QP-SUBDUE: PROCESSING QUERIES OVER GRAPH DATABASES

by

ANKUR GOYAL

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2015

To my father Vijay and my mother Shashi

ACKNOWLEDGEMENTS

I express my sincere gratitude to my supervising professor, Dr. Sharma Chakravarthy who has been a great motivating factor and a constant source of encouragement throughout my masters' research. Without his guidance and excellent foresight this thesis would have only remained a great idea. I am sincerely thankful to Dr. Ramez Elmasri and Prof. David Levine for giving valuable suggestions and serving on my committee.

I would like to thank the administrative staff, specilcally Ms. Pam McBride and Ms. Sherri Gotcher for their valuable support and services. Special thanks to Ms. Camille Costabile in helping me through the final requirements of my thesis, and the department of computer science, UTA. I am grateful to Irie Bito for maintaining a well-administered research environment.

My heartfelt thanks to my family for continuous support and inspiration. I would like to express my appreciation to Soumyava Das, Jay D. Bodra, Abhishek Santra and other friends in ITLAB. I also greatly appreciate my numerous other friends for their love and support.

November 13, 2015

ABSTRACT

QP-SUBDUE: PROCESSING QUERIES OVER GRAPH DATABASES

ANKUR GOYAL, M.S.

The University of Texas at Arlington, 2015

Supervising Professor: Dr. Sharma Chakravarthy

Graphs have become one of the preferred ways to store structured data for various applications such as social network graphs, complex molecular structure, etc. Proliferation of graph databases has resulted in a growing need for effective querying methods to retrieve desired information. Querying has been widely studied in relational databases where the query optimizer finds a sequence of query execution steps (or plans) for efficient execution of the given query. Until now, most of the work on graph databases has concentrated on mining. For querying graph databases, users have to either learn a graph query language for posing their queries or use provided customized searches of specific substructures. Hence, there is a clear need for posing queries using graphs, consider alternative plans, and select a plan that can be processed efficiently on the graph database.

In this thesis, we propose an approach to generate plans from a query using a cost-based approach that is tailored to the characteristics of the graph database. We collect metadata pertaining to the graph database and use cost estimates to evaluate the cost of execution of each plan. We use a branch and bound algorithm to limit

the state space generated for identifying a good plan. Extensive experiments on different types of queries over two graph databases (IMDB and DBLP) are performed to validate our approach. Subdue a graph mining algorithm has been modified to process a query plan instead of performing mining.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

With the advent of automated data collection tools, massive amounts of data are being generated. Technology has given us inexpensive ways to store this considerable amount of data. Google receives millions of searches/queries per minute; in addition, the volume of email messages being exchanged is ever increasing when other forms such as tweets and others (e.g., Whatsapp) are included. DBLP [1], a computer science bibliography, has around 2.8 million records (conference, workshop papers, and journal articles). Freebase [2], an online collection of structured data, harvested from various sources, has around 47 million topics and 2.7 billion facts. Internet Movie Database (IMDB) [3] contains information about 3.3 million movie titles and 6.6 million people associated with movies, TV-series etc. This clearly shows that the amount and types of data being generated are significantly large and querying them is becoming a useful alternative and a challenge.

Data is collected and stored to find useful information (either by mining or by search or by querying) that is beneficial either for business or individuals. For example, DBLP has the information of authors, journals, and conference papers. If an employer in a research lab wants to hire people who have published papers on a particular topic, s/he may want to query the DBLP data. Another example could be to find authors who have published or co-authored papers in prestigious conferences (e.g., SIGMOD) during a specific period. Moreover, people use Freebase to find information, such as find Vietnamese restaurant in Palo Alto, California [4]. In addition, nowadays social networking has also become very popular which gives a platform to

build social relations among people who share interests, activities, backgrounds, or real-life connections. Currently, many social networking sites are available, such as Facebook, Twitter, LinkedIn, etc. and users may want to query these sites to find relevant and useful information. Most of this information is structured in that relationships (e.g., works for, founder of) are explicit in these data sets unlike traditional applications such as payroll, airline reservation, etc. Despite the rich data sets and their structures, they cannot be queried the way user wants. Queries are limited and decided by the vendors or corporate stake holders. For example, DBLP provides an option to find papers of a particular author, but instead if a user is interested in finding a paper, where author1 has collaborated with author2, but not with author3 in a certain period of time, then these types of queries are not supported. IMDB, which contains the movie database, can be queried to find information about movies, genres, actors, etc. But again, if a user wants a movie in a particular period of time and with a specific genre, company, or actor, these queries are not supported. The goal is to support arbitrary queries over these data sets that have rich representation already captured in the form of a graph. Providing a capability to allow queries over these representations will avoid conversion of this data into a traditional relational representation for querying purposes. However, these queries need to be processed using a different representation of data than relations or tables. As a result, both query processing and optimization techniques need to be developed for this purpose. Our focus is to design a platform for a user, where the users queries should be driven by his needs. Therefore, we need a way to process this data to extract meaningful information.

Relational database management systems (RDBMS) have been widely used across industries. Research and development over several decades have resulted in efficient storage alternatives, optimization of queries, and other tools. One of the

drawbacks is that the application needs to be modeled first using this representation in order to avail the benefits. More importantly, the relationships within the data are lost during the mapping process thereby making it difficult to understand the connections in data and to formulate queries. New applications are emerging whose data representation may benefit from alternative ways of data storage and management. Nowadays, data is generated or evolved at a higher rate. Also, data is typically not structured. Therefore, traditional databases with a need to create schema first may not be the best choice for these applications. A graph is more intuitive and also has a sound theoretical basis. Moreover the graph model can natively support many of these applications thatmay require forcing them into a RDBMS representation. Batra and Tyagi [5] explain that graph databases are able to accept all types of data- structured, unstructured, and semi-structured – more easily than relational databases, which rely on a predefined schema.

Moreover, graph databases are more flexible forthe aforementioned applications and provide the capability to keep data in their native representation where relationships are explicit. Relational models, on the other hand, work best when there are a relatively small and static number of relationships between objects. When the data sets become larger and the query contains results from many tables, then it may require expensive join operations. Conversely, in a graph database, query processing does not have to scan the entire graph to find the nodes that meet the search criteria. It looks only at records that are directly connected to other records. For such applications graph database may perform better than traditional RDBMS. Once the data is modeled we can query this data.

As mentioned earlier, current systems such as DBLP, IMDB, and Freebase support a limited class of queries. DBLP, which has all the information of authors, publications, conferences and publication year, can be accessed through a web interface.

We can find interesting information, such as, find all the papers written by particular author, DBLP returns all the publication papers, journals, books, etc. written by that author. Moreover, it provides various ways to refine the result set such as refine by author, venue and year. However, there are queries which are not supported by DBLP. For example, Find an author who has written a paper with author1 but not with author2 or Find papers written by author1 after 2005 etc. In the case of IMDB, queries such as find movies where person1 and person2 have worked as actors and person3 has worked as director in 1995 and movie genre should not be drama cannot be supported by the existing interface. To support these and other more expressive types of queries, we model this data as a graph, where each individual entity can be represented as a node and the relationship between them can be represented by an edge. After modeling this data into graph structure, we have to find patterns in this data for a query having similar graph form but includes conditions, wild cards etc.

The abundance of graph data in a variety of domains implies that graph querying and graph mining have become important for information retrieval and analysis. Graph mining identifies frequent and interesting graph patterns, classifies new graphs based on knowledge of known graphs, or clusters graph into subclasses. A graph query, on the other hand, takes a graph pattern as an input and retrieves patterns that satisfy the input graph that includes conditions, ranges, and wild card specifications from graph databases. As in any other form of querying, graph querying needs to find all the occurrences of a given substructure. Typically, a query is in the form of a connected sub graph which is relatively very small as compared to the size of the graph database. Therefore, a query is given as an input to find exact matches in graph data. A given query can be processed in many ways to find these exact matches. Given a query graph with many vertices and edges, its processing can theoretically start from any node and. compute answers. There will be several possible ways to compute the

results for the same query depending on the start node. For a given query,number of potential plans depends on the number of possible starting nodes of the query graph. In a query of n nodes, if there are k unknown nodes, there can be n-k different plans (all nodes except unknown nodes can be the starting point). If unknown nodes are also used as starting points then they match every node in the graph which increases the search space significantly. Clearly, if there are multiple alternatives (plans), a cost metric is needed to differentiate with respect to the computation needed for its evaluation. Each plan, when executed generates different number of intermediate substructures. Intuitively, a plan that generates more intermediate substructures is costlier. In order to compute the cost, typically, selectivity, cardinality and other information are used in relational databases. Along the same lines, there are two issues that need to be identified for graph databases. First, what meta information needs to be collected from the graph that can be used to evaluate query plans, and second, how to generate that information efficiently. Once we find a good query plan, based on cost of processing that plan, results can be obtained from the query processor. This requires an algorithm that accepts a query plan and computes the results from the graph database efficiently.

In this thesis, we address general-purpose query processing over a graph database. For that, we develop techniques and approaches for generating alternative plans and for computing their estimated costs. The goal is to develop a cost metric that is applicable to a graph database query evaluation. We also develop a query processor to process a plan generated by our plan generator. We evaluate our approach experimentally by using a couple of real-world graph databases. Our approach will use query specification that is more expressive than what can be queried over earlier work in this area.

The remainder of the thesis is organized as follows. Chapter 2 presents the related work in the areas of relational query processing and optimization. Chapter 3 discusses an overview of graph mining and querying. Chapter 4 discusses graph representations and how to specify a general/arbitrary query. Chapter 5 explains the generation of alternative plans, cost model, and heuristics used for state space generation and pruning. Chapter 6 details the implementation of a graph query optimizer and processing of a query plan using subdue. Chapter 7 shows experimental results to validate our approach. Conclusions and future work are outlined in Chapter 8.

CHAPTER 2

RELATED WORK

Graph querying is useful for retrieving information from emerging graph databases such as Freebase and knowledgebase. Querying a database, although different from mining, is useful for retrieving desired information which satisfies the conditions we already know. For example, the query list all founders of a company who attended Harvard or Yale is more specific than mining where one is looking for patterns that may be of interest because it occurs a lot of times in the database. For querying a graph database we need a query which is in the form of graph to obtain the exact matches. As mentioned earlier, a query can be evaluated in multiple ways by generating alternate plans and a good plan (i.e., cheaper to evaluate) is chosen among alternative plans generated, Therefore, the problem of graph querying boils down to explore alternate viable plans, estimate a cost for each of these plans, have an approach to avoid exhaustive generation of search space, and finally choose a plan for actual evaluation. Query processing and optimization processing have been well researched and a number of techniques have been proposed. This chapter briefly presents an overview of some of the widely used approaches for query optimization and query processing. Various techniques proposed for query optimization and query processing include query optimization in relational databases [6], Graph-grep [7], G-index [8], G-ray [9], and Subdue [10]. The following sections describe some of the query optimization and query processing approaches. A discussion of some of these systems is presented when we consider the related work in the area of graph mining.

2.1 Query Optimization in RDBMS

Query optimization is one of the accomplishments of relational database research. A query optimizer attempts to determine a good execution plan for a given query by considering a large number of possible query plans. There is a trade-off between the amount of time spent on optimization and the cost savings provided by the chosen plan. Therefore, the optimizer uses techniques to keep the search space generated manageable and still obtain a good plan for evaluation. A given query can be evaluated in many possible ways and the difference between the best and the worst plan, in terms of cost of its computation, can be significantly different. Relational databases use different parameters to evaluate the cost. According to Jakre and Koach [11], query optimization cost model typically is an objective function consisting of the following costs;

1. Secondary storage access cost: The cost of loading data pages from secondary storage into main memory. This is influenced by the number of pages to be retrieved, clustering of data on physical pages, and size of available buffer space.
2. Storage cost: Each operator is associated with a CPU cost and the cost of using the CPU during that operator evaluation is called computation cost.

Query optimization algorithm is strongly influenced by these cost components. For each operator, there are algorithms available in RDBMS and several factors (i.e. the size of each table, availability of an index, etc.) affect the cost of an algorithm. There are some common techniques used in developing an evaluation algorithm and one of them is the concept of access path, which is a way of retrieving data from a table. An access path consists of a file scan or an index plus matching selection condition. Each relational operator takes one or more tables as an input and access methods are used to retrieve tuples.

