Graph Databases for Relationship Querying

Bela Demchouk, Kelsey Neis, Ashwin Sridhar
July 14, 2021

Abstract

This paper will explore graphical and relational databases in terms of how well they perform when used on a specific real world data set. In regards to the databases, we will discuss several possible queries such as an ER schema to represent the relations between Twitter users and a graph schema that was created on Tiger Graph Cloud as well as Degree centrality and Shortest path. The goal of this paper is to gain a better understanding of the methods and materials of these two approaches and which yield better results.

Keywords. MySQL,TigerGraph,Relational database, graph database, relational schema, graph schema, cycles, shortest path, sql, gsql

1 Introduction

Relational databases are a type of database that stores and provides access to data points that are related to one another. Relational databases are based on the relational model, an intuitive, straightforward way of representing data in tables. One of the major benefits of using a Relational Database is that this type of Database allows the user to simply classify the data into different categories and store them efficiently. A very popular use case is online transaction processing because they support the ability to insert, update, or delete small amounts of data; they accommodate a large number of users, and they support frequent queries and updates as well as fast response times.
A graph database is a database that uses graph structures for semantic queries with nodes, edges, and properties to represent and store data. Graph databases are used in cases where the most important data is the relationships. A graph database is purpose-built to handle highly connected data, and the increase in the volume and connectedness of today's data presents a tremendous opportunity for sustained competitive advantage. A common use case is fraud detection which uses graph databases to detect suspicious patterns of customer activity and cross-referencing with previously identified fraud, we can flag up potential fraud that may still be ongoing.
Calculation of the shortest path between two nodes in a graph is a popular operation used in graph queries in applications such as map information systems, social networking services, and biotechnology. Recent shortest-path search techniques based on graphs stored in relational databases are able to calculate the shortest path efficiently, even in large data using frontier-expand-merge operations.
For our algorithms, we used GSQL, a very specific graph query language designed specifically for TigerGraph, the graph database we used to test our methods on. GSQL is a Turing-complete language that incorporates procedural flow control and iteration, and a facility for gathering and modifying computed values associated with a program execution for the whole graph or for elements of a graph called accumulators. GSQL graphs are then composed from these vertex and edge sets.
We will find out which database is more efficient and easier to use for this particular data set.

2 Objectives

The goal of this project is to explore relational database schema with a graph database schema side-by-side in a use case involving real world data set.

3 Data

We will be modeling the Higgs Twitter Dataset from Stanford using a relational database and a graph database and comparing their performance. The Higgs Twitter Dataset includes mention, retweet, reply, and follow data from Twitter during the time when the Higgs Boson particle was discovered. The Higgs dataset has been built after monitoring the spreading processes on Twitter before, during and after the announcement of the discovery of a new particle with the features of the elusive Higgs boson on 4th July 2012. The messages posted in Twitter about this discovery between 1st and 7th July 2012 are considered.[5]

4 Relational Model

We used MySQL 8 for the creation of our relational database and ran it locally on one of our laptops, which has a 2.6 GHz 6-Core CPU and 16GB of memory. In modeling the data, we emulated a graph structure by creating the

"twitter_users" table to represent nodes and the "higgs_twitter_edges" to represent edges.

In contrast to the graph database, we combined the types of relations — retweet, mention, and reply — into one table and differentiated between them with an action_type attribute. We did this for ease of finding multiple relation types in one query.

