High-level query language for Graph Databases

Manasi Sunil Bharde
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
msb4977@cs.rit.edu

***Abstract*—To extract information from the relational database it is necessary to perform operations like Join, Select and filtering the data with criteria. Each tuple in a relational database table is like subgraph in graph database with labels and\or values. Relational algebra has structured query language which uses relational algebra. Optimizer for relational database considers span of tables in a query, span of columns in query and indexes for choosing optimal query execution plan. Similar to relational algebra, graph algebra and algorithms can be used to design a system to query specific information, patterns from the graph database. This system builds a query language for graph databases which will follow intuition of SQL for a relational database. The performance of the system will be tested for different representations of the data in Neo4j. Results are verified against the ground truth by SQL on RDBMS.**

***Index Terms*—Graph database; GraphQL**

I. Introduction

Query operations: selection, projection, join, group by and order by are required to extract meaningful information from a relational database. When relational data is transformed to a graph database, query graph needs to be operated on graph database to query data. In graph databases data is represented with relations without the use of redundant space that occurs in the de-normalized relational database. Using graph database eliminates a need for joining the multiple tables and accessing different directory pages to retrieve records belonging to different tables. Use of graph algebra to implement query operations on graph database enables optimization based on graph structure of data [1].

A. Limitations with Neo4j API

Neo4j graph database API considers the disconnected graphs as subgraphs of a single large graph. Hence to consider them as different graph tuples, candidate set generation is necessary at each stage. The graphs are operated as undirected graphs. Neo4j also has a limitation that it cannot deal with two different databases having same definition or partial overlapping definition. Existing query language for Neo4j: Cypher has a format which is difficult to understand for a programmer using SQL. Cypher has following limitations:

- Creates candidate set by computing cross product of all possible values for all pattern variables. E.g. if there are vi unique values for all variables, the candidate set will contain candidates equal to the product of all such vi.
- Follows relationship uniqueness. Single node entity can be related to a unique relationship in one match clause. E.g. to define Cypher query with a pattern where node v1 has two different relationships, two match operations are performed. The query will fetch matching subgraphs using 1st subpattern and then apply 2nd subpattern to create a composite pattern.

B. Query Operations for Tree Algebra XML

Tree database in the form of XML files allows heterogeneity in the definition of data which makes TAX schema-free or NoSQL database [2]. Nodes in the tree have tag and content as attributes. It also stores pedigree which represents a history of the node through different query operations. Timber framework uses XML files as a database. To perform queries on TAX, a query is represented in the form of pattern tree. Pattern tree is represented by pre-order code and set of edges. Edges in pattern tree are represented in terms of Ancestor-Descendant (AD) or Parent-Child (PC). Where PC represents direct edge and AD represents the transitive edge. In order to match tuple in tree database, it should have a same pre-order code and set of edges. Listing 1 is an example of a tuple in an XML database.

Listing 1: XML tuple

```
<movie name = Sense and Sensibility>
   <year year = 1995></year>
   <actor name = Emma Thompson></actor>
</movie>
```

Following are query operations that can be performed with Tree Algebra XML Database.

1) Selection: Matching the structurally related nodes and their descendants using pattern tree

2) Projection: List of matching nodes from projection list and their descendants

3) Product Join: New root node is created for each pair of the tree. The root of left and right part of join connect to the new root node.

4) Group by: Unlike SQL the Group by operator is separated from aggregation operator in TAX. It provides pattern tree and grouping list as a query. Group by resultset contains trees with New root node with group by criteria as one child and tree tuple matching group as another child. Partitions are created based on Group by value. Default comparison of a node is shallow.

5) Aggregation: Aggregation is performed on top of partitions created using group by.

6) Order by: Ordering list is applied to the candidate set.

C. Adapting Query Operations for using Graph Algebra to Graph DB

Similar to Tree Database, to run queries on graph database, queries are converted to query graph. To identify matching graph tuples from graph database, subgraph isomorphism is performed. GraphQL is a query language over graph databases that defines and uses graph algebra [1]. Similar to tree algebra instead of flattening the the graph structure, graph algebra operates on graph tuples. Hence it uses graphs as input and gives graphs as output. Following are query operations that can be performed with Graph Algebra on Graph Database.