Consider a selection operation that is a conjunction of multiple conditions of the form attr op value where op is one of the comparison operators <, <=, =, ! =, >= or >. These selections are said to be in conjunctive normal form. Each condition is called a conjunct. A hash index matches a CNF selection if there is an equality condition. A tree index matches CNF selection for non equality conditions. An index can match a subset of the conjuncts in a selection condition. The most selective access path is the one that retrieves the fewest pages. For each conjunct there are some tuples from the table that satisfy the condition; this fraction of tuples is called a reduction factor. So based on reduction factor, optimizer estimates the number of tuples.

A query optimizer uses different heuristics to reduce the cost of an execution plan. One of the main heuristics is to apply select and project operations before applying the join because a join is a relatively expensive operation and a good heuristic used is to reduce the sizes of intermediate results. In other words, the size of the file resulting from a join operation is usually a multiplicative function of the size of the input files. Similarly, the size of intermediate tables is reduced by carrying forward only the required attributes. The select and project operations reduce the size of an intermediate file. The query optimizer generally applies these operations before join operation and joins are implemented based on the availability of index on relations.

To estimate the costs of various execution strategies, some meta information is needed. This information is stored in the DBMS catalog which is used by a query optimizer. A catalog typically contains the size (Number of pages) of each file, number of records in each file, record size, and number of blocks. The records may be unordered, ordered by an attribute, or ordered without a primary or clustering index; all this information is kept in the catalog. Another important parameter is the number of distinct values of an attribute and its selectivity, which is the fraction of records

satisfying equality condition on the attribute. This gives an estimation of selection cardinality of an attribute, which is the average number of records that will satisfy an equality selection condition on that attribute. Different query plans generated are evaluated using this information and the plan with the least cost is selected, and finally that plan is executed to evaluate the query.

In our work, we have adopted a similar approach and use meta information associated with a graph database. We analyze the graph once to glean the meta information such as different node labels, their connection information, edge label information, etc, which will be discussed in detail in the following chapters.

According to Ramkrishnan and Gehrke [12], to identify equivalent expressions for a given query, a relational query optimizer uses relational algebra equivalences. For all such expressions of the query, all available implementation techniques are considered, thereby generating several alternative query evaluation plans. The query optimizer estimates the cost of each plan and chooses an optimal plan. Two plans over the same set of input tables are said to be equivalent if the results produced by them are the same.

A typical SQL query consists of *select*, *from*, *where*, *group by* and *order by* clauses. The fields in each select clause are projected from the Cartesian product of tables in the from clause. The use of equivalences enables us to convert this initial representation into equivalent expressions. Particularly, selections and cross-products can be combined into joins and joins can be reordered. Consider a query having multiple joins. Three relational algebra operator trees that are equivalent to query would be left-deep tree, right-deep tree, and bushy tree. Optimizers typically use a dynamic programming approach to efficiently search the class of all left-deep plans. If an optimizer considers all possible plans, the number of plans would be exponential which differs in join order. Of course, the decision rules out many alternative plans

that may cost less than the best plan using a left-deep tree. The optimizer only considers the left deep plans because, as the number of joins increases, the number of alternative plans also increases rapidly and it becomes necessary to prune the search space. Moreover, using left deep trees, fully pipelined plans can be generated, which means intermediate results are not materialized.

After considering a set of plans, the query optimizer estimates the cost of each plan. The cost of a plan is typically the sum of costs of all operators in the query plan. The cost of individual relational operators in the plan is calculated using the information from system catalog. I/O costs play major role in determining the total cost. The I/O cost of a plan consists of a cost of reading input tables from the disk, cost of writing intermediate tables if necessary (e.g., for a sort-merge join), and sorting the final results if query specifies an output order. The cost of a fully pipelined plan is dominated by the cost of reading input tables which depends on the access paths used to read input tables. For plans that are not fully pipelined, the cost of materializing temporary tables needs to be taken into account. The number of tuples in the result of a selection is estimated by multiplying the input size by the reduction factor for the selection condition while the number of tuples in the result of projection is the same as input. In the case of a projection query optimizer carries forward only required attributes thus reducing the size of intermediate results. The result size for a join can be estimated by multiplying the maximum result size which is the product of the input table sizes, by the reduction factor of the join condition.

In the case of graph data, we use a similar approach. Rather than expanding all the plans, we only consider subset of plan space to be expanded. Cost model for our approach estimates the number of intermediate substructures that will be generated. We evaluate the cost of each plan by its number of intermediate substructures which will be discussed further in upcoming chapters.

2.2 Query processing on Graph Data

Graph is a powerful tool for representing and understanding objects and their relationships in various application domains. Due to increasing popularity of graph databases, graph query processing has been widely researched. However, as mentioned earlier, existing querying systems have limitations and do not support queries containing operators ($<$, $>$, $=$, $!=$). Moreover, most of the querying systems do not follow a plan to evaluate queries. Existing research has been conducted mainly on two types of graph databases. The first involves, one large graph (such as social networking graph, web graph etc.) and the second consists of multiple small graphs (such as chemical compounds, blog graphs etc.). In this section we discuss about various techniques for graph querying such as Graph-grep, G-Index, and G-Ray. We also establish the relationship between graph mining and graph querying in the upcoming sections.

Graph querying is the process of finding similar query patterns in the graph data. A lot of research has been done in this field earlier. Shasha, Wang and Guino [7] propose an algorithm called Graph-grep which is a variable path index approach. The first step in this algorithm is to construct an index which essentially is done by finding all possible paths up to length l, from all nodes. Then, it stores all possible paths in a hash table. The second step is to filter the database; the query graph is parsed to build its fingerprint (hashed set of paths). Database is filtered by comparing the fingerprint of the query with the fingerprint of the database. A graph, for which at least one value in its fingerprint is less than the corresponding value in the fingerprint of the query, is discarded when looking for an exact sub graph match. The last step is finding sub graphs matching the query; after filtering, this algorithm searches for all the matching sub graphs in the remaining graphs. The branches of a depth-first traversal tree of the query are decomposed into sequences of overlapping label-paths,

which are called patterns. Then, it joins all the sub paths on an overlapping node to get the final results. Since all the paths are already hashed so this approach provides fast results. However if the graph size is large, keeping all paths up to length l takes huge space. Since this approach breaks the query and graph into different paths so it loses the structural information which makes it difficult for the chemical compounds that requires structural information to be preserved. This approach cannot handle queries containing comparison operators.

This approach evaluates the query graph from the main graph using indexing techniques. In our approach, instead of indexing we use an exploration approach. A node is fully explored and, with the exception of required nodes every other node is discarded. Unlike graph-grep approach, structural information is also preserved.

Another approach by Yan, Yu and Han [8] suggests indexing frequent structures, called the G-index. In this approach the authors define the substructure to be frequent if its threshold is greater than the minimum support threshold and all frequent substructures are indexed. Each substructure is associated with an id list. Given a graph query q, if q is frequent, the graphs containing q can be retrieved directly since q is indexed. The G-index only indexes structures which are distinct, which means that if the same substructure is generated twice, it indexes only once.

G-index uses Depth First Search (DFS) coding to translate the graph into unique edge sequence called canonical label. If two substructures are the same that means, they must share the same canonical label; the G-index holds canonical labels in a prefix tree. Given a query, G-index enumerates all its fragments up to a maximum size and locates them in the index and then it intersects the id lists associated with these fragments, which is the candidate answer set. After getting the candidate answer set, it verifies whether the graphs in answer set really contain the query graph.

However, this approach does not answer infrequent queries because it only indexes the frequent substructures, and if graph is large then index size becomes greater.

In our querying approach, we do not use indexing scheme because in a large graph index size becomes very large and updating indexes incurs an additional cost; instead we use exploration approach where we explore the nodes in a specific order. Therefore, our querying system answers all queries (both frequent and infrequent) unlike G-index. In addition, G-index does not handle queries with comparison operators and does not follow any plan to evaluate the query.

Another approach presented by Gallagher, Faloutsos and Eliasi-Rad [9], called G-Ray, finds both exact and inexact matches. This approach first finds a seed node and then expands the seed node by finding a matching node followed by bridging both nodes by the best possible path. G-Ray proposes a goodness score which is a measure of proximity between two nodes. Based on this goodness score, it ranks the results. In this approach each vertex stores the information of remaining vertices. Therefore, space requirement is significant and it also does not differentiate between two results having the same goodness score. Unlike other approaches, G-ray keeps the attribute information of each node. For example, if there are two nodes California and Steve Jobs, their attribute could be state and person, respectively. In our approach we use the similar approach and keep the attribute information of each node. Our approach, on the other hand, does not deal with inexact results, which means if any node from the query is not found during the exploration, the result set contains null.

In this chapter, we have discussed the overview of related work in the area of query optimization and query processing. With this overview of the related literature in the area of graph query processing, the discussion on graph mining techniques is presented in the next chapter.

CHAPTER 3

OVERVIEW OF GRAPH MINING AND GRAPH QUERYING

3.1 Overview of Graph Mining

Data mining is the process of discovering hidden patterns in large data. The goal of the data mining process is to extract non-intuitive information from a data set and use it for making business decisions. Data, in many applications, have an inherent structure and converting them to non-structural (RDBMS) format will result in loss of information. Graph representation provides a natural format for preserving the inherent structural characteristics. If processing can be done directly on this representation, it will provide better results as the semantics of the applications (in the form of relationships) is preserved during processing. Complex structural relationships can be modeled as graphs if no constraints are assumed (such as cycles, multiple edges, only directional edges, and constraints on vertex and edge labels). Graphs model the data in the form of a vertex (to characterize the data), and edges (that typify extra information). Graph mining is used to mine structural data such as DNA sequences, electrical circuits, chemical compounds, social networks, schemes (such as money laundering and fraud) that have associations and relationships among transactions, etc. A graph representation comes across as a natural choice for representing complex relationships as the data visualization process is relatively simple as compared to a data in traditional RDBMS representation. Data representation in the form of a graph preserves the structural information of the data which may otherwise be lost if it is translated into other representation schemes.

3.2 Overview of Graph Mining System Subdue

Subdue [10], the earliest work on graph mining, uses information-theoretic model for determining the best substructure given a forest of unconstrained graphs. This substructure discovery system was developed by Cook and Holder. The Subdue discovery algorithm discovers repetitive patterns and interesting substructures in graph representations of input data. A substructure is a connected sub graph within the graph representation. In a graph, entities and objects are mapped to the vertices and the relationship between these objects is represented as the edge between the corresponding pair of vertices. An instance of a substructure in an input graph is a set of vertices and edges from the input graph that matches the graphical representation of the substructure.

The input to Subdue is a forest of graphs and the output is a set of substructures that are ranked based on their ability to compress the input graph using the Minimum Description Length (MDL) principle. The compression technique is elaborated in detail in the following sections.

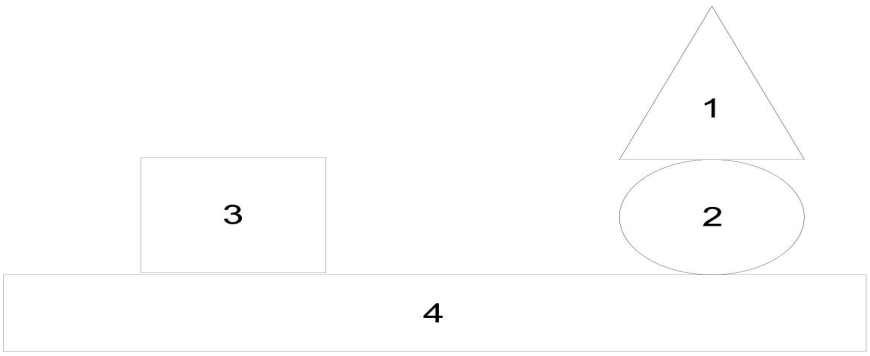

Figure 3.1. High-level view of shapes.

The input is in the form of a table consisting of a list of unique vertices in the graph and edges between them. The output is a list of representative substructures

discovered in the input graph that compress the graph most and each is qualified by its size and occurrence frequency in the input graph. Consider the example in Figure 3.1 . It is a high-level view of shapes resting on a table. The graphical representation of these shapes is shown in Figure 3.2 below.

