Plan Before You Execute: A Cost-Based Query Optimizer for Attributed Graph Databases

Soumyava Das, Ankur Goyal and Sharma Chakravarthy

IT Laboratory & Department of Computer Science and Engineering
The University of Texas at Arlington, Arlington, TX 76019.
soumyava.das@mavs.uta.edu, ankur.goyal@mavs.uta.edu, sharma@cse.uta.edu

Abstract. Proliferation of NoSQL and graph databases indicates a move towards alternate forms of data representation beyond the traditional relational data model. This raises the question of processing queries efficiently over these representations. Graphs have become one of the preferred ways to represent and store data related to social networks and other domains where relationships and their labels need to be captured explicitly. Currently, for querying graph databases, users have to either learn a new graph query language (e.g. Metaweb Query language or MQL [6]) for posing their queries or use customized searches of specific substructures [13]. Hence, there is a clear need for posing queries using the same representation as that of a graph database, generate and evaluate alternate plans, develop cost metrics for evaluating plans, and prune the search space to converge on a good plan that can be evaluated directly over the graph database.
In this paper, we propose an approach for effective evaluation of queries specified over graph databases. The proposed optimizer generates query plans systematically and evaluates them using appropriate cost metrics gleaned from the graph database. For the time being, a graph mining algorithm has been modified for evaluating a given query plan using constrained expansion. Relevant metadata pertaining to the graph database is collected and used for evaluating a query plan using a branch and bound algorithm. Experiments on different types of queries over two graph databases (Internet Movie Database or IMDB and DBLP) are performed to validate our approach. Experimental results show that the query plan generated by our system results in exploring significantly fewer portions of the graph as compared to any other query plan for the same query.

1 Introduction

Relational database management systems (RDBMSs) are good at managing transactional data. With the proliferation of applications rich in relationships (e.g., social networks) graphs are becoming the preferred choice as the data model for representing/storing data with relationships. Ability to efficiently query this representation using intuitive ways of querying is central for its ubiquitous usage. Given a graph G and a user-defined query Q, we want to retrieve a set of subgraphs of G that are isomorphic to Q. Graph query finds use in a plethora of domains. For example, in a bibliography network such as DBLP [1], users are eager to extract coauthor and paper information across years; and in a movie database like Internet Movie Database (IMDB) [2], movie enthusiasts tend to look for movies or series belonging to particular genres or containing specific actors or directors. Note that queries can contain

various relational conditions on node and edge labels and unbounded variables to be instantiated to appropriate values during query evaluation. Queries can also include logical operators. With growing graph sizes, number of answers to queries typically increase, making a clear case for the need of query optimization for graph databases.

Graph query answering problem has three major challenges: (1) graph query answering involves graph isomorphism which is NP Complete [8]; (2) querying on graphs need users to either learn a specialized query language (like Metaweb Query Language for Freebase [6] and Cypher Query Language [11] for Neo4j [3]) or use customized query patterns; and (3) large graphs suffer from random access problem and hence need to take advantage of indexing techniques. Note that, graph query answering typically starts from a set of matching vertices in the query, expand them in constrained ways until all edges in the query are expanded. This results in the generation of a set of intermediate results. Intuitively, lesser the number of intermediate results, lesser is the query response time. Given a vertex/edge, indexing helps to quickly find it in a graph. However, indexing does not guide the choice of choosing an initial set of nodes or the set of subsequent nodes and edges needed for query expansion. Index-based approaches do not handle unbound variables and relational conditions. Therefore there is a growing need and strong motivation to take advantage of well known database query optimization techniques to address the problem of reducing the number of intermediate results in graph query answering.

Query optimization on a relational database uses metadata extracted from the database [12]. This metadata can be effectively used to estimate the cost of a query plan to help minimize the desired cost (disk I/O and cpu cost) of processing a query in an RDBMS. Similarly, a query plan in a graph database can be thought of as a sequence of vertex/edge expansions to cover the query. The choice of starting point(s) and the sequence of subsequent vertices/edges expanded, affect the intermediate results generated following *that* particular query plan. Hence, given a query plan, the goal is to *estimate* the number of intermediate results generated while answering the query. Note that determining the exact set of intermediate results is possible only when the query is executed. However, we need a technique to quickly estimate the size of these intermediate results while generating a query plan. Graph metadata can play a crucial role in approximating the size of these intermediate results. Label frequency and average node degree are examples of metadata specific to a graph database. It is also not useful to collect a large amount of metadata as it takes space, time to compute, and resources to update. Therefore, one of the challenges is to identify a relatively small set of graph metadata that can help to generate "good" plan for a given query. In this paper we identify the metadata needed and used by a graph query plan generator. We also discuss the use of metadata for computing the cost of a partial query plan. The contributions of the paper are:

- Identifying a set of metadata and algorithms for extracting and storing them in a graph catalog. We show its applicability in handling graph queries containing both logical and relational operators
- Cost formulas to estimate the average number of intermediate substructures generated by a partial plan. This is used to prune the search space of query plans.
- Modification of the SUBDUE mining algorithm [10] to accept a query plan and evaluate that query in a constrained manner without generating all substructures as is typically done in mining

– Experimental validation of the cost estimate and evaluation of plans consisting of relational and logical operators on different real world databases

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 introduces the preliminaries and problem definition. Section 4 discusses the catalog, cost formulas and the query plan generator, and Section 5 shows the related experiments while Section 6 concludes the paper.

2 Related Work

Graph databases have received a lot of attention over the recent years due to proliferation of data with relationships such as bioinformatics, social networks, web and telecommunication networks. Hence it becomes necessary to manage large graphs in relational DBMSs [5]. However, existing database models and query languages lack the native support for (large) graphs. The popularity of graph data calls for newer techniques to tackle graph data and search for user defined patterns in them. Typically, there are two types of graph databases. The transaction graph database consists of a large number of small graphs while a single-graph database is a large single graph. In all of these graph databases, the common problem is to answer a user defined query, which can be formulated as a combination of logical and relational operations on graph databases.

In any graph database, the graph query problem is to find the set of subgraphs isomorphic to the input query graph. The major challenge is to reduce the number of pairwise graph comparisons and a number of graph indexing techniques have been developed for it [9, 7, 15, 14]. A graph database can also be comprehended as a forest of disconnected components, so solving graph querying for a single graph would also fit into the graph transaction setting. A straightforward approach is to store the underlying graph as relational tables and make use of SQL queries to address graph querying. However, SQL queries are not the best for graph querying as evaluating queries leads to a large number of self-joins. Even though database specific optimizations, like query plan generator tries controlling join order, the number of intermediate results can grow excessively large, especially when the graphs are large. Hence there is a clear motivation to query a graph using graph traversal techniques and devise optimizing techniques for querying in graph databases.

Indexing on a single graph has received some attention in recent times. GraphGrep [9] proposed a path length based indexing scheme. In TALE [14], the authors made use of a neighborhood based indexing technique to match nodes. GRAY [15] used a label propagation based indexing scheme. The common aspect among all these indexing schemes is given a node they quickly find matches of that node to give a head start into query answering. However, none of them uses an educated approach to decide the choice of a start node and the subsequent edges. We demonstrate that metadata can be more useful than indexing for queries with logical and relational operators.

3 Preliminaries and Problem Definition

This section presents the key concepts, notations and terminologies used in this paper.

3.1 Attributed Graph

A graph is attributed where each vertex belongs to a category/type. Formally, an attributed graph is defined as $G = (V, E)$ with V vertices, E edges. The vertices are again categorized into two types: instance vertices (I) and type attributes (T) where $V = I \bigcup T$. There exists a mapping ϕ: $I \rightarrow T$ such that, for each v in I, $\phi(v)$ is the category or type node connected with that v. Each instance vertex is connected to only one type node and to multiple instance vertices. Optionally, edges can have edge labels. All edges going out of a type node has the same label. This is typical of many real-world graph databases with multiple labels in some cases and absence of type nodes in some cases.

Figure 1 shows an example of an attributed graph. The colored circles (Person, movie, and year) are the three type nodes in the graph. Type nodes are connected to instances which are represented using white circles. Note that type nodes are connected to instance nodes containing numeric types (e.g. year) or non numeric types (e.g. Movie). The graph is typically represented as a set of vertices with vertex labels followed by a list of edges between vertices taking $O(V + E)$ space. In case of a directed graph, the edges showcase the source destination relationship. Any other representation that captures this information can also be used.