1) Selection With Pattern: Retrieves subgraphs matching with the query graph. As seen in equation (1), selection over a pattern matches value of attribute as well as pattern.

$$\sigma_{P=p}(T) \quad = \quad \Phi_{P=p}(G) \quad | \quad G \quad \epsilon \quad T \tag{1}$$

2) Selection Without pattern: Retrieves nodes satisfying attribute value criteria. This operation filters nodes as per criteria on attribute. List of nodes is returned as per list of criteria as seen in equation (2).

$$\sigma_P(N_1, N_2, ..N_N) = \phi_1(n_1) + \phi_2 n_2 + .. + \phi_n n_n | n_n \quad \epsilon \quad N_n \tag{2}$$

3) Projection With Pattern: Retrieves subgraphs matching with the query graph and node labels in projection list. This operation returns subgraphs in pattern without a criteria.

$$\Pi_p(T) \quad = \quad \Phi_p(G) | G \quad \epsilon \quad T \tag{3}$$

4) Projection Without Pattern: Retrieves nodes from the projection list.

$$\Pi_P(N_1, N_2, ..N_N) = \phi_1(n_1) + ..\phi_n(n_n) \quad | \quad n_n \quad \epsilon \quad N_n \tag{4}$$

5) Composition: Perform combination of multiple operations in a possible order. Operations in query can be reordered. Composition operation specified in equation (5) shows one composition for given query.

$$M_{o_1 o_2 .. o_n}(T) = M_{o_1}(M_{o_2}(..M_{o_n}(T)) \tag{5}$$

6) Join:

- Perform cartesian product of candidate subsets from both sides of join. Then apply join pattern to find target tuples from Cartesian product set.
- Perform subgraph matching on each side of join. Join on condition node using new type of edge: JoinEdge.
- Perform subgraph matching on each side of join. Then unify tuples that join on each side by using only one node of join condition. This operation is seen for movie-actor and movie-year relations. Consider two databases actor and year in figure 1 and figure 2 respectively. To retrieve data for a movie, join is performed on movie name as seen in figure 3.

$$J(L_P, R_Q) = \sigma_{P=p}(L) \quad join \quad \sigma_{Q=p}(R) \tag{6}$$

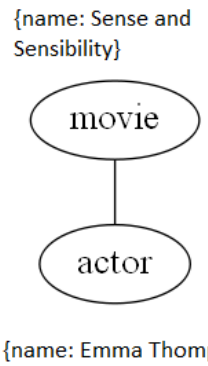

Fig. 1: Movie-Actor database tuple

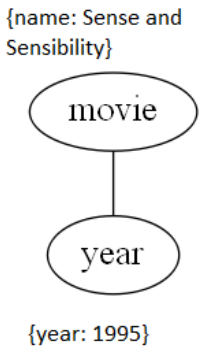

Fig. 2: Movie-Year database tuple

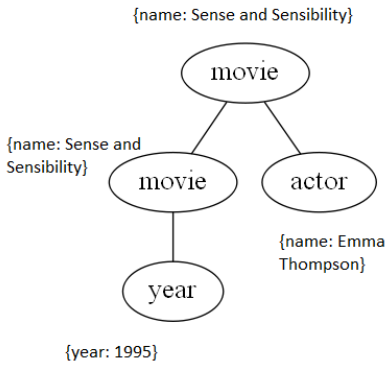

Fig. 3: After join: Movie-Movie-Actor-Year subgraph

7) Group By: Perform subgraph matching for group by clause. Perform shallow comparison.

8) Aggregation: Perform in memory aggregation operation on group by partitions.

9) Order By: Order graph tuples by order list.

II. Implementation Details

A. Data Model

Neo4j is NoSQL or Schema-Free database however in order to make queries more efficient, logical or high-level partition of data can be done by using following data representation methods:

1) Single node representing one partition is connected to all nodes belonging to a partition.
2) Add label representing a partition to all nodes belonging to a partition.
3) Using two different databases.