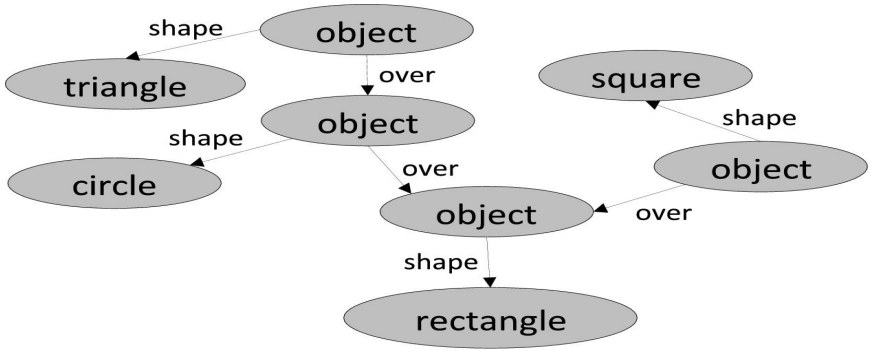

Figure 3.2. Graph representation of shapes example.

The input for Subdue (for this particular example) is as shown in Figure 3.3. This input is in the form of a file consisting of the list of vertices and the edges between the vertices.

```
v 1 object
v 2 object
v 3 object
v 4 object
v 5 triangle
v 6 circle
v 7 square
v 8 rectangle

u 1 2 over
u 2 3 over
u 2 4 over
u 3 4 over
u 1 5 shape
u 2 6 shape
u 3 7 shape
u 4 8 shape
```

Figure 3.3. Subdue Input for shapes example.

Subdue generates the best substructures that compress the input graph the most and lists out the top n substructures. The output given by subdue for the example in Figure 3.2 is displayed in Figure 3.4.

```
Best 3 substructures:

(1) Substructure: value = 0.96959, pos instances = 1, neg instances = 0
 Graph(2v,1e):
  v 1 object
  v 2 object
  u 1 2 over

(2) Substructure: value = 0.953003, pos instances = 1, neg instances = 0
 Graph(2v,1e):
  v 1 object
  v 2 square
  u 1 2 shape

(3) Substructure: value = 0.953003, pos instances = 1, neg instances = 0
 Graph(2v,1e):
  v 1 object
  v 2 rectangle
  u 1 2 shape
```

Figure 3.4. Subdue Output for shapes example.

3.2.1 Parameters for control flow

There are a number of parameters that Subdue provides the user in order to control the flow of the substructure discovery process. The input to Subdue is the file containing the list of vertices and corresponding edges as shown in Figure 3.2. The parameters which drive the discovery process of Subdue are as follows:

1. BEAM: This parameter specifies the number of top substructures. Top BEAM substructures are retained for the expansion in each iteration of the discovery algorithm. The default value of the beam is 4.
2. ITERATIONS: Iterations is used to specify the number of iterations to be made over the input graph. The best substructure from the previous iterations is taken to compress the graph for the next iteration. The default is no compression.
3. LIMIT: Limit specifies the number of different substructures to be considered in each iteration. The default value is (number of vertices + number of edges)/2.
4. NSUBS: This parameter is used to specify the number of substructures to be returned as the result from the total number of substructures that Subdue discovers.
5. OVERLAP: Specifying this parameter to Subdue allows the algorithm to consider overlap in the instances of the substructures. Instances of substructures are said to overlap if they have a common substructure in them.
6. PRUNE: If this parameter is specified, then the child substructures whose value is lesser than their parent substructures are ignored. Since the evaluation heuristics are not monotonic, pruning may cause SUBDUE to miss some good substructures, however, it will improve the running time. The default is no pruning.
7. SIZE: This parameter is used to limit the size of the substructures that are considered. Size refers to the number of vertices in the substructure. A minimum and maximum value is specified that determines the range of the size parameter.
8. THRESHOLD: This is the parameter that provides a similarity measure for the inexact graph match. Threshold specifies how different one instance of a substructure can be from the other instance. The instances match if match-cost(sub, inst) <= size(inst)$*$ threshold. The default value is 0.0, which means

that the graphs should match exactly. Currently, Subdue supports threshold values up to 0.3.

3.2.2 Substructure Discovery in Subdue

The substructure discovery in Subdue is done by using a beam search and progresses in an iterative manner starting with substructures of size 1 and expanding to successively larger substructures. A list consisting of a set of substructures to be expanded is maintained. The input graph is compressed by replacing the instances of these substructures by a single node. The resulting input graph is then used for the next iteration to find other interesting substructures. This process continues until the number of iterations specified by the user is reached or it meets one of the several halting conditions such as the total number of substructures needed provided by the user. The occurrences of substructures that have an exact match are unlikely to occur in most domains. Substructure instances that are not exactly the same but are similar can also be discovered by Subdue. Subdue is capable of discovering both exact and inexact (isomorphic) substructures in the input graph. Subdue employs a branch and bound algorithm that runs in polynomial time for inexact graph match and discovers graphs that differ by a threshold given by the user. This discovery process is used to find repetitive and interesting substructures or patterns. After that it compresses the graph by replacing the instances of these patterns by a single node in order to provide a hierarchical view of the original input graph. Subdue compresses the input graph using the substructures generated. In order to determine which of those substructures compress the graph best, Subdue uses the MDL principle to evaluate the compressed substructure. After compressing the substructure in multiple iterations, the best substructures are output. Subdue system finds interesting and repetitive patterns using graph mining.

However, if a user wants matches of a specific pattern then graph mining may not be the best idea because the user is interested in specific pattern. Graph querying, on the other hand, takes graph pattern as an input and retrieves similar patterns from the data graph. In graph querying, we find all the occurrences of a given substructure. In the following section we discuss the relationship between graph mining and graph querying and also discuss how a graph mining system can be re-purposed into a graph querying system.

3.3 Graph Mining to Graph Querying

The abundance of graph data in a variety of domains implies that graph querying is needed in addition to graph mining as a form of information retrieval and analysis. Graph querying can be seen as a special case of graph mining where exploration looks for specific patterns that match a query and hence can be deemed as restrictive mining. Graph mining identifies frequent and significant graph patterns, classifies new graphs based on the knowledge of known graphs, or clusters graphs into subclasses according to the mutual relevance. A graph query, on the other hand, takes a graph pattern as input and retrieves exact match patterns from the data. In graph querying, we want to find all the occurrences of a given substructure. Graph mining, unlike graph querying, starts with all the vertices in the graph and hence, is called unrestricted search. Therefore, for querying specific patterns, general purpose mining approach can be modified based on the nodes in a query. We can make use of information from query graph to improve our search.

In general, a mining system can be converted to a querying system by restricting its start nodes for expansion followed by expanding starting nodes to desired nodes. Intermediate substructures that do not follow the query patterns are discarded. Thus a querying system can be developed with modifications to a mining system. Having

provided an overview of graph mining, graph querying, and an introduction to the Subdue discovery system, in the next chapter, we elucidate how data graphs and query graphs are represented as well as the types or classes of queries that can be handled in our approach.

CHAPTER 4

GRAPH AND QUERY REPRESENTATION

4.1 Graph Representation

In this section, we discuss the representations used for the graph databases. A graph consists of nodes and edges. A node typically contains a node label, a node id and an edge connecting to the other node. An edge is a connection between two nodes which can be labeled or unlabeled, directed, or undirected. Consider the graph shown in Figure 4.1.

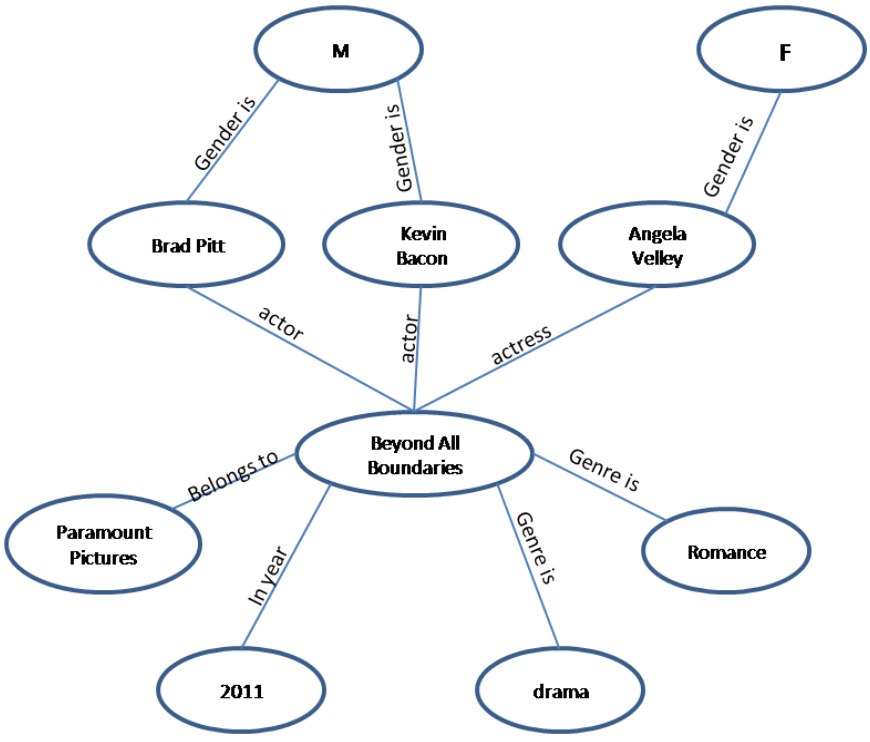

Figure 4.1. An example graph containing movie information.

Figure 4.1 shows a portion of a graph database consisting of a movie and its related information. Nodes are connected to each other based on the relationship between them. If a user wants to retrieve the information from this graph, s/he can query this graph. For example, Find male actors in the movie Beyond all boundaries in the year 2011. In this case all the nodes which are connected to the movie Beyond all boundaries with the edge label actor, would be its answers. Another example of a query could be Find persons who have worked in the movie Beyond all boundaries in the year 2011. In this case, the person is an additional information which denotes the general category for actors/actress in the graph. Since this additional information is missing in the graph (Figure 4.1), so this query cannot be answered with the given information. This necessitates the categorization of the nodes based on their attributes. Moreover, graphs (i.e., IMDB, DBLP, Freebase, Knowledge graph) nowadays come with the property information along with the node. In other words, each node belongs to some category. Therefore, to group node labels with the same semantics (e.g., author, city) in the graph, the concept of type nodes can be used. Non-type nodes are viewed as instance nodes. Every instance node is connected to its respective type node and other instance nodes. The concept of type node is analogous to a super class and all instance nodes belong to the super class. For example, in the case of a social network graph, if two instance nodes John and Mary belong to the type Person then John and Mary nodes would be connected with its type node Person. In case of DBLP there are four types of nodes: author; paper; year; and conference. Each type of node is connected to all instances which are of that type. For example, instances of conferences are the values of conferences such as sigmod, cikm, etc., and the instances for author are names of author who have published a paper, such as Jeffrey D. Ullman, Shantunu Sharma, etc. Consider the graph in Figure 4.2, which is an extended version of the graph in Figure 4.1 including the type information.

