
  - To Share a large read-only piece of data to be used in multiple parallel operations
- Accumulators:
  - These are variables that workers can only “add” to using an associative operation, and that only the driver can read.

MapReduce Vs Spark

- Shuffle: data is shuffled between computational stages, often involving sort, aggregation and combine operations.
- Execution model: parallelism among tasks, overlapping computation, data pipelining among stages
- Caching: reuse of intermediate data across stages at different levels

WordCount

- Spark is 3x faster
- Fast initialization; hash-based combine better than sort-based combine
- For low shuffle selectivity workloads, hash-based aggregation in Spark is more efficient than sort-based aggregation in MapReduce due to the complexity differences in its in memory collection and combine components.

PageRank

- For graph analytic algorithms such as BFS and Community Detection that read the graph structure and iteratively exchange states through a shuffle, compared to MapReduce,
- Spark can avoid materializing graph data structures on HDFS across iterations, which reduces overheads for serialization/de-serialization, disk I/O, and network I/O.

Sort

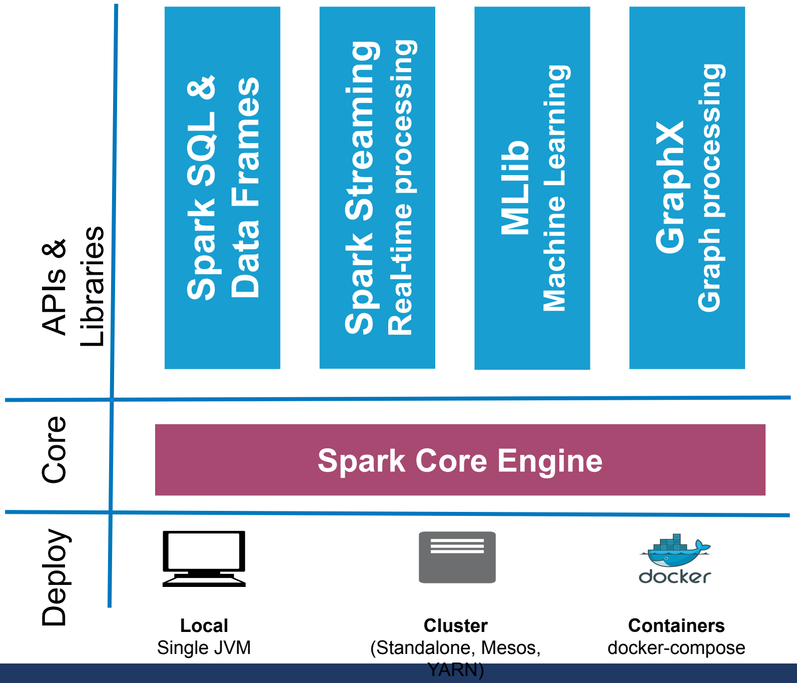
- In MapReduce, the reduce stage is faster than Spark because MapReduce can overlap the shuffle stage with the map stage, which effectively hides the network overhead.
- In Spark, the execution time of the map stage increases as the number of reduce tasks increase. This overhead is caused by and is proportional to the number of files opened simultaneously.

Iterative Algorithms

- CPU-bound iteration *caching the raw file* (to reduce disk I/O) may not help reduce the execution time since the disk I/O is hidden by the CPU overhead.
- In disk-bound, caching the raw file can significantly reduce the execution time.
- RDD caching can reduce not only disk I/O, but also the <u>CPU overhead</u> since it can cache any intermediate data during the analytic pipeline. For example, the main contribution of RDD caching for k-means is to cache the Point object to save the transformation cost from a text line, which is the bottleneck for each iteration.

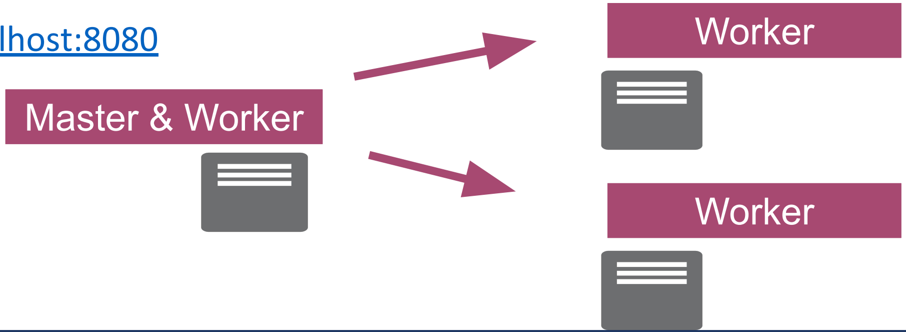
Spark Stack

Spark Stack – A unified analytics stack



Installing Spark Standalone

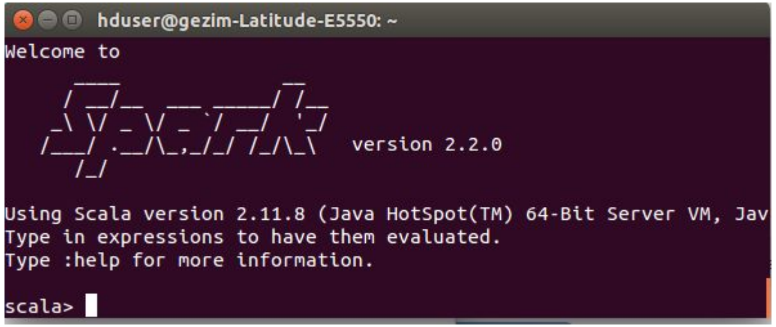
- ❖ Spark runs on Windows and Linux-like operating systems.
- ❖ Download the Hadoop distribution you require under "Pre-build packages"
  - ➢ Place that compiled version on each node on the cluster
  - ➢ Run ./sbin/start-master.sh to start a cluster
  - ➢ Once started, the master will show the spark://HOST:PORT url which you may need to connect workers to it.
    - ■ Spark Master UI : http://localhost:8080



Check out Spark's website for more information.

Spark jobs and shell

- Spark jobs can be written in Scala, Java, Python or R.
- Spark native language is Scala, so it is natural to write Spark applications using Scala.
- The course will cover code examples written in Scala.