Listing 2: Cypher query

```
MATCH (a1)--(a2),(a3)--(a4)
WHERE a1:movie AND a1:actor-db
AND a2:actor AND a2:actor-db
AND a3:movie AND a3:year-db
AND a4:year AND a4:year-db AND
a1.name='Sense and Sensibility' AND
a3.name='Sense and Sensibility' AND
a2.name='Emma Thompson'
RETURN a1,a2, a3,a4
```

1) Comparison between representations: For the purpose of this project, I worked with 2nd representation. Since labels are indexed by default in Neo4j, using this approach improves the performance of querying data from Neo4j over identifying relation which is required in 1st representation. 3rd representation is unarguably most poor due to multiple connection and disconnection cost involved.

B. Parser for query language

I worked on creating a parser for query language using JavaCC [3].

C. Join Operation

In Listing 2, two disjoint graphs G1 and G2 are queried with independent predicate and nodes from these disjoint graphs are projected. Since there is no way to specify join condition in Cypher query [4], Cypher performs expensive Cartesian product of two disjoint graphs when projection contains node labels from disjoint graphs as observed in Figure 4. The Figure 4 is generated using the profile clause in cypher query.

Since join over predicate is less costly than join without a predicate, in proposed query language join is performed by executing left and right parts of join separately and creating the join by adding the new relationship between nodes satisfying predicate. This avoids Cartesian product of disjoint graphs. Performance of this operation is seen in Table I for small number of nodes. Algorithm 1 is expected to perform better over traditional Cypher Cartesian product for large number of nodes.

Algorithm 1 Join

Input: Values for left attribute of join

1: **for** $left\_value$ in $Values$ **do**
2: Construct $query$ by adding match clause
3: Add predicate to get left node with value $left\_value$ and right node with value $left\_value$
4: Add 'Join' relationship between such left and right node
5: Run $query$
6: **end for**

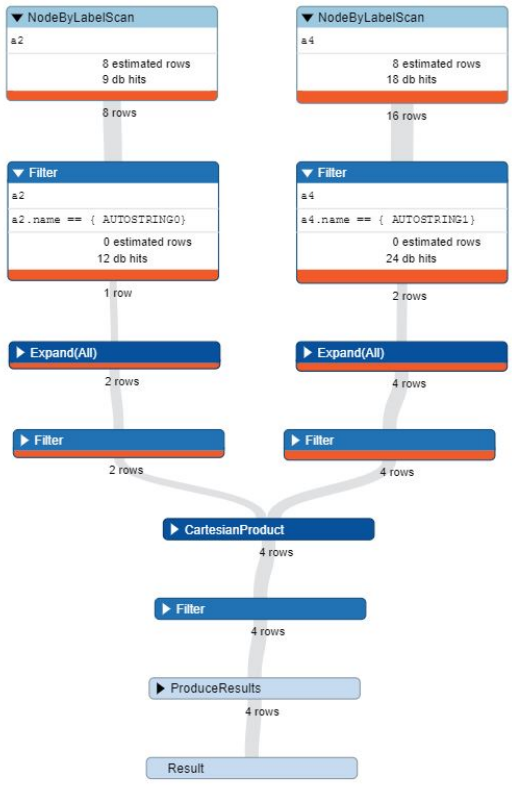

Fig. 4: Profile for query over disjoint graphs

Listing 3: Query for join in proposed high-level query language

```
SELECT m1 movie a actor
FROM actor-db
SELECT m2 movie y year
FROM year-db
PROJECT a m y
MATCH
m1 a
m2 y
WHERE
a.name= "Emma Thompson"
JOIN
actor-db.m1.name = year-db.m2.name
```

TABLE I: Cost of join operations

Number of Nodes	4	8	16
Number of Nodes in projection	2	4	8
Number of relations	2	4	8
Time for decomposed query (ms)	2689	2617	3572
Time for complete query (ms)	2341	2921	3145

D. Groupby Operation

The Group by operator is used to extract aggregated information from data. In a graph database, data can be grouped by labels by performing simple match operation. However, no operator is available in Cypher or Neo4j API to

group nodes of a label which follow provided subgraph pattern and criteria on attribute. In proposed query language, group by operator enables a user to specify the label and attribute value to calculate aggregation of some other attribute value. Multiple attributes can be specified in the group by list as seen in Listing 4.