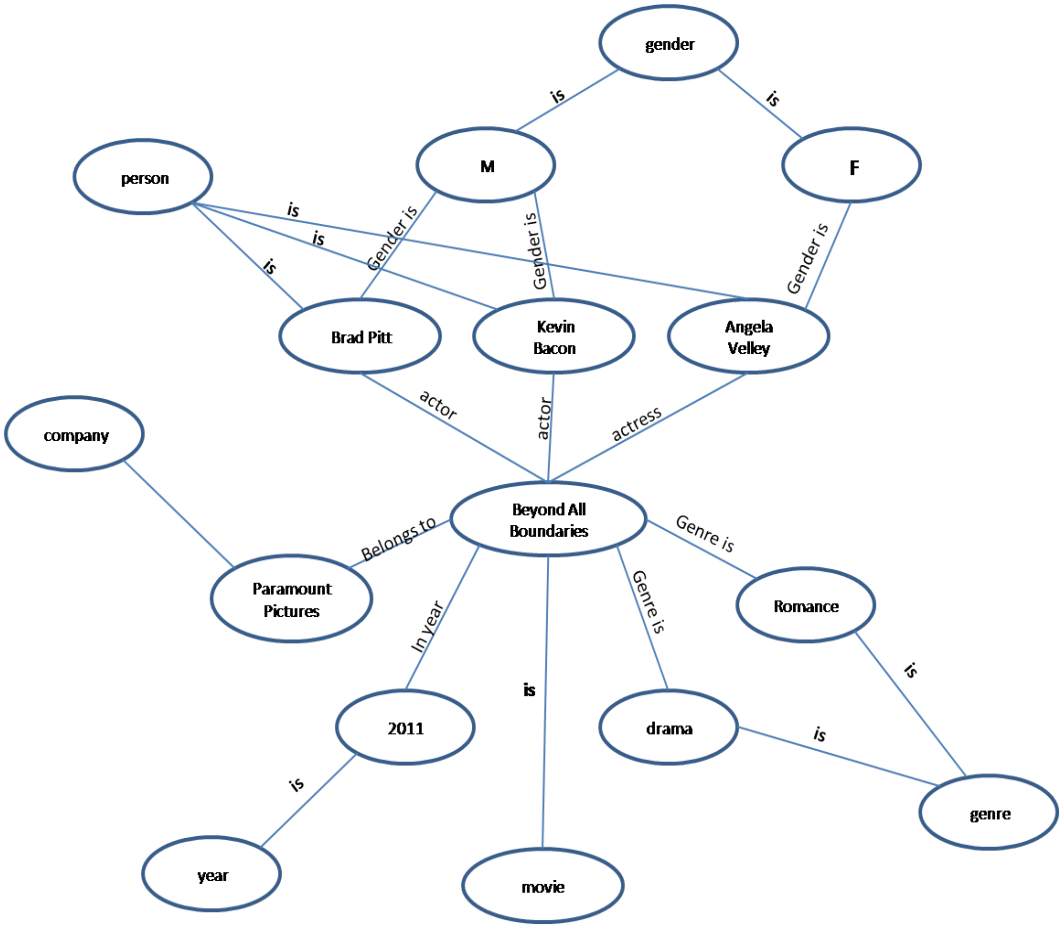

Figure 4.2. An example graph with the type information .

In Figure 4.2 there are six type nodes shown. Number of type nodes is application dependent. This number is an indicator of how many different categories of the nodes exist in the dataset. All instance nodes are connected to their respective type nodes and with other instance nodes as appropriate. This signifies the connection among instances. It captures both property relationship of an entity and relationship across entities.

4.2 Graph Query and its Representation

A query is a request to retrieve the information from the graph database and it can be as simple as, Find names of restaurants in Arlington, or more complex like, Find movies with all its cast information where the movie genre should not be Drama, and all movies should be before 2005, cast should be male and working as an actor, and movies should belong to the company Paramount pictures. Based on the different types of queries, it can be classified into several categories. In the graph database model, several types of queries can be specified:

1. Single relation queries
2. Join queries (multi-relational queries)
3. Queries with logical operators (AND, OR)
4. Range queries (<, >, =, >=, <=, ! =)
5. Aggregate queries (Count)

Similarly, a general graph query processing system should be able to answer all query categories. Consider a conjunction of conditions of the form *attr op value*, where *op* is one of the comparison operators (<, >, =, >=, <=, ! =), and *attr* could be a type or an instance. This is called conjunctive normal form (CNF). General query can be represented as follows.

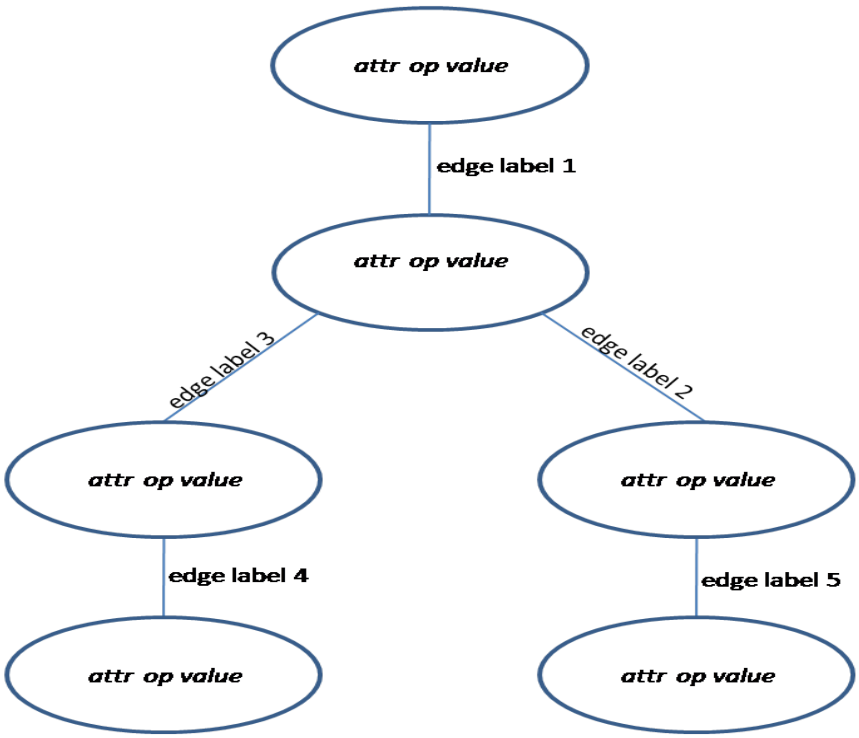

Figure 4.3. General query graph .

Figure 4.3 is a general representation of a query graph. There can be any number of nodes in a query. The above mentioned query categories can be represented using this general representation. In this section, we discuss types of queries with the help of examples.

1. Single/Multi-relational queries: For each instance node there is only the equal operator present in the query. For instance, Find papers published by the author Karl Aberer in the conference BNCOD in the year 2011, is an example of a simple query. Its graphical representation (Figure 4.4) is as follows.

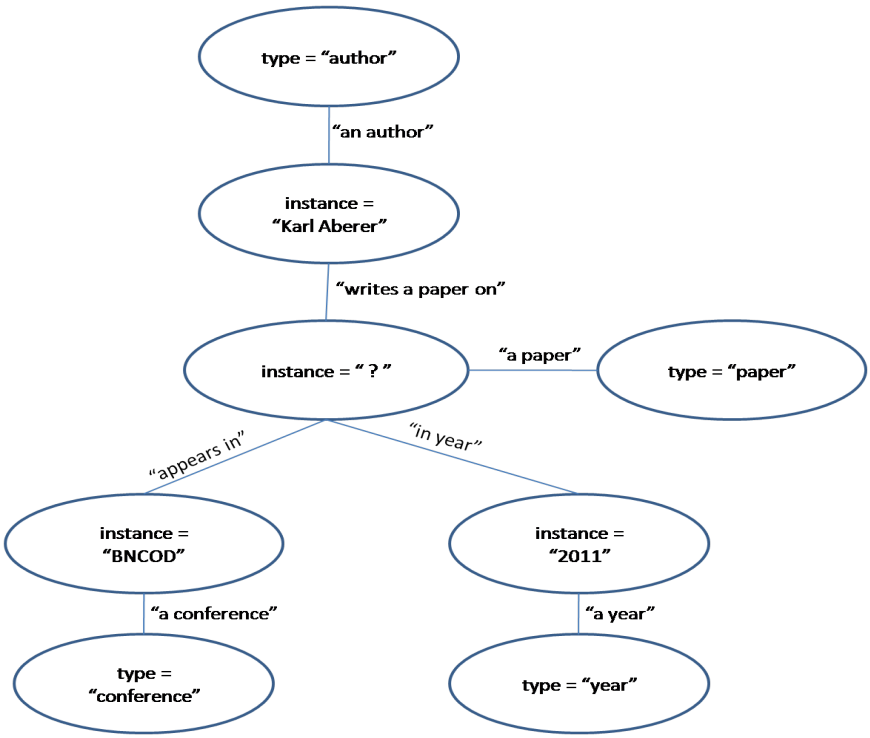

Figure 4.4. Representation for a Simple Query .

Figure 4.4 represents the graph form of the aforementioned simple query. Each instance node is connected to its respective type node. In this query we have to find papers, so the node connected to the type paper is ?. All the edge labels in this query are the same as the edge labels in the main graph.

2. Queries with logical operators: Logical operators include AND, OR, NOT operations. Queries typically include one or more logical operators. Each of the above operators is represented in a slightly different manner.
   (a) AND: This operator is used in queries where all the conditions have to be true in order to fetch the results. Essentially each AND condition is a different node in the query. For instance, Find paper written by Saket Sathe and Karl Aberer in 2011 is an example of a query with AND operator.

In this query Saket Sathe and Karl Aberer are two conditions which have to be fulfilled. Following is the representation of the given query.

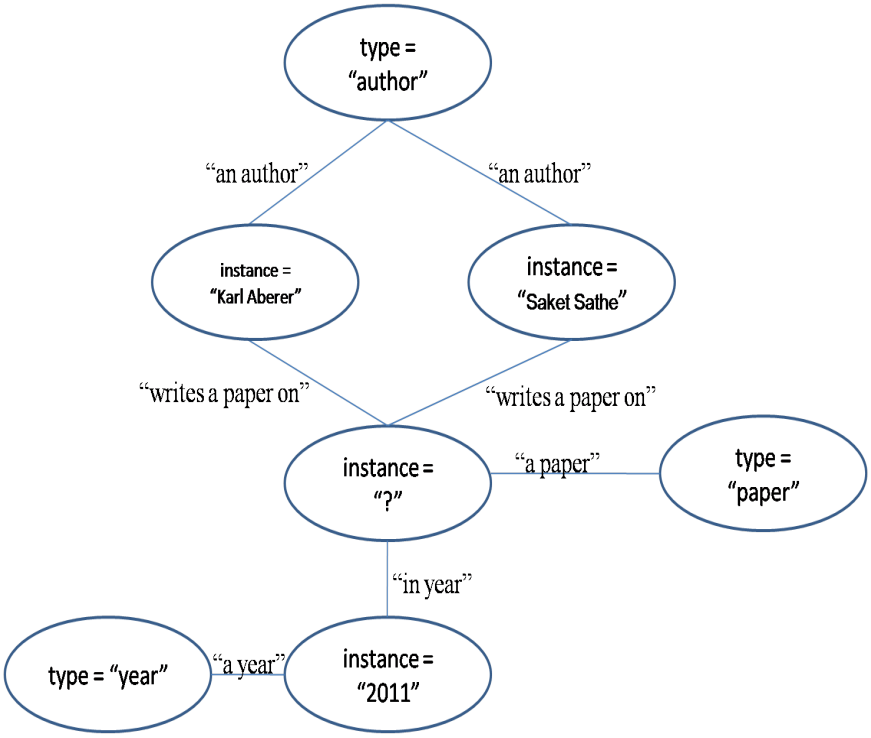

Figure 4.5. Representation for an AND Query .

Figure 4.5 represents the graph form of the AND query. In the case of an AND condition both the author nodes are connected to the paper node, that means both the conditions should be satisfied. Each instance node is connected to its respective type node and other instance nodes. In this query we have to find papers, so the node connected to the type paper is ?.

(b) OR: This operator is used if any one out of all given conditions is true. So in this case, result would be the union of all the OR conditions. For example, Find paper published by Karl Aberer in 2010 or 2011 in the

conference sigmod. in this query results from both the year would include in the answer set. OR queries are represented as follows.

