CSE 291H Course Project
Planning and Evaluation of Graph Queries Based on Wander Join

Xiuwen Zheng

June 2018

Contents

1 Project Framework Overview

In this graph data management project, we create a query processor for a fragment of the Cypher language. We parse Cypher queries and generate efficient execution plan using dynamic programming algorithm in which wander join is used to estimate cost to improve the performance of the system.

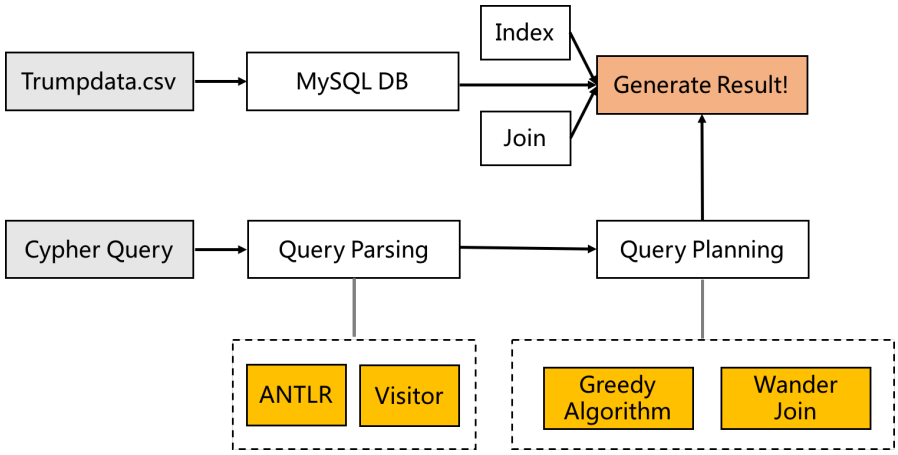

Figure 1: Project Framework.

The Framework of the The original dataset is a .csv file which contains the properties and label for each node and entity, and the connectivity information between nodes. This file is transformed to a RDBMS, a MySQL database specifically. Given a Cypher query, it is firstly parsed by using ANTLR. The constraints on each variable are generated, each of which is regarded as an execution unit and the return variables are also generated. The query execution units are then be ordered by using a dynamic programming algorithm combining with wader join for cost estimation in order to improve the performance of the system. After generating the execution order, the DBMS operation, SELECT and WHERE, and a hash join algorithm are used to execute each execution unit and after execution we can answer the cypher query.

2 MySQL Database Schema

I transfered the trumpdataset.csv file into two tables in MySQL database, entity table and connection table. For the entity table, the schema is "node_id, node_name, node_type" (node_type in the schema denotes node label). We get all node names and corresponding labels from the original file, make sure there is no duplicate node and then assign each node a node id. For nodes whose names contain "Bank" or "Hotel",

we assign them an extra label. Because one node can has multiple labels, *(node_id, node_type)* are used as primary key.

For the connection table, the schema is "edge_id, fromNode_id, toNode_id, connection". We get the connection relationship of a pair of nodes from original file and use their node id to denote them. We maintain a list of key words and if one key word appears in the connection string of two nodes, we use the key word as the label (connection) of the edge.

Because each edge here only has one property, sources, which I think is not an important property. I do not build a table to maintain the property information. Considering the number of nodes with multiple labels is very few, the duplication in the entity table is tolerable and is not a issue.

3 Query Parsing

For any valid Cypher query, we should parse it into several execution units and catch the information of what need to be returned. Here ANTLR is used to generate the parse tree of a fraction of Cypher gramma and visitor methods are extended to collect information which is needed to execute and answer the query.

3.1 Parse Results

We collect the constraints on each variable from the MATCH and WHERE clause. The constraints are divided into two types: Select condition and Join condition. Select condition is the label or property constraints on each node or edge. Join condition is the connection constraints of two node variable and one edge variable. I use two ArrayLists to store the two types of constraint respectively. In addition, there is another return ArrayList that collect the variables needed to be returned parsed from the RETURN clause.