For computing multiple group by for aggregation, distinct values of each specified attribute are calculated. As seen in algorithm 2 All combinations of one attribute value with each value of other attribute are computed using a recursive function. Aggregation operator value is calculated for each combination and it is stored as another attribute in the last label in the group by list.

Algorithm 2 group_by

Input:$distinctValues$, $label$, $query$

1: **for** each value from $distinctValues$ of attribute at $index$ **do**
2: Append value to $label$
3: Add where clause $query$
4: **if** index is last **then**
5: Calculate aggregate function $\rightarrow aggrValue$
6: Add attribute of $label$ at node of $index$ with value = $aggrValue$
7: **else**
8: groupBy($index$+1, $query$, $label$)
9: **end if**
10: **end for**

Listing 4: Query for group by in proposed high-level language

```
SELECT
d director
m movie
g genre
MATCH
m g
m d
GROUP BY
d.first g.genre
COUNT(m.year)
WHERE
d.first="Timothy" AND d.last="Wilde"
```

III. Experimental setup and Results

For computing the performance of the system single Neo4j node residing on 8GB RAM and 500 GB hard-disk is used. While executing queries, Neo4j stores the query plan in cache. Page cache memory is set to minimum value of 256 MB to lessen the impact of caching while measuring performance. String block size is set to 60 and array block size is set to 300. IMDB dataset is used for testing.

A. Relabeling intermediate results

Relabeling intermediate result data graph re-indexes it. With this intuition when many operations are chained together, it makes difference to have indexed intermediate result. After each operation in the query graph is applied to data graph, resultant data graph is relabeled. The query graph is also modified to represent new labels. Relabeling results at each stage makes it possible to perform all the computations at the database and eliminates the need for in-memory computation.

B. Altering order of operations

Neo4j: Cypher query chooses best execution plan by altering the order of operations in the query. As observed in Figure 4, Cypher executes all the predicates in the query in early stages to perform minimum data handling in later expensive subgraph matching stages of execution. Cypher query engine makes use of indexes created on node attributes to identify efficient query plan. In section C, performance of different filter plans is evaluated by altering order of operations.

C. Computing different compositions of Query

Proposed query language provides two query optimization strategies to choose from.

1) Filter plan based on span of predicate: One query plan orders predicates by comparing the node sizes for labels in the predicate. Then applies the predicate on result set obtained from pattern matching in decreasing order of node sizes so that maximum nodes are filtered first. It finds the optimal predicate order by sorting the composite where clauses by the size of individual where clauses in it and sorting the individual where clauses by a number of nodes in target graph that the labels in where clause cover.

The hypothesis is since a predicate covers more number of nodes it will filter more hence priority is given to a predicate having a label with more number of nodes. During query execution, subgraph matching for the query is performed by executing Cypher query over graph database. Relabeling is used to eliminate in-memory processing of intermediate results. However such optimization does not lead to significant improvement in case of queries containing a small number of nodes as seen in Table II.

TABLE II: Comparison of queries with optimizer based on number of nodes covered by predicates

Number of Nodes	6	5	2
Number of Nodes in projection	5	4	2
Number of relations	8	4	2
Predicate then match (ms)	12879	13351	7167
Match then predicate (ms)	3665	3925	3826
Complete Cypher query (ms)	4012	3057	2551

2) Filter plan based on selectivity of predicate: Second query plan finds selectivity of each predicate by querying distinct values of an attribute in predicate. Predicates are ordered in decreasing order of selectivity. This filter plan is used in relational databases like h2.

