Apache Spark Notes

Spark essentials

Advantages of Apache Spark:

- Compatible with Hadoop
- Ease of development
- Fast
- Multiple language support
- Unified stack: Batch, Streaming, Interactive Analytics

Transformation vs. Action:

- A transformation will return an RDD. Since RDD are immutable, the transformation will return a new RDD.
- An action will return a value.

Spark Components

- The **Spark core** is a computational engine that is responsible for task scheduling, memory management, fault recovery and interacting with storage systems. The Spark core contains the functionality of Spark. It also contains the APIs that are used to define RDDs and manipulate them.
- **Spark SQL** can be used for working with structured data. You can query this data via SQL or HiveQL. Spark SQL supports many types of data sources such as structured Hive tables and complex JSON data.
- **Spark streaming** enables processing of live streams of data and doing real-time analytics.
- **MLlib** is a machine learning library that provides multiple types of machine learning algorithms such as classification, regression, clustering.
- **GraphX** is a library for manipulating graphs and performing graph-parallel computations.

Launch Spark Application

Local Mode

- driver & worker are in the same JVM
- RDD & variable in same memory space
- No central master
- execution started by user

Standalone/Yarn Cluster Mode

- the driver is launched from the worker process inside the cluster
- async,no wait

Standalone/Yarn Client Mode

- the driver is launched in the client process that submitted the job
- sync, need to wait

MESOS replaces Spark Master as cluster Manager and provides two modes:

1. Fine-grained mode: each task as a separate MESOS task; useful for sharing; start-up overhead
2. coarse mode: launches only one long-running task; no sharing; no start-up overhead

General operation

Spark provides **Transformation & Action**, Transformation is lazily evaluated.

Read files

- Text files with one record per line > `sc.textFile()`
- SequenceFiles > `sc.sequenceFile[K,V]`
- Other Hadoop inputFormats > `sc.hadoopRDD`
- A (filename, content) pairs > `sc.wholeTextFile`

Transformations

Function	Details
Map	Returns new RDD by applying `func` to each element of source
filter	Returns new RDD consisting of elements from source on which function is true
groupByKey	Returns dataset (K,iterable) pairs on dataset of (K,V)
reduceByKey	Returns dataset (K,V) pairs where value for each key aggregated using the given reduce function
flatMap	return a sequence instead of single item
distinct	Returns new dataset containing distinct elements of source

Actions

Function	Details
count()	number of elements in the RDD
reduce(func)	Aggregate elements of RDD using `func`
collect()	Returns all elements of RDD as an array to driver program
take(n)	Returns an array with first n elements
first()	Returns the first elements
takeOrdered(n,[ordering])	Return first n elements of RDD using natural order or custom operator

Different Types of RDD

- ParallelCollectionRDD
- MapPartitionsRDD
- PairRDD
- CoGroupedRDD
- HadoopRDD
- ShuffledRDD

ParallelCollectionRDD

`ParallelCollectionRDD` is an RDD of a collection of elements with `numSlices` partitions and optional `locationPrefs` => the result of `SparkContext.parallelize` and `SparkContext.makeRDD` methods.

MapPartitionsRDD

`MapPartitionsRDD` is an RDD that applies the provided function `f` to every partition of the parent RDD.

By default, it does not preserve partitioning — the last input parameter preservesPartitioning is false. If it is true, it retains the original RDD's partitioning.

`MapPartitionsRDD` is the result of the following transformations:

- `map`
- `flatMap`
- `filter`
- `glom`
- `mapPartitions`
- `mapPartitionsWithIndex`
- `PairRDDFunctions.mapValues`

- `PairRDDFunctions.flatMapValues`

PairRDD

check PairRDD

CoGroupedRDD

A RDD that cogroups its pair RDD parents. For each key k in parent RDDs, the resulting RDD contains a tuple with the list of values for that key.

Use RDD.cogroup(…) to create one.

HadoopRDD

`HadoopRDD` is an RDD that provides core functionality for reading data stored in HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI using the older MapReduce API (org.apache.hadoop.mapred).

