Introductory Comment on Spark

Nathaniel Osgood

Key Characteristics of Spark

- Platform for distributed (cluster) computing
- A central use area: data science
- To key desiderata:
  - Generality (middleware support across diverse types of applications)
  - Performance (enable interactive exploration, even for iterative applications)
- Common stack
  - Mixing & matching mechanisms across pipeline
  - Maintaining & updating one system vs. fragmented, minimally-compatible systems
- Cross-language support
- Support for existing infrastructures: SQL RDBMSs, Cassandra, Hadoop, etc.
- Multiple components: ML/MLlib, GraphX, Spark Streaming, Spark SQL
- Rapid evolution

Key Performance Advantages

- Distributed processing (different subsets of data handled on different units)
- Memory-conscious computation
  - Enabling much in memory computation for iterative algorithms
  - Lazy (and thus incremental) evaluation
  - Immutable
    - Ability to share
    - And provenance recording: Capacity to discard and recompute results when necessary
  - User-specification of memory constraints

Anti-Patterns that Seeking to Avoid

- Need to pre-join all tables in database
  - Often far too expensive, if feasible at all
- Loading entire datasets into memory
- Transforming full dataset to full dataset
- Need to process all on one computer
- Need to “shuffle” (send) lots of data across the network to perform computation
- Slow performance of iterative algorithms in classic map-reduce frameworks

Important Concepts

- Immutability
- Provenance
- Resilient Distributed Data Sets
- Laziness: Transformers vs. actions
- Memory consciousness
  - Preferences over persistence & memory hierarchy level
- Distributed computation consciousness
  - Partitions, shuffling
- Implicit conversions between datatypes (e.g., for numeric operations)
- Key-value pairs
- Dataframes
- Type safety

Collection Abstractions: Common Features

- **Lineage (provenance) preserving**: Remember their history
- **Fault tolerant**: With knowledge of lineage, can redo computation if required
- **Memory conserving**: Can discard and recompute data, place in lower levels of the memory hierarchy (e.g., on disk), etc.
- **Lazy (and therefore potentially incremental)**: Computation deferred until action; distinction between
  - Transformation
  - Action
- **Target multiple machines**: Transparently distributed for processing to different machines
- **Interoperable**: Easy conversion to
  - Other Spark collection types
  - Scala collection types

Three Types of Collections

- 1st Generation: Resilient Distributed Datasets (RDDs)
  - Developer needs to specify schema to use (fields & types that apply)
  - Lower efficiency in moving data, querying
- 2nd Generation: DataFrames (previously SchemaRDDs)
  - Named columns (can add additional columns being over time)
  - Conscious of schema
  - Optimization of querying, data movement, memory management
  - Schema known statically: Lack of type safety using fields
  - Inability to recover type-safe data turned into a data frame
- 3rd Generation: DataSets
  - Improved (but not fully complete) type safety
  - Enhanced optimization
  - Greater interoperability: Movement of JVM objects into DataSets and recovery of objects from DataSets
  - Query optimization, memory management

All three types of collections are co-existing and can be readily used, with conversions between them

Areas of Strength of Each API

- 1st Generation: Resilient Distributed Datasets (RDDs)
  - Non-tabular data
  - Need lower level control
  - Less structured data
- 2nd Generation: DataFrames (previously SchemaRDDs)
  - Tabular data: Named columns (can add additional columns being over time)
  - Higher performance: Optimization of querying, data movement, memory management
  - Seek higher level operations (e.g., with user defined functions)
  - R use
- 3rd Generation: DataSets
  - Enhanced type safety
  - Benefits of DataFrames
  - Enhanced optimization
  - Greater interoperability: Movement of JVM objects into DataSets and recovery of objects from DataSets
  - Query optimization, memory management
- References:
  - http://www.agildata.com/apache-spark-rdd-vs-dataframe-vs-dataset/
  - https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

Machine Learning Libraries

- MLLib
  - Lower-level
    - Works with RDDs
    - Focused on particular algorithms
  - Older & more established
  - Additional features
- ML
  - Higher level: Pipeline based
  - Works with DataFrames