TABLE III: Comparison of queries with optimizer based on selectivity of predicate

Number of Nodes	6	5	2
Number of Nodes in projection	5	4	2
Number of relations	8	4	2
Predicate then match (ms)	36699	35037	11032
Match then predicate (ms)	3926	4085	4107
Complete Cypher query (ms)	3693	3360	2485

IV. Conclusion

A graph database can gain the power of relational database by adding operations such as Join and group by. Implementing a Join operator using relabeling gets rid of Cartesian product cost. Relabeling a result in the database allows reordering of different operations in the query without using extra memory or computation. Hence relabeling is used to compare the performance of different order of query operations. Performing subgraph matching before applying predicate always gives better performance over applying predicate and then performing subgraph matching. This is the result of the linked list like storage of data and weak indexing support for attributes in Neo4j. Query optimization based on the selectivity of a predicate is slower than query optimization based on span of predicate since the earlier involves the cost of calculating distinct values for an attribute in the predicate.

Acknowledgment

I would like to thank Prof. Carlos Rivero for guiding me through the capstone and giving all essential inputs. I would also like to thank Prof. Richard Zanibbi for important guidelines on how to design, build and implement capstone.

References

[1] H. He and A. K. Singh, "Query language and access methods for graph databases," in *Managing and Mining Graph Data*, 2010, pp. 125–160. [Online]. Available: https://doi.org/10.1007/978-1-4419-6045-0_4

[2] H. V. Jagadish, L. V. Lakshmanan, D. Srivastava, and K. Thompson, "Tax: A tree algebra for xml," in *DBPL*, vol. 1. Springer, 2001, pp. 149–164.

[3] The java parser generator. [Online]. Available: https://javacc.org/

[4] F. Holzschuher and R. Peinl, "Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j," in *Joint 2013 EDBT/ICDT Conferences, EDBT/ICDT '13, Genoa, Italy, March 22, 2013, Workshop Proceedings*, 2013, pp. 195–204. [Online]. Available: http://doi.acm.org/10.1145/2457317.2457351

APPENDIX

Listing 1: Definitions

```
AGGR_Operator = MIN\MAX\AVG\SUM\COUNT
operator = =\!=
```

Listing 2: Select..from syntax

```
SELECT
node1 node_label1
node2 node_label2
FROM db_label
MATCH
node1 node2
```

Listing 3: Select syntax

```
SELECT
node1 node_label1
node2 node_label2
MATCH
node1 node2
```

Listing 4: Project syntax

```
SELECT
node1 node_label1
node2 node_label2
MATCH
node1 node2
PROJECT node1
```

Listing 5: Select... where syntax

```
SELECT
node1 node_label1
node2 node_label2
MATCH
node1 node2
WHERE
node1.property1 = "value1" AND\OR node1.
    property2 = "value2"
```

Listing 6: Select.. where.. project syntax

```
SELECT
node1 node_label1
node2 node_label2
MATCH
node1 node2
PROJECT node2
WHERE
node1.property1 = "value" AND\OR node1.
    property2 = "value2"
```

Listing 7: Group By syntax

```
SELECT
node1 node_label1
node2 node_label2
node3 node_label3
MATCH
node1 node2
node1 node3
GROUP BY
node1.property1 node2.property2
COUNT(node3.property3)
```

Listing 8: join

```
SELECT
node1 node_label1
node2 node_label2
FROM db_label1
SELECT
node3 node_label3
node4 node_label4
FROM db_label2
PROJECT node1 node2 node3
MATCH
node1 node2
node3 node4
WHERE(optional)
node1.property1 <operator> "value1" AND\OR
    node1.property2 <operator> "value2"
JOIN
db_label1.node1.property1 = db_label2.node2
    .property2
```

Listing 9: Group By.. where syntax

```
SELECT
node1 node_label1
node2 node_label2
node3 node_label3
MATCH
node1 node2
node1 node3
GROUP BY
node1.property1 node2.property2
AGGR_Operator(node3.property3)
WHERE node1.property1 <operator> "value"
    AND\OR node1.property2 <operator> "
    value2"
```

Listing 10: Group By.. project .. where syntax

```
SELECT
node1 node_label1
node2 node_label2
node3 node_label3
MATCH
node1 node2
node1 node3
GROUP BY
node1.property1 node2.property2
AGGR_Operator(node3.property3)
PROJECT node3
WHERE node1.property1 <operator> "value"
    AND\OR node1.property2 <operator> "
    value2"
```