HadoopRDD is created as a result of calling the following methods in SparkContext:

- `hadoopFile`
- `textFile` (the most often used in examples!)
- `sequenceFile`

Partitions are of type `HadoopPartition`.

ShuffledRDD

`ShuffledRDD` is an RDD of (key, value) pairs. It is a shuffle step (the result RDD) for transformations that trigger shuffle at execution. Such transformations ultimately call `coalesce` transformation with shuffle input parameter `true` (default: `false`).

It can be the result of RDD transformations using Scala implicits:

- `repartitionAndSortWithinPartitions`
- `sortByKey`
- `partitionBy` (only when the input partitioner is different from the current one in an RDD)
- `groupBy`

Data Frame

```
//1. create sqlContext
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
//2. used to convert RDD implicitly into a DataFrame
```

```
import sqlContext.implicits._
//3. define schema using a Case class
case class mySchema(name1: type, name2: type)
//4. create the RDD first
val rdd=sc.textFile("../.csv").map(_.split(","))
//5. Map the data to the Auctions Class
val mappedRDD = rdd.map(a=>mySchema(a.(0), a.(1),...))
//6. convert RDD to dataframe
val df = mappedRDD.toDF()
//7. register the DF as a table
df.registerTempTable("datatable")
```

Commonly used actions for data frame

Function	Details
collect	returns an array with all of rows in df
count	returns number of rows in the df
describe(cols)	computes statistics for numeric columns including `count`, `mean`, `stddev`, `min`, `max`
first(), head()	returns first row
show()	displays the first 20 rows in tabular form
take(n)	returns the first n rows

Commonly used functions for data frame

Function	Details
cache()	cache DF same as `persist(StorageLevel.MEMORY_ONLY)`
columns	return all column names as an array
explain()	prints the physical plan to the console for debugging purposes
printSchema()	Print the schema to the console in a tree format
registerTempTable(tableName)	Registers this DF as a tempory table using the given name
toDF()	returns a new data frame

Commonly used language integrated [transformation] queries

Function	Details
agg(expr, exprs)	Aggregates on the entire DF without groups
distinct	returns a new unique Dataframe
filter(conditionExpr)	filters based on given sql expression
groupBy(col1, cols)	groups DF using specified columns
select(cols)	selects a set of columns based on expressions

Stand-alone Application

Default project setting:

- `project_name.sbt`
- `src/main/scala/source_name.scala`

project_name.sbt example

```
name := "Auctions Project"

version:= "1.0"

scalaVersion := "2.10.5"

libraryDependencies += "org.apache.spark" %% "spark-core" % "1.6.1"
```