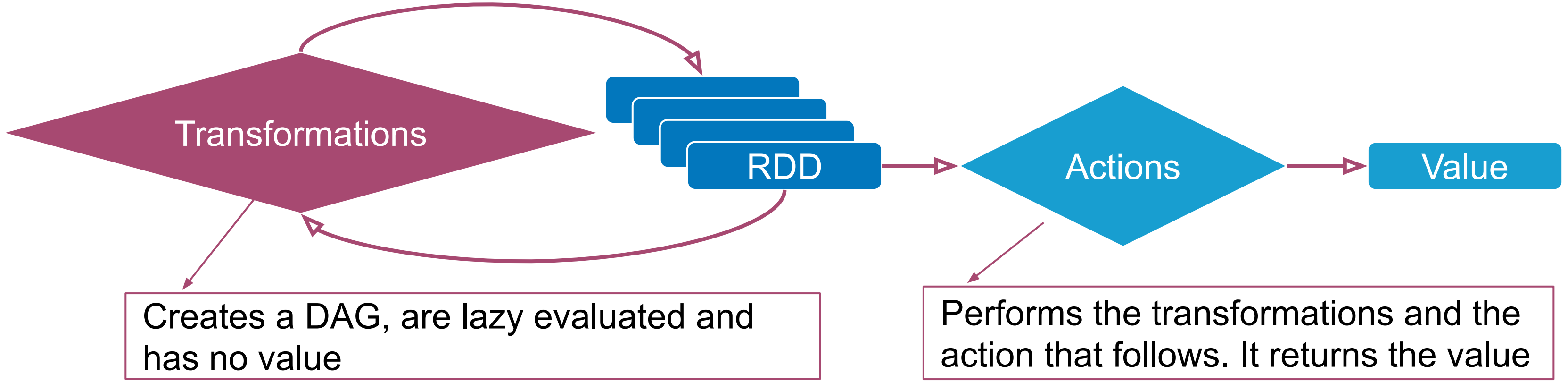


```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Jav
Type in expressions to have them evaluated.
Type :help for more information.

scala> 
```

Resilient Distributed Dataset (RDD)

Resilient Distributed Dataset (RDD)

- Spark's primarily abstraction.
- Distributed collection of elements, partitioned across the cluster.
  - **Resilient**: recreated, when data in-memory is lost.
  - **Distributed**: partitioned in-memory across the cluster
  - **Dataset**: list of collection or data that comes from file.



Creates a DAG, are lazy evaluated and has no value

Performs the transformations and the action that follows. It returns the value

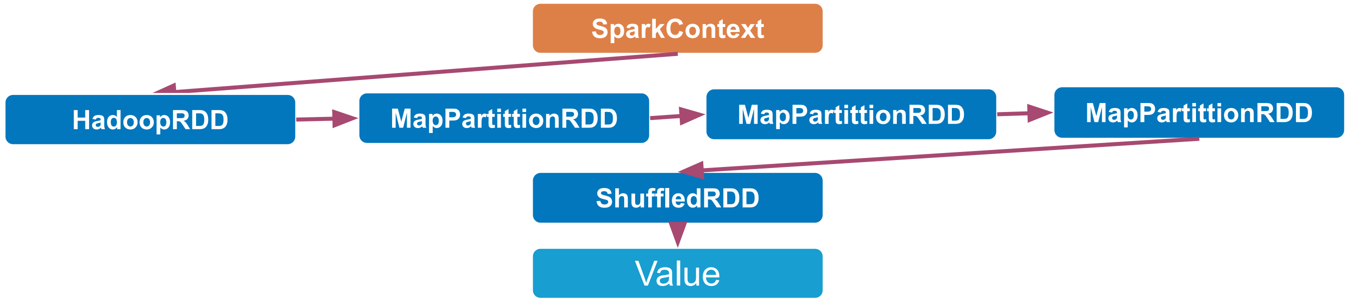
Creating RDDs

- Launch the Spark shell
  - `./bin/spark-shell` *//sc: SparkContext instance*
- Create some sample data and parallelize by creating and RDD
  - `val data = 1 to 1000`
  - `val distData =sc.parallelize(data)`
- Afterwards, you could perform any additional transformation or action on top of these RDDs:
  - `distData.map { x => ??? }`
  - `distData.filter { x => ??? }`
- An RDD can be created by external dataset as well:
  - `val readmeFile = sc.textFile("README.md")`

RDD Operations

Word Count example

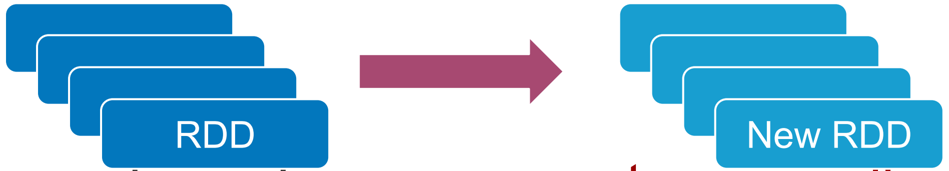
```
val textFile = sparkSession.sparkContext.textFile("hdfs://...")
val wordCounts = textFile.flatMap(line => line.split(" "))
                         .filter(!_.isEmpty())
                         .map(word => (word, 1))
                         .reduceByKey(_ + _) //(a, b) => a + b
wordCounts.take(10)
```



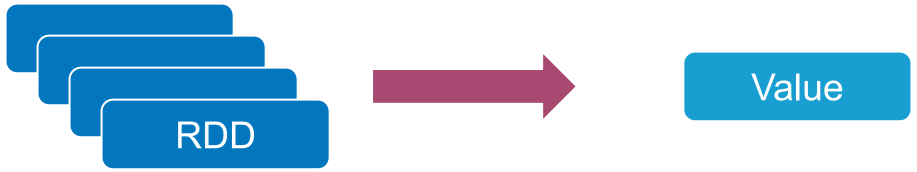
Directed Acyclic Graph (DAG) for Word Count example

RDD Operations

- **Transformations**: Return new RDDs based on existing one (f(RDD) => RDD), e.g filter, map, reduce, groupBy, etc.



