Implementing Recursive Queries on Hive and Hadoop

Yang, Shijing
syang6@wpi.edu

Catherman, Davis
dscatherman@wpi.edu

1 Introduction

We live in the data age. We are creating tons of data everyday. There are roughly 2.5 quintillion bytes of data we are generating each day at our current pace, but the pace is only accelerating with the growth of internet. [1] However, traditional database system such as SQL, is designed for handling a limited amount of data efficiently. When it comes to huge volume data, using SQL might take a long time to process, therefore a new way needs to be considered in order to process big data efficiently.

Apache Hadoop is an open-source software as well as a data warehouse that is designed for solving problems involving huge amount of data by using multiple computers. [2] Hadoop applies MapReduce concept into the software to achieve efficiency, but it requires reading JAVA code to trigger the process [3], which is less user-friendly to data engineers who are used to writing SQL code.

Hive is created based off the concept of utilizing MapReduce framework as well as having the syntax of SQL. Therefore, it becomes extremely efficient for data engineers to switch to Hive without being frustrated by having to learn a new programming language. Even though Hive can perform some particular functions that cannot be achieved by relational databases [4], it can also be the other way around. The recursion feature in SQL does not exist in Hive, but it can be very useful to handle data with a hierarchical structure.

Recursion in SQL is generaly under the category of "common table expression (CTE)". [5] The core idea of recursion in SQL is that having the key word "recursive", the sub-query inside the CTE would repeadily refer to the CTE itself, which is how recursion is done. Once the termination condition is achieved, the sub-query would return 0 row and output the result. Since Hive does not support recursion, it can be time-consuming to write sub-queries line by line if we want to use recursion-like approach on hierarchical-structured data.

Our goal is how can recursion be implemented on Hive in an efficient and logical manner that feels as if it is naturally supported while working with different tables and data. It has significant meanings to achieve our goal. First, as we mentioned previously, recursion is a useful tool to retrieve hierarchical data. Second, switching database is too impractical. If a company's data is completely stored on Hive and the company is switching to another database management system just because there is one function that Hive does not support, it becomes very costly especially when there is terabytes or petabytes of data stored. Last but not least, it is also a bad practice to switch a programming language since it will create an issue of making maintaining the database harder and harder. Consequently, finding a way to connect recursion and Hive can fix this problem easily.

2 Existing Methods

Several methods to accomplish similar behavior currently exist. Many of such capabilities have significant limitations or are unfeasible in certain situations. The following section will further detail these options and when they may work and what is insufficient about them.

2.1 Naive Methods

The first valid approach is to simply use a language that supports recursion to begin with. This method allows development to fluid and continous in one language and database method. This is often not an option since many databases may have been created months or years prior and already contain terabytes to petabytes of data. This knowledge also makes it infeasible to simply switch databases to one that supports such features. Given these limitations, we will quickly move onto the next.

2.2 Hadoop

Since Apache Hive is built on top of an Apache Hadoop implementation, the data is already stored in an Hadoop HDFS system and therefore readily available. Hive does not use any special storage techniques making the data still readable by Hadoop. Given this information, it would be feasible to directly implement MapReduce jobs in Java on Hadoop with the main function acting as the recursion loop. This would mean that in the event a database engineer wishes to recursively access data, they would be required to switch languages. The primary limitation of this method comes due to that fact; the code would not flow as well and harder to maintain a detailed repository. This also implies that the user of the database would need to be familiar with both SQL and Java, further limiting the workers available for the position.

2.3 Programming Recursion

As in the Hadoop implementation, the main Java function would be handling the recursion. This would behave in the way recursion is handled in other traditional programming languages. When traditional languages are used in conjunction with the database software used, these style of implementations become possible since recursion is a well developed capability in most modern languages.

2.4 SQL Recursion

Hive is an SQL like language, and as a result it becomes worthwhile to examine SQL method of recursion even though it would not be feasible to transfer the information to an SQL database. SQL uses the command $WITH\ RECURSIVE$ in order to perform recursion. There are four main components in a SQL recursion query. The initial query is called an anchor members, which can be treated as a base case. Recursive query is the query that constantly calling the CTE itself to perform recursion. $UNION\ ALL$ or $UNION\ DISTINCT$ are often used to separate the first query and the recursive query. Once the termination condiction is achieved, a final query is needed to present the result generated by the CTE table.

We created an example with a simple family tree. Figure 1 shows the relationship of the members in the family. Based on the information given from Figure 1, we input the information into a SQL database. The schema of the table structure is like Table 1. If we want to know all the ancestors of Hannah, the SQL command would be similar to Listing 1. As we can see from Listing 1, the first query is the base case, which is search for the row that is equal to Hannah; the second query is joining the row from the base case with the entire table, which will give us Hannah's parents Andrew and Sarah. Having Andrew and Sarah as output, the query is joining the output with the entire table again. When it returns 0 row, the termination condition will be triggered. By having $UNION\ ALL$, it will join all the outputs together. In our case, according to Figure 1, the final output should be "$Hannah, Andrew, Sarah, Tim, Penny\ and\ Cheryl$".