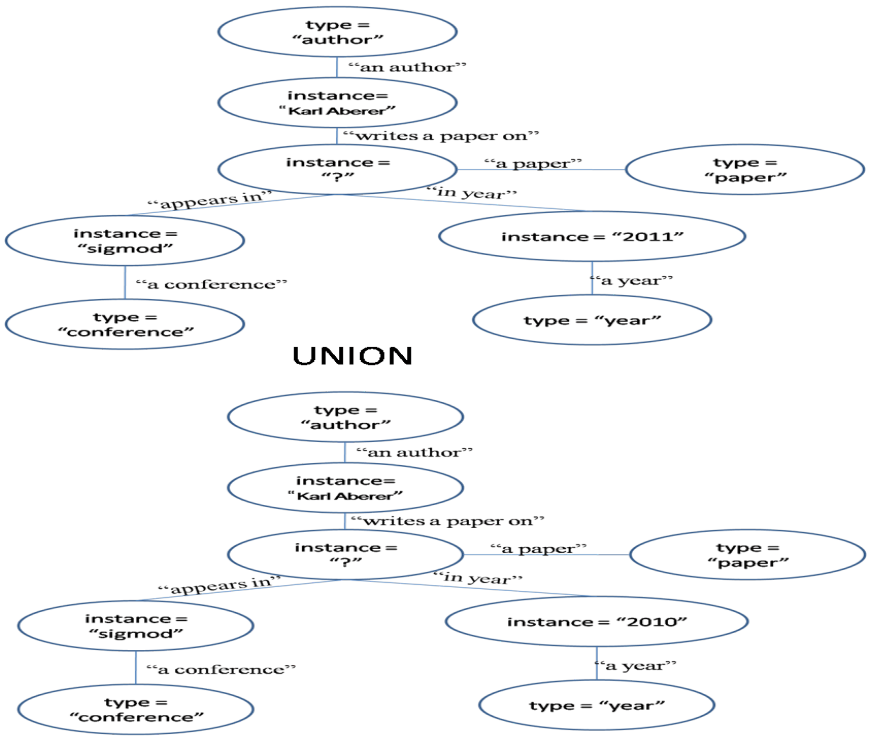

Figure 4.6. Representation for an OR Query .

Figure 4.6 represents the graph form of the OR query. In the case of OR conditions both the year conditions are denoted in two different plans. The result would be the union of both the plans having unique elements. In the case of OR queries, all OR conditions are represented in different sub plans and the result would be the union of all the sub plans. Each instance node is connected to its respective type node and other instance nodes.

(c) NOT: If we want all the results except for a particular condition, the NOT operator is used. In this case, the result set would include all the results except the given NOT condition. To represent NOT query, != symbol is

used. For instance, Find all the movies where Brad Pitt has worked as a director except in the year 2005 shows an example of a NOT query. In this query, results from all years except 2005 would be in the result set. NOT queries are represented as follows.

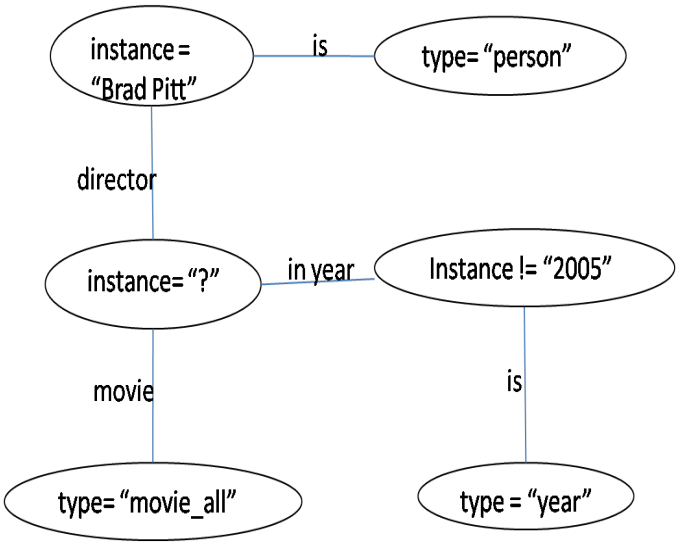

Figure 4.7. Representation for a NOT Query .

Figure 4.7 represents the graph form of the NOT query. In the case of a NOT condition, instead of equality sign we use ! = symbol to represent the condition, that means except the given value, all others should satisfy the condition.

Moreover, queries may contain a combination of more than one logical operator and range operators. In this case each logical operator is represented as mentioned above and the result set would include the answers satisfying all the conditions. For instance, Find movies with all its cast where movie genre should not be Drama, and all movies should be before 2005, cast should be male and working as an actor and movies should belong to the company Paramount pictures contains more than one

logical operator along with a specified range. Following is the representation of this query.

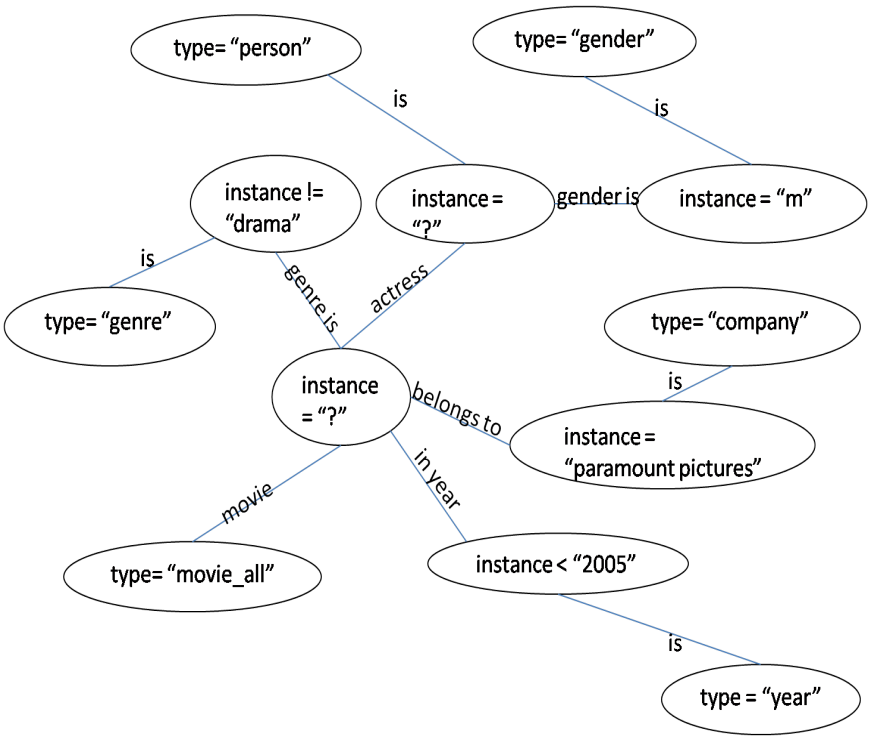

Figure 4.8. Representations for Queries with a Combination of Logical and Range Operator .

Figure 4.8 represents the graph form of the query having a combination of multiple logical and range operators. Each instance node is connected to its respective type node along with other instances.

In this chapter we have discussed several categories of the queries and their representation. A graph query generally consists of a small number of nodes and edges compare to the graph database. A querying system takes a query as an input and retrieves all the exact matches of the query in the main graph. A general query answering system can start from any node in the query and expand to a sequence

of edges in the query graph until all matches are found. An ordering of nodes in which each node (with the desired label) is expanded exactly once forms a query plan. Having provided a detailed description of the query categories, in the next chapter we discuss about alternative query plans and how one plan differs from other plan. We also discuss the cost metric to evaluate plans.

CHAPTER 5

PLANS AND THEIR COST EVALUATION

Queries are the primary mechanism for retrieving desired information from any dataset. Query results are generated by accessing the relevant data by traversing graph databases, in a way that yields the requested information. A query may have multiple results. For a given query, based on the different starting points there may be multiple ways to evaluate the same query and these alternatives are called plans. It is critical that all alternatives give the same result. This chapter discusses the cost metric for the evaluation of alternative plans and when one plan is considered better than the other. We also discuss the catalog information and cost formulas needed for evaluating a plan.

5.1 Query and its matches

A query is a small graph or a pattern (as compared to the graph database) which may have multiple exact matches in the data graph. A query is input to the graph query processor and all the matching patterns are returned. Consider Figure 5.1 which shows the main graph, the query and its exact matches.

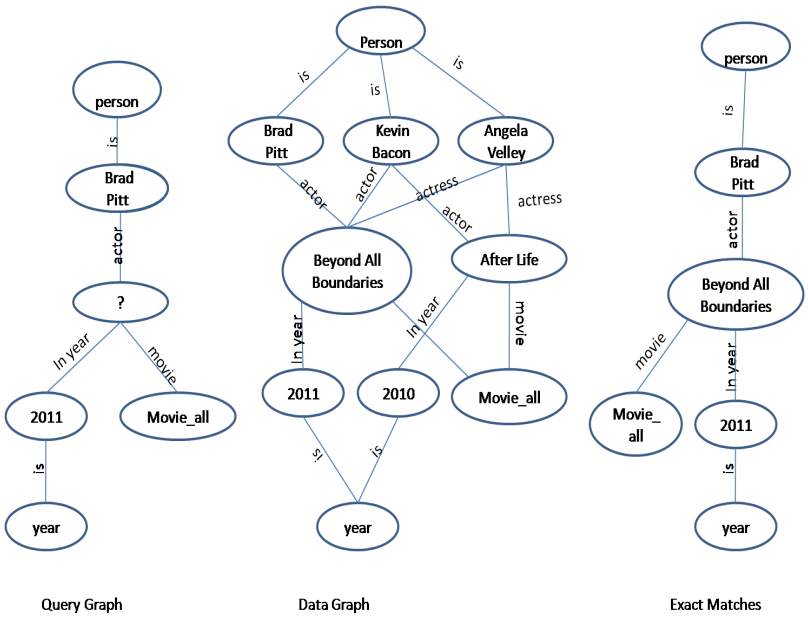

Figure 5.1. A query graph, a data graph and its exact matches.

Figure 5.1 shows the graphical representation of the query (Find movies by Brad Pitt where he has worked as an actor in the year 2011), the graph data and its exact matches. It is evident from the figure that there is only one exact match present in the data graph for the corresponding query. As mentioned earlier, a graph query is a connected graph generally consists of a small number of nodes (10s to 100s) and edges. A general query answering system can start from any node in the query. For the query shown in Figure 5.1, there are five possible starting nodes (except the node having ?) and we can start with any one of them and can expand the node based on the query. An ordering of nodes, in which each query node is expanded exactly once, forms a query plan. Obviously there will be multiple query plans, each having a different ordering of nodes for a single query. Clearly, in Figure 5.1, we have five possible plans based on different starting points. If there are multiple plans then we need some way to evaluate these plans and assess why one plan is better than the others. Intuitively,

a query plan that generates more intermediate results is costlier.This generation of intermediate results needs to be quantified in some way based on the graph properties to convert it into a cost. As discussed earlier, given a query of n nodes with k unknown nodes, number of plans in the query would be n-k. Clearly, evaluating each and every plan will be computationally expensive. Since unknown nodes match every node in the graph, we do not consider them as an initial heuristics (this is similar to the postponing or delaying Cartesian product in a relational model.) Therefore, finding a query plan that minimizes the size of intermediate results is a challenge. Cardinality, selectivity, and join ordering are used in the relational query optimization. Similar to the relational model, for graph databases, the requirement is to collect appropriate metadata information that can be used to evaluate query plans. In the next section we discuss the parameters to evaluate the cost of a plan and the analogy between the metadata of the RDBMS and the graph database.

5.2 Cost Metric for evaluation of plan

As discussed in the previous chapter,a query may contain comparison operators (such as $<$, $>$, $<=$, $>=$, $!=$, $=$), logical operators (AND, OR) and combinations of these. Before we can evaluate the cost, some meta information is needed to estimate the cost of an operation.

As mentioned earlier, in relational databases some meta information is kept to estimate the cost of a plan. Graph databasesare analogous to the relational databases in some ways. For instance, in the graph databases, the type nodes are similar to the attribute names in relational databases. Similarly, the number of instances of a type node is analogous to the number of values of an attribute. Another important parameter is the number of connection an instance node (of a particular type) has to other instance nodes of a specific type with specific edge label. In the case of a

graph, we can identify the number of connection from a type node to its instance node based on the given condition; this gives an estimation of the selection cardinality of a node. Therefore, we need to store some graph information in order to estimate the cost of a plan. This information can be gleaned by processing the graph, which can be effectively used for estimating the number of intermediate substructure. This information is collected and stored in agraph catalog. A traversal of the graph is likely to be needed for collecting this information. If the graph evolves the additional information can be gathered incrementally. In this section we discuss the graph catalog which contains the relevant graph statistics which can be utilized for evaluating a plan.The following metadata is extracted from the graph.