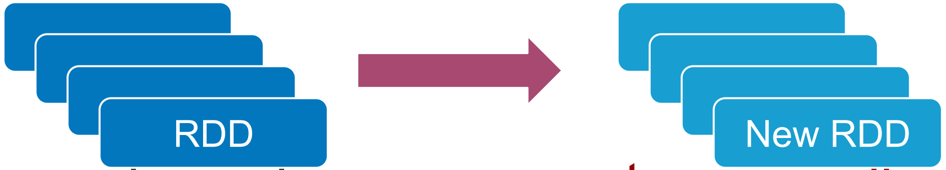
- **Actions**: Computes values, e.g count, sum, collect, take, etc.
  - Either returned or saved to HDFS



RDD Operations

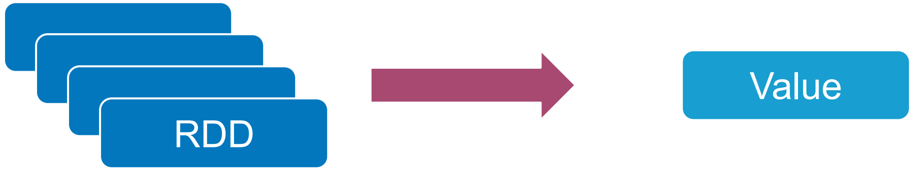
Lazy

- **Transformations**: Return new RDDs based on existing one (f(RDD) => RDD), e.g filter, map, reduce, groupBy, etc.



Eager

- **Actions**: Computes values, e.g count, sum, collect, take, etc.
  - Either returned or saved to HDFS



RDD Transformations (Lazy)

- **map:**Apply function to each element in the RDD and return an RDD of the result.
- **flatMap** Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned
- **Filter**: Apply predicate function to each element in the RDD and return an RDD of elements that have passed the predicate condition, pred.
- **distinct:**Return RDD with duplicates removed.

RDD Actions (Eager)

TakeSample

takeOrdered

saveAsTextFile

saveAsSequenceFile

RDD Actions (Eager)

collect: Return all elements from RDD.

count: Return the number of elements

take(n) Return the first n elements of the RDD.

reduce:Combine the elements in the RDD together using an “op” function and return the result.

foreach:Apply a function to each element in the RDD

Transformations on Two RDDs (Lazy)

union: Return an RDD containing elements from both RDDs.

intersection: Return an RDD containing elements only found in both RDDs

subtract: Return an RDD with the contents of the other RDD removed

cartesian:Cartesian product with the other RDD.

RDD Operations

Expressive and Rich API

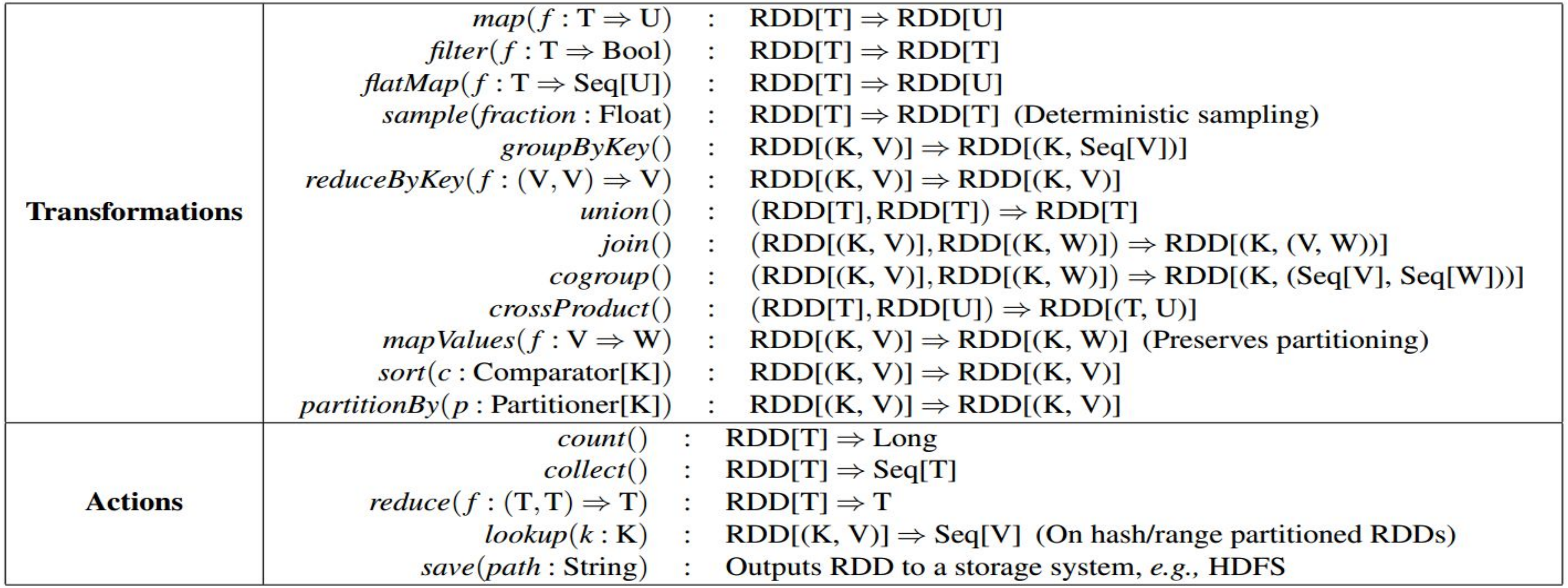
map
filter
groupBy
sort
union
join
leftOuterJoin
rightOuterJoin

reduce
count
fold
reduceByKey
groupByKey cogroup
cross
zip

sample
take
first
partitionBy mapWith
pipe
save
...

RDD Operations

Transformations and actions available on RDDs in Spark.