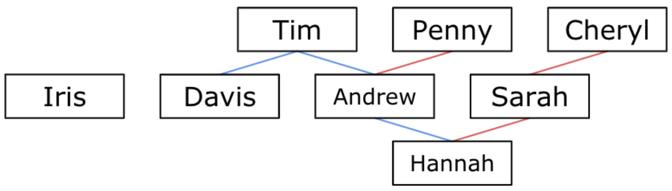


Figure 1: Simple Family Tree

ID	Name	Father	Mother
1	Davis	6	null
2	Iris	null	null
3	Andrew	6	7
4	Sarah	null	8
5	Hannah	3	4
6	Tim	null	null
7	Penny	null	null
8	Cheryl	null	null

Table 1: Table Structure of Family Tree

```sql
with recursive ancestor as
(select * from people where Name = "Hannah"
union all
select people.* from people
inner join ancestor on
ancestor.father = people.ID or ancestor.mother = people.ID)
select * from ancestor;
```

Listing 1: SQL Recursion example

3 Assumptions

In order to simplify our implementation and solution, we lay out a few key assumptions. First, we assume the tables a command will be operated on already exists. This is a straightforward assumption since most SQL or Hive commands mostly treat this as an assumption, except maybe perform some error checking. Regardless, the Hive commands should throw a related error that is already covered internally. Second, we assume an intelligent developer is manipulating the system in that they will type the code correctly. Again, this assumption is not a hard to believe requirement. We also assume that the use cases will be limited for the program. While ideally we would support an vast number of cases, the goal of this research is to demonstrate the capabilities and provide analysis for further development to exist. We do, however, assume that the solution must be malleable in terms of parameters and query commands versus hard coded cases.

4 Solution

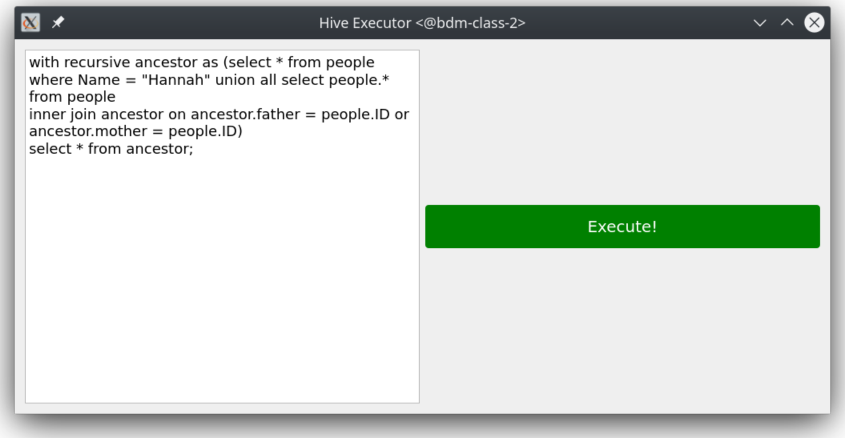
For our solution we chose to leverage existing capabilities of systems. First, we leverage that Python can be connected to Hive. This allows the implementation to work with a traditional language and expand the capabilities of the system. Within Python we leverage the recursive logic of Java Hadoop jobs. These combined mean that in the same way a main function in Java may provide recursion capabilities when submitting jobs to Hadoop, we can use Python to submit jobs recursively to Hive. Python provides the easily capability of implementing recursion and connecting to Hive providing a strong performance and result. Python can receive a query in a generic method and parse the information to generate commands to send to Hive, it will execute those commands before receiving a response and checking if another layer of recursion is necessary. Python can repeat this inner loop as much as necessary. In order to specify the query in an easily to understand fashion that feels as if you are still programming directly for Hive, we leverage the logic that Hive is an SQL like language. This is possible efficiently since many Hive commands are exactly the same in SQL and SQL already supports recursive queries. Python will simply parse the SQL commands and convert them to Hive's equivalent commands in a recursive loop.

5 Implementation

The solution was implemented to support command line queries and a graphical user interface. The interface is what will be detailed in this section as it is an example implementation of how to leverage the functionality. The interface was completed with $PyQt5$ following a Model-View-Controller design paradigm. The controller section depicts the interaction with the Hive query tool. The interface is able to accept both recursive and non-recursive commands entered into the multi-line text box. The user may then click $Execute!$ to process the commands. The recursive results will be saved in a $.tmp/$ directory, however, this can be easily modified. Figure 2 depicts the main screen of the graphical user interface. On the left side is where a user may enter or copy and paste a query. The user may continue to edit the query until they are ready and may select the green execute button shown on the right. Once the query is executing, the button should transition to the state shown in Figure 3. This depicts that the button has changed to red and that the current query is executing. Once it completes, it will return to green. The query continues to execute in the terminal window and the status can be watched from there.

