Spark

Project 2, COMP30770 Programming for Big Data (Spring 2021)

Rajit Banerjee, 18202817

Introduction

This project looks at using Spark for data analysis, including graph processing. The report covers the following sections:

- Spark: Exploring starred GitHub projects on "big-data".
- Graph Processing: Exploring the DBLP co-authorship dataset.
- Reflection: Review on 'Spark: Cluster Computing with Working Sets'.

The scripts and data files have been submitted alongside this report, as described in README.md. To set up the environment, the following Bash scripts need to be executed.

- Set up a Docker cluster for Spark (`spark-master`, `spark-worker-1` containers) after stopping and removing any preexisting containers with the same names. The script also adds `/spark/bin` to the `PATH` environment variable.

  ```
  $ ./docker-setup-spark.sh
  ```

- Clean and download required `data/` files: `github-big-data.csv` and `dblp_coauthorship.csv`. Next, copy all data and `.scala` files to the `spark-master` container, then start a Bash prompt in `/root/`.

  ```
  $ ./docker-start.sh
  ```

- The following commands can be executed in the container to check that the setup is successful.

  ```
  bash-5.0# ls
  01-github.scala  02-dblp.scala    data

  bash-5.0# ls data/
  dblp_coauthorship.csv  github-big-data.csv
  ```

1. Spark: GitHub Starred Projects

Dataset: `data/github-big-data.csv`
Scala source code: `01-github.scala`

The aim is to explore the dataset of the 100 most starred GitHub projects in the "big-data" topic. First, the CSV file is loaded into a Spark DataFrame called `github`, specifying that the file has a header row, and the schema should be deduced from the file organisation.

To answer questions about the dataset below, an SQL table is created using `registerTempTable`. All queries are written using Spark SQL and the results are stored in RDDs (Resilient Distributed Datasets).

1. Determine which project has the most stars. If multiple projects have the same number of stars, list all of them.

   All fields are selected from the `github` table under the condition that `stars` matches the maximum number of stars in the table (calculated using a subquery with the `MAX()` function).

2. Compute the total number of stars for each language.

   Grouping the rows by the `language` attribute, the name of the language and the number of stars (`SUM(stars)`) in each group (i.e. language) are displayed. The results are also sorted in descending order of total stars, to improve readability.

3. (a) Determine the number of project descriptions that contain the word "data".

   The count of all rows where the description field contains "data" is returned. Any substring in project descriptions is matched, using the `LIKE` operator with `%data%`. The `%` wildcard character matches zero or more characters, i.e. any number of characters before or after the word "data" are matched. The case is ignored by applying the `LOWER()` function on the project description before looking for "data", so only lowercase characters need to be checked.

   (b) Among those, how many have their language value set (not empty/null)?

   To count how many of these entries have their language attribute not null/empty, a similar SQL statement as above is used, with an added condition `language <> ''`, which does the necessary non-emptiness check.

4. Determine the most frequently used word in the project descriptions.

   First, all project descriptions are extracted and stored in a DataFrame. Then, the individual words from every line are obtained using the `explode()` function and space (`" "`) as split delimiter to identify words. Grouping is done by words, and the row count of every group is calculated. The result is arranged in descending order of count, and taking the first entry shows the most frequent word in the project descriptions ("Apache", 35 occurrences).

Script execution:

```
bash-5.0# spark-shell -i 01-github.scala
...
1. Which project has the most stars?
+-----+--------------------+--------+-----+
| name|         description|language|stars|
+-----+--------------------+--------+-----+
|spark|Apache Spark - A ...|   Scala|28798|
+-----+--------------------+--------+-----+

2. How many stars does each language have?
+---------------+----------+
|        language|sum(stars)|
+---------------+----------+
|            Java|    102331|
|           Scala|     59888|
|          Python|     39761|
|            null|     30305|
|      JavaScript|     25344|
|             C++|     23899|
|Jupyter Notebook|      8819|
|              Go|      6558|
|          Erlang|      4820|
|      TypeScript|      3274|
|               C|      2038|
|         Clojure|       774|
|           Julia|       489|
|            HTML|       383|
+---------------+----------+

3a. How many project descriptions contain the word 'data'?
[35]

3b. Among those, how many have their language value set (not empty/null)?
[33]

4. What is the most frequently used word in the project descriptions?
[Apache,35]
```

2. Graph Processing: DBLP Co-authorship

Dataset: `data/dblp_coauthorship.csv`
Scala source code: `02-dblp.scala`

This section explores a dataset extracted from the DBLP co-authorship dataset, representing a graph of scientists connected if they have authored a paper together.