Transformations	*map*(f : T ⇒ U)	RDD[T] ⇒ RDD[U]
	filter(f : T ⇒ Bool)	RDD[T] ⇒ RDD[T]
	flatMap(f : T ⇒ Seq[U])	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(f : (V, V) ⇒ V)	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(f : V ⇒ W)	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(c : Comparator[K])	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(p : Partitioner[K])	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	RDD[T] ⇒ Long
	collect()	RDD[T] ⇒ Seq[T]
	reduce(f : (T, T) ⇒ T)	RDD[T] ⇒ T
	lookup(k : K)	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	Outputs RDD to a storage system, *e.g.*, HDFS

Pair RDDs

- Transformations
  - groupByKey
  - reduceByKey
    - Only values of Keys are used for the Grouping
    - More performant
  - mapValues
    - Applies a function to only values in a PairRDD

Pair RDDs

- Join
  - Inner join, lossy, **only** returns the values whose keys occur in both RDDs
- leftOuterJoin/rightOuterJoin

- Actions
  - countByKey
    - Counts the number of elements per key , returns a regular map, mapping keys to count

Shuffling

- The shuffle is Spark’s mechanism for re-distributing data so that it is grouped differently across partitions.
- This typically involves copying data across executors and machines, making the shuffle a **complex and costly operation.**
- Certain operations within Spark trigger an event known as the shuffle.
- GroupbyKey can cause shuffling

groupByKey and reduceByKey differ in their internal operations

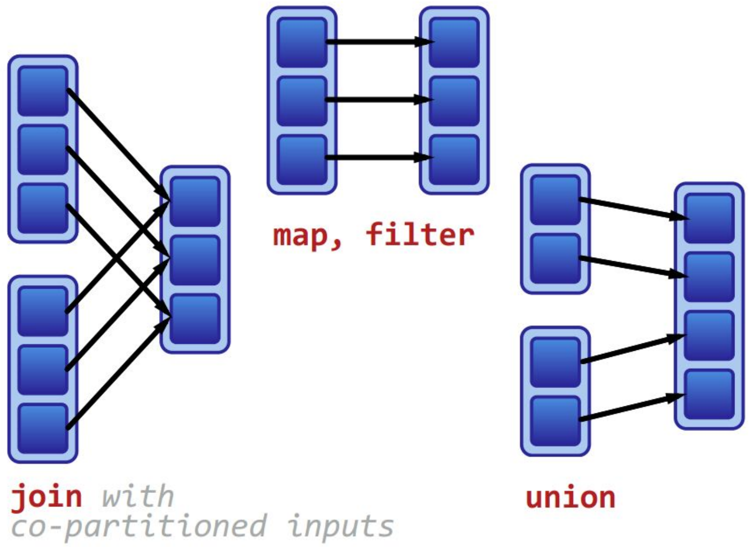
Dependencies / Shuffling

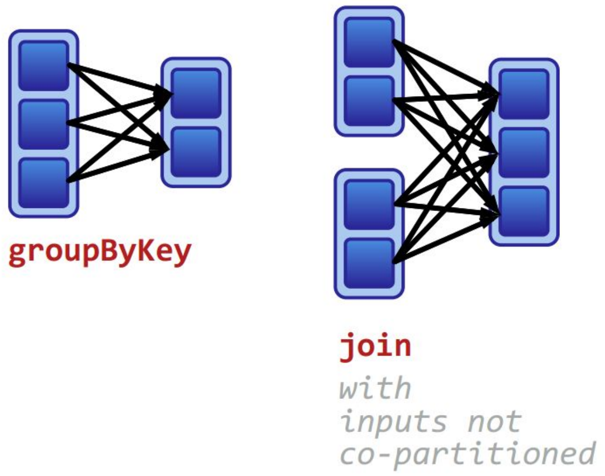
Narrow dependencies:

Each partition of the parent RDD is used by at most one partition of the child RDD.

Wide dependencies:

Each partition of the parent RDD may be depended on by multiple child partitions.





Partitioning

- One partitioning on one machine, or a machine can have many, depending on cores
  - Default = number of cores
- Hash-
  - Hash on key and modulo core size
- Range
  - keys that can have an ordering
- Custom Partitioning , based on keys
    - Only on Pair RDDs

partitionBy

- Range partition
  - Number of partitions
  - PairRdd with ordered keys
- Always **persist**
- Or data will be shuffled in each iteration

Partitioning using transformations

- Partitioner from Parent RDD
  - The RDD that is the result of a transformation on parent RDD usual configured to use the same partitioner as parent
- Automatically set Partitioners
  - e.g. sortByKey uses RangePartitioner
  - groupByKey uses HashPartitioner
- Map and Flatmap loose the partitioner as we can change the key itself
- Use mapValues instead !!

Spark SQL

Motivation

- Support relational processing both within Spark programs
- Provide high performance with established DBMS techniques
- Easily support new data sources, including semi-structured data and external databases amenable to query federation
- Enable extension with advanced analytics algorithms such as graph processing and machine learning

Motivation

- Users:
  - Want to perform ETL-relational processing
    - data Frames
  - Analytics - procedural tasks
    - UDFs

Spark SQL

- A module that integrates relational processing with Spark’s Functional programming API
- Spark SQL allows relational processing
- Perform complex analytics
  - Integration between relational and procedural processing through declarative Data Frame
  - Optimizer (catalyst)
    - Composable rules
    - Control code generation
    - Extension points
    - Schema inference for json
    - ML types
    - Query federation

Spark SQL

Three main APIs

- SQL Syntax
- DataFrames
- Datasets

Two specialised backend components

- Catalyst
- Tungsten

Data Frame

- DataFrames are collections of structured records that can be manipulated using Spark’s procedural API,
- Supports relational APIs that allow richer optimizations.
- Created directly from Spark’s built-in distributed collections of Java/Python objects,
- Enables relational processing in existing Spark Programs
- DataFrame operations in SparkSQL go through a relational optimizer, Catalyst

