# Implementation of a basic CYPHER query processor using a PostgreSQL backend

AMARNATH RAJU VYSYARAJU, University of California, San Diego, USA

This is an attempt to implement a basic Cypher query processor that takes in a simple cypher query and renders the results in an intelligent fashion by using minimum cardinality estimation of the nodes and the connections in the query. This is done as part of a graduate course project for the course CSE 291H- Management of Large scale graph data.

Additional Key Words and Phrases: Graph data, Cypher, PostgreSQL, Minimum cardinality estimation

**ACM Reference Format:**
Amarnath Raju Vysyaraju. 2010. Implementation of a basic CYPHER query processor using a PostgreSQL backend. *ACM Comput. Entertain.* 9, 4, Article 39 (March 2010), 11 pages. https://doi.org/0000001.0000001

## INTRODUCTION

### Back-end

The back-end database has been implemented with the help of PostgreSQL. These are the Tables and their respective columns.

(1) Nodes
  (a) id
  (b) name
  (c) node_type (This is equivalent to the node label)
(2) Edges
  (a) id
  (b) source
  (c) description
(3) Connections
  (a) edge_id
  (b) a_node_id (Id of the first node of the edge)
  (c) b_node_id (Id of the second node of the edge)
  (d) relationship (This is the edge label)

Author's address: Amarnath Raju Vysyaraju, University of California, San Diego, CA, USA, avysyara@eng.ucsd.edu.


Manuscript submitted to ACM

**Data (TRUMPWORLD data)**

Although the system can be easily used on any data with the above schema, the data used to test and build the system has been derived from https://neo4j.com/blog/buzzfeed-trumpworld-dataset-neo4j/. These are some salient features of the data:

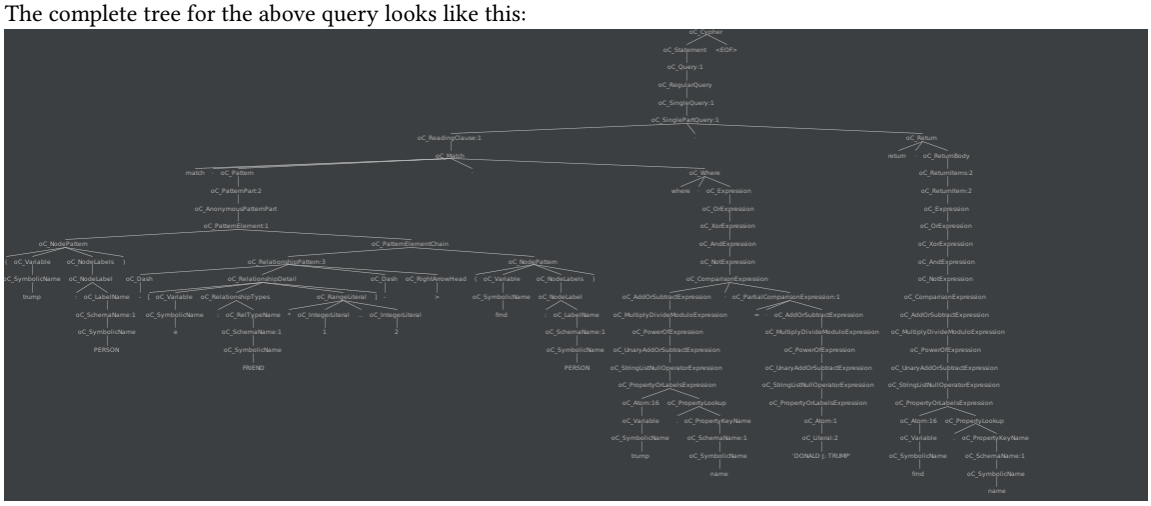
(1) Node Labels:
  (a) PERSON
  (b) ORGANIZATION
    ORGANIZATIONs can also have the following labels:
    HOTEL, BANK, BAR, CLUB
  (c) FEDERAL_AGENCY
(2) Relationship Types:
  (a) PERSON-to-PERSON relations have the following edge labels:
    MARRIED, SAT_IN, APPEARANCE, REPRESENTATIVE, LUNCHED, DIRECTOR, WHITE_HOUSE, AIDED, PARTNER, NOMINEE, ASSISTANT, RELATED, ADVISOR, CONSULTED, PARENT/CHILD, FRIEND, MET, COUNSELOR, GAVE, WORKED, BOUGHT, CLOSE, CAMPAIGN
  (b) ORGANIZATION-to-ORGANIZATION relations have the following edge labels:
    SHAREHOLDER, NEGOTIATION, INVOLVED, TENANT, LOAN, NKA/FKA, LICENSES, SUPPLIER, SUBSIDIARY, MEMBER, AFFILIATED, PARTNER, RELATED, SALE, DONOR, LOBBIED, SOLD, OWN, TIES, DBA, BOUGHT, PAID, AKA, INVESTOR
  (c) PERSON-to-ORGANIZATION/ORGANIZATION-to-PERSON relations have the following edge labels:
    SHAREHOLDER, LAWYER, INCOME, FOUNDER, APPOINTEE, MANAGER, PRESIDENT, BANKER, CONNECTED, PUBLISHER, SECRETARY, DIRECTOR, ADVISOR, INVOLVED, EXECUTIVE, COUNSEL, HIRED, STAFF, OWNER, CHAIR, RELATED, FELLOW, CEO, TRUSTEE, BOARD, AMBASSADOR, INVESTOR, DEVELOPER, SPEECH, PARTNER