source_name.scala example

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf


object AuctionsApp{
    def main(args:Array[String]){
        val conf = new SparkConf().setAppName("AuctionsApp")
        val sc = new SparkContext(conf)
        val aucFile = "../../Downloads/data/auctiondata.csv"
        val auctionRDD = sc.textFile(aucFile).map(_.split(",")).cache()
        val totalnumber = auctionRDD.count()
        println("totalnumber: %s".format(totalnumber))
    }
}
```

Once `sbt` installed, run `sbt package` in the root of project to compile the `jar`.

PairRDD

PairRDD: key-value Pairs > `val pairRDD = RDD.map(data => (data(key), data(value)) )`

Many formats will directly return pairRDD: sequenceFiles create pairRDD, sc.wholeTextFile on small files creates pairRDD.

Transformations & Actions in RDD

- Transformations specific to pairRDD
  - `reduceByKey(func: (V, V) ⇒ V, numPartitions: Int): RDD[(K, V)]` : combines first locally then send at most one `V` per `K` to shuffle and reduce
  - `groupByKey(): RDD[(K, Iterable[V])]` : very expensive use following two instead, returns a key/value pairs where the value is an iterable list
  - `aggregateByKey[U](zeroValue: U)(seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U)(implicit arg0: ClassTag[U]): RDD[(K, U)]` : given initial value `U` , first operation with `V` , then the result value `U` will be combined by `K` .
  - `combineByKey[C](createCombiner: (V) ⇒ C, mergeValue: (C, V) ⇒ C, mergeCombiners: (C, C) ⇒ C): RDD[(K, C)]` : transform `(K, V)` => `(K, C)` by `createCombiner` to transform each `V` , then `mergeValue` with `C` to be `C` , finally for each `K` , `mergeCombiners` to `C` .
  - `sortByKey(ascending: Boolean = true, numPartitions: Int = self.partitions.length): RDD[(K, V)]`
- Transformations that work on two pair RDD
  - `join[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (V, W))]` : inner join > keep common `K` present
  - `cogroup[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (Iterable[V], Iterable[W]))]` : full outer join > keep all key
  - `subtractByKey()` : left subtract join > only left without common key
  - `leftOuterJoin[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (V, Option[W]))]`
  - `rightOuterJoin[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (Option[V], W))]`
- Transformations that apply to RDD as pairRDD, the input is `(K, V)` , so use `._1` to access the `K` , or `._2` to access the `V`
  - `filter()`
  - `map()`
- Traditional actions on RDD available to PairRDD
  - `countByKey(): Map[K, Long]` : To handle very large results, consider using `rdd.mapValues(_ => 1L).reduceByKey(_ + _)` , which returns an `RDD[T, Long]` instead of a map > a transformation
  - `collectAsMap(): Map[K, V]` : collect result as a map
  - `lookup(key: K): Seq[V]` : return all values associated with the provided key

Partition

- The data within an RDD is split into several partitions.
- System groups elements based on a function of each key: set of keys appear together on the same node
- Each machine in cluster contains one or more partitions: number of partitions is at least as large as number of cores in the cluster
- Two kinds of partitioning: **Hash Partitioning**, **Range Partitioning**

Hash Partitioner

Apply the hash partitioner to an RDD using the `partitionBy` transformation at the start of a program. It will shuffle all the data with the same key to the same worker.

```
val pair = rdd.map(x=>(K,V))
//HashPartitioner(partitions: Int)
val hpart = new HashPartitioner(100)
val partrdd2 = pair.partitionBy(hpart).persist()
// Result of partitionBy should be persisted to prevent partitioning from being applied each time the
```

Range Partitioner

The range partitioner will shuffle the data for keys in a range to the same workers especially a pair RDD that contains keys with an ordering defined.

`partitionBy(partitioner: Partitioner): RDD[(K, V)]`:

```
val pair = rdd.map(x=>(K,V))
// RangePartitioner(partitions: Int, rdd: RDD[_ <: Product2[K, V]], ascending: Boolean = true)(implic
val rpart = new RangePartitioner(4, pair)
val partrdd1 = pair.partionBy(rpart).persist()
// Result of partitionBy should be persisted to prevent partitioning from being applied each time the
```

Specify partitions in transformations

Some operations accept additional argument: numPartitions or type of partitioner. Some operations automatically result in RDD with known partitioner: `sortByKey` -> `RangePartitioner`; `groupByKey` -> `HashPartitioner`.

To change partitioning outside of aggregations and grouping operations:

- `repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T]` shuffles data across network to create new set of partitions
- `coalesce(numPartitions: Int, shuffle: Boolean = false)(implicit ord: Ordering[T] = null): RDD[T]` decreases the number of partitions, use `rdd.partition.size()` to determine the current number of partitions

DataFrames in Spark

SparkSQL is a library that runs on top of the Apache Spark Core and provides DataFrame API. The Spark DataFrames use a relational optimizer called the Catalyst optimizer.

Spark DataFrame:

- is a programming abstraction in sparkSQL: a distributed collection of data organized into *named columns* and scales to PBs
- supports wide array of data formats & storage systems, can be constructed from structured data files, tables in Hive, external databases or existing RDDs > equivalent to a DB table but provides a much finer level of *optimization*
- has API in scala, python, java and sparkR

Create DataFrames from existing RDD

Two ways:

- Infer schema by reflection
  - convert RDD containing case classes
  - use when schema is known and under 22 columns - limitation of Case class
- Construct schema programmatically
  - use to construct DF when columns and their types not known until runtime
  - when the fields pass 22

Infer schema by reflection is best for the users who are going to see the same fields in the same way.

Infer schema by reflection