Below is the SQL for creating the relational schema, and the ER diagram we used as a guide:

```
CREATE TABLE twitter_users (
        uid VARCHAR(15) PRIMARY KEY
);

CREATE TABLE higgs_twitter_edges (
    source_id VARCHAR(15),
    target_id VARCHAR(15),
    action_time INT,
    action_type VARCHAR(2),
                FOREIGN KEY(source_id) REFERENCES twitter_users(uid),
                FOREIGN KEY(target_id) REFERENCES twitter_users(uid),
    PRIMARY KEY(source_id, target_id, action_time, action_type)
    )engine=innodb;
```

ER diagram

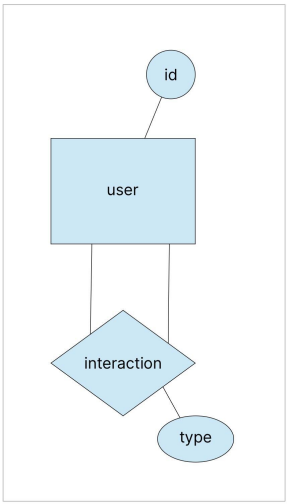

Figure 1: The ER diagram for the relational schema

5 Graph Model

Our initial attempt was to run the graph database locally. We used the docker tutorial however the performance was very slow. Our next choice was to use the tiger graph cloud service to have the database in the cloud. As this project has no budget we were limited to the free tier. The free tier was set as 4vCPU and 8GB of memory.
Just like relational database, tiger graph also supported direct csv import to specify relationships between vertices. Below is the gsql to create the graph schema.

```
CREATE VERTEX person (
    PRIMARY_ID name STRING,
    name STRING, age INT,
    gender STRING, state STRING
)
```

twitterHiggs Schema

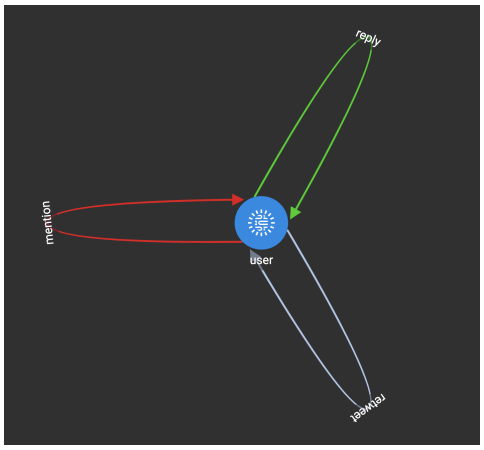

Figure 2: The above is the visual representation of the higgsTwitter schema

```
CREATE DIRECTED EDGE reply (FROM person, TO person)

CREATE DIRECTED EDGE retweet (FROM person, TO person)

CREATE DIRECTED EDGE mention (FROM person, TO person)

CREATE GRAPH higgsTwitter (person, reply, retweet, mention)
```

6 Algorithms

The algorithms were selected on the basis of their recognition by the programming community. Initial choice fell on "Shortest Paths" and "Cycles". However, the size of the data set and the limitations in computing power necessitated simplifying both algorithms into "Shortest Path from Source" and "Triangles"

Shortest paths finds the shortest path from source to target vertex .

Triangles finds an interaction between at least 2 vertices even if the vertices are identical (self directed replies or tweets). Another limitation placed on the execution of the algorithm was that we could only look at one interaction at a time. Either reply, or mention or retweet.

6.1 Shortest Path

The first algorithm we used to evaluate the two models was one to find a single source, unweighted shortest path. The query would find, for a given node, a list of reachable nodes and the shortest path to each of them in the form of a sequence of the intermediary nodes.

6.1.1 Relational Database

We created a "paths" table in order to facilitate keeping track of paths to each node and a "nextPaths" view to find the next set of nodes at a given iteration of running the algorithm.

```
CREATE TABLE paths (
  id INT PRIMARY KEY AUTO_INCREMENT,
  source VARCHAR(15),
  target VARCHAR(15),
  hops SMALLINT,
  path CHAR(250)
);
```

```
INSERT INTO paths
  SELECT
    NULL,
    source_id,
    target_id,
    1,
    CONCAT(source_id,',',target_id)
  FROM higgs_twitter_edges;

CREATE OR REPLACE VIEW nextpaths AS
  SELECT
    paths.source, higgs_twitter_edges.target_id, paths.hops+1 AS hops,
    CONCAT(paths.path, ',', higgs_twitter_edges.target_id) AS path
  FROM paths
  JOIN higgs_twitter_edges ON paths.target = higgs_twitter_edges.source_id AND LOCATE(higgs_twitter_edges.ta
  GROUP BY paths.source,higgs_twitter_edges.target_id;
```