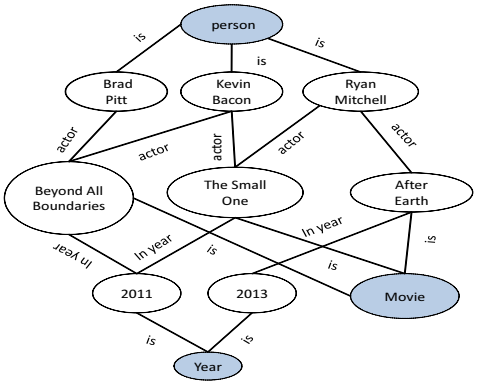


Fig. 1. Example attributed graph from IMDB

3.2 Graph Query

A graph query on an attributed graph is defined as $Q = (V_q, E_q)$ where V_q is the set of vertices, E_q is the set of edges and $V_q = I_q + T_q$ where T_q is the set of type nodes associated with the I_q instance vertices. The user can optionally leave vertex labels unbound (or unspecified) in the query. Following the graph representation, queries are also represented as a set of V_q vertices followed by a set of E_q edges. In graph databases some interesting queries that can be specified are: (i) Queries with one or more type nodes (ii) Queries where instance nodes can have multiple relational operators (*e.g. year* $>$ 1980, *actor* $\neq$ *Brad Pitt etc.*) (iii) Queries with unbounded vertex and edge labels (iv) Queries with logical AND/OR operators on instance node values.

4 Methodology and Approach

For a query on a relational database, the query optimizer generates a "good" plan using the database catalog and the cost formulas for estimating the execution cost of *that* query. We model our system along similar lines. However, the cost in the case of graph databases need to be properly identified. Figure 2 discusses the architecture of our query optimizer. We extract metadata from the graph and store it in a graph catalog. For any given query, the *partial plan generator* generates alternative plans and uses the

cost estimator to estimate the cost of evaluating that plan. The *pruner* helps control the search space of query plans. Below, we explain in detail our three major contributions: the graph catalog, the partial query plan generator and the cost formulas used by the cost estimator.

4.1 Graph Catalog

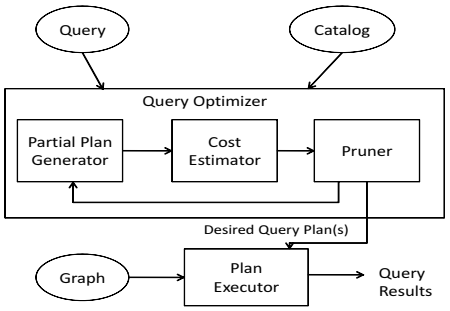


Fig. 2. Graph Database Query Optimizer (GDQP)

As mentioned earlier, in relational databases, metadata information is used to estimate the cost of a query plan. Graph databases are analogous to relational databases in some ways. For example, in an attributed graph database, the type nodes are similar to the attribute names in an RDBMS. Similarly, the number of instances connected to type node is analogous to the attribute cardinality. The connectivity between instance nodes of different types, with different edge labels, estimates the connection cardinality which is similar to the join cardinality in a database. Depending on how a graph is used (whether materialized in memory or not), the cost that needs to be taken into account varies. In this paper, since we are using Subdue which materializes the graph, we do not consider the I/O cost but only the number of intermediate substructures generated. The cost formulas may change under a different assumption. We here discuss the relevant graph statistics which can be gleaned from the graph database for catalog creation. Table 1 introduces the notations that we use throughout the paper.

Table 1. Symbols

Symbol	Description
$I(t)$	Number of instances connected with type t
n_i^t	i^{th} instance of type t where $1 \leq i \leq I(t)$
$edge(n_i^{t1}, n_j^{t2}, el)$	edge between n_i^{t1} and n_j^{t2} with edge label el
$Max(t)$	Maximum value across all instances of type t (if numeric)
$Min(t)$	Minimum value across all instances of type t (if numeric)

I **Type Cardinality (TC):** Type cardinality is defined as the number of instances of a particular type node. Intuitively, if query answering starts from a type node, the type cardinality helps estimate the number of substructures generated after exploring the type node. For type node t, the type cardinality is $I(t)$. Given the type node, degree of that type node is its type cardinality.

II **Average Instance Cardinality (AIC):** Once the query answering reaches an instance node, the number of substructures generated in the next round depends on the degree of the instance node. Since there are multiple instances of a particular type node, with different degrees, we introduce average instance cardinality to

estimate the average number of neighbors to be searched from an instance of type t. For a given instance node of a specific type, the average instance cardinality is computed by taking the average of degrees of instances of that type. Average instance cardinality of an instance node with type t is

$$AIC(instance\ of\ type\ t) = \frac{1}{I(t)} \times \sum_{i=1}^{i=I(t)} degree(n_i^t) \tag{1}$$

III **Average Connection Cardinality (ACC):** Query answering explores the neighborhood of an instance but only chooses a few connected instances, based on the query connectivity (edge label and connected instance node) for expansion. Since multiple instances of type $t1$ can be connected with multiple instances of type $t2$ by the same edge label el, we again derive the average connectivity information for our catalog. Intuitively, this value gives us the expected number of expansions from an instance node to another with a particular edge label. The average connection cardinality starting from an instance node of type $t1$ and reaching another instance of type $t2$ with edge label el is defined as

$$ACC(t1, t2, el) = \frac{1}{I(t1)} \times \sum_{i=1,j=1}^{i=I(t1),j=I(t2)} edge(n_i^{t1}, n_j^{t2}, el) \tag{2}$$