Figure 2 is the structure for each ArrayList.

3.2 Examples of Output

Figure 4 shows a cypher query and the corresponding parse results is in Figure 3, and the corresponding parse tree is in Figure 5.

```
Class Relation{
        String edgeAlias
        String toVertexAlias
        String fromVertexAlias
}
ArrayList  < Relation >


Class Constraint{
        String edge(vertex)Alias
        String property
        String rightConstant
}
ArrayList < Constraint >

Class Return{
        String edge(vertex)Alias
        String property
}
ArrayList < Return >
```

Figure 2: ArrayList Structure

Figure 3: Parsing Result for a query

```
MATCH (a)-[e] -> (b)
WHERE a.node_type = "Person" AND b.node_type = "Person"
RETURN a.node_id, b.node_name, e.edge_id
```

Figure 4: A simple Cypher query example.

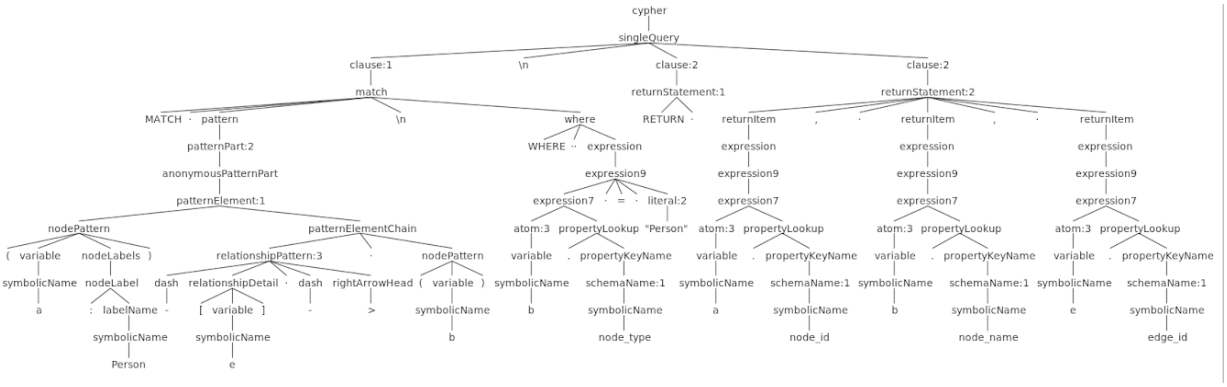

Figure 5: Parsing Tree for a query.

4 Query Planning

Figure 6 and 7 show a pseudo-Cypher query and the corresponding join graph. In the pseudo query, a, b, c, d, e denote nodes with labels $L1$ to $L5$ respectively, $r1$ to $r4$ denote edges with labels $N1$ to $N4$ respectively. In the graph, A is the table consisting of all nodes with label $L1$ and so on. After query parsing, we can generate this join graph. In this part, we need to determine the effective join ordering.

```
Match (a: L1) – [r1: N1] -> (b: L2) –[r2: N2] -> (c:L3),
Match (d: L4) – [r3: N3] -> (b),
Match (b) – [r4: N4] -> (e:L5)
Where a.name = xxx
Return a.id, d.id, e.id
```

Figure 6: A pseudo Cypher query.

4.1 Order Planning Algorithm

Greedy algorithm was used initially to determine the join order, while the performance is not good, because even though the greedy strategy choose best one in each step, the plan overall may be much different from the optimal one. Thus a DP-like strategy is used to calculate the cost of each plan and generate the optimal join ordering. I generated all possible join ordering plans and estimate their cost iteratively. The cost function I choose is:

$$Cost(A, B) = Cost(A) + Cost(B) + Size(A) * Size(B) \quad (1)$$

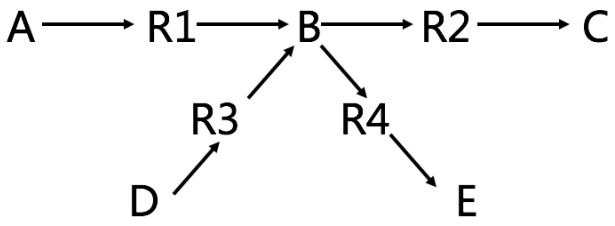

Figure 7: Corresponding Join Graph.

