Intro to Spark / RDDs

Apache Spark

Spark is a fast and general engine for large-scale data processing. The user writes a Driver Program containing the script that tells spark what to do with your data, and Spark builds a Directed Acyclic Graph (DAG) to optimize workflow. With a massive dataset, you can process concurrently across multiple machines.

Let's take a second to discuss the components of spark:

- **Spark Core** - All functionalities are build on this layer (task scheduling, memory management, fault recovery, and storage systems). Also has the API that defines Resilient Distributed Datasets (RDDs), which we will discuss later. There are 4 modules on this layer:
  - **Spark Streaming** - Live data streaming of data, such as log files. These APIs are similar to RDD.
  - **MLib** - Scalable machine learning library
  - **Spark SQL** - Library for working with structured data. Supports Hive, parquet, json, csv, etc.
  - **GraphX** - API for graphs and graph parallel execution. Clustering, classification, traversal, searching, and path-finding is possible in the graph. We'll come back to this much later on.

Why Scala?

You can use Spark with many languages; primarily *Python, R, Java and Scala*. I like Scala because it's functional, type-safe and JVM-friendly language. Also, since Spark is written in Scala, there is a slight overhead on running scripts in any other language.

Besides a knowledge of programming, a familiarity with SQL with make Spark very easy to learn.

The Resilient Distributed Dataset

The RDD is the backbone of Spark Core. This is not the same as a DataFrame, that's a different level of the API. This is a dataset made up of rows of information, that can be divided (distributed) on to many computers for parallel processing, and Spark makes sure we can find a way to get the data even if a node goes down during an operation (resilient).

Creation

We don't have to write the code to make sure we can handle al of the, the RDD is within the **SparkContext.** The first thing the Driver Program does is start the SparkContext automatically. And the RDD can come from any structured data stores; a database, file, in-memory object, or anything else.

Transformation

Once you have an RDD you'll probably want to transform it. There are many types of row-wise operations, such as:

- map - map an operation to each row, one-to-one
- flatmap - map an operation to create new rows from each row, one-to-many
- filter
- distinct
- sample
- Set operations - union, intersection, subtract, cartesian

Since Scala is a functional language, functions are objects. Meaning, there are many functions that also take functions. This is helpful in mapping operations to RDDs:

```
def sqaureIt(x: Int) {
return x*x
}


rdd.map(sqaureIt)
```

Actions

Finally we want the results of our RDD, to do this we can summarize the data with functions such as:

- collect - view all the raw data
- count
- countByValue - how many rows exist for each given unique value
- take - sample the rows
- top - view the top rows
- reduce

> **Important: You can call a transformation on an RDD but until you call an action nothing will be executed! This is part of Spark's lazy evaluation.**

Key Values RDDs

The map function can return key-value pairs by returning a tuple, and the values in the tuple can be other tuples or a complex object. This is not unique to Scala

```
rdd.map(x => (x,1))
```

And there are unique functions that can be used with key-value RDDs:

- reduceByKey() - combine values by key
  - ex. ((x,y) => x + y) to add them
  - think of x like the current row and y as the next row
- groupByKey() - group keys with the same values
- sortByKey() - sort by keys
- keys(), values() - return rdd of keys or values
- SQL style joins are also possible, but in practice Dataframes or the SQL API are usually used for this
- mapValues() / flatmapValues() - apply operation only to the values

Map vs Flatmap

map is a fairly straightforward concept, for each element of an RDD a transformation is applied in a 1:1 manner

```
val lines = sc.textFile("words.txt")
val bigWords = lines.map(x => x.toUpperCase) // covert all words to uppercase
```

Flatmap removes that 1:1 restriction. We can create many new rows for each row:

```
val lines = sc.textFile("words.txt")
val words = lines.map(x => x.split(" ")) // split words on spaces
val wordCount = words.countByValue() // count occurances of each word
```

Revision #8
Created 5 February 2024 23:29:09 by Elkip
Updated 9 April 2024 15:31:35 by Elkip