IV **Min and max values of type nodes:** Note that some special type nodes are only connected to numeric instance nodes. This gives user the opportunity to ask range based queries using these attributes. Hence for selectivity, we keep the minimum and maximum numeric value associated with such special type nodes. This intuitively hints at the number of substructures to be generated following user-defined selectivity criteria. For a type node t with numeric or categorical attributes the selectivity is defined as

$$Selectivity = \begin{cases} \frac{1}{I(t)} & \text{node=value} \\ 1 - \frac{1}{I(t)} & \text{node}\neq\text{value} \\ \frac{Max(t) - value}{Max(t) - Min(t)} & \text{node>value} \\ \frac{value - Min(t)}{Max(t) - Min(t)} & \text{node<value} \end{cases} \tag{3}$$

The three major operations to answer a graph query are: (1) choosing a starting node (type or instance); (2) expanding the neighborhood; and (3) retaining substructures using query conditions. Type cardinality and instance cardinality help choose start nodes and estimate neighborhood expansion cost. Average connection cardinality, analogous to joins helps determine the number of expansions. Min and max values estimate number of intermediate substructures based on query conditions on numeric or categorical attributes. Table 2 shows a graph catalog created from the example graph in Figure 1. **AIC** is depicted as an instance followed by "*" while **ACC** is between two instances with defined labels. Fields starting with a type node and ending with an instance node indicate **TC**.

Catalog generation for the system is outlined in Algorithm 1. We make a single pass over the input graph data file using very little memory unlike other algorithms that require graph construction in main memory and its traversal. The catalog is created by reading one input graph line at a time and maintaining a few counters.

Table 2. Graph Catalog extracted for Graph in Figure 1

Node and its type	Node and its type	Edge Label if used	Avg Cardinality
person	person instance	is	3
person instance	person	is	1
person instance	movie instance	actor	5/3
person instance	*	*	8/3
movie	movie instance	is	3
movie instance	movie	is	1
movie instance	person instance	actor	5/3
movie instance	year instance	in year	1
movie instance	*	*	11/3
year	year instance	is	2
year instance	movie instance	in year	3/2
year instance	year	is	1
year instance	*	*	5/2
Node	Min	Max	#Unique values
year instance	2011	2013	2

Processing of vertices provide information on the instance and type nodes. Processing of the edges is used to populate the type cardinality, instance cardinality and connection cardinality. If the instance node has a numeric or categorical attribute, the min and max value associated with its type is computed accordingly. The number of lines in the graph file is equal to the number of vertices and edges hence catalog creation requires $O(|V| + |E|)$ time. Hence the catalog generation time scales linearly with graph size. The number of counters depends on the unique type nodes and unique edge labels present in the graph, which are orders of magnitude lesser than $|V|$ and $|E|$. Since the underlying graph does not change, the catalog incurs an one time generation cost. Once created, the catalog can be used to answer any number of queries. If new vertices and edges are added, the catalog need not be recreated but incrementally updated.

4.2 Cost Formulas

We propose cost formulas for estimating the number of intermediate results generated during query execution as the cost of that plan. Typically, answering a query requires expanding a node to connected nodes in the graph matching edge and node label conditions from the query. Our total estimated cost of a plan is a cumulative cost of partial query plans generated for evaluating a query. A partial query match grows by expanding matched node(s) incurring an expansion cost. Out of these expansions, only the ones matching the query conditions are retained, hence controlling the number of substructures. We introduce two parameters to keep track of each query plan in any iteration i: $(i) cost_i$ and $(ii) currSub_i$. The $currSub_i$ parameter keeps track of estimated number of substructures generated until iteration i. The $cost_i$ parameter

Algorithm 1 *Graph Catalog Generator*

```
1: function CatalogGenerator(V,E)
2:     vertexTypeMap = null
3:     for each vertex v in V do
4:         if v is instance
5:             vertexTypeMap[v]=instance
6:         else
7:             vertexTypeMap[v]=type
8:     end for
9:     for each edge e in E do
10:        source = vertexTypeMap[e.source]
11:        dest = vertexTypeMap[e.dest]
12:        source==type and dest==instance
13:            update type cardinality
14:            update min and max value if e.dest has numeric type attribute
15:        if source==instance and dest==instance
16:            update average instance cardinality
17:            update average connection cardinality
18:    end for
19: end function
```

estimates the cumulative neighborhood expansion cost till the i^{th} iteration. A query must have at least one node. The node can be a type node or an instance node of a known type with or without relational operators. Given a start node, the initialization of cost and currSub in iteration 1 is computed as shown below:

$$cost_1 = I(t) \times Selectivity\ (from\ equation\ 3)\ ; \quad currSub_1 = cost_1 \tag{4}$$