**CYPHER QUERY PROCESSING:**

The Open Source cypher grammar has been used as the standard for the software. Antlr4 has been used to generate the Lexer and Parser for the grammar. The Parser helps to get the features from the cypher statement in a structured manner. Below is an example of how the parse tree for an example query looks like.

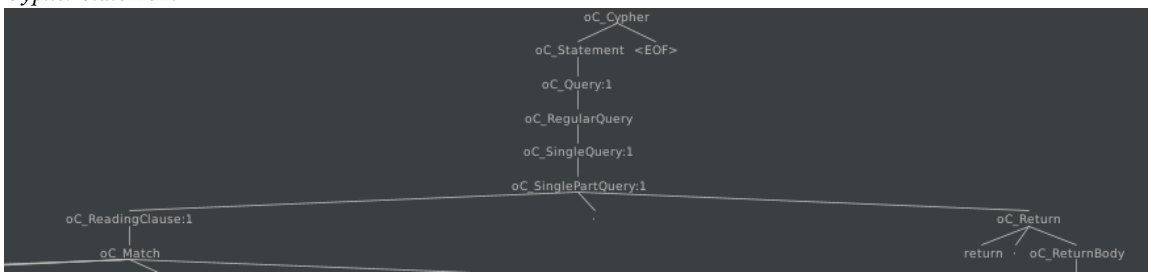
*MATCH (trump:PERSON)-[e:FRIEND*1..2]->(frnd:PERSON) WHERE trump.name = 'DONALD J. TRUMP' RETURN frnd.name*

(1) The complete tree for the above query looks like this:

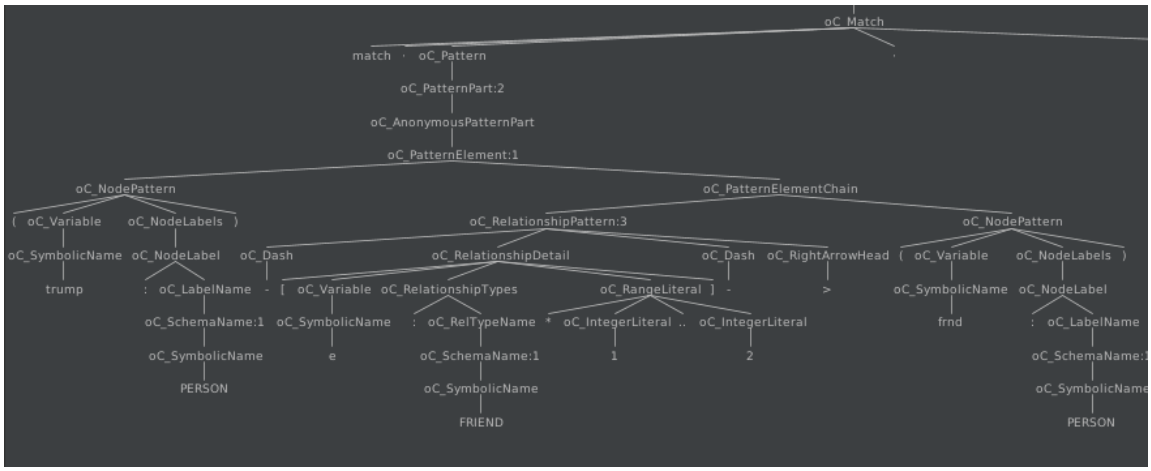


Let us look at each sub tree separately as follows:

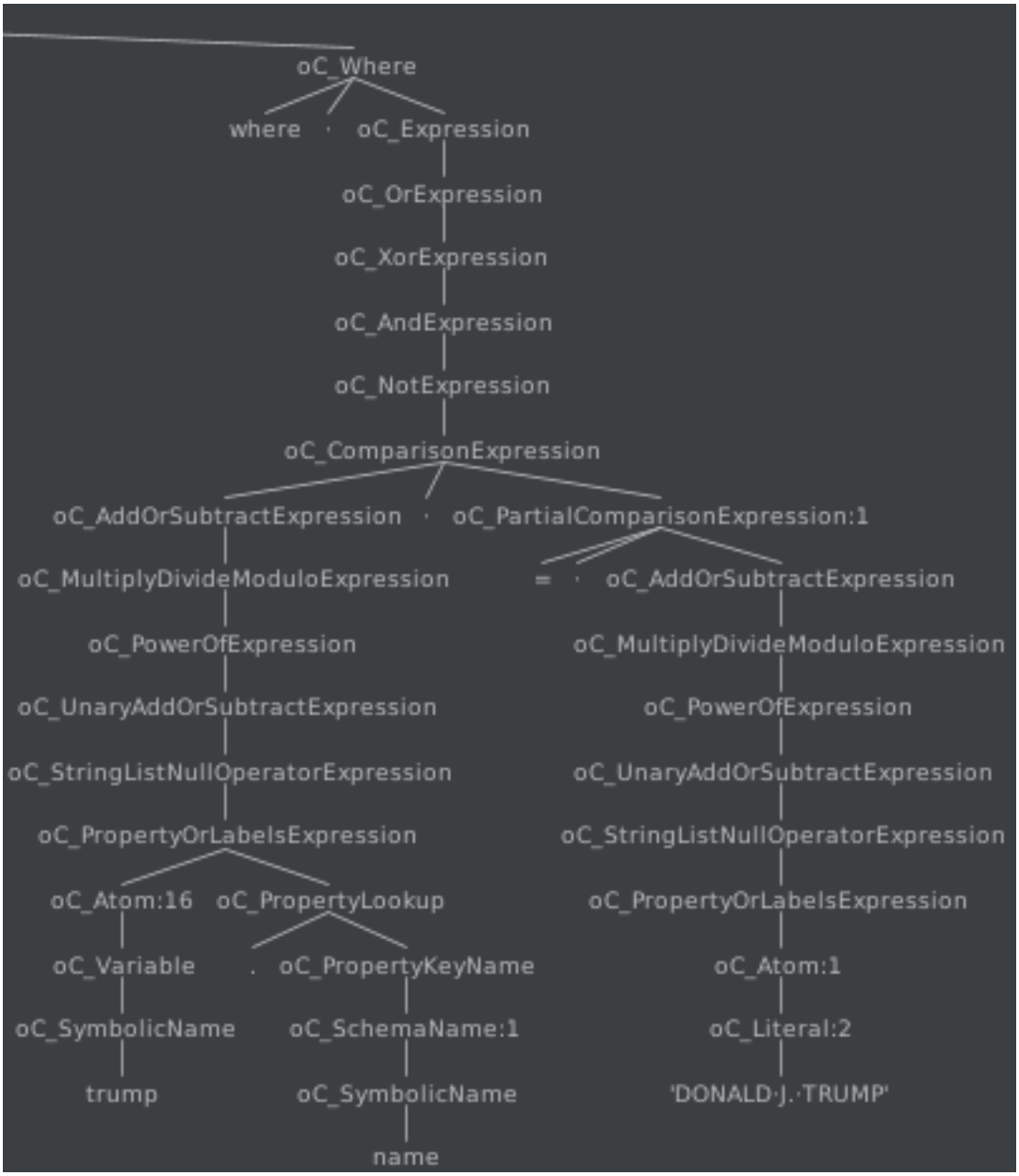
(2) *Cypher* statement:



