Report

GraphXQL

A graph database on top of HDFS,
using Apache GraphX

Student: Paul Legout

Fall 2022

Abstract

While relational databases are the foundations of the database world, they also have some drawbacks, such as a lot of constraints. New types of databases emerged in the last decade to fill the gap. Graph databases are one of them, where the data items are stored as a collection of nodes and edges. Edges represent the relationship between nodes. Graph databases hold the relationships between data as a priority, hence they are well suited for data items heavily linked.
Moreover, explosion of data in the last decade led to the emergence of new file systems such as the Google File System or the Hadoop File System. Apache Spark is a well known data processing framework, that can quickly and efficiently perform processing tasks on very large datasets, and has become the framework of choice when processing Big Data. Spark includes many APIs, such as Spark SQL, Spark Streaming and GraphX.
GraphX is Apache Spark's API for graphs and graph-parallel computation.
For this project, I decided to link graph databases with the Apache GraphX API.

Contents

1. Overview

The solution I implemented uses the Apache GraphX API [7] to easily process large amount of data, using the Spark ecosystem while storing persistently the data. The solution is implemented using Java. Since the Hadoop ecosystem is written in Java and Spark in Scala, hence using the Java Virtual Machine, I found it appropriate. Moreover, Java is a strongly typed, object oriented language, portable and vastly used. The goal is to be able to easily load, process and save the data, with a Spark-like interface.

During this report, I will use the terms ”Nodes” and ”Relations” / ”Relationships” to refer to ”Vertices” and ”Edges”, since these are the terms used in a graph database.

2. Design

GraphXQL API capabilities include creating, updating, deleting databases, and manipulate data like any database. This type of database is more suited for read-heavy workloads and data processing. One database consists of one big graph. Each node of the graph has a NodeType, which can be compared to a table in a relational database. However, a NodeType do not have a schema, but key-value pair fields. This approach introduces high flexibility.
Each Node has a unique identifier, a NodeType and a set of fields.

Nodes and relationships can easily be added to the database, and since it uses the GraphX API, querying the data is done via a graph instance. There is no need to learn a new language. If you are familiar with GraphX, you will be able to query and process the data from the database.

GraphXQL is a Maven project, hence it can easily be integrated to any Java project, simply by importing the package, without the need to download the dependencies.

2.1 Configuration

A little configuration is needed. Of course, Hadoop and Spark need to be installed and running.

The project includes a ”config” folder, where instructions and scripts are provided to create a cluster of LXD containers [1], install and set up Hadoop and Spark. Moreover, another folder ”installation”, includes two bash scripts.

- graphxql-env.sh contains the environment variable GRAPHXQL_HOME refering to the HDFS absolute path of the root
- install.sh creates the folders needed by the program

The installation is simple, but does require a UNIX-based Operating System.

Then, a few more configuration steps are required. The root directory of the database must be provided to Spark, and the property "fs.defaultFS" must be provided to Hadoop (Configuration steps in README.md file). When starting the database, it will get the Hadoop and Spark configurations in order to run properly.

2.2 Properties

GraphX competes on performance with the fastest graph systems while retaining Spark's flexibility, fault tolerance and ease of use. Moreover, it comes with a variety of graph algorithms, which can be used to transform the data.
The data is stored on HDFS, which provides replication to ensure fault-tolerance.

3. Architecture

3.1 Hadoop File System

First, a class to manage HDFS operations has been implemented. The methods include rename, copy, read, write, append, list files / directories... The methods are statics to easily be called anywhere in the program.
The class loads the Hadoop configuration, including the fs.defaultFS property and perform the tasks.

3.2 Storage

As show in 1, each database is stored in its own folder, under root/databases. Then, each database has two folders containing the nodes content and relations between them. Several options are available to store a graph, however I found this method to be the most flexible.
The nodes of a given nodetype, are stored in one folder. This enable efficient modifications.

3.2.1 Partitioning

We can see that the nodes and relations are partitioned into several files. Since the Hadoop File System does not allow random writes, updating a file means to load its content, make the modifications and then right it back to HDFS. Therefore, one big file per nodetype / relations is not conceivable to efficiently update or delete a node.