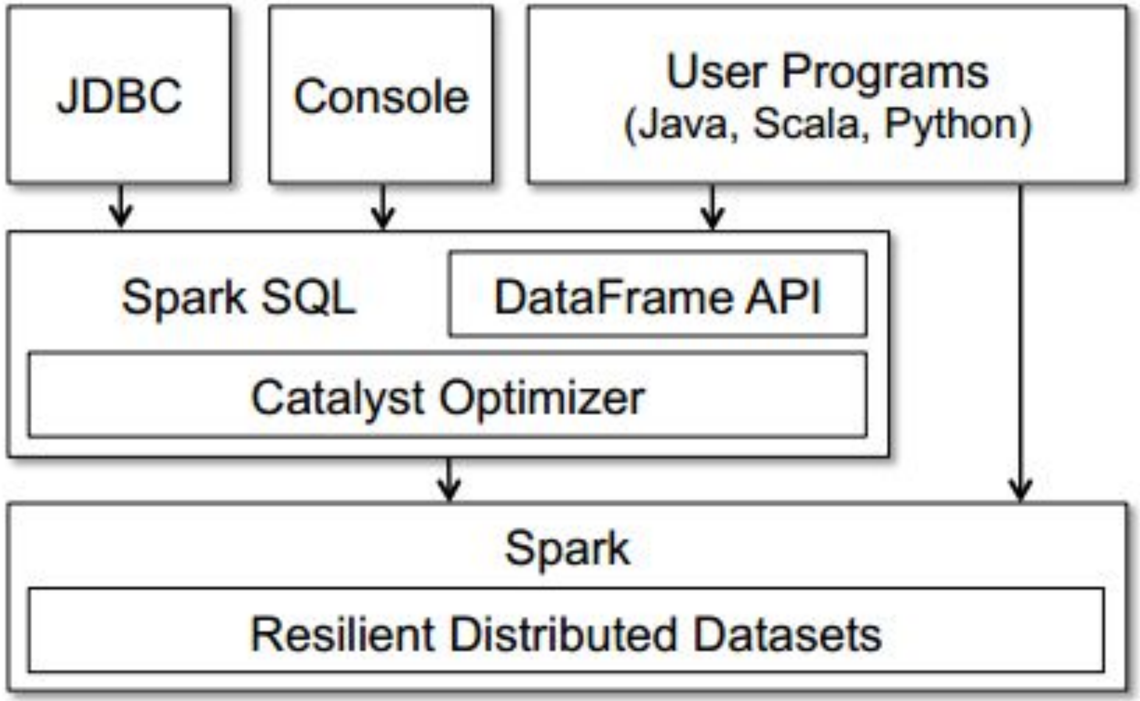
Catalyst

- Catalyst is the first production quality query optimizer built on such functional language.
- It contains an extensible query optimizer
- Catalyst uses features of the Scala programming language,
  - Pattern-matching
  - Express composable rules
  - Turing complete language

Catalyst

- Catalyst can also be
  - extended with new data sources,
  - semi-structured data
    - such as JSON
    - “smart” data stores to use push filters
    - e.g., HBase
    - user-defined functions;
    - User-defined types for domains e.g. machine learning.
- Spark SQL simultaneously makes Spark accessible to more users and improves optimizations

Spark SQL


JDBC
Console
User Programs (Java, Scala, Python)
Spark SQL
DataFrame API
Catalyst Optimizer
Spark
Resilient Distributed Datasets

DataFrame

- DataFrame is a distributed collection of rows with the “Known” schema like table in a relational database.
- Each DataFrame can also be viewed as an RDD of Row objects, allowing users to call procedural Spark APIs such as map.
- Spark DataFrames are lazy, in that each DataFrame object represents a logical plan to compute a dataset, but no execution occurs until the user calls a special “output operation” such as save

DataFrame

- Created from an RDD using .toDF()
- Reading from a file ()

Example

- ctx = new HiveContext()
- users=ctx.table("users")
- young = users.where(users("age")<21)
- println(young.count())

Data Model

- DataFrames support all common relational operators, including
  - projection (select),
  - filter (where),
  - join, and
  - aggregations (groupBy).
- Users can break up their code into Scala, Java or Python functions that pass DataFrames between them to build a logical plan, and will still benefit from optimizations across the whole plan when they run an output operation.

Optimization

- The API analyze logical plans eagerly
  - identify whether the column names used in expressions exist in the underlying tables,
  - whether the data types are appropriate
- Spark SQL allows users to construct DataFrames directly against RDDs of objects native to the programming language.
- Spark SQL can automatically infer the schema of these objects using reflection

Optimization

- Uses <u>columnar cache</u>
  - reduce memory footprint by an order of magnitude because it applies columnar compression schemes such as dictionary encoding and run-length encoding.
- In Spark SQL, UDFs can be registered inline by passing Scala, Java or Python functions, which may use the full Spark API internally.

Catalyst - Extension

- Easy to add new optimization techniques and features to Spark SQL
- Enable external developers to extend the optimizer
  - E.g. adding data source specific rules that can push filtering or aggregation into external storage systems,
  - support for new data types.
- Catalyst supports both rule-based and cost-based optimizations

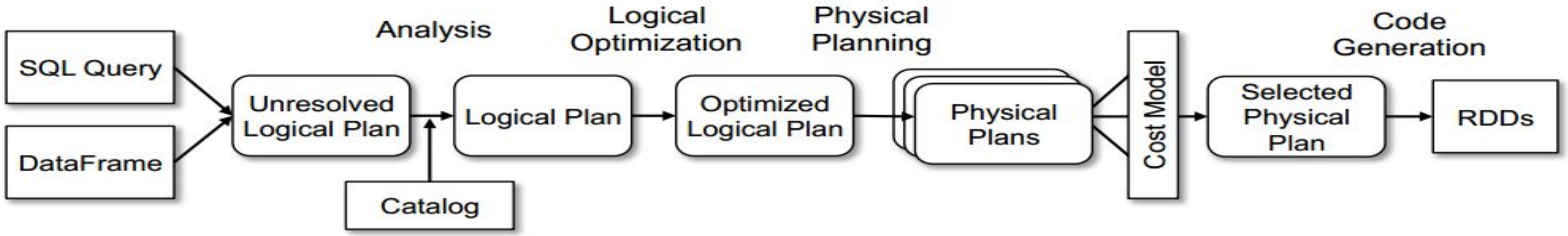
Catalyst

- Catalyst contains a general library for representing trees(Abstract Syntax Tree) and applying rules to manipulate them
- Catalyst offers several public extension points, including external data sources and user-defined types.

Tree Transformation

❖ Catalyst’s general tree transformation framework works in four phases

- ➢ analyzing a logical plan to resolve references
- ➢ logical plan optimization
- ➢ physical planning
- ➢ code generation to compile parts of the query to Java bytecode.