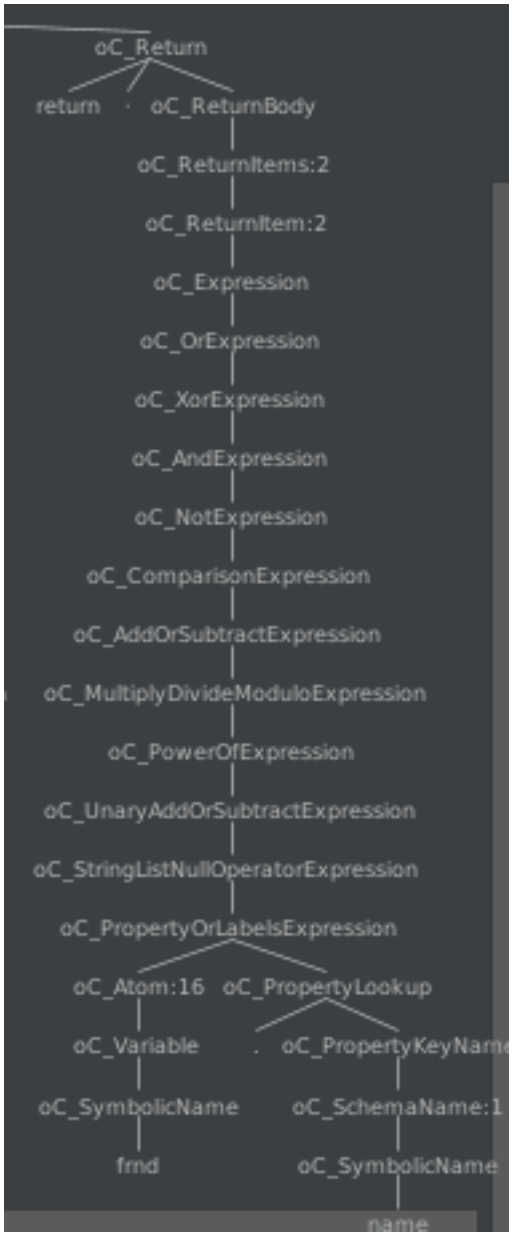

(3) *MATCH* statement:



(4) *WHERE* statement:



(5) *RETURN* statement:




**Elements captured:**

When a Cypher query is passed in, we capture the following elements.

(1) Capture all *Node* and *Connection* variables. Implicit variables are created if the variable names are not mentioned.
(2) For each Node variable we capture the following:
  (a) Labels

(b) Filter conditions in the *WHERE* clause
(c) The connection variables connected to it.

(3) For each Connection variable we capture the following:
(a) Direction : Left, Right, Both
(b) Relationship types
(c) The Minimum and the Maximum lengths of the connection if specified. Else, they are both defaulted to 1
(d) The two Node Variables that it connects.

**SQL CREATION:**

The system leverages heavily on the following methods. These methods take the necessary inputs, translate that into the appropriate SQL query, query the Backend database, and fetch the results.

(1) getNodeCandidates
(a) Input : nodeVar, nodeTypes, node Constraints, existing list of Candidates(if any)
(b) Output : List of candidate ids for nodeVar
(c) This method creates a query to the backend on the nodes table with the given nodeTypes and nodeConstraints as filters and also ensures that the output nodes are a subset of the existing list of candidates, if specified.

(2) getConnCandidates:
(a) Input: ConnVar, direction, minLength, maxLength, relationshipTypes, candidates for a_node, candidates for b_node
(b) Output: List of Candidate 'ConnectionInfos' for connVar
(c) This method creates a sub-query n the connections table with the filter on the relationshipTypes and re-uses the same sub-query by joining them to create the effect of the length of the connection and filters at the end based on the candidates of a_node and b_node.

The ConnectionInfo class is a custom class that has been defined with the following member variables: aNodeId, bNodeId, Sequence of edgeIds that connect from aNodeId to bNodeId

**BACKEND INDEXING:**

To Take advantage of the PostgreSQL database on the backend the following indexes have been added.

(1) nodes
(a) id, name, node_type
(2) connections
(a) edge_id, relationship, a_node_id, b_node_id
(3) edges
(a) id

All of these indexes have been implemented based on BTrees.

**GRAPH QUERY EXECUTION PLAN**

To take advantage of the graph structure of the data, we follow a greedy method to reduce the cardinality of the result as early as possible so that we don't need to work with an unnecessarily big result set in the end. In order to do this we need some information about the data we have in the database.