1. Building a graph to represent the co-authorship data.

   A class named `CoAuthors` is created to store the pair of authors in every line from the dataset. Then, the `parseAuthors` method is defined, which takes a String as input (line), splits it on the comma delimiter, and returns a `CoAuthors` object, i.e., a pair of author Strings. The file `dblp_coauthorship.csv` is read into `textRDD`, which is then filtered to remove the header row. Finally, the CSV lines stored in `textRDD` are properly parsed into author pairs, using a `map()` with the `parseAuthors` function defined earlier. The result is stored in `coauthorsRDD` and the `cache()` action is used to persist the dataset.

   The individual authors are extracted from every pair using `map()` on the `coauthorsRDD`, followed by `distinct` to get the unique authors. The `zipWithIndex` function is used to create ID numbers for the authors, then the order of each item is reversed, i.e. (name, ID) becomes (ID, name), using another `map`. Using the `authors` RDD, the `authorIdMap` is created for faster lookup for author IDs based on author names as keys.

   To create a graph, an RDD for edge information is also required. First, the `idPairs` RDD is created to store the pair of author ID numbers for every pair of author names in the `coauthorsRDD` created earlier. Lookup is performed using the `authorIdMap` to substitute the author names with IDs. Next, the `edges` RDD is created, which uses a `map` operation on `idPairs` to create `Edge` objects, with attribute "1", signifying a distance of 1 between every pair of directly connected authors (i.e. nodes).

   The co-authorship `graph` is created using the following RDDs: `authors`, `edges` as well as a `default` vertex. To demonstrate successful graph creation, the number of vertices and edges are displayed. Three sample vertices and edges are also shown.

2. What is the maximum Erdös number of authors in the dataset?

   To compute the maximum Erdös number, the author ID number for "Paul Erdös" is obtained from the `authorIdMap`. Then, the `ShortestPaths.run()` function is used to compute the minimum distance between authors in the dataset and Erdös, i.e., the authors' Erdös number. The function returns a graph in which each vertex has the ID of an author and contains a map with the shortest distance to the target (called "landmark") authors. The resultant graph vertices (`paths`) are sorted in descending order, by the distance between Paul Erdös and landmark authors (each vertex's 1st item is the landmark author ID, and the 2nd item is a map, with a key for Erdös' ID and value as distance to Erdös). The top entry in `sortedPaths` gives an author with the maximum Erdös number, which is extracted and displayed.

Script execution:

```
bash-5.0# spark-shell -i 02-dblp.scala
...
Building co-authorship graph...
Done!

Number of authors (vertices): 58874
Number of co-authorships (edges): 217750

Sample vertices:
(41234,Helge A. Tverberg)
(28730,Kim B. Bruce)
(23776,Andrew M. Marshall)

Sample edges:
Edge(0,11176,1)
Edge(0,20844,1)
Edge(4,10140,1)

Maximum Erdös number of authors in the dataset: 3
```

3. Reflection

Spark was developed at UC Berkeley, with the codebase later donated to the current maintainers: the Apache Software Foundation [1]. To better understand its main principles, the original Spark paper [2] by Zaharia et al. provides a great discussion.

Motivation

The primary motivation behind developing Spark was to overcome the limitations of the highly successful MapReduce [3] model and its variants. Although MapReduce is proven to be effective for large-scale data-intensive applications using distributed computing, it suffers from a major drawback of having to read and write from disk too frequently. By using acyclic data flow graphs created by users, MapReduce achieves locality-aware scheduling, fault tolerance, as well as load balancing. However, several types of applications require a "working set" of data to be re-used for multiple parallel operations, which is highly inefficient using the basic MapReduce model, especially in use cases such as iterative jobs and interactive analysis.

In machine learning/optimisation tasks, there is a certain degree of repetition involved. Using MapReduce, each iteration would be expressed as a separate job involving expensive disk operations. Similarly, for interactive analytics tasks such as performing queries, there is a significant delay incurred due to reading large datasets from disk.

Spark introduces its fundamental data structure to support working sets: resilient distributed datasets (RDDs), which partition objects across nodes, and can be rebuilt in case of failure; thus maintaining MapReduce's fault tolerance and scalability properties as well.

Related Work

Spark's RDDs can be considered as Distributed Shared Memory (DSM) architecture, which has been studied since the early 1990's [4]; with a few key differences such as the "move code to data" model, and failure recovery via lineage information instead of checkpointing. Similar systems providing different approaches include Munin [5], Linda [6], and Thor [7], among others.

Regarding cluster computing, Spark differs from related work (such as Hadoop [8]) mainly in the aspects of data persistence and re-usability across parallel operations. Comparisons have also been made to IPython (interactive Python interpreter), and SMR (Scala Map Reduce, a former Scala wrapper for Hadoop) which inspired the use of Scala for Spark.