For subsequent iterations (i+1 where ($i \geq 1$)), the current number of substructures is updated based on the query expansion conditions on edge labels and the relational operator on the connecting instance node. This involves both connection cardinality (for edge label) and selectivity (for connecting instance node).

$$currSub_{i+1} = currSub_i \times ACC \times Selectivity \tag{5}$$

In each iteration, partially completed substructures incur a neighborhood expansion cost for satisfying the query conditions. Note that in any iteration, we are adding only one edge on a particular node (type or instance) as the underlying system is implemented that way. Hence $cost_{i+1}$ captures the cumulative cost of neighborhood expansion for the $currSub_i$ matched patterns and the previous cost ($cost_i$). Mathematically,

$$cost_{i+1} = cost_i + currSub_i \times node\ cardinality \tag{6}$$

$$where\ node\ cardinality = \begin{cases} TC & if\ expanding\ on\ type\ node \\ AIC & otherwise \end{cases}$$

4.3 Query Plan Generator

The query plan generator takes a graph query and the metadata generated for that graph as input and generates the best cost query plan using a branch and bound

algorithm. The cost formulas are used as a heuristic to guide the branch and bound algorithm to limit the search space. The generator uses each node in the query as a start node for the generation of alternate query plans. From each start node partial query plans are generated by adding an edge from the query and computing cost of partial plans using the formulas discussed earlier. At every iteration, top-k plans (that correspond to the lowest costs) are considered as candidates for the next iteration. When a complete plan is generated (number of iterations equal to the number of edges in the query), all plans that have a cost higher than the completed plan are pruned. Note that partial plans with a cost lesser than the cost of the current completed plan are still expanded to guarantee an optimal plan. The plan generator emits the optimal completed plan. Note that, in case of relational queries (like OR) the emitted query plan correspond to multiple plans and the query result is an union of results of those plans. Algorithm 2 outlines the algorithm used by our generator. The user can generate top-k plans by explicitly specifying a k value. Obviously, $k = 1$ emits the best plan and k with a large value generates all plans along with their estimated cost information.

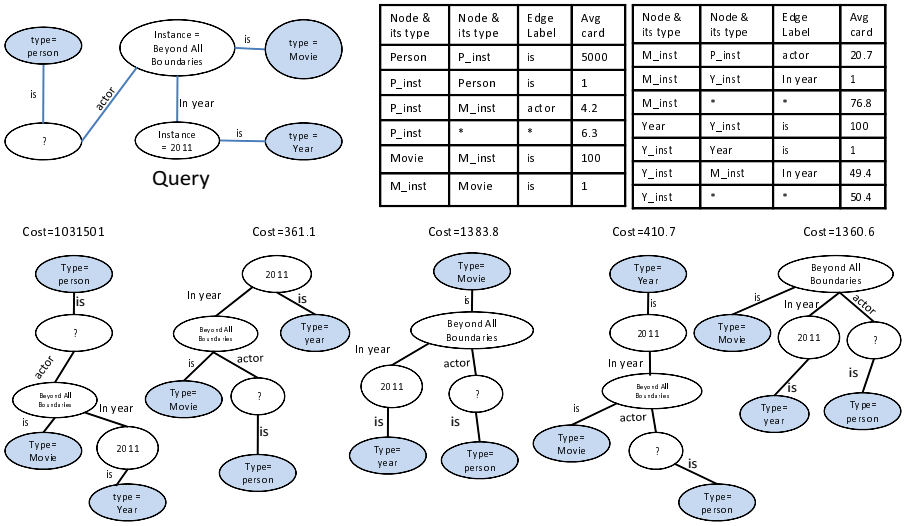
Algorithm 2 *Query Plan Generator*

```
1: Input: Query graph G_q, Catalog C, user defined k
2: Output: k alternative plans with their estimated cost
3: planList = NULL
4: for each node q in G_q do
5:     add q as start point of plan
6:     initialize q.cost using equation 4
7:     initialize q.currSub
8: end for
9: while number of plans < k do
10:    for each plan p in k lowest cost plans do
11:        expand plan by adding an edge
12:        update p.cost using equation 6
13:        update p.currSub using equation 5
14:    end for
15:    update k lowest cost plans
16:    update planList on plan completion
17: end while
18: emit k lowest cost plans
```

Figure 3 shows how above-described plan generator works on a query on the IMDB graph, along with the catalog used by the plan generator. We show all plans and their final costs to demonstrate the cost differences between the best and worst plans. For this example, it is clear that there is orders of magnitude difference between the costs. Once a good (or best) query plan is generated, any system can be used to execute the query plan on the graph. A good query plan would require the least execution time in any graph querying system assuming node-wise expansion of the plan.