**Pre-Computed statistics:**

To help us in doing the greedy cardinality minimization we need to have some statistics about the database. These are the following:

(1) $C_n$ : Number of Nodes
(2) $C_e$ : Number of Edges
(3) $C_l(label)$ : Count of Nodes for a given Node label (i.e. node_type)
(4) $C_r(rel_type)$ : Count of Nodes for a given Edge label (i.e. relationship type)
(5) $C_{nr\_in}(node, relationship\_type)$: Count of incoming relationship_type edges to node
(6) $C_{nr\_out}(node, relationship\_type)$: Count of outgoing relationship_type edges from node
(7) $C_{lr\_in}(label, relationship\_type)$: Count of incoming relationship_type edges to all nodes with label
(8) $C_{lr\_out}(label, relationship\_type)$: Count of outgoing relationship_type edges from all nodes with label
(9) $C_{name\_word}(word)$: The number of node names where the 'word' occurs.

We fetch these statistics at the start of the application to help us with the Cypher inquiries when the database is queried.

**Node Cardinality estimation:**

We look into a probability based heuristic for choosing the first node to query the backend. This node should be the one which we presume to have the least cardinality.
We assign the following values of probability for each of the constraints on a Node Variable and multiply these constraints like we do for a Naive Bayes probability estimation.

- "Equal to" : $p(" = ") = 1/C_n$
- $p("Contains" or "Startswith" or "Endswith") = \Pi_{w \in QueryString} p_{name\_word}(w)$
  If 'w' is at the start/end of the QueryString (depending on the type of enquiry), we do not consider it as a QueryString.
- $p(label) = C_l(label)/C_n$

The estimated probability for the node is

$$P(nodeVar) = \Pi_{constraint \in constraints(nodeVar)} p(constraint)$$

The estimated Node-Cardinality for nodeVar is $P(nodeVar) * C_n$.

**Edge Cardinality:**

For a pattern that looks like : $(a : X : Y) - [: T1| : T2] - > (b : W : Z)$. Here, a and b are variable nodes each with two labels. The upper bound on the cardinality: C is calculated as follows.

- Estimate from a, $C_a$:

$$C_a = min(X - [: T1] - > (), Y - [: T1] - > ()) + min(X - [: T2] - > (), Y - [: T2] - > ())$$

$$= min(C_{lr\_out}(X, T1), C_{lr\_out}(Y, T1)) + min(C_{lr\_out}(X, T2), C_{lr\_out}(Y, T2))$$

- Estimate from b, $C_b$:

$$C_b = min(() - [: T1] - > W, () - [: T1] - > Z) + min(() - [: T2] - > W, () - [: T2] - > Z)$$

$$= min(C_{lr\_in}(W, T1), C_{lr\_in}(Z, T1)) + min(C_{lr\_in}(W, T2), C_{lr\_in}(Z, T2))$$

- Estimate for $C$:

$$C = min(C_a, C_b)$$

If we already have the candidate list for one of the nodes. Then the cardinality function changes only to consider the candidate nodes. Let us assume we have a candidate set for nodeVar 'a' . Then the estimate of $C_a$ changes as:

$$C_a = \Sigma_{n \in candidatesOf(a)}(C_{nr\_out}(n, T1) + C_{nr\_out}(n, T2))$$

*Variable Length Connections:* Estimation of variable length edge pattern (Assuming we do not have the candidate nodes for any of the variables) looks like this:
For Pattern $(a : X) - [: T1 * 1..l] - > (b : Y)$

- If $l = 1$, it is a single edge
- If $l = 2$, split it
  - We calculate the cardinality for the first edge, $L : (a : X) - [: T1] - > ()$
  - We calculate the cardinality for the last edge, $R : () - [: T1] - > (b : Y)$
  - The value of cardinality, C is given by $C = |L|.|R|$
- If $l > 2$, split it to 3 parts : $L, M, R$
  - $M$ accounts for all the edges between the first and the last edge i.e. $M : () - [: T * 1..(l-2)] - > ()$
  - Cardinality of $M : (l-2)^{deg(:T)} = (l-2)^{C_r(T)}$
  - The value of cardinality, C is given by $C = |L|.|M|.|R|$

For Pattern $(a : X) - [: T1 * i..j] - > (b : Y)$, we calculate the cardinality for $I : (a : X) - [: T1 * i..I] - > (b : Y)$ and $J : (a : X) - [: T1 * i..J] - > (b : Y)$ and calculate $C$ as, $C = |J| - |I|$

## DB QUERY PLAN

We implement a greedy cardinality minimization technique that focuses on pruning the result set at each step. We do this we the help of the cardinality estimation process that has been discussed above.