, where A and B are sub-plans and they can be joined. The Cost for each node in the join graph is initially the size of the table that the node represents. We iteratively generate all join plans and calculate the cost based on this equation. In the order1 table, the join table entries are all the nodes in join graph, in the order2 table, it includes all pair of nodes that can be joined and stores the cost. In the order N table, it will include all the plans generated by combining table entries from (order1, orderN-1) tables, (order2, orderN-2) tables to (orderN/2 , orderN/2) tables. Figure 8 shows an example. To generate order 3 table entries, we traverse the order 1 table entries and the oder 2 table entries respectively, take each entry from each table, generate a new order 3 table entry if the two entries can be joined.

4.2 Introduction to Wander Join

In the order planning algorithm, when we calculate the cost of join, we need to estimate size of join sub-plans. Instead of using the pre-computed statistics, I utilized a dynamic method to estimate the intermediate join size. Here I introduce an online aggregation method, wander join, to estimate join size. Online aggregation is mainly used to estimate the aggregation function without executing full join on tables. Estimating join size is equivalent to estimate the COUNT aggregation function. Thus, it is reasonable to introduce this method to estimate the intermediate join size.

Wander join is one of the efficient online aggregation methods which uses random walk to sample paths in the join graph. It uses the Thompson estimator to estimate aggregation function. Suppose a path γ is sampled with the probability $p(\gamma)$, and the expression on the path to be aggregated is $v(\gamma)$, then $v(\gamma)/p(\gamma)$ is an unbiased estimator for the aggregation function value. We independently perform multiple random walks, and take the average of the estimators. Since each $v(\gamma_i)/p(\gamma_i)$ is an unbiased estimator, their average is also unbiased estimator.

In order to calculate the sampling probability of path, we should build index on the variable to be joined, which is the most burdensome work in the implementation of this method.

4.3 Implementation Details

4.3.1 Walk order of Wander Join

To estimate the result join size of a sequence of join operations, we need to first find out a valid path to walk through all tables. Here, I only consider chain join and use backtracking method to find a valid path. In order to determine whether two tables can be joined, I utilize a simple structure to store join conditions which is a map where each table's name is a key and the corresponding value is a set of tables' name which can join with this table. Different walk order would result in different convergence rate of wander join, but I do not consider it further and just simply choose one valid path.

4.3.2 Wander Join

For wander join, building index is one of the most important parts. For each variable that two tables join on, we need to build index on this variable for the second table in order to estimate the sampling probability of the path. So we need to maintain the information that based on what variables that the two tables join on. The map structure in the last part is not applicable now because it only contains information that whether two tables can join.

I utilize a google graph package to build a join graph. The template is *<String, Map<String, String>>* which keeps all the node and edge information. The edge here stores the name of the variables that two nodes join on. After generating a valid path in last section, here we get the join variables from this graph structure and then implement wander join on this.

4.4 Order Planning Result

For the Cypher query shown in Figure 1, we generated the following plan table as Figure 8 shows that contains all possible join plans built from order1 join to final order3 join with estimated cost.

The optimal join plan is chosen as the plan with minimum cost.