1. Unique Type Names: This captures all the names of unique type nodes present in the graph. In other words, it contains all the category information of nodes in the graph. For example, in Figure 5.1 there are three categories (person, year, and movie) of nodes. This information is analogous to the attribute name in RDBMS. We can determine the category (type) of that instance node by its connection to a type node.
2. Type cardinality: Type cardinality is defined as the number of instances of a particular type node. If there are multiple edge labels from the type node to its instance nodes then the type cardinality is taken with all the different edge labels. This gives us the type cardinality of that type node. As mentioned earlier, this type cardinality is used to estimate the number of substructures generated after exploring the type node.
3. Average Instance cardinality: This number defines the number of connections of an instance with other instances. For example, if an instance of type1 is connected with on an average k instances of type2 and l instances of type3 then average instance cardinality of that node would be k+l. This instance

cardinality is used for the estimation of the number of substructures which are generated after exploring an instance node.

4. Average Connection cardinality: Connection cardinality is the number of the connection of an instance with another instance with particular edge label. This information is analogous to the join cardinality in relation databases. While exploring an instance node to another instance node, this information is used to determine the number of intermediate substructures with the particular edge label.
5. Min and Max values of each type instances: For numeric attributes (such as year), this information tells the range of instances. For example, if there are 100 unique values of the instances of a year type node then, given a value we can estimate the number of substructures based on the condition in node. For non-numeric attributes min and max does not exist.

All nodes in a query follow the form *attr op value* where *op* is one of the comparison operators $<$, $<=$, $=$, $!=$, $>=$ or $>$ and *attr* is either a type or an instance. Based the on value of *op*, selectivity is calculated. For example, if there is an equality operator, the selectivity of equality condition would be 1/ type cardinality, which means the intermediate result size would select only one out of total instances of that particular type node. Similarly, for $>$ and $<$ condition, the selectivity would be (max-value)/(max-min) and (value-min)/(max-min) respectively. For $!=$ condition, the selectivity would be 1- selectivity of an equal condition.

Given a query, the purpose of a query plan generator is to generate a good plan and eliminate potentially bad ones.Any cost computation based on meta data is only an estimate and hence the estimate needs to be accurate in order to identify a good plan. Having provided a detailed description of the graph catalog that contains all necessary information to estimate the query plans, we now elaborate the design

and architecture of query optimizer in the following chapter. A general graph query optimizer will accept a given a query (from a file) and use the catalog information to find a good plan that will be evaluated over the graph database. Note that even after identifying a good query plan, the query plan needs to be executed for finding exact matches from the graph. This calls for query answering techniques which shall also be discussed in the next chapter.

CHAPTER 6

DESIGN AND ALGORITHM

The general approach for the query evaluation is to develop a strategy that can generate an efficient plan and then execute that plan to obtain the results. In this chapter, we discuss the meta data collected for the catalog, a branch and bound algorithm for the generation of one or more plans, and finally modifications to the mining algorithm used by Subdue to convert it into the plan processor. Following is the architecture of our query processor.

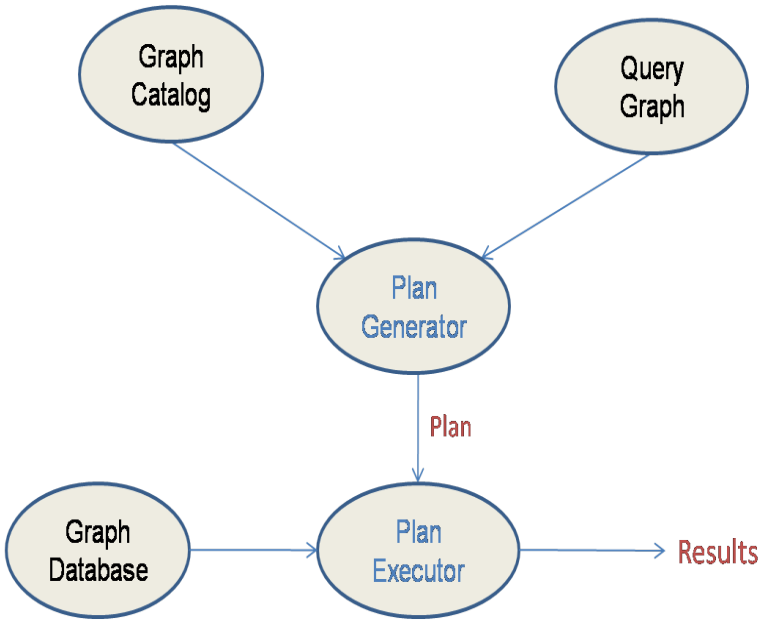

Figure 6.1. System architecture .

Figure 6.1 explains the architectural diagram of the query processor. The query processing is mainly divided into two phases: plan generation and plan execution. As

shown in the figure, generated catalog file and the query are input to the plan generator which outputs a good (i.e., low cost) plan for the given query. After generating the plan, it is used by the plan executor along with the graph database to execute that plan. In the following sections we discuss details of each component described in Figure 6.1.

6.1 Catalog generation

A graph needs to be processed in order to collect the metadata associated with it. Given a graph along with its node and edge labels, we collect all the catalog information described in the chapter 5. Consider a graph G(V, E), which has k type nodes (t_1,t_2,.....,t_k) and each type node t_i ($i \in (1, k)$ is associated with m_i instance/s in the G(V, E). Catalog information is calculated as follows:

1. Type name: Each node in the graph has some attributes associated with it. All vertex label of attribute/type node (t_1,t_2,.....,t_k) presented in G(V, E) are therefore type nodes.
2. Type cardinality: For type node t_i, the type cardinality would be m_i. Type cardinality for a type node can be computed by counting all the nodes in G(V, E) having similar attribute. In other words, given the type node, degree of that type node is the type cardinality.
3. Average instance cardinality: For a given type, average instance cardinality can be computed by taking average of degrees of instances of that type. Average instance cardinality for a type node t_i is

$$\sum_{j=1}^{j=m_i} degree(m_{ij})/m_i$$

where m_i is the number of instances of type t_i and m_{ij} is a particular instance of type t_i where $j \in (1, m_i)$.

4. Average connection cardinality: Connection cardinality for a given type is computed by taking an average of number of connections to other type instances with particular edge label in G(V, E). For two type nodes t_i and t_j, if there are n_i and n_j number of instances (out of m_i and m_j respectively) involved with edge label e then the connection cardinality from t_i to t_j for edge label e would be e_{ij}/m_i where e_{ij} is the total number of edges between m_i and m_j with edge label e .
5. Min and Max values: If t_i is a numeric node type (such as year) then out of m_i instances, minimum and maximum values are kept.

Consider the graph in Figure 6.2. This graph is a snap shot of the IMDB database. The catalog computation explained above is shown for this example graph.

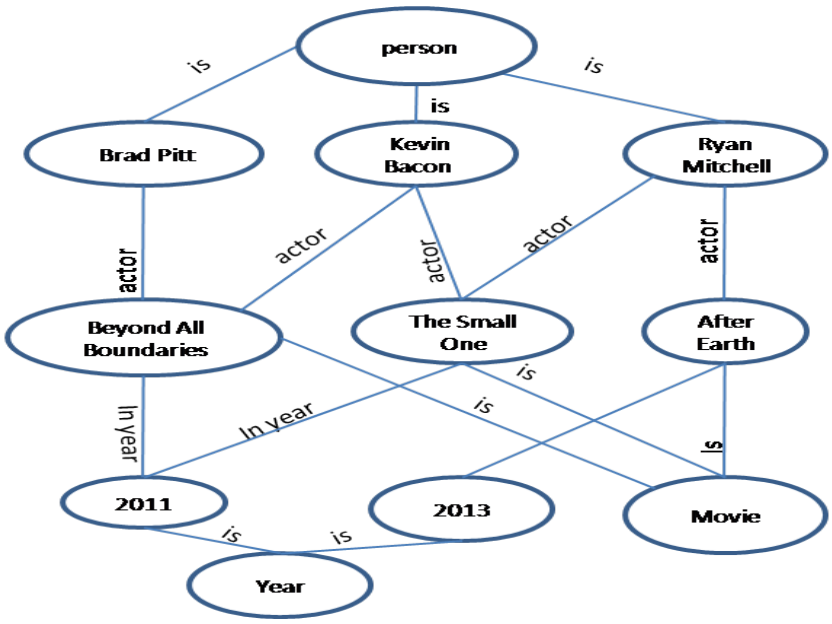

Figure 6.2. IMDB example graph .

Using the formulae described above, the following catalog information is generated.

From Node	To Node	Edge Label	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	year	is	1
year instance	movie instance	in year	3/2
year instance	*	*	5/2

Node	Min	Max
year instance	2011	2013

Figure 6.3. Catalog for graph in Figure 6.2 .

Figure 6.3 has four columns. First two columns are *From Node* and *To Node*, third column contains the *edge label* between first two columns, and in the last column average cardinality information is kept. If we traverse from *From Node* to *To Node* with that *edge label*, then the average cardinality in the catalog is used in the cost estimation. An $*$ in the catalog represents *all*, which essentially means the instance cardinality of *From Node*. For the numeric attributes min and max values are also computed to estimate the selection cardinality. With the discussion of catalog generation, in the section below, we elaborate on how this catalog information is used to estimate costs of partial plans as they are generated.

6.2 Cost estimation of partial plans

For any query, assuming the availability of a catalog, the goal of plan generator is to generate a good plan. In plan generation we use the number of substructure generated during the answer computation as a cost metric. Intuitively, a plan which generates more number of substructures while computing an answer is costlier. To keep track of the cost of a partial plan, two parameters *cost* and *currSubs* are maintained. *Cost* signifies the total number of substructure generated by the plan so far while *currSubs* maintains the number of estimated current substructures during the expansion in that iteration. The algorithm first finds the seed node to start with. For a query with n nodes, if there are m unknown nodes, this algorithm starts from all the n-m nodes and the estimated cost is associated with each of the starting nodes. The unknown nodes (i.e., nodes with a question mark) are not used as a starting nodes as its cardinality is the number of nodes in the graph as a ? matches any label. The plan generation algorithm uses the branch and bound algorithm and beam size (k) which determines the number of plans to be expanded after each iteration. The algorithm picks k (out of n-m) least cost plan for expansion. All connections of a node are known from the query. In a plan, a node is expanded to the node which generates less number of substructures compared to other nodes. After each expansion, cost is updated. This algorithm continues expanding k least cost plans until it covers all edges in the query for all k plans. The sequence of nodes in which they are expanded is returned as a plan.

6.2.1 Cost estimation in plan generation

In the plan generation algorithm cost is incurred in each expansion. Cost is typically the function of instance cardinality of the node (which we are expanding on) and the number of current substructure in the plan. If the plan which is expanded

on t_i (or an instance of t_i) to t_j (or an instance of t_j) with an edge label e then the cost would be the multiplication of instance cardinality of t_i and number of current substructure. Similarly number of estimated substructure would be the product of current number of substructure, connection cardinality (t_i to t_j with edge label e), and the selectivity of a query node. In each iteration substructure having n node is expanded to substructure with n+1 nodes and corresponding cost and number of current substructures are updated.

6.2.2 Algorithm of the plan generator

Above is the algorithm for generation of the plan. This algorithm takes catalog information and a query graph as input and generates a good plan as an output (line 1 to line 2). This algorithm uses the branch and bound technique to limit the search space and for that one parameter *Beam Width* (k) is kept. It means that at a time only k out of all possible plans will be expanded. This algorithm starts with taking all the known nodes in the query as starting point of the plans (line 3 to line 6). All the plans are now initialized with the estimated cost based on the catalog information (line 7 to line 9). This algorithm now picks best k cost plans and expand these plans from n edge substructure to n+1 edge substructure. After each iteration best k plans are updated. Algorithm continues until k least cost plans get completed (line 10 to line 17).

Consider a query "Find persons who have worked as actors in the movie Beyond all boundaries in 2011". The graphical and subdue representation of the query is shown in the Figure 6.4 and Figure 6.5 respectively. The beam size (k) in this example is 5. The catalog is shown in the Figure 6.6.