### Algorithm/Pseudo code:

(1) connectionsToConsider = {} //Global variable
(2) nodesToConsider = {All nodeVars} //Global variable
(3) queriedNodes = {} //Global variable
(4) queriedConnections = {} //Global variable
(5) While(nodesToConsider not empty())
  (a) minNodeVar = extractNodeWithMinCard(nodesToConsider)
  (b) processNodeVar(minNodeVar)

*processNodeVar(nodeVar):*

(1) Query(nodeVar) // Populate the candidate list of nodeVar
(2) queriedNodes.add(nodeVar)
(3) nodesToConsider.remove(nodeVar)
(4) connectionsToConsider.add(connectionsWith(nodeVar))
(5) minConnVar = extractConnWithMinCard(connectionsToConsider)

(6) processConnVar(minConnVar, neighbor(nodeVar, minConnVar))

*processConnVar(connVar, otherNodeVar):*

(1) Query(connVar) // Populate the candidate list of connVar & otherNodeVar
(2) queriedConnections.add(connVar)
(3) connectionsToConsider.remove(connVar)
(4) processNodeVar(otherNodeVar)

Note that, Whenever we query a node/conn we update the candidate list of the node/conn and the estimated cardinality of its variable edges/node. If on querying we find no results then we quit and declare no results.

**Example**

Query : *MATCH (a)-[e1]->(b:PERSON)-[e2]->(c)<-[e3]-(d:CLUB)-[e4]-(e) WHERE b.name = 'DONALD J. TRUMP' and d.name= 'MAR-A-LAGO CLUB, INC' RETURN a.name, c.name, e.name* DB Query plan:

- The program picks 'b' to query because both 'b' and 'd' are the nodes with the least estimated cardinality as explained before.
- After 'b' has been queried, 'e1' and 'e2' are now in the connectionsToConsider.
- As per our data DONALD J. TRUMP has fewer incoming nodes compared to out going nodes so 'e1' is estimated to have lower cardinality compared to 'e2'. Therefore 'e1' and 'a' are queried next in the database, respectively.
- Since, the 'e2' is still present in connectionsToConsider, that is next connection to be queried.
- The next node that has minimum estimated cardinality is 'd', so this node is queried next in the database.
- Now the connections to Consider has both e3 and e4. Since c, already has a set of candidate nodes because of 'e2', the program choses to query 'e3' and 'c' respectively.
- Then the program queries 'e4' and the last node 'e', respectively

## CREATION OF RESULT GRAPHS

After the Database has been queried for all the nodes and connections, we filter out all the node and connection candidates by checking the node candidates and seeing if there are any connection candidates that support the node candidates, and vice-versa.

We construct result graphs before we render the results for the query to ensure the *isomorphic* property of Cypher. We do this by using a DFS-like approach where we assign each candidate element for the respective component node/connection and find all the permutations where we are able to create a complete graph satisfying all the given constraints.

**Example:**

Consider the following query : *match (a:PERSON)-[]-(b:PERSON)-[]-(c:PERSON) where b.name = 'AJIT PAI' return a.name,c.name limit 10* Candidates for both a and c are : "SAM BROWNBACK", "DONALD J. TRUMP", "JEFF SESSIONS". To filter out results where a=b, we construct the graph using the DFS-like approach and make sure we don't assign the same values to multiple variables thus maintaining the Isomorphic property.

**Result correctness and completeness:**

Note that the approach we are taking starts with bigger results and prunes the results as we keep refining the candidates for the components of the graph. We are also making sure to have the same set of assumptions and properties that Cypher has (Example: Isomorphic results). Since we are getting the results for each of nodes and candidates separately from the backend and combining the results by pruning the result sets and trying all the permutations of the candidates resulting in the result graphs that hold the same set of properties as expected by Cypher, we are sure that the generated results are correct and complete.

## SPECIAL FUNCTIONS IMPLEMENTED:

(1) labels(nodeVar)
(2) inDegree(nodeVar)
(3) outDegree(nodeVar)
(4) *Path Length* :length(nodeVar1, nodeVar2,...) or length(connectionVar1, connectionVar2, ...)
We can add up the length of the edgesList for each of the connVar and get the pathLength once the graphs constructed.
(5) *Minimum Path Length* : minLength(nodeVar1, nodeVar2,...) or minLength(connectionVar1, connectionVar2, ...)
Get the minimum pathLength for all the results with the same list of nodeVars/connectionVars.