```
//1. import necessary class
import sqlContext._
import sqlContext.implicits._

//2. create RDD
val sfpdRDD = sc.textFile("/user/user01/data/sfpd.csv").map(inc=>inc.split(","))

//3. define case class
case class Incidents(incidentnum:String, category:String, description:String, dayofweek:String, date:

//4. convert the RDD into an RDD of case objects using map
val sfpdCase=sfpdRDD.map(inc=>Incidents(inc(0),inc(1), inc(2),inc(3),inc(4),inc(5),inc(6),inc(7),inc(

//5. RDD is then converted to DF
val sfpdDF=sfpdCase.toDF()
```

```
//6. Registered as Table for enabling SQL query
sfpdDF.registerTempTable("sfpd")
```

In Python, no toDF or Case, just use `SQLContext.createDataFrame(data, schema=None, samplingRatio=None)` over RDD with:

Parameter	Value
data	an RDD of Row/tuple/list/dict, list, or pandas.DataFrame.
schema	a StructType or list of column names. default None.
samplingRatio	the sample ratio of rows used for inferring

Example in python:

```
# sc is an existing SparkContext.
from pyspark.sql import SQLContext, Row
sqlContext = SQLContext(sc)

# Load a text file and convert each line to a Row.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))

# Infer the schema, and register the DataFrame as a table.
schemaPeople = sqlContext.createDataFrame(people)
schemaPeople.registerTempTable("people")
```

Construct schema Programmatically

1. Create an RDD of `Rows` from the original RDD
2. Create the schema represented by a `StructType` matching the structure of Rows in the RDD created in Step 1: `StructType(Array(StructField(name, dataType, nullable),...))`
3. Apply the schema to the RDD of `Row`s via `createDataFrame` method provided by `SQLContext`

Example:

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// Create an RDD
val people = sc.textFile("examples/src/main/resources/people.txt")

// The schema is encoded in a string
```

```
val schemaString = "name age"

// Import Row.
import org.apache.spark.sql.Row;

// Import Spark SQL data types
import org.apache.spark.sql.types.{StructType,StructField,StringType};

// Generate the schema based on the string of schema
val schema = StructType(
    schemaString.split(" ")
                .map(fieldName => StructField(fieldName, StringType, true))
                )

// Convert records of the RDD (people) to Rows.
val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))

// Apply the schema to the RDD.
val peopleDataFrame = sqlContext.createDataFrame(rowRDD, schema)

// Register the DataFrames as a table.
peopleDataFrame.registerTempTable("people")
```

In Python:

```
# Import SQLContext and data types
from pyspark.sql import SQLContext
from pyspark.sql.types import *

# sc is an existing SparkContext.
sqlContext = SQLContext(sc)

# Load a text file and convert each line to a tuple.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: (p[0], p[1].strip()))

# The schema is encoded in a string.
schemaString = "name age"

fields = [StructField(field_name, StringType(), True) for field_name in schemaString.split()]
schema = StructType(fields)

# Apply the schema to the RDD.
schemaPeople = sqlContext.createDataFrame(people, schema)

# Register the DataFrame as a table.
schemaPeople.registerTempTable("people")
```

SQL data types reference

Loading/saving from/to data source

- Read: `sqlContext.read.format("format").load("filename.extension")`
- Write: `sqlContext.write.format("format").save("filename.extension")`
- to JDBC: `sqlContext.write.jdbc(url: String, table: String, connectionProperties: Properties): Unit`
- format: `json`, `parquet`, `json`

Transformations & Actions

Actions

- `collect()`
- `count()`
- `describe(cols:String*)`

Functions

- `cache()`
- `columns`
- `printSchema()`

Language integrated queries

- `agg(expr, exprs)` : aggregates on entire DF or `groupBy(df.name).agg(exprs)` by `df.name`
- `distinct`
- `except(other)` : returns new DF with rows from this DF not in `other` DF
- `filter(expr)` : filter based on the SQL expression or condition
- `sort($"col1", $"col2".desc)` : sort the column(s)

User defined function (UDF)