Then we created a stored procedure in MySQL that would:

1. Add rows to the paths table from the nextpaths view
2. Join with paths on source and target using sort-merge join where there is not yet a row for the matching source and target in the paths table
3. Continue until the query returns no results

Below is the SQL for the stored procedure [3]:

```
SET sql_mode="";
DROP PROCEDURE IF EXISTS BuildPaths;
DELIMITER go
        CREATE PROCEDURE BuildPaths(IN start_node VARCHAR(15))
BEGIN
  DECLARE row_count INT DEFAULT 0;
  TRUNCATE paths;
  -- STEP 1: SEED ROUTES WITH 1-HOP PATHS
  INSERT INTO paths
  SELECT
    NULL,
    source_id,
    target_id,
    1,
    CONCAT(source_id,',',target_id)
  FROM higgs_twitter_edges
       WHERE source_id=start_node;
  SET row_count = ROW_COUNT();
  WHILE (row_count > 0) DO
    -- STEP 2: ADD NEXT SET OF PATHS
    INSERT INTO paths
          SELECT
            NULL,
            nextpaths.source,
            nextpaths.target_id,
            nextpaths.hops,
            nextpaths.path
          FROM nextpaths
          LEFT JOIN paths ON nextpaths.source = paths.source
                AND nextpaths.target_id = paths.target
          WHERE paths.source IS NULL AND paths.target IS NULL;
    SET row_count = ROW_COUNT();
```

```
  END WHILE;
END;
go
DELIMITER ;
```

6.1.2 Graph Database

Similar to the approach for the relational database, we applied a greedy algorithm to find shortest paths in the graph database. The algorithm follows these steps [2]:

1. Get all connected vertices that have not been visited
2. Update the length of the path for the current target
3. Add the neighbor vertex to the path and mark as visited
4. Continue iterating until no neighbors are found

Below is the GSQL for creating the query in TigerGraph [4]:

```
CREATE DISTRIBUTED QUERY ShortestPath (VERTEX<user> source, BOOL display=TRUE) FOR GRAPH higgsTwitter {

        MinAccum<int> @dis;
        OrAccum @visited;
        ListAccum<vertex> @path;
        SetAccum<edge> @@edgeSet;

  ##### Initialization  #####
        Source = {source};
        Source = SELECT s
                 FROM Source:s
                 ACCUM s.@visited += true,
                       s.@dis = 0,
                       s.@path = s;
        ResultSet = {source};

  ##### Calculate distances and paths #####
        WHILE(Source.size()>0) DO
                Source = SELECT t
                         FROM Source:s -(:e)-> :t
                         WHERE t.@visited == false
                         ACCUM t.@dis += s.@dis + 1,
                               t.@path = s.@path + [t],
                               t.@visited += true;
                ResultSet = ResultSet UNION Source;
        END;

  ##### Print the results #####

        PRINT ResultSet[ResultSet.@dis, ResultSet.@path];
        IF display THEN
                ResultSet = SELECT s
                            FROM ResultSet:s -(:e)-> :t
                            ACCUM @@edgeSet += e;
                PRINT @@edgeSet;
        END;


}
```

6.2 Triangles

Another algorithm considered for comparison was a simple triangle search. Specifically we attempted to find an interaction among at least three users in the form of a reply, a re-tweet or a mention.