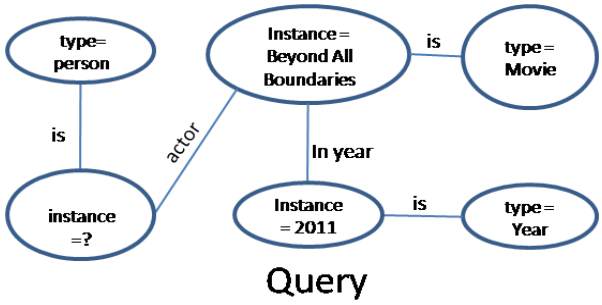

Figure 6.4. Graphical representation of the query.

```
V    1    type = "person"
V    2    instance = "Beyond all boundaries"
V    3    type =  "movie"
V    4    type = "year"
V    5    instance = "2011"
V    6    instance = "?"
E    1    6    "is"
E    6    2    "actor"
E    2    5    "in year"
E    2    3    "is"
E    5    4    "is"
```

Figure 6.5. Subdue representation of the query.

From Node	To Node	Edge Label	Avg. Cardinality
person	person instance	is	5000
person instance	person	is	1
person instance	movie instance	actor	4.2
person instance	*	*	6.3
movie	movie instance	is	100
movie instance	movie	is	1
movie instance	person instance	actor	20.7
movie instance	year instance	in year	1
movie instance	*	*	76.8
year	year instance	is	100
year instance	year	is	1
year instance	movie instance	in year	49.4
year instance	*	*	50.4

Figure 6.6. Catalog for IMDB dataset (12K vertices and 30K edges).

The query plan is evaluated for the given query in Figure 6.4. In this query there are 6 nodes and out of that only 1 node is unknown, therefore the algorithm starts from all the remaining 5 nodes. *cost* and *currSubs* are initialized by 1 for each of the plan, since there are only one substructure in each of the plan. Following the algorithm plans are generated as shown in the figure below.

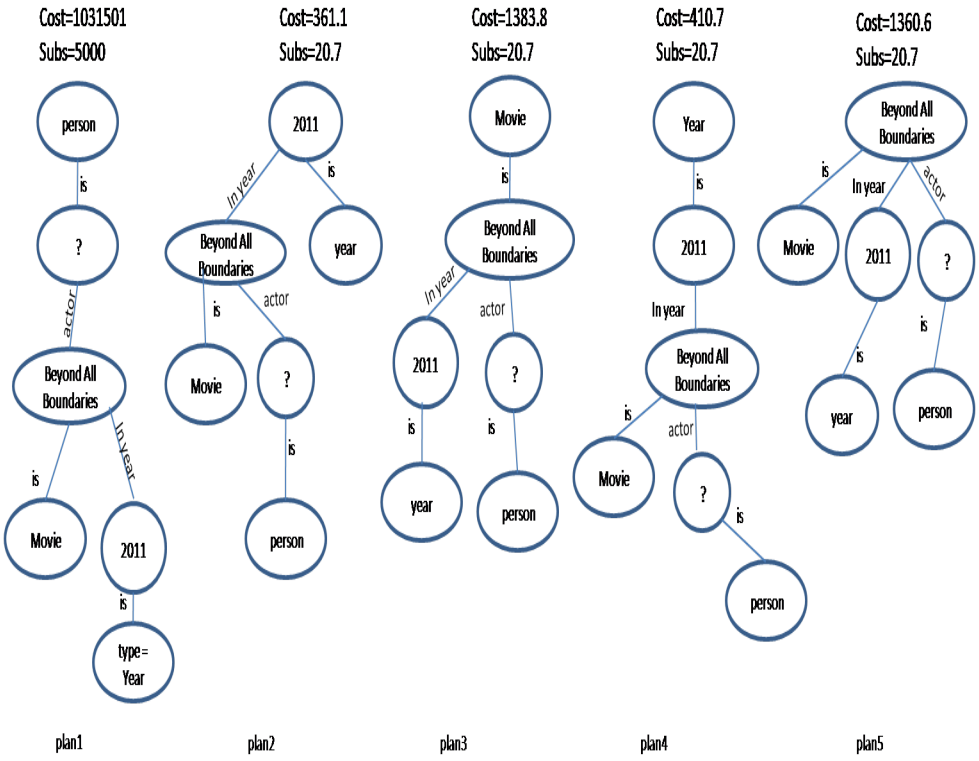

Figure 6.7. Output plans are shown for the given query graph.

In above Figure 6.7 all five plans are shown with their estimated cost (number of intermediate substructure). Each plan gives the same result set but is different in terms of generating intermediate number of substructure. Intuitively if a plan generates more number of substructure, it will take more amount of time to process, hence is a costlier plan than the one which generates less number of intermediate substructure. There are two factors which determine the cost of a plan. First is the degree (cardinality) of a node. If we start with a node which has higher degree than other node, that means it will generate number of substructure equal to its degree. In all the plans shown in above figure, plan 1 starts from a node which has degree higher than other starting nodes so initial cost becomes high. Second is position of an

unknown node, if algorithm expands to an unknown node first, then the number of substructure in the plan becomes more, so each expansion is performed for all number of substructures in the plan. Therefore the cost of the plan becomes high. This is analogous to pushing selection and projection down in the tree in the RDBMS. In the plan 1 in above figure, first expansion is on unknown node, due to this plan keeps all the generated substructure. Each expansion after that is performed on all these number of substructures, that is why the estimated cost of the plan 1 is more. In all the plans except plan 1 number of expansions after encountering unknown node are comparatively less, hence the costs of other plans are less compare to the plan 1. In plan 2 algorithm expands on all possible known nodes first, that is the reason why plan 2 performs best among all the plans. Plan 2 and Plan 4 encounter unknown node at same position, so the difference in cost is because of the cardinality of starting nodes. After generating a good plan by the plan generator, the plan is fed to the plan executor with the graph database. In the section below we elaborate the modifications in an existing system called Subdue and the working of plan executor.

6.3 Plan Execution

As discussed earlier in previous chapters that Subdue is a graph mining system which takes the graph database as an input and finds the interesting substructure from the data. We also elaborated the relationship between graph mining and graph querying. A querying (also called as restrictive mining) is a special case of mining. In this section we discuss the modifications in the Subdue, and how a mining system can be converted to a querying system.

In mining algorithm the only input is the graph databases and the output is hidden interesting substructures. On the other hand, in the case of querying system, query plan (sequence of nodes to be traversed) also needs to be input to find the

matches. Graph mining algorithm starts from all the unique node labels in the graph while in our case, we start from a unique vertex label specified in the plan. Instead of expanding in all possible ways (in case of mining), we restrict the expansion (to only the particular node with specified edge label) based on the plan. In addition, to support comparison operators we have implemented <, <=, >, >=, ! =, and = operators. When the node is expanded to the desired node label, operators are checked to ensure that whether the node (which algorithm is going to expand upon) meets the condition. If the node does not meet the condition it is discarded. We continue until all the nodes and edges in the plan get covered.

The algorithm for plan execution takes graph database and query graph as input and outputs all the exact matches to the query (line 1 to line 2). Each plan is a sequence of nodes to be traversed. This algorithm starts from finding the first node label in the graph database (line 3) and expands this node to next node as specified in the plan . It keeps on expanding on the nodes to the desired nodes according to the plan until all the nodes and edges get covered (line 4 to line 7).

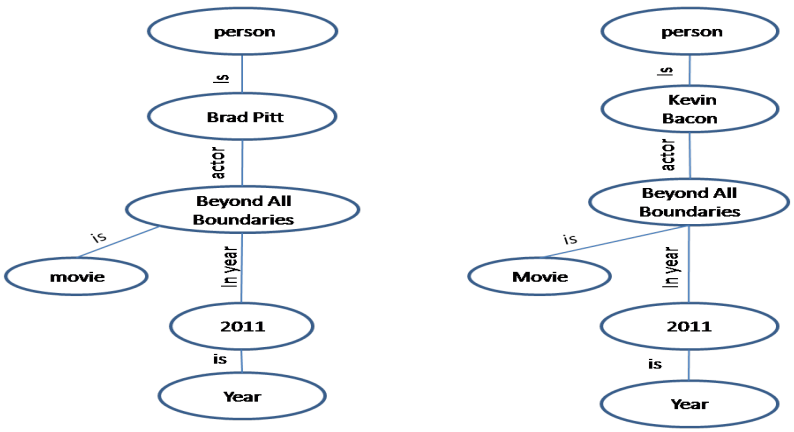

Figure 6.8. Results from the plan executor for the input query plan.

After executing the plan in QP-Subdue results are shown in the form of a graph. For the query in Figure 6.4 results are shown in the above figure, which are the exact matches of the specified query. In this chapter we have explained the design of the system architecture and also elaborated the detailed algorithm of each component involved. Having provided this discussion, in the next chapter we elaborate the experiments and their results on various plans.

Algorithm 1 Plan Generation algorithm

1: ***Input***: Query Graph, Catalog information, Beam Size (k)
2: ***Output***: k alternative plans with their estimated cost
3: initialize planNodeList with null
4: **for** each node q in the query add q as a starting point of a plan **do**
5: add q to planNodeList
6: **end for**
7: **for** each node p in planNodeList **do**
8: initialize p with the estimated cost using the catalog information
9: **end for**
10: **while** number of completed plans ¡ k **do**
11: **for** each of k lowest cost plans **do**
12: expand each plan to the next node
13: cost = previous cost + added cost;
14: **end for**
15: update k lowest plans
16: update number of completed plans
17: **end while**
18: write out k plans

Algorithm 2 Plan Execution algorithm

1: ***Input***: Graph database, Query graph

2: ***Output***: Exact matches of the query graph in the graph database

3: Initialize current node with the starting node based on the plan

4: **while** all the nodes and edges in the plan get visited **do**

5: Expand the current node to the desired node with specific edge based on the plan

6: Update the current node according to the plan

7: **end while**

CHAPTER 7

EXPERIMENTAL EVALUATION

This chapter presents the results of extensive experimental analysis performed on various queries of different databases. The experimental result reinforce our premise that the number of intermediate substructures generated in any plan determines the execution time of the plan. The consistent performance of the plan generator across different types of queries and databases establishes the applicability of our proposed approach for plan generation. As we have considered different real time graph databases and queries, the performance of the plan generator for each database is presented in detail in a separate section. The experimental setup and a brief description of the data set used is also provided. Before we elaborate the performance of the plan generator, a brief overview of the system implementation is presented below.

7.1 Implementation

The plan generator is implemented in Java. Java has been chosen as the language of choice, as it provides excellent support for string processing, since the labels in a typical graph database are strings. The plan execution algorithm is implemented in C as it is a modification of the Subdue system that has been implemented in C. In order to make it easier for a user, the inputs from an end user is kept to a minimum and a configuration file is used to provide either defaults or choices. A scripting language is used to accept the configuration file containing all the information. The QP-Subdue system consists of the following modules: i) catalog generator, ii) plan

generator which can output various types of plans including all plans for a query, a query processor that takes a graph database, a plan, and outputs the exact answers to that plan. The QP-Subdue system generates several intermediate outputs including the catalog file, a plan file consisting one or more plans for each query, and an output file for each query. The configuration file contains all the information needed for the QP-Subdue system to process one or more queries on a graph database.

In the implementation of the various routines, we have been able to optimize greatly due to the use of Java, which is geared towards string processing and extraction. The availability of pre-developed functions for many routine tasks and the ability of handle complex data structures that have been utilized in the implementation justify its use. In the discussion that ensues we will briefly describe some of the implementation aspects of the various modules and parameters used.

7.1.1 Plan generation

Plan generator takes catalog information and query graph as input. The catalog information needs to be stored before we can utilize this. As described in the previous chapter that for expansion in the query plan, we need to find the average cardinality information for the unique tuple (combination of *from node*, *to node* and *edge label*). The combination of unique tuple is stored as a key in a hash map and the value of the hash map is the average cardinality. After processing the catalog files query file is also processed. In query processing we create an adjacency list for each node based on the connectivity and assign a boolean flag with each node to keep track of whether the node is a type or an instance node. A query node has the vertex label in the form of *attr op value*. However, in the graph database, vertices have only *value*. *attr* and *op* in the query node carry the information of type of node and operator respectively. This information is kept separately during the parsing of query. In other words, *attr*

and *op* is stored separately than the actual vertex label (*value*). This algorithm also uses branch and bound technique to limit the search space, which means only *beam width*(k) number of plan are expanded instead of expanding all the plans. We keep track of top k plans with their plan id in a hash map and after each iteration, hash map is updated. In each expansion estimated cardinality information is taken from the catalog and estimated cost of a plan is updated. The output is a plan file which consists of top k plan with their estimated cost.