- In Spark, UDF can be defined inline, no need for registration
- No complicated registration or packaging process
- 2 types of UDF
  - to use with Scala DSL (with Data Frame Operations)
  - to use with SQL

User Defined functions in (Scala DSL Domain Specific Language)

In scala use `org.apache.spark.sql.functions.udf((arguments)=>{function definition})` :

```
// Defined a UDF that returns true or false based on some numeric score.
```

```
val predict = udf((score: Double) => if (score > 0.5) true else false)
// Projects a column that adds a prediction column based on the score column.
df.select( predict(df("score")) )
```

In python use `pyspark.sql.functions.udf(f, returnType=StringType)`:

```
from pyspark.sql.types import IntegerType
slen = udf(lambda s: len(s), IntegerType())
df.select(slen(df.name).alias('slen')).collect()
##[Row(slen=5), Row(slen=3)]
```

User defined functions in SQL query

```
// Option 1
// First define a function (funcname)
def funcname
//funcname _ turns the function into a partially applied function that can be passed to register
sqlContext.udf.register("func_udf", funcname _)
// Option 2
// or inline definition
sqlContext.udf.register("func_udf", func def)
```

N.B. The underscore tells scala that we don't know the parameter yet but want the function as a value. The required parameter will be supplied later.

Repartition DataFrame

The partition of DF is set by default in `spark.sql.shuffle.partitions` as value 200, and can be change by `sqlContext.setConf(key, value)`, i.e. `sqlContext.setConf("spark.sql.shuffle.partitions", partitionNumber: Int)`.

Repartition is possible by `df.repartition(numPartitions: Int)`, and the number of partitions can be determined by `df.rdd.partitions.size`.

Best practices:

- each partition to be 50 MB - 200 MB
- small dataset is ok to have few partitions
- large cluster with 100 nodes should have at least 100 partitions, e.g. 100 nodes with 10 slots in each executor require 1000 partitions to use all executor slots.

Monitor Spark Applications

A **RDD Lineage Graph** (aka RDD operator graph) is a graph of all the parent RDDs of a RDD, is built as a result of applying transformations to the RDD and creates a *logical execution plan*: `rdd.toDebugString()` will return the lineage for an RDD.

No **calculation** is performed until an *action* while the lazy evaluation creates a **Directed Acyclic Graph (DAG)**.

While applying an **action**, the scheduler outputs a computation stage for each RDD in the DAG. If an RDD can be computed from its parent **without** movement of Data, they are collapsed into a single stage referred as **pipelining**.

Situations when lineage truncated as usually the number of jobs equal to the number of RDDs in DAG:

1. **Pipelining**
2. RDD **persisted** in cluster memory or disk
3. RDD materialized due to earlier **shuffle**: since shuffle outputs in Spark are written to disk => *Built-in optimization*

When an **action** is encountered, the **DAG** is translated into a **physical plan** to compute the RDDs needed for performing the **action**.

Components	Description
tasks	unit of work within a stage
stages	group of tasks
shuffle	transfer of data between stages
jobs	work required to compute RDD, has one or more stages
pipelining	collapsing of RDDs into a single stage when RDD transformations can be computed without data movement
DAG	Directed Acyclic Graph: Logical graph of RDD operations
RDD	Resilient Distributed Dataset: Parallel dataset with partitions

Phases during Spark Execution:

1. User code defines the DAG
2. Actions responsible for translating DAG to physical execution plan
3. Tasks scheduled and executed on cluster

Debug and tune spark applications

- **Skewed** partition: some partition(s) take more time than others.
- Node Problem: some node(s) have issues

Common issues leading to slow performance:

- the level of **parallelism**: tune the level of parallelism by `numPartitions` in the shuffle ops or `repartitions()`, `coalesce()`
- the **serialization format** used during shuffle operations:
  - Spark serializes data during data transfer thus shuffle operations
  - java built in serializer is default, *Kryo* serialization is often more efficient
- managing **memory** to optimize your application:
  - 60% *RDD storage* vs. 20% *Shuffle* vs. 20% *user program*
  - Refer to persist options
  - `MEMORY_ONLY_SER` will cut down on garbage collection -> serialized in memory cache.