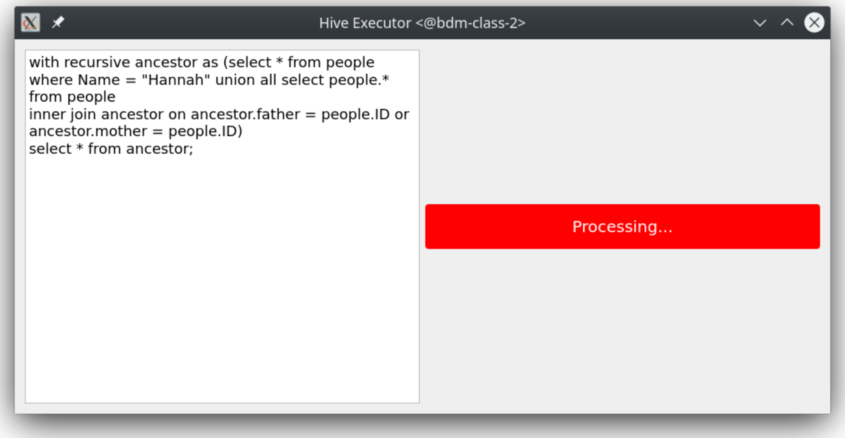
5.1 Limitation

One limitation that exist in the GUI is the ability to see the results, a terminal window could be added for better clarity as to what is currently happening, however, this was determined to be a



```
with recursive ancestor as (select * from people
where Name = "Hannah" union all select people.*
from people
inner join ancestor on ancestor.father = people.ID or
ancestor.mother = people.ID)
select * from ancestor;
```

Figure 2: The main screen of the GUI, ready to execute a query.



```
with recursive ancestor as (select * from people
where Name = "Hannah" union all select people.*
from people
inner join ancestor on ancestor.father = people.ID or
ancestor.mother = people.ID)
select * from ancestor;
```

Figure 3: The main screen of the GUI, currently executing a query.

feature that is ultimately unrelated to the scope of the project.

6 Experimentation

We performed a series of experiments to test our application in order to verify that it is not hard-coded and is able to handle different situations. We created three edge cases that are for each of the query component within a recursive SQL query.

6.1 *Change in Initial Query*

The first test case is to change our base case from searching for Hannah's ancestors to Davis's ancestors. The SQL command is shown as Listing 2 below. From Figure 1, the output should be *Davis* and *Tim*.

```sql
with recursive ancestor as
(select * from people where Name = "Davis"
union all select people.* from people
inner join ancestor on ancestor.father = people.ID)
select * from ancestor;
```

Listing 2: SQL Recursion example

6.2 *Change in Recursion Query*

The second test case is to change the condition of the recursion query. Other than looking for father that matches the corresponding person, we are also trying to look for father plus one as well. The SQL command is shown as Listing 3 below. The expected output result should be $Hannah$, $Andrew$, $Sarah$, Tim and $Penny$ since Sarah does not have a father listed.

```
with recursive ancestor as
(select * from people where Name = "Hannah"
union all select people.* from people
inner join ancestor on
ancestor.father = people.ID and ancestor.father+1 = people.ID)
select * from ancestor;
```

Listing 3: SQL Recursion example

6.3 *Change in Final Query*

The third test case is to change the condition of final query. Instead of choosing all the columns, we only select the Name column from the ancestor CTE table. The SQL command is shown as Listing 4 below. The expected output should be $Hannah$, $Andrew$ and Tim.

```
with recursive ancestor as
(select * from people where Name = "Hannah"
union all select people.* from people
inner join ancestor on ancestor.father = people.ID)
select Name from ancestor;
```

Listing 4: SQL Recursion example

7 Conclusion

After implementing several testings on edge cases, we all get positive results. We can conclude that we've successfully created a GUI written in Python that will interpret a SQL recursive command and convert it to a way that Hive can process. This is a completely unique way to perform a recursive query on Hive.

However, since we are using only one node to process, it is much more slowly than implementing in SQL. For future development, we will use multiple nodes to accelerate the process and take the advantage of Hive. Also, we are considering adding more terminal support for the GUI to make the process more convenient and user-friendly.

References

[1] Marr, B.(2019, September 05).How Much Data Do We Create Every Day?The Mind-Blowing Stats Everyone Should Read. Retrieved December 11, 2020, from https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/?sh=5269a70c60ba

[2] Apache Hadoop. (2020, November 21). Retrieved December 12, 2020, from https://en.wikipedia.org/wiki/Apache_Hadoop

[3] White, T. (2015). Part I. Hadoop Fundamentals. In Hadoop: The Definitive Guide (pp. 29-31). Sebastopol, CA: O'Reilly.

[4] What is Hive? Architecture and Modes. (n.d.). Retrieved December 12, 2020, from https://www.guru99.com/introduction-hive.html

[5] A Definitive Guide To MySQL Recursive CTE. (2020, April 11). Retrieved December 12, 2020, from https://www.mysqltutorial.org/mysql-recursive-cte/