6.2.1 Relational Database

Below is the sql query for finding explicit triangle relationship between three vertices.

```
select e1.source, Count(*)
from higgs_twitter_edges e1 join
     higgs_twitter_edges e2
     on e1.dest = e2.source join
     higgs_twitter_edges e3
     on e2.dest = e3.source and e3.dest = e1.source and e2.source < e3.source
where action_type = 'RE'
group by e1.source;
```

6.2.2 Graph Database

To create the triangle algorithm we used the Pattern Recognition feature of the TigerGraph service. This simplified creating the triangle algorithm. The example below is just for the reply triangle.
Exact same procedure was used in an attempt to identify mention and retweet as well as mix of edges triangles.

```
CREATE QUERY reply_triangle() FOR GRAPH higgsTwitter API("v2") SYNTAX v2 {

  SetAccum<edge> @@FinalEdgeSet_5;
  SetAccum<edge> @@FinalEdgeSet_6;
  SetAccum<edge> @@FinalEdgeSet_4;
  SetAccum<vertex<user>> @@FinalVertexSet_1;
  SetAccum<vertex<user>> @@FinalVertexSet_3;
  SetAccum<vertex<user>> @@FinalVertexSet_2;

  VertexSet_1 =
        SELECT two
        FROM user:one -(reply>:alias_schema_5)- user:two,
             user:two -(reply>:alias_schema_6)- user:three,
             user:three -(reply>:alias_schema_4)- user:one
        ACCUM @@FinalEdgeSet_5 += alias_schema_5,
              @@FinalEdgeSet_6 += alias_schema_6,
              @@FinalEdgeSet_4 += alias_schema_4
        POST-ACCUM @@FinalVertexSet_1 += two
        POST-ACCUM @@FinalVertexSet_3 += one
        POST-ACCUM @@FinalVertexSet_2 += three
        ;

  PRINT @@FinalEdgeSet_5;

  PRINT @@FinalEdgeSet_6;

  PRINT @@FinalEdgeSet_4;

  VertexSet_1 = { @@FinalVertexSet_1 };
  PRINT VertexSet_1[" as no_attributes];

  VertexSet_3 = { @@FinalVertexSet_3 };
  PRINT VertexSet_3[" as no_attributes];

  VertexSet_2 = { @@FinalVertexSet_2 };
  PRINT VertexSet_2[" as no_attributes];
```

}

7 Results

7.1 Shortest Paths

Both the graph database and the relational database handled the single source shortest path problem fairly well, with no noticeable difference in results or run time. However, analyzing the estimated run time for both implementations yields vastly different results. If we were to run these queries on millions instead of hundreds of thousands of nodes, the difference would likely be more noticeable.

The run time for the graph database is $O(E)$ [2], whereas the estimated run time for the relational database is $O(E^2 log(E))$. This is due to the fact that with the relational query, it was necessary to join the "nextpaths" view with the "paths" table on every iteration, whereas with the graph database there was an inherent ability to retrieve a node's neighbors in constant time, and the algorithm ran by traversing the graph without needing to query for the next node. We estimated the run time based on the assumption that the query would use sort-merge to join the view and table at each iteration [6].

7.2 Triangles

Executing the Cycles algorithm in the free tier would consistently time out. We attempted to create a smaller subset of the graph with unsuccessful results. The simplified "Triangle" algorithm did yield interactions that showed cycle involvement.

triangles

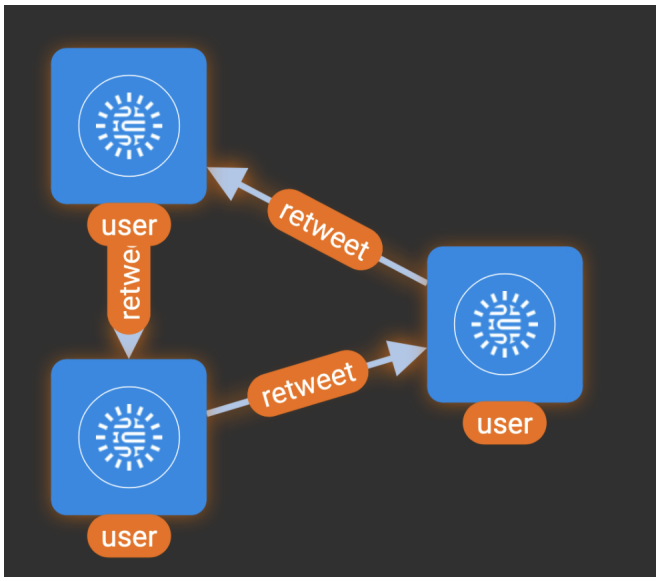

Figure 3: The above is the visual representation of the triangles pattern used to generate the gsql query