Best Practices and Tips

1. avoid shuffling large amounts of data: `aggregateByKey` for aggregations, `reduceByKey` | `combineByKey` | `foldByKey` instead of **heavy shuffle** `groupByKey`
2. do not copy all elements of RDD to driver: not `collect()` every time
3. filter sooner than later
4. many idle tasks (\~10k) -> `coalesce()`
5. not using all slots in cluster -> `repartition()` up

Spark Streaming

Spark Streaming Architecture

1. Data stream divided into batches
2. Process data using transformations
3. Output operations push data out in batches

Streaming Programing

Create Streaming Context

a. Existing `SparkContext sc`:

```
import org.apache.spark._
import org.apache.spark.streaming._

val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1))
```

a. Existing `SparkConf conf`

```
import org.apache.spark.streaming._

val sc = ...                // existing SparkContext
val ssc = new StreamingContext(sc, Seconds(1))
```

N.B.: the second parameter is `batchDuration` the time interval at which streaming data divided into batches.

After a context is defined, you have to do the following:

1. Define the input sources by creating input DStreams
2. Define the streaming computations by applying transformation and output operations to DStreams
3. Start receiving data and processing it using `streamingContext.start()`
4. Wait for the processing to be stopped (manually or due to any error) using `streamingContext.awaitTermination()`
5. The processing can be manually stopped using `streamingContext.stop()`, add `stopSparkContext=false` not to stop `sparkContext`

Key Concept

- Data sources:
  - File based: HDFS
  - Network based: TCP sockets
  - Twitter, Kafka, Flume, ZeroMQ, Akka Actor
- Transformations: create new DStream > cf.
  - Standard RDD operations: `map`, `countByValue`,...
  - Stateful operations: `UpdateStateByKey(function)`,...
- Output operations: trigger computation
  - `print`: prints first 10 elements
  - `saveAsObjectFile`, `saveAsTextFiles`, `saveAsHadoopFiles`: save to HDFS
  - `forEachRDD`: do anything with each batch of RDDs

Window Operations

- Apply transformations over a sliding window of Data
- window length [duration of the window] vs.[normally >] sliding interval [interval of operation]
- cf.

Window Operation	Description
window(windowLength, slideInterval)	Returns new DStream Computed based on windowed batches of source DStream
countByWindow(windowLength, slideInterval)	Returns a sliding window count of elements in the stream
reduceByWindow(func, windowLength, slideInterval)	Returns a new single-element stream created by aggregating elements over sliding interval using `func`
reduceByKeyAndWindow(func, windowLenght, slideInvterval, [numTasks])	Returns a new DStream of (K,V) pairs from DStream of (K,V) pairs; aggregates using given reduce function `func` over batches of sliding window
countByValueAndWindow(windowLength, slideInterval, [numTasks])	Returns new DStream of (K,V) paris where value of each key is its frequency within a sliding window; it acts on DStreams of (K,V) pairs

Fault tolerance

Fault Tolerance in **Spark RDDs**

- RDD is immutable
- Each RDD remembers lineage
- if RDD partition is lost due to worker node failure, partition recomputed

- Data in final transformed RDD always the same provided the RDD transformation are deterministic
- Data comes from fault-tolerant systems -> RDDs from fault-tolerant data are also fault tolerant

Fault Tolerance in **Spark Streaming**

- Spark Streaming launches receivers within an executor for each input source. The receiver receives input data that is saved as RDDs and then replicated to other executors for fault tolerance. The default replication for each input source is 2.
- The data is stored in memory as cached RDDs.
- If instead the node with the receiver fails, then there may be a loss of data that was received by the system but not yet replicated to other nodes. The receiver will be started on a different node.
- Write Ahead Logs (WAL), when enabled, all of the received data is saved to log files in a fault tolerant system, and sends acknowledgment of receipt, to protect against receiver failure.

Checkpointing

- Provides fault tolerance for driver
- Periodically saves data to fault tolerant system
- Two types of check pointing:
  - Metadata > for recovery from driver failures
  - Data checkpointing > for basic functioning if using stateful transformations
- Enable checkpointing > `ssc.checkpoint("hdfs://...")`
- If you use sliding windows, you need to enable checkpointing

GraphX

A way to represent a set of *vertices* that may be connected by *edges*.