By default, Spark creates partitions when saving an Resilient Distributed Dataset (RDD) to HDFS. However, when updating a node, we want to know the file it is written on, to load and rewrite this only file, and keeping track of the nodes location on disk with the partitioning Spark provides is not easy.

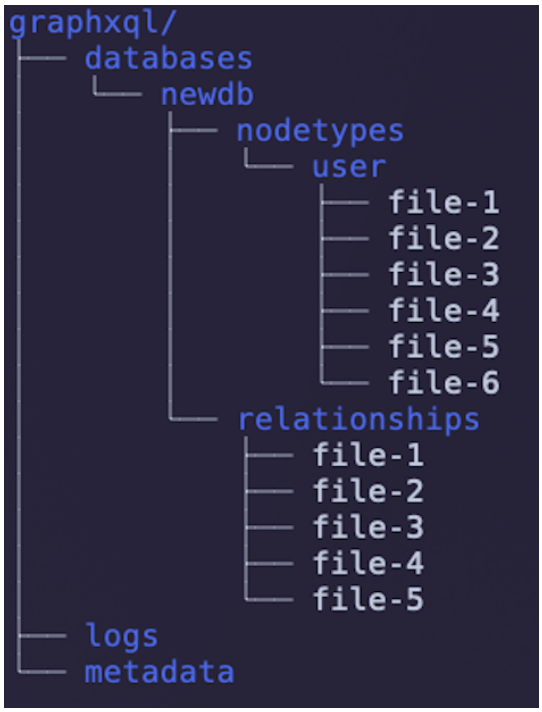

Figure 1: Tree Structure of the root

To deal with this, I created a simple but effective partitioning. The nodes are partitioned by their UUID and the relations by the source node UUID. A simple hash function takes this UUID and returns the corresponding file. The implemented hash function is monotone : $UUID_1 < UUID_2 => f(UUID_1) <= f(UUID_2)$
NodeTypes and relationships files have a maximum UUID they can contain so that each file can be loaded and rewritten without too much cost when an update / delete occur.
In case of a new node / relation added to the database, an append to the corresponding file is performed.

3.3 NodeTypes

As said previously, a NodeType refers to a type of node. When creating the Java object, it can be added to the database by calling the methods on the database object. When doing so, a folder is created in the given database directory.
Each Java object nodetype is a singleton. One nodetype is referred to by its name. However, two nodetypes, from different databases, can have the same name. To avoid conflicts, each database has its own set of singletons nodetype.

3.4 Nodes

When a Java object Node is added, the database affects the object a unique identifier. The way the UUID is generated is essential because it impacts the queries, but also the location of the node on disk. Since GraphX Edge attributes source and destination id

fields are defined as a long type, the nodes UUID are also of type long. Since Java long types are 64 bits of length, one limitation is that one database cannot contain more than $2^{63} - 1$ nodes (number of possible values for a signed long).
To generate unique identifiers, I used an accumulator, provided by Spark's API. Indeed, when running on cluster mode, an incremental object's attribute can perform in an unpredictable way, this is why the accumulator is here for.
Whenever a new node is added, the accumulator's value is incremented and the node UUID is set to the accumulator's value. When the database is loaded, i.e. a user creates its Java object (discussed in detail in the database section), the accumulator value is set to the maximum created UUID. The partitioning here is very important. Instead of looking for the maximum UUID, going through each node, the maximum file (i.e. the file "file-n", where n is the maximum), for each nodetype, is loaded to get the maximum UUID per nodetype, and then the global maximum. It works because of the monotony property of the implemented hash function.

At first, I wanted to use the AVRO file format [6] to store the nodes. This format uses a schema and can be well integrated with Java [2] to manipulate samples as Java objects. However, I ran into issues with the non-serializable implementation of the classes. When trying to save or load into HDFS, a NonSerializableException was thrown. Finally, I decided not to use this file format and went for a set of key-value pairs to define the content of a node. The value has a general type for more flexibility.

3.5 Relationships

Relationships between nodes are Edge<String>, which take three parameters: the UUID of the source node, the UUID of the destination node and the value of the relation / edge. I solely implemented relationships with a String value, which might prevent some graph algorithms to be performed, however a sub-graph can be defined, with another type (if cast is possible).