We have modified SUBDUE [4, 10], a popular graph mining system into a querying system, Query Processor-Subdue (QP-Subdue) by making the following changes. Instead of starting from all nodes (as in the case of mining), QP-Subdue starts from

node(s) specified by the plan. We replace the unconstrained expansion strategy in SUBDUE by a constrained expansion strategy which supports checks of relational query conditions on node labels and edge labels. Moreover, support for logical operations (AND/OR) on node values is also added to QP-Subdue. We believe that any other querying system will show the same trend as QP-Subdue. However, QP-Subdue carries forward the advantages and disadvantages of SUBDUE. Typically SUBDUE expands one edge at a time and hence requires multiple traversals of the adjacency list of the same vertex. This can be avoided by performing simultaneous expansions on a node. The cost formulas need to be adjusted to reflect the approach used.



Node & its type	Node & its type	Edge Label	Avg card
Person	P_inst	is	5000
P_inst	Person	is	1
P_inst	M_inst	actor	4.2
P_inst	*	*	6.3
Movie	M_inst	is	100
M_inst	Movie	is	1

Node & its type	Node & its type	Edge Label	Avg card
M_inst	P_inst	actor	20.7
M_inst	Y_inst	In year	1
M_inst	*	*	76.8
Year	Y_inst	is	100
Y_inst	Year	is	1
Y_inst	M_inst	In year	49.4
Y_inst	*	*	50.4

Fig. 3. Query plan generation steps for an example query

Note that, in Figure 3 the plans with lower costs have always expanded to known nodes before unknown nodes thereby achieving better pruning during the expansion phase. This is analogous to query plans in a relational model where selections and projections are pushed down the query plan tree. Intuitively, expanding to smaller number of nodes (similar to lower intermediate result cardinality in an RDBMSs) achieves lesser query computation cost. We believe that the actual running time of these plans on the graph will have a positive correlation with the estimated query cost. We shall verify our conjecture in Section 5 on a diverse set of queries to show the effectiveness of our approach.

5 Experimental Analysis

In this section, we present the results of our experimental analysis performed on various queries over different databases. The experimental results reinforce our premise that generating a "good" plan before evaluating a query is beneficial and the execution time of the plan is directly proportional to the number of intermediate substructures

generated. The consistent performance of the plan generator across different types of queries and databases establishes the validity of our proposed approach of cost-based plan generation for graph querying.

All experiments have been carried out on Dual Core AMD Opteron 2 GHz processor machine with 4 GB memory. To evaluate the performance of the plan generator, we used IMDB [2] and DBLP [1] data sets. DBLP data set contains the information of publications along with the information of their authors, conferences, and years. IMDB graph database contains the information of movies, actors, genres, year, company, etc. Since the focus is on good plan generation, we carefully extracted a section from both the graphs for query answering with at least one known query pattern. To see the performance of the system on graphs of various sizes, we extracted small graphs (12,000 vertices and 30,000 edges) to big graphs(350,000 vertices and 1100,000 edges.) This gives us better control in doing targeted queries with the assurance that they would be discovered in the graph.

For the above mentioned graph databases, we used queries having different characteristics such as queries with a comparison operator ($<, \leq, >, \geq, \neq, =$), queries with a combination of multiple comparison operators, queries with logical operator (OR, AND) and queries with a combination of logical and comparison operators. In an AND query all query conditions need to be satisfied, while an OR query internally translates into an union of individual queries. Note that, our system is parameterized to generate minimum, maximum, or all cost plans. Note the our cost metrics is based on the number of intermediate substructure generated, but can be tweaked to match the expansion used in the algorithm for query evaluation. In order to see the effectiveness of our cost formulas among various plans, we picked minimum(**Min**), median(**Med**) and maximum(**Max**) cost plans for experimental analysis. We do not include the catalog creation cost and graph loading time (both being one time operations) as part of the query response time across all our experiments.