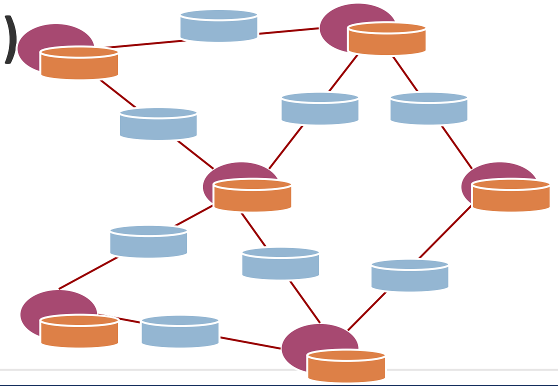
Spark GraphX

Spark GraphX

- 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)

Spark GraphX

- Spark GraphX - stands for graph processing
  - For graph and graph-parallel computation
- At a high level, GraphX extends the Spark RDD by introducing a new Graph abstraction:
  - a directed multigraph with properties attached to each vertex and edge.
- It is based on Property Graph model → **G(V, E)**
  - Vertex Property
    - Triple details
  - Edge Property
    - Relations
    - Weights



Resilient Distributed Graph (RDG)

- A tabular representation of the efficient vertex-cut partitioning and data-parallel partitioning heuristics
- Supports implementations of the
  - PowerGraph and
  - Pregel graph-parallel
- Preliminary performance comparisons between a popular dataparallel and graph-parallel frameworks running PageRank on a large real-world graph

Graph Parallel

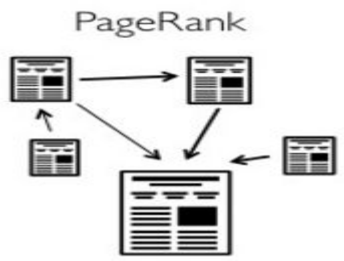
- Graph-parallel computation typically adopts a vertex (and occasionally edge) centric view of computation
- Retaining the **data-parallel metaphor,** program logic in the GraphX system defines transformations on graphs with each operation yielding a new graph
- The core data-structure in the GraphX systems is an immutable graph

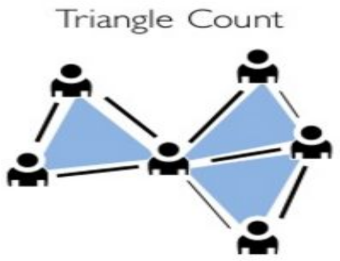
GraphX operations

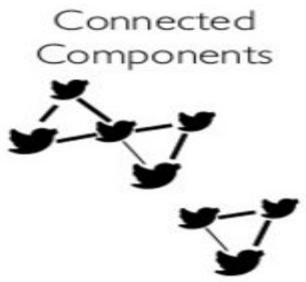
```
class Graph[VD, ED] {
// Information about the Graph
val numEdges: Long
val numVertices: Long
val inDegrees: VertexRDD[Int]
val outDegrees: VertexRDD[Int]
val degrees: VertexRDD[Int]

// Views of the graph as collections
val vertices: VertexRDD[VD]
val edges: EdgeRDD[ED]
val triplets: RDD[EdgeTriplet[VD, ED]]

// Functions for caching graphs
def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
def cache(): Graph[VD, ED]
def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]
// Change the partitioning heuristic
def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]
// Transform vertex and edge attributes
def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
     ----
```

GraphX build-in Graph Algorithms

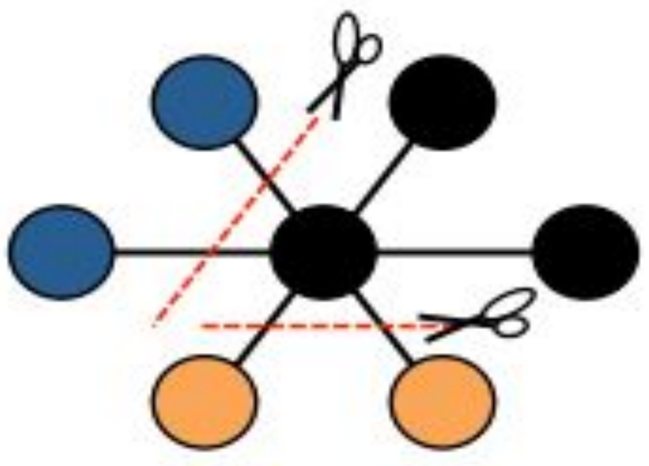
```
// Basic graph algorithms
========================================
def pageRank(tol: Double, resetProb: Double = 0.15): Graph[Double, Double]
def connectedComponents(): Graph[VertexId, ED]
def triangleCount(): Graph[Int, ED]
def stronglyConnectedComponents(numIter: Int): Graph[VertexId, ED]
```