3.6 Database

At first, I imagined the database inheriting from the class Graph. However, I found it more convenient to simply have the graph defined as an attribute of the class. Graph operations are performed using the getter method, which returns the graph.

I thought a lot about the structure of the database, and I came up with one big graph, with various nodetypes. This approach makes it easy to create relations between different nodetypes. This is one of the major drawbacks of relational databases: if a possible relation has not been thought between tables, it is difficult to add it later. One big graph enables high flexibility.

The database does not implement transactions, hence does not respect the ACID properties. However, it respects the Consistency property.
The consistency of the graph is achieved with two elements:

- First, the graph is immutable, thanks to the Scala implementation
- Second, each database object is a singleton, therefore two users connected to the database will manipulate the same Java object, so the same graph object

The consistency between the graph object and the values stored on disk is achieved by first updating the values on disk, then the graph. If an error occurs during the modification on disk, an error is thrown and the graph is not updated.

The updates and deletes are the most difficult to handle and the more costly operations. Adding a node or a relationship is a simple append. As discussed before, due to HDFS properties, an update to a file involves overriding it. The more costly operation is the deletion of a node. First, the node is deleted from disk, then every relation involving the node is deleted from disk. Finally, the graph is updated, without the node and its relationships. To that extend, this type of database is not well suited for write heavy workload, but for read and data processing. Adding nodes or relationships is OK, but updating and deleting are definitely operations to minimize.

4. Results and Discussion

4.1 Results

To test the application, I performed simple operations:

- Create a new database
- Add nodes and relationships (One node is a User with a name and a favorite number, relationships are ”Friends” and ”Best friends”)
- Display graph
- Select and display nodes having the ”Best friends” relationship
- Update a node
- Update a relationship
- Display graph to see updates
- Add more nodes and relations
- Display final graph

The output of the above operations can be found under the logs folder, ”res.log” file.
The corresponding values on disk can be found under logs/graphxql folder (I copied from HDFS to local the database folder, after the above operations performed).

4.2 Future directions

Some future improvements / features are:

- Implementation of queues for updates, to avoid too much ”pressure” with a lot of writes on disk. Moreover, the queues would allow some optimizations, for examples to perform several updates at a time, instead of one by one.
- Implementation of users and access writes. Since HDFS supports access rights, and that a database is one folder, each database would have its own defined group and a user has access if he is part of the group (UNIX access rights...)
- Implementation of a Rest API, for example using Spring Boot, to easily manage the database with HTTP requests

References

[1] Linux Containers. https://linuxcontainers.org.

[2] Documentation, A. A. Apache Avro (Java). https://avro.apache.org/docs/1.11.1/getting-started-java/.

[3] Documentation, A. Y. Apache Hadoop YARN. https://hadoop.apache.org/docs/r3.1.0/hadoop-yarn/hadoop-yarn-site/ResourceModel.html.

[4] Documentation, A. Y. Hadoop - Multi-Node Cluster. https://www.tutorialspoint.com/hadoop/hadoop_multi_node_cluster.htm.

[5] Documentation, H. Hadoop Java FileUtil. https://hadoop.apache.org/docs/r2.6.0/api/org/apache/hadoop/fs/FileUtil.html.

[6] Documentation, S. Apache Avro. https://spark.apache.org/docs/latest/sql-data-sources-avro.html.

[7] Documentation, S. Apache GraphX. https://spark.apache.org/graphx/.

[8] Documentation, S. Apache GraphX Examples. https://spark.apache.org/docs/latest/graphx-programming-guide.html.

[9] Documentation, S. Running Spark on YARN. https://spark.apache.org/docs/latest/running-on-yarn.html.

[10] Documentation, S. Spark SQL. https://spark.apache.org/docs/1.3.1/sql-programming-guide.html.

[11] Zone, J. D. Java Read and Write files HDFS. https://javadeveloperzone.com/hadoop/java-read-write-files-hdfs-example/.

[11] [5] [10] [9] [6] [2] [3] [4] [7] [8]