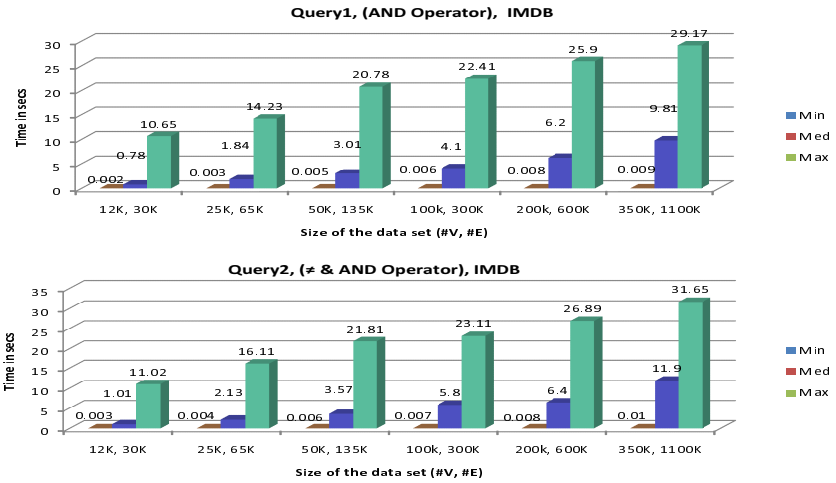


Fig. 4. Query response times for queries 1 and 2 with growing graph sizes

5.1 Performance of Plans

We used the following queries to verify our query plan.

Query 1:"*Find tv-series and its company name, where Kelsey, Wagner has worked as an animator and genre of the tv-series is animation and comedy*" is an example query to IMDB graph database containing AND logical operator. This query inspects how our system performs on queries that needs multiple satisfiability. Experiments in Figure 4 shows us that our best plan performs significantly better than others. The max cost plan typically started exploring unknown nodes first, thereby generating a much larger set of intermediate results which increased the runtime.

Query 2: "*Find tv-series and its company by Soler, Rebecca where the genres are drama and family and the year is not equal to 1996*" is a query to IMDB graph database which contains a combination of both comparison ($\neq$) and logical operator (AND). This query verifies how our system handles combination of elementary query types. Results in Figure 4 show the effectiveness of the plan found by our system. The inequality operation (because of selectivity) adds a little bit to the response time as compared to the equality condition.

Query 3: "*Find tv-series and its company where Kelsey, Wagner has worked as an animator OR Soler, Rebecca has worked as an actress*" is a query to IMDB graph database having an OR operator. This query helps understand the difference in runtime between queries with logical operators. Note that the OR query in Figure 5 takes considerably more time than the AND query in Figure 4. Since OR queries are divided into multiple sub plans based on the OR condition, execution of an OR query is internally translated as union of multiple plans thereby needing more time. We still see that our system generated query plan is the best among all query plans in terms of runtime.

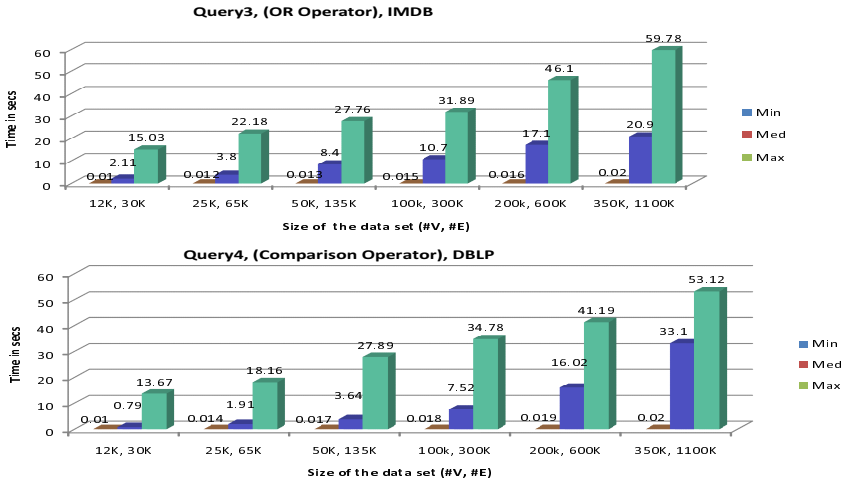


Fig. 5. Query response times for queries 3 and 4 with growing graph sizes

Query 4: "*Find papers published by the author Eric Hanson prior to the year 2009*" is an example of query which contains a range (<) operator on the DBLP graph database. This query is important to understand how our system performs for range queries. Once again Figure 5 shows that minimum cost plan executes in considerably less amount of time compared to other plans.