Proposed Solution

The primary abstraction that makes Spark possible is an RDD, a resilient distributed dataset. It is read-only, may be rebuilt if necessary, and can even be cached by users in memory to be re-used in parallel operations. Fault tolerance is achieved by "lineage", i.e., an RDD has adequate information about deriving itself from other RDDs in case it needs to be rebuilt when a partition is lost. A driver program is written to implement an application's high-level control flow and perform parallel operations. Spark RDDs are represented by Scala objects, and can be created in multiple ways: from a file stored in HDFS or similar, by parallelizing a Scala collection in the driver program, by transforming existing RDDs, or by changing the persistence of an existing RDD. On the last point, it is worth noting that RDDs follow a "lazy evaluation" model, i.e. dataset partitions are only stored in memory so long as a parallel operation is in progress. Hence, a dataset can be persisted in memory if necessary using the *cache* and *save* actions.

Spark also defines a number of parallel operations that can be performed on RDDs: *reduce* (combine elements), *collect* (return elements to driver program), *foreach* (apply a function to every element). Furthermore, two kinds of shared variables are also supported: *broadcast variables* (large read-only structures like lookup tables are wrapped and copied to each worker once instead of being packaged with every closure) and *accumulators* (used to implement fault-tolerant MapReduce counters, allowing a worker to "add" and the driver to read).

Implementation

Spark is implemented in Scala, which uses the Java Virtual Machine but is a functional programming language with an interactive interpreter available as well. Mesos [9], a cluster manager, which also started as a research project at UC Berkeley, provides the foundation for Spark, and allows data sharing with other frameworks. Internally, Spark RDDs implement a simple 3-method interface: *getPartitions*, *getIterator(partition)* and *getPreferredLocations(partition)*, used for partition ID listing, iteration, and task scheduling respectively.

Using a parallel operation creates a Spark task to be sent to worker nodes, using a "delay scheduling" approach. Shipping closures to workers is achieved by serializing them, just like Java objects, which makes "moving code to

data" much easier. Shared variables, i.e., broadcast variables and accumulators, are implemented using classes and special serialization tricks described in the paper.

Results

Spark was arguably the first system to provide an efficient programming language for interactive, large-scale, distributed data processing. It was proven to perform 10 times better than Hadoop in iterative machine learning, and provide sub-second latency while interactively querying a 39 GB Wikipedia dataset.

Thoughts

Over a decade since its inception, Spark (now a top-level Apache project) remains one of the most widely used technologies for big data analytics. The core principle of a resilient distributed dataset (RDD) is easy to understand: prevent frequent read/write operations on disk by moving computation to data, processing in memory, and rebuilding to achieve fault tolerance. Although the paper is a short one, it effectively describes the design and implementation of Spark, along with useful examples and performance comparisons for algorithms such as text search, logistic regression, and alternating least squares.

In terms of limitations, the authors note that at the time of writing, Spark did not support "shuffle" operations, or provide other high-level interactive interfaces for the Spark interpreter. However, much of that has been overcome in the past few years, with *repartition*, *'ByKey* and *join* operations performing shuffles, and interactivity is provided by the Python, R, and SQL shells.

Conclusion

Getting hands-on experience with Spark, although for simple analytics tasks, helped me familiarise myself with Scala and parallel operations in general. I certainly hope to work on more complex problems in the future using big data analytics tools. Finally, reading the original Spark paper provided much-needed clarity on some of the fundamental concepts that I applied while writing code.

References

[1] Wikipedia. Apache Spark. Retrieved from *https://en.wikipedia.org/wiki/Apache_Spark*.

[2] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud'10). USENIX Association, USA, 10.

[3] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107–113. DOI:*https://doi.org/10.1145/1327452.1327492*.

[4] Bill Nitzberg and Virginia Lo. 1991. Distributed Shared Memory: A Survey of Issues and Algorithms. Computer 24, 8 (August 1991), 52–60. DOI:*https://doi.org/10.1109/2.84877*.

[5] John B. Carter, John K. Bennett, and Willy Zwaenepoel. 1991. Implementation and performance of Munin. SIGOPS Oper. Syst. Rev. 25, 5 (Oct. 1991), 152–164. DOI:*https://doi.org/10.1145/121133.121159*.

[6] David Gelernter. 1985. Generative communication in Linda. ACM Trans. Program. Lang. Syst. 7, 1 (Jan. 1985), 80–112. DOI:*https://doi.org/10.1145/2363.2433*.

[7] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. 1996. Safe and efficient sharing of persistent objects in Thor. SIGMOD Rec. 25, 2 (June 1996), 318–329. DOI:*https://doi.org/10.1145/235968.233346*.

[8] Apache Hadoop. Retrieved from *http://hadoop.apache.org/*.

[9] Wikipedia. Apache Mesos. Retrieved from *https://en.wikipedia.org/wiki/Apache_Mesos*.