The results of the Triangle algorithm are below: The big O run time for graph database is estimated to match DFS, $O(V + E)$. For relational database estimation I used the inner nested loops index from chapter 12 [6]. Assuming I index the edges with hash index, the run time can be estimated as $O(E^3(E + V))$

7.3 Discussion

Creating schema for both relational databases and graph databases was relatively easy and quick. Ability to visualize results as dynamic clusters makes analyzing data and relationships much more intuitive then static tables.

triangles

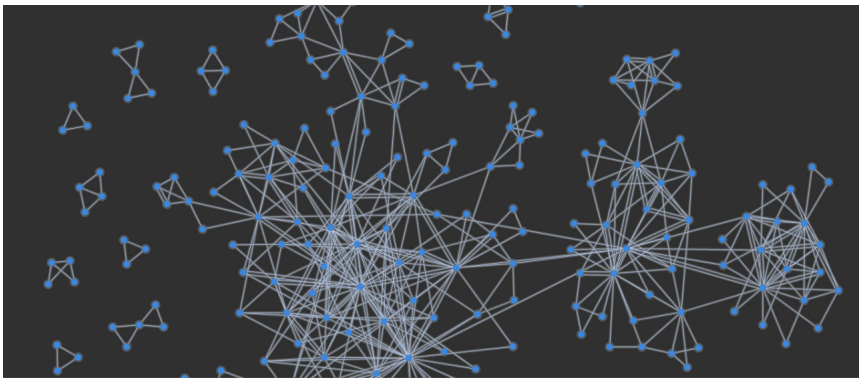

Figure 4: The above is the visual representation of the triangles algorithm result

Our original effort had the idea of comparing each database on identical algorithm for performance. However, as soon as we set up the database it became obvious that in the scope of our project this will be impossible. This is since the relational database existed on a local machine while the graph database existed remotely in the cloud.

Another factor that contributed to our inability to assess the actual run time was our choice of data. While we tried to choose a real life data set, the fact that we were using a free tier for graph database severely limited computing capacity. The computation would time out even after increasing timeout to over an hour. Thus we had to limit ourselves to such algorithms as could be run in these restricted environments.

Implementing algorithms also was not difficult. However, the relatively new status of gsql as a graph querying standard. Except for documentation on Tiger Graph and a few other scant resources, the community around gsql is nascent. Considering the limited time for this project the pattern functionality of tiger graph made implementing the algorithm easier. For an explicit comparison StackOverflow.com is numbering over five hundred thousand questions tagged with [sql] and zero tagged with [gsql].

When it came to replicating graph algorithms in a relational database, ancillary tables and views were needed. In the execution of shortest path, it was also clear that the estimated run time would be much higher than for the graph database because of the need for querying tables to find a single node's neighbor; an operation which a graph database could do in constant time.

References

[1] TigerGraph Documentation. Accessed July 2nd, 2021. https://docs.tigergraph.com/.

[2] TigerGraph Algorithms library. Accessed July 5th, 2021. https://docs.tigergraph.com/tigergraph-platform-overview/graph-algorithm-library.

[3] Representing graphs with MySQL. Accessed July 6th, 2021. https://www.artfulsoftware.com/mysqlbook/sampler/mysqled1ch20.html.

[4] Code for implementing graph algorithms in GSQL. Accessed July 4th, 2021. https://github.com/tigergraph/gsql-graph-algorithms/.

[5] Higgs Twitter Data set. Accessed June 30th, 2021. http://snap.stanford.edu/data/higgs-twitter.html

[6] R, Ramakrishnan, Database Management Systems. Boston, MA: McGraw Hill, 2003.