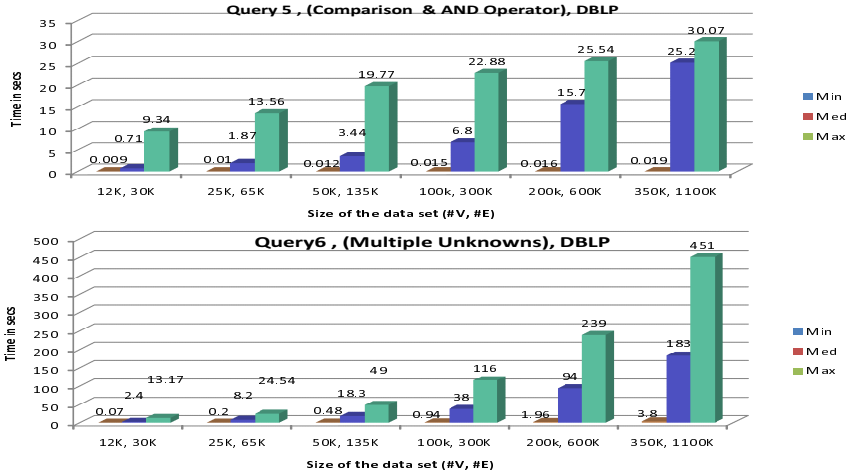


Fig. 6. Query response times for queries 5 and 6 with growing graph sizes

Query 5: "*Find papers where Yuri Breitbart AND Abraham Silberschatz have collaborated together after the year 1980*" shows an example a query with a combination of comparison and logical operators. In Figure 6, the time varies for minimum cost plan from 9 millisecond to 19 millisecond on different data sets, while for the maximum cost plan, it varies approximately from 9 seconds to 30 seconds. The order of magnitude difference is indicative of the need for this approach for graph query processing.

Query 6: "*Find authors with their papers and conference information in year 2005*" is an example of a query which has multiple unknown nodes (authors, papers, conference). This query helps us to validate the quality of our generator in the presence of a high number of unknown values in the query. Figure 6 shows that our method still performs better than the median and maximum cost plans. However, the time taken to answer this query is significantly greater than other queries due to the presence of multiple unknowns in the query. Still the difference is in order of magnitude.

All the queries show that the best cost plan generated by our system always generates minimum query response time. And the median and max plans are orders of magnitude slower than the best plan. This validates our premise that different, but appropriate cost measures are needed for estimating graph query plans. The consistent performance of the plan generator validates the feasibility of the proposed approach for graph databases. Moreover, our plan generator is able to handle different query types.

6 Conclusions and Future Work

We have developed an initial framework that allows us to generate query plans for various types of queries containing one or more comparison and logical operations. Our choice of the number of intermediate substructures generated as a good cost estimator is validated by the time taken for the execution of different plans. We have minimally modified a mining algorithm to evaluate queries. The proposed approach overcomes some of the limitations of the conventional techniques used in graph databases for evaluating a query without generating query plans.

Future work includes evaluation of queries on partitioned graphs to achieve scalability, overcoming some of the limitations of Subdue to expand multiple edges in each iteration which requires re-examination of cost formulas, generalize to arbitrary graphs – with or without type nodes, with multiple node and edge labels, and multiple edges between nodes as well as cycles. An intuitive User interface for specifying queries as well as the underlying graph to facilitate querying will be useful.

References

1. `http://www.informatik.uni-trier.de`.
2. `http://www.imdb.com/`.
3. `http://neo4j.com/`.
4. `http://ailab.wsu.edu/subdue`.
5. Shalini Batra and Charu Tyagi. Comparative analysis of relational and graph databases. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(2):509–512, 2012.
6. Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1247–1250, New York, NY, USA, 2008. ACM.
7. James Cheng, Yiping Ke, Wilfred Ng, and An Lu. Fg-index: towards verification-free query processing on graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 857–872, 2007.
8. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.
9. Rosalba Giugno and Dennis Shasha. Graphgrep: A fast and universal method for querying graphs. In *16th International Conference on Pattern Recognition, ICPR 2002, Quebec, Canada, August 11-15, 2002.*, pages 112–115, 2002.
10. Lawrence B. Holder, Diane J. Cook, and Surnjani Djoko. Substucture Discovery in the SUBDUE System. In *Knowledge Discovery and Data Mining*, pages 169–180, 1994.
11. Florian Holzschuher and René Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 195–204. ACM, 2013.
12. Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Computing surveys (CsUR)*, 16(2):111–152, 1984.
13. Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *World Wide Web Conference Series*, pages 607–614, 2011.
14. Yuanyuan Tian and Jignesh M. Patel. TALE: A tool for approximate large graph matching. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 963–972, 2008.

15. Hanghang Tong, Christos Faloutsos, Brian Gallagher, and Tina Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, California, USA, August 12-15, 2007*, pages 737–746, 2007.