Functions on RDDs

- **Cache:** Request in-memory cache
- **toDebugString**: This shows the provenance (lineage) of this RDD
  - The series of operations that gave rise to it
- **persist**: Can pass in the persistence level sought (e.g., Memory only, Memory and Disk, etc.)
- unpersist
- Checkpoint
- isCheckpointed
- **val partitioner:** returns Option[SparkPartitioner]
- **partitionBy**: For Key-Value RDDs: Return RDD partitioned by specified partitioner

Input/Output

- saveAsTextFile
- saveAsObjectFile

Laziness Distinction

- Transformation: Lazy operation
- Actions: Force computation

Some Actions and Transformations

Transformations

- distinct : returns RDD with distinct values of this one
- Mapping
  - Map
  - flatMap
    - Note that the function given to flatMap returns a TraverseableOnce[U] -- doesn't have to be an RDD!)
- mapPartitions [applies on each partition, Can indicate if preserves partitions]
- Filter
- randomSplit
- sample

Actions

- Aggregate
- foreach
- foreachPartition
- reduce/fold
- groupBy
- Set transformations: intersection, union, subtract, cartesian
- Uncertain: sortBy (?), zip, zipWithIndex
- Summary statistics:max,min, count, countApprox, countByValue, countByValueApprox, countDistinctApprox
- Extractions: first, take [first *n*], takeOrdered [*n* that are smallest according to user-specified ordering], top [n that are largest according to user-specified ordering]
- keyBy [Returns RDD of key-value pairs]
- collect [Like scala's:filter & map at same time]

With respect to persistent performance

- Narrow: only process data in the same partition, e.g.,
  - map
  - flatMap
- Wide e.g., reduceByKey
  - If those with same key aren’t all mapped to same partition, shuffling will go on

Useful Input/Output

- sc.textFile()

Manipulating Cassandra Tables

- https://docs.datastax.com/en/datastax_enterprise/4.8/datastax_enterprise/spark/sparkSCcontext.html
- **sc.cassandraTable[CaseClassGivingSchema](“strKeyspace”, “strTableName”)**
  - This returns a CassandraRDD[CaseClassGivingSchema]
- **To get column names:** cassandraRDD.columnNames
- cassandraRDD.toArray
- cassandraRDD.select(“amount”).as((a:Int) => a).sum
- cassandraRDD.select(“age”).filter(“sex = ?”,”Male”), as((a:Int) => a).sum

DataFrame Features

- Creation
  - toDF(collOfCaseClass)
  - sqlContext.createDataFrame(sc.makeRDD(coll))
- Use $”colname” to indicate column in current DataFrame
- show [Displays in tabular format]
- printSchema [Displays schema]
- dfA.join(dfA(“id”) === dfB(“id”))
- df(“colName”) gets the column
  - Can combine element-wise with other columns e.g., df(“colName1”)/ df(“colName2”)
- Agg(op(col)): Avg, Max, CountDistinct, Count, Sum, Min, First
- withColumn(column) returns a DataFrame with an added column
- df.select(“colName1”,”colName2”…) selects columns
- write.json(“filename”)
- Can define user-defined functions to apply to data & use in queries
- *Call as[CaseClass] to return equivalent (strongly typed) Dataset*

Special Syntax

- $”colName” indicates the column name in the context of the current dataframe

Spark SQL

- Register data as temporary table
  - createOrReplaceTempView
- Code

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

Creation

- Via calls to spark.sql(sqlExpression)
- Via mapping read data into instances of a case class and calling .toDF
- spark.read.load(strParquetFilePath)
- spark.read.format(json).load(strJsonFilePath)

Spark Dataset Features

- count
- map & flatMap
- filter
- reduce
- groupBy
- as[ClassClassName]: [convert to DataSet[ClassClassName]
- show
- coalesce [consolidate at n partitions]
- distinct [Returns new dataset of unique]
- joinWith [specify condition as well]
- Intersection/union/subtract
- cache
- persist [save with Memory_and_disk]/unpersist
- printScheme
- rdd [to RDD]
- toDF [to DataFrame]
- sample [sample fraction of records]
- select (column expressions)
- take [first n as array] & take [first n as list]
- transform [allows applying function to map entire Dataset)]