Apache Spark GraphX:

- Spark component for graphs and graph-parallel computations
- Combines data parallel and graph parallel processing using in *single API*
- View data as graphs and as collections (RDD) > without the need for duplication or movement of data
- Operations for graph computation
- Provides graph algorithms and builders

Regular, Directed, and Property Graphs

Regular Graph: graph where each vertex has the same number of edges, e.g. users on Facebook.

Directed Graph: graph in which edges run in one direction from one vertex to another, e.g. Twitter follower.

Property graph: the primary abstraction of Spark GraphX; a directed multigraph which can have multiple edges in parallel； every edge and vertex has user defined properties associated with it, and therefore is unique > immutable, distributed and fault-tolerant. Characteristics:

1. Edges and vertices have user-defined properties associated with them
2. Property graphs are directional
3. Every edge and vertex is unique
4. Property graphs are immutable

Create Property Graph

Step 1. Import required classes

```
import org.apache.spark._
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
```

Step 2. Create vertex RDD

Vertex RDD is a tuple with a vertex ID, followed by an array of the properties of the vertex > `tuple(vertexID,case(properties))`

Step 3. Create edge RDD

An edge in a property graph will have the source ID, destination ID and properties > `tuple((srcID,dstID),case(properties))` then construct `Edge(srcID, destID, case(properties))`.

Step 4. Create graph

```
val graph = Graph(vertexRDD, edgeRDD, defaultVertex)
```

Perform operations on graph

Graph Operators

Type	Operator	Description
Info	numEdges	number of edges (Long)
Info	numVertices	number of vertices (Long)
Info	inDegrees	The in-degree of each vertex (VertexRDD(int))
Info	outDegrees	The out-degree of each vertex (VertexRDD(int))
Info	Degrees	The degree of each vertex (VertexRDD(int))
Cache	cache()	caches `MEMORY_ONLY`
Cache	persist(newLevel)	caches `newLevel`
Structure	vertices	An RDD containing the vertices and associated attributes
Structure	edges	An RDD containing the edges and associated attributes
Structure	triplets	An RDD containing the edges and associated attributes > `RDD[EdgeTriplet[VD, ED]]`
Structure	subgraph(epred, vpred)	Restricts graph to only vertices and edges satisfying the predicates
Structure	reverse	Reverses all edges in the graph
Structure	mask	The mask operator constructs a subgraph by returning a graph that contains the vertices and edges that are also found in the input graph
Property	mapVertices	Transforms each vertex attribute using map function
Property	mapEdges	Transforms each edge attribute, a partition at a time, using map function - does *not* pass the vertex value to the edge
Property	mapTriplets	Transforms each edge attribute, a partition at a time, using map function - does pass the vertex value to the edge

In addition to the vertex and edge views of the property graph, GraphX also exposes a triplet view. The triplet view logically joins the vertex and edge properties yielding an `RDD[EdgeTriplet[VD, ED]]` containing instances of the EdgeTriplet class. This join can be expressed in the following SQL expression:

```
SELECT src.id, dst.id, src.attr, e.attr, dst.attr
FROM edges AS e LEFT JOIN vertices AS src, vertices AS dst
ON e.srcId = src.Id AND e.dstId = dst.Id
```

Spark MLlib

Algorithms and Utilities	Description
Basic statistics	summary statistics, correlations, hypothesis testing, random data generation
classification and regression	linear models, decision trees, Naive Bayes
Collaborative filtering	model-based collaborative filtering using alternating least squares (ALS) algorithm
Clustering	Supoorts K-means clustering
Dimensionality reduction	singular value decomposition (SVD) and principal component analysis (PCA)
Feature extraction and transformation	common feature transformations

Alternating Least Squares

- ALS approximates a sparse user item rating matrix of dimension K as the product of two dense matrices (user [U*K] and item [I*K] factor [latent] matrices).
- ALS tries to learn the hidden features of each user and item
- The algorithm alternatively fixes one factor matrix and solves for the other until it converges.