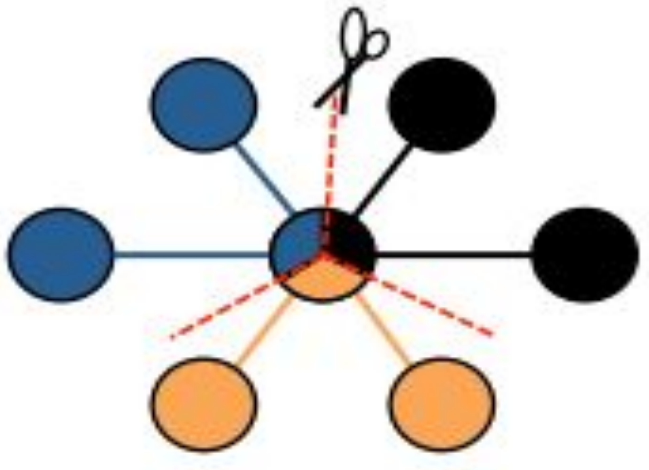




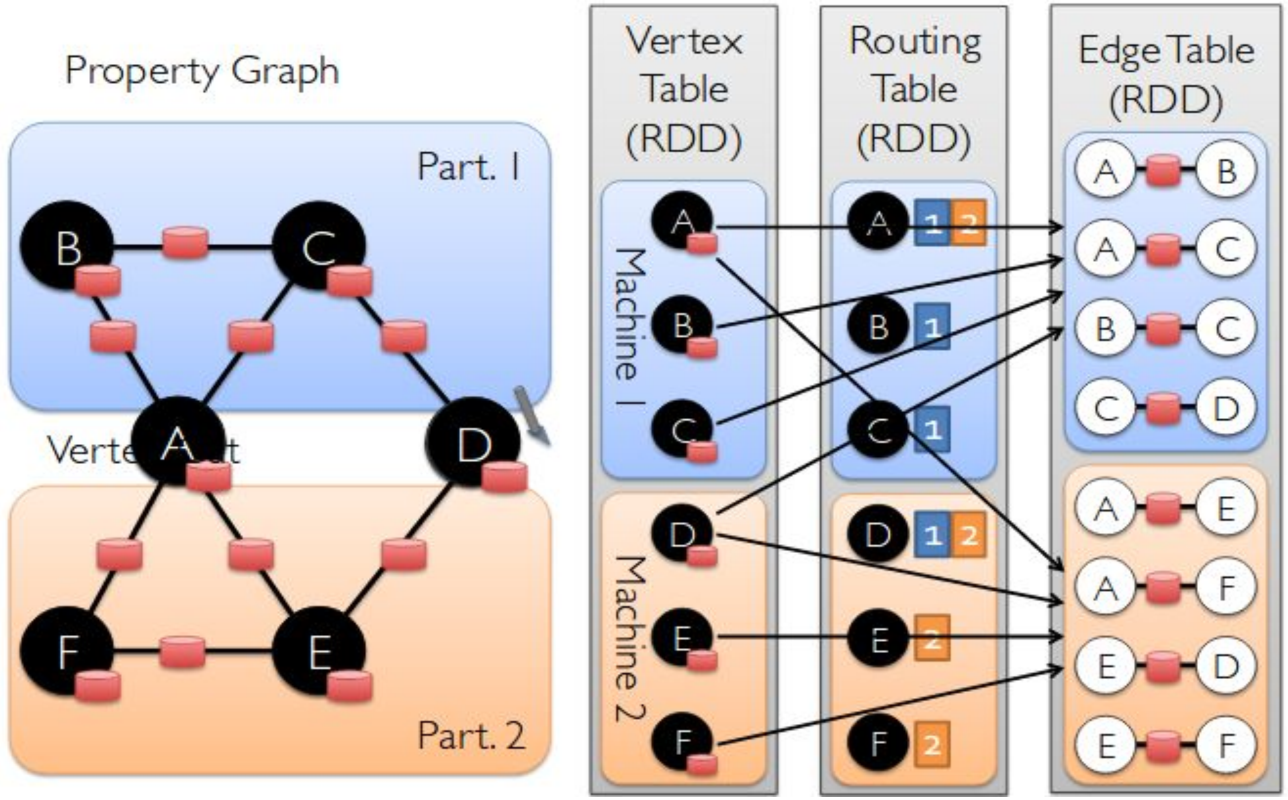


Edge-Cut vs Vertex-Cut


(a) Edge-Cut


(b) Vertex-Cut

Encoding Property Graphs as RDDs



Edge Table

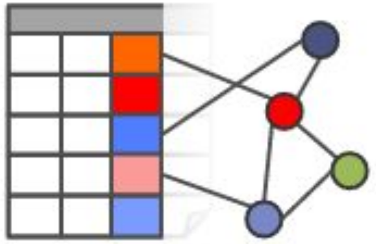
- EdgeTable(pid, src, dst, data): stores the adjacency structure and edge data
- Each edge is represented as a tuple consisting of the
  - source vertex id,
  - destination vertex id,
  - user-defined data
  - virtual partition identifier (pid).

Vertex Data Table

- VertexDataTable(id, data): stores the vertex data, in the form of a vertex (id, data) pairs
- VertexMap(id, pid): provides a mapping from the id of a vertex to the ids of the virtual partitions that contain adjacent edges

New API

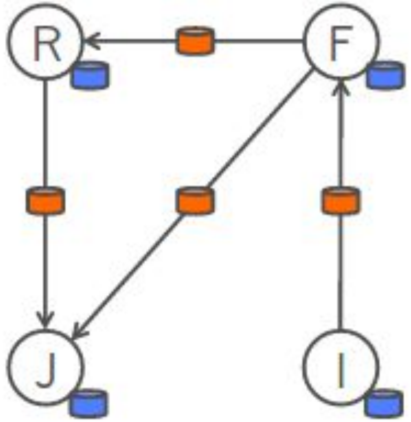
Blurs the distinction between Tables and Graphs



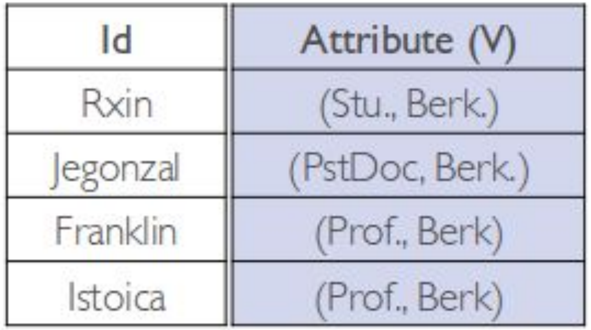
New Library

Embeds Graph-Parallel model in Spark

Property Graph

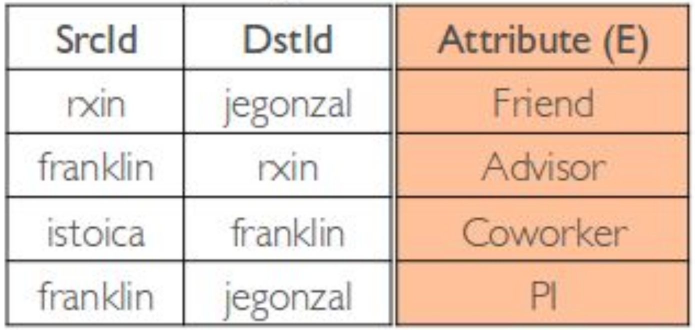


Vertex Table



Id	Attribute (V)
Rxin	(Stu., Berk.)
Jegonzal	(PstDoc, Berk.)
Franklin	(Prof., Berk)
Istoica	(Prof., Berk)

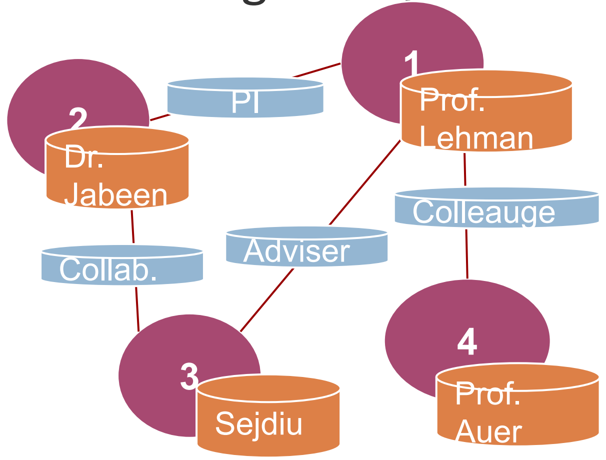
Edge Table



SrcId	DstId	Attribute (E)
rxin	jegonzal	Friend
franklin	rxin	Advisor
istoica	franklin	Coworker
franklin	jegonzal	PI