5 Cypher Query Results Generating

5.1 Overview

Firstly, after parsing the Cypher query, we get an ArrayList which stores all the label or property constraints on node and edge variables. We transform it to an MySQL query "SELECT * from table where" and generate a result table for each variable. Then executing the join ordering algorithm to generate join plan. After generating join plan,

```
plan table:
+-------+-------------------------+----------------+
| order | join_expression         | estimated_cost |
+-------+-------------------------+----------------+
| 1     | [ a ]                   | 0              |
+-------+-------------------------+----------------+
| 1     | [ b ]                   | 0              |
+-------+-------------------------+----------------+
| 1     | [ e ]                   | 0              |
+-------+-------------------------+----------------+
| 2     | [ (a JOIN e) ]          | 2156440        |
+-------+-------------------------+----------------+
| 2     | [ (b JOIN e) ]          | 2156440        |
+-------+-------------------------+----------------+
| 2     | [ (e JOIN a) ]          | 2156440        |
+-------+-------------------------+----------------+
| 2     | [ (e JOIN b) ]          | 2156440        |
+-------+-------------------------+----------------+
| 3     | [ (a JOIN (b JOIN e)) ] | 442222044      |
+-------+-------------------------+----------------+
| 3     | [ (a JOIN (e JOIN b)) ] | 293310930      |
+-------+-------------------------+----------------+
| 3     | [ (b JOIN (a JOIN e)) ] | 107628686      |
+-------+-------------------------+----------------+
| 3     | [ (b JOIN (e JOIN a)) ] | 327269118      |
+-------+-------------------------+----------------+

optimal join plan is: (b JOIN (a JOIN e))
stack pop sequence:
e
a
(e JOIN a)
b
```

Figure 8: Plan Table with Estimated Cost.

use the nested join algorithm written by myself to execute join. After executing all units, return the variable attributes in the Return ArrayList.

5.2 Examples

Figure 9 is the partial result for the query in figure 1. The execution series is to select tables for each variable based on their label or property constraints, then conduct (e join a) and then (e join a) join b.

6 Future Work and Drawback

6.1 Drawbacks

This project innovatively introduce an online aggregation method, wander join, to estimate intermediate join size. However, there are a lot of remaining work to do because of limited time.

Firstly, I do not implement the walk plan optimizer, which could increase the convergence rate of wander join. I do not use benchmark to estimate the performance of this method including the convergence rate and percentage error. In addition, I do not do comparison experiment between my join size estimation method based on wander join and other traditional join size estimation methods. Most importantly, the wander join algorithm implemented here can only deal with the chain join, and can not deal with acyclic chain or cyclic chain, which need to be further considered.

In addition, I only do the planning for join operations without considering cardinality. I just simply assume that all the selections by label or property are executed first and then do the join planning for the tables generated, which harms the efficiency of the system.

6.2 Not Finished Work

I used nested join algorithm to execute join, which is not very efficient. If time is enough, I would use other algorithms, like hash join. I wrote some graph algorithm, like reachability query, but because of the limited time, I have not add it into Cypher language. For the path index, I initially decide to do the 2-path index, however, I found that in the dataset, there are not many long-length path, I do not want to waste space for building index. However, for other datasets which are densely connected, path index would be helpful to improve the efficiency.

query result is:

a.node_id	b.node_name	e.edge_id
557	ANNIE DONALDSON	10
557	DON MCGAHN	11
593	BRAD PARSCALE	41
2325	JACK WEISSELBERG	46
921	ZIYA MAMMADOV	74
1110	MIKE FERGUSON	81
1110	MITCH MCCONNELL	82
1110	SCOTT WALKER	85
203	WILLIAM P. FOLEY	101
2263	KIRILL DMITRIEV	110
236	JARED KUSHNER	124
1970	BARRY BENNETT	148
1383	DANIEL STEINMETZ	160
1189	KEVIN MARINO	219
1422	DONALD J. TRUMP	257
484	ALAN HAMMER	267
484	BENJAMIN NETANYAHU	268
484	DARA ORBACH	269
484	ESTHER KUSHNER	270
484	JIM MCGREEVEY	271
484	JOSHUA KUSHNER	272
484	MURRAY KUSHNER	274
484	NICOLE KUSHNER MEYER	275

Figure 9: Query Results.

7 Reference

[1] Haas, Peter J., and Joseph M. Hellerstein. ”Ripple joins for online aggregation.”ACM SIGMOD Record28.2 (1999): 287-298.
[2] Li, Feifei, et al. ”Wander join: Online aggregation via random walks.”Proceedings of the 2016 International Conference on Management of Data. ACM, 2016.