## EXAMPLE QUERIES:

(1) *match (a:ORGANIZATION)-[e:OWN*6..6]->(b:ORGANIZATION) return a.name, b.name*
Results:

```
THE DONALD J. TRUMP REVOCABLE TRUST , TRUMP COMMERCIAL CHICAGO LLC
THE DONALD J. TRUMP REVOCABLE TRUST , TRUMP RUFFIN COMMERCIAL LLC
THE DONALD J. TRUMP REVOCABLE TRUST , TRUMP PAYROLL CHICAGO LLC
THE DONALD J. TRUMP REVOCABLE TRUST , NITTO WORLD CO. LIMITED
THE DONALD J. TRUMP REVOCABLE TRUST , TRUMP COMMERCIAL CHICAGO LLC
THE DONALD J. TRUMP REVOCABLE TRUST , TRUMP RUFFIN COMMERCIAL LLC
THE DONALD J. TRUMP REVOCABLE TRUST , TRUMP PAYROLL CHICAGO LLC
THE DONALD J. TRUMP REVOCABLE TRUST , TRUMP COMMERCIAL CHICAGO LLC
THE DONALD J. TRUMP REVOCABLE TRUST , TRUMP PAYROLL CHICAGO LLC
Total number of results: 9
```

Explanation: Querying for an organizations that is an umbrella organization at very top level (level 6).

(2) *match (a:PERSON)-[e*1..4]-(b:PERSON) where a.name = 'DONALD J. TRUMP' and b.name contains 'PUTIN' return b, length(a,b), e*
Results:

```
{ name=VLADIMIR PUTIN, labels=[PERSON] } , 2 ,  ['Secretary of State' , 'In 2012, Putin awarded Order of Friendship to Tillerson']
Total number of results: 1
```

Explanation: Finding the paths from Trump to Putin as per the data we have.

(3) *match (a:PERSON),(b:PERSON),(c:PERSON),(d:PERSON) where a.name = 'DONALD J. TRUMP' AND b.name = 'DONALD TRUMP JR.' and c.name = 'IVANKA TRUMP' and d.name = 'MELANIA TRUMP' return a.name, inDegree(a),outDegree(a), b.name, inDegree(b),outDegree(b),c.name,inDegree(c),outDegree(c),d.name, inDegree(d),outDegree(d)*

Results:

```
DONALD J. TRUMP , 7 , 750 , DONALD TRUMP JR. , 2 , 35 , IVANKA TRUMP , 8 , 16 , MELANIA TRUMP , 1 , 9
Total number of results: 1
```

Explanation: Checking how busy the nodes corresponding to Trump, Trump Jr., Ivanka and Melania Trump are, by looking at the indegree and outdegrees of these nodes.

(4) *match (a:ORGANIZATION)-[e*1..4]-(b:PERSON) where a.name = 'GOLDMAN SACHS' and b.name= 'DONALD J. TRUMP' Return a.name, b.name, length(e), minLength(e)*

Results:

```
GOLDMAN SACHS , DONALD J. TRUMP , 2 , 2
GOLDMAN SACHS , DONALD J. TRUMP , 4 , 2
GOLDMAN SACHS , DONALD J. TRUMP , 3 , 2
GOLDMAN SACHS , DONALD J. TRUMP , 3 , 2
GOLDMAN SACHS , DONALD J. TRUMP , 2 , 2
GOLDMAN SACHS , DONALD J. TRUMP , 2 , 2
GOLDMAN SACHS , DONALD J. TRUMP , 3 , 2
GOLDMAN SACHS , DONALD J. TRUMP , 4 , 2
GOLDMAN SACHS , DONALD J. TRUMP , 3 , 2
GOLDMAN SACHS , DONALD J. TRUMP , 2 , 2
Total number of results: 10
```

Explanation: How many ways are Goldman Sachs and Donald Trump connected with a maximum path length of 4. We see that the answer is 10 ways with the minimum path length being 2.

## SUMMARY

This program has been developed as part of a course project to demonstrate the understanding of various database and graph data concepts. The software is a very basic one which analyses very simple cypher queries. But, it can be extended to add more functionality and handle more complex data. The project will soon be uploaded to my github account at : https://github.com/amarnathraju