Spark GraphX - Getting Started

❖ Creating a Graph



```
type VertexId = Long
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))]=
spark.sparkContext.parallelize(
                    Array((3L, ("sejdiu", "phd_student")),
                          (2L, ("jabeen", "postdoc")),
                         (1L, ("lehmann", "prof")),
                          (4L, ("auer", "prof"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
spark.sparkContext.parallelize(
                    Array(Edge(3L, 2L, "collab"),
                          Edge(1L, 3L, "advisor"),
                          Edge(1L, 4L, "colleague"),
                          Edge(2L, 1L, "pi")))
// Build the initial Graph
val graph = Graph(users, relationships)
```

Vertex RDD	
vID	**Property(V)**
1L	(lehmann, prof)
2L	(jabenn, postdoc)
3L	(sejdiu, phd_student)
4L	(auer, prof)

Edge RDD		
sID	**dID**	**Property(E)**
1L	3L	advisor
1L	4L	colleague
2L	1L	pi
3L	2L	collab

GraphX Optimizations

- Mirror Vertices
- Partial materialization
- Incremental view
- Index Scanning for Active Sets
- Local Vertex and Edge Indices
- Index and Routing Table Reuse

Spark application, configurations, monitoring and tuning

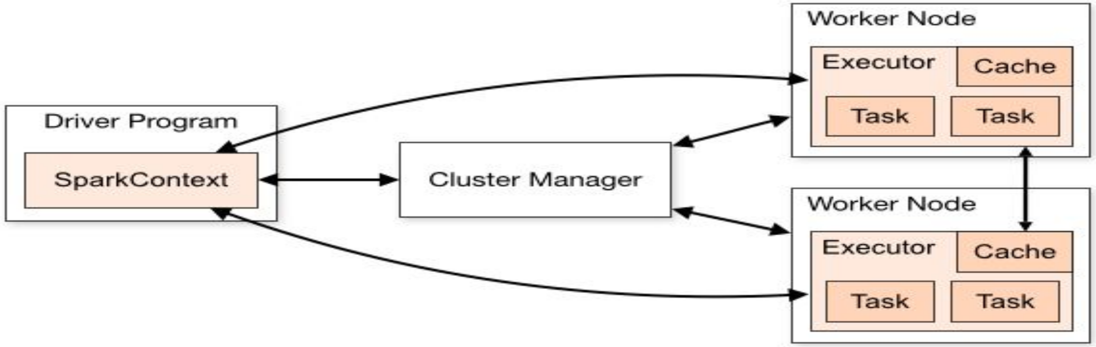
Spark application

- Spark Standalone Application on Scala
  - `git clone https://github.com/SANSA-Stack/SANSA-Template-Maven-Spark.git`
  - Import it on your IDE as Maven/General(sbt) project and create a new Scala class.
- Run a standalone application
  - Use any tools (sbt, maven) for defining dependencies
  - Generate a JAR package containing your application's code
    - Use sbt/mvn build package
  - Use `spark-submit` to run the program

    -
      ```
      ./bin/spark-submit   --class <main-class>   --master <master-url> \

      <application-jar> [application-arguments]
      ```

Spark configurations

- Spark Cluster Overview
  - **Components**
    - Driver aka `SparkContext`
    - Cluster Manager (Standalone, Apache Mesos, Hadoop YARN)
    - Executors



Spark monitoring

- Web Interfaces
  - **WebUI**
    - Every `SparkContext` launches a web UI, on port 4040, that displays useful information about the application.
  - **Metrices**
    - Spark has a configurable metrics system based on the Dropwizard Metrics Library. This allows users to report Spark metrics to a variety of sinks including HTTP, JMX, and CSV files.
  - **Advanced Instrumentation**
    - Several external tools can be used to help profile the performance of Spark jobs.

Spark tuning

- Tuning Spark
  - **Data Serialization**
    - It plays an important role in the performance of any distributed application.
      - Java serialization
      - Kryo serialization
  - **Memory Tuning**
    - The amount of memory used by your objects (you may want your entire dataset to fit in memory), the cost of accessing those objects, and the overhead of garbage collection (if you have high turnover in terms of objects).
  - **Advanced Instrumentation**
    - Several external tools can be used to help profile the performance of Spark jobs.

References

[1]. “Spark Programming Guide” - http://spark.apache.org/docs/latest/programming-guide.html

[2]. “Spark Streaming Programming Guide”- http://spark.apache.org/docs/latest/streaming-programming-guide.html

[3]. Spark SQL: Relational Data Processing in Spark by Armbrust, Michael, Reynold Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi and Matei Zaharia *in SIGMOD Conference,* 2015.

[4]. “Spark SQL, DataFrames and Datasets Guide” - http://spark.apache.org/docs/latest/sql-programming-guide.html

[5]. GraphX: Graph Processing in a Distributed Dataflow Framework by Gonzalez, Joseph, Reynold Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin and Ion Stoica *in OSDI,* 2014.

[6]. “GraphX Programming Guide” - http://spark.apache.org/docs/latest/graphx-programming-guide.html

[3]. “Spark Cluster Overview” - http://spark.apache.org/docs/latest/cluster-overview.html

[4]. “Spark Configuration” - http://spark.apache.org/docs/latest/configuration.html

[5]. “Spark Monitoring” - http://spark.apache.org/docs/latest/monitoring.html

[6]. “Spark tuning” - http://spark.apache.org/docs/latest/tuning.html

THANK YOU !

< http://sda.cs.uni-bonn.de/teaching/dbda/ >

Dr. Hajira Jabeen
jabeen@cs.uni-bonn.de
Room 1.062
(Appointment per e-mail)

Gezim Sejdiu
sejdiu@cs.uni-bonn.de
Room 1.068
(Appointment per e-mail)

Denis Lukovnikov
lukovnikov@cs.uni-bonn.de
Room 1.062
(Appointment per e-mail)