Input for the plan generator is a catalog and a query file, and output is the plan file. Catalog is shown in the Figure 7.1.

From Node	To Node	Edge Label	Avg card
Person	person instance	is	5000
person instance	Person	is	1
person instance	movie instance	actor	4.2
person instance	*	*	6.3
Movie	movie instance	is	100
movie instance	Movie	is	1
movie instance	person instance	actor	20.7
movie instance	year instance	In year	1
movie instance	*	*	76.8
Year	year instance	is	100
year instance	Year	is	1
year instance	movie instance	In year	49.4
year instance	*	*	50.4

Node	Min	Max
year instance	1965	2013

Figure 7.1. Catalog file.

Similarly, the graph query is represented using the subdue representation. Graphical query and subdue representation are shown in the Figure 7.2 and Figure 7.3 respectively.

Query: Find persons who have worked as actors in the movie "Beyond all boundaries" in 2011.

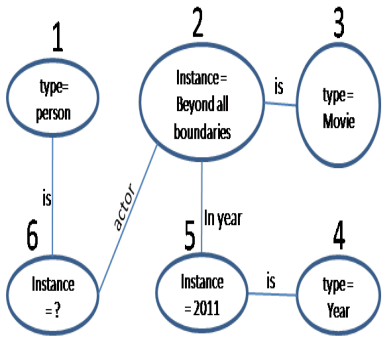

Figure 7.2. Graphical representation of a query.

V	1	type = "person"	
V	2	instance = "Beyond all boundaries"	
V	3	type = "movie"	
V	4	type = "year"	
V	5	instance = "2011"	
V	6	instance = "?"	
E	1	6	"is"
E	6	2	"actor"
E	2	5	"in year"
E	2	3	"is"
E	5	4	"is"

Figure 7.3. Input file of the query in Figure 7.2.

After processing the query, plan generator outputs a plan which is fed into plan processor. Figure 7.4 is the representation of the output plan.

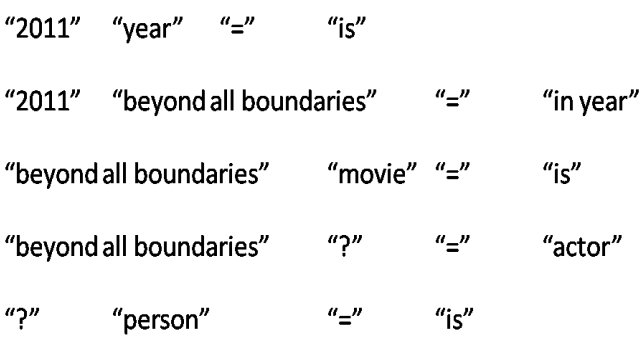

Figure 7.4. Plan file (input to the plan processor) .

In above figure, plan is the sequence of edges, to be visited. In this case the first node would be *2011*, which is expanded on *year* node with edge label *is*. = operator means that the node is equal to *year*. Similarly all the edges are covered in the plan. In the section below, implementation of plan executor is discussed.

7.1.2 Plan Execution

Plan execution takes place after the plan generation. Plan file and graph database are input to the plan executor. Plan executor first processes the graph database and stores the information of vertices and edges. Plan file contains the sequence of nodes with operators and edges. Subdue starts the discovery process by finding the starting node mentioned in the plan file. In this step, all vertices with this label become the starting point (instances of the substructure having one node). From the plan file, now next node along with edge label and operator is looked up. After first iteration, all these instances (starting nodes) are expanded in all possible ways but only those substructures are kept which meet the conditions (next node label, edge label connecting both the nodes and operator) specified in the plan file. This process continues until all the nodes and edges in the plan file get visited. Structures

which do not meet the conditions in the plan file, are discarded. After processing the query results are shown in the form of subdue graph representation as shown in the figure below.

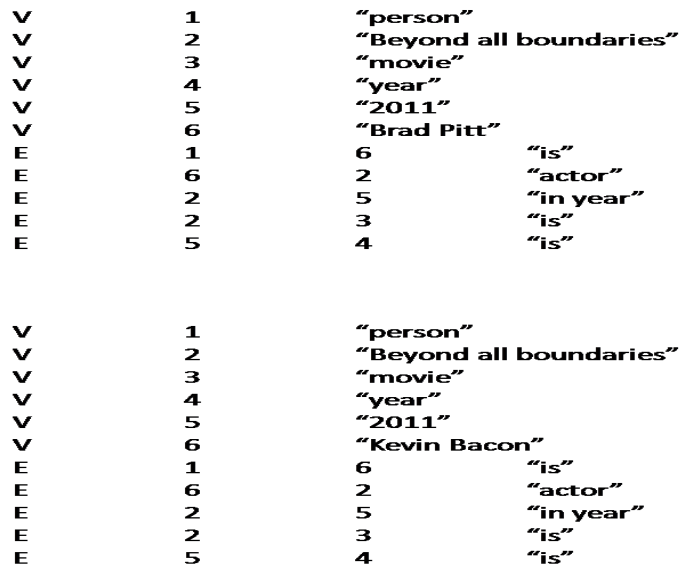

Figure 7.5. Query matches after processing the plan on the graph database .

The query matches are shown in the above figure, which have representation similar to the query. In the section below we discuss the experimental analysis of various categories of queries and the performance of the plan generator.

7.2 Experimental Analysis

With the important implementation details outlined, we now elaborate the performance of the plan generator for various queries over different graph databases. All experiments have been carried out on Dual Core AMD Opteron 2 GHz processor machine with 16 GB memory. Extensive experiments on different queries with diverse

characteristics have been carried out to study the performance of the plan generator. In the section below we discuss the data set and query characteristics.

To evaluate the performance of the plan generator, we used IMDB and DBLP data sets for the experiments. The maximum size we could handle for these graph databases is with 350000 nodes and 1200000 edges on 16 GB machine, since the whole graph loads in the main memory. DBLP data set contains the information of publications along with the information of their authors, conferences and years. Similarly, IMDB graph database contains the information of movies, actors, genres, year, company, etc.

For the above mentioned graph databases, we took queries having different characteristics such as queries with a comparison operator ($<$, $<=$, $>$, $>=$, $!=$, $=$), 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 the section below we demonstrate the performance of the plan evaluator on different categories of a query. Plan generator generates a good plan using the information of catalog and query. Intuitively, a plan which generates less number of intermediate substructure, should take less amount of time to obtain results. Therefore the minimum cost plan would be a plan which generates minimum number of intermediate substructure among all possible plans. Similarly the maximum cost plan would be the plan which generates maximum number of intermediate substructure. In order to see the difference among various plans, we generated all the possible plans from the query and picked minimum, median and maximum cost plans to see the efficiency of the plan generator and to evaluate the difference in time taken to execute various plans.

7.2.1 Performance of plans

- AND Query: "Find tv-series and its company name, where *Kelsey, Wagner* has worked as an *animator* and genre of the tv-series should be *animation* ***and*** *comedy*" is an example query to IMDB graph database containing *AND* logical operator. The *AND* operation is between two genres *animation* and *comedy*. The result set would contain tv-series which belong to both the specified genres. Below are the results of running different plans (min, med, max cost plans) on different sizes of IMDB data set.

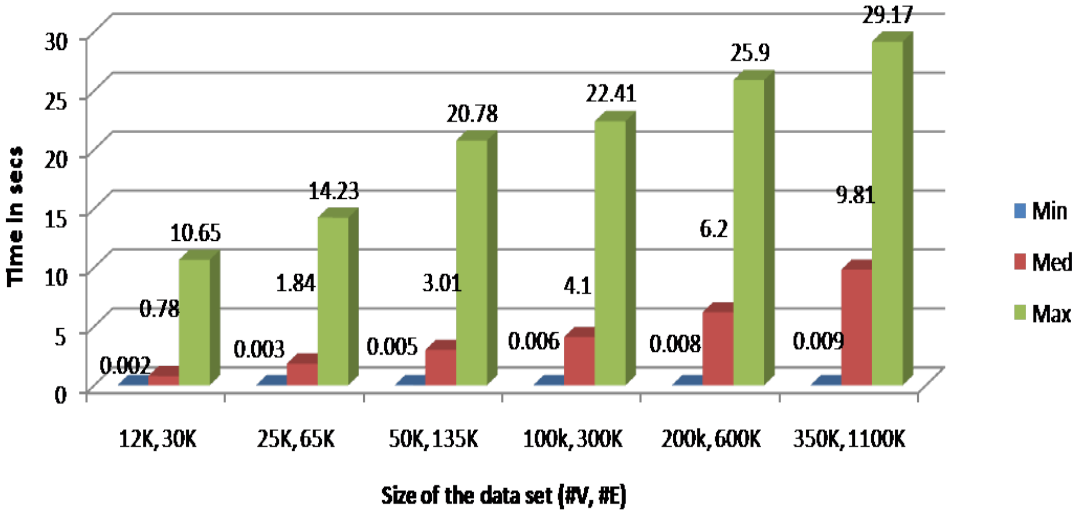

Figure 7.6. Execution time (in secs)of an AND query on different sizes of databases for min, med, and max cost plan .

As it is clear from the figure that in each data set, minimum cost plan takes significantly less amount of time. Clearly, the difference in execution time between min cost plan versus other plans is due to the number of intermediate substructure generated/processed. The minimum cost plan performs at least

300 times better than the worst plan in all the cases. It evidently shows the efficiency of the plan generator.

- COMB Query: "Find tv-series and its company by *Soler, Rebecca* where the genres should be *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 and logical operator. In this query the result set would contain tv-series which belong to both *drama* and *family* in all the years except *1996*. In different plans, different number of intermediate substructures are generated. Execution time for different plans over different sizes of data set is shown in the following figure.

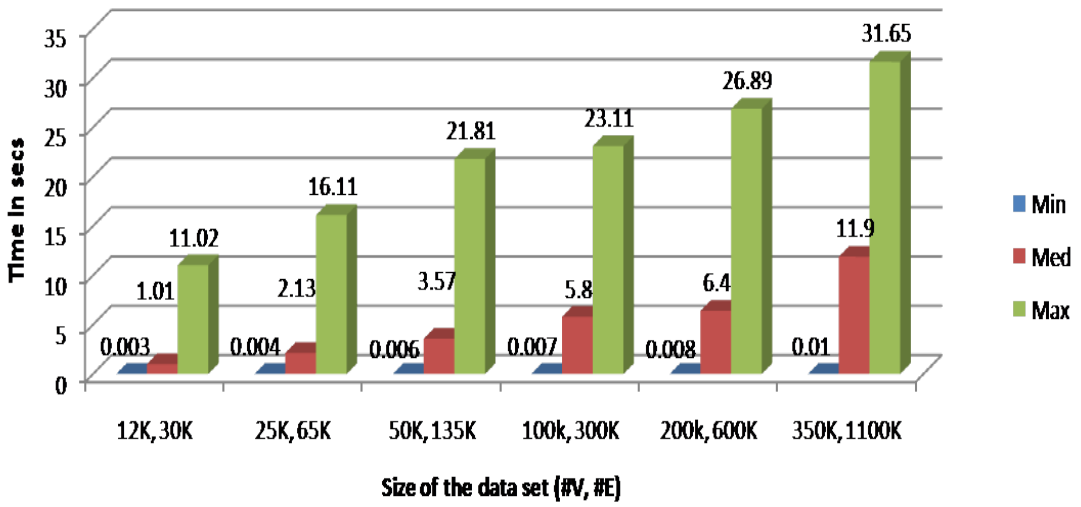

Figure 7.7. Execution time (in secs) of a COMB query on different sizes of databases for min, med, and max cost plan .

In the figure above min, med, and max cost plans are run on the different data set and in each data set, performance of the minimum cost plan is significantly higher than other plans.

- OR Query: "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. As explained earlier, *OR* queries are divided into multiple sub plans based on the *OR* condition specified. Therefore, this query first finds the tv-series where *Kelsey, Wagner* has worked as an *animator* and then the second part finds tv-series where *Soler, Rebecca* has worked as an *actress*. The union of both the sub plans would be the result set. The same query is run on different sizes of IMDB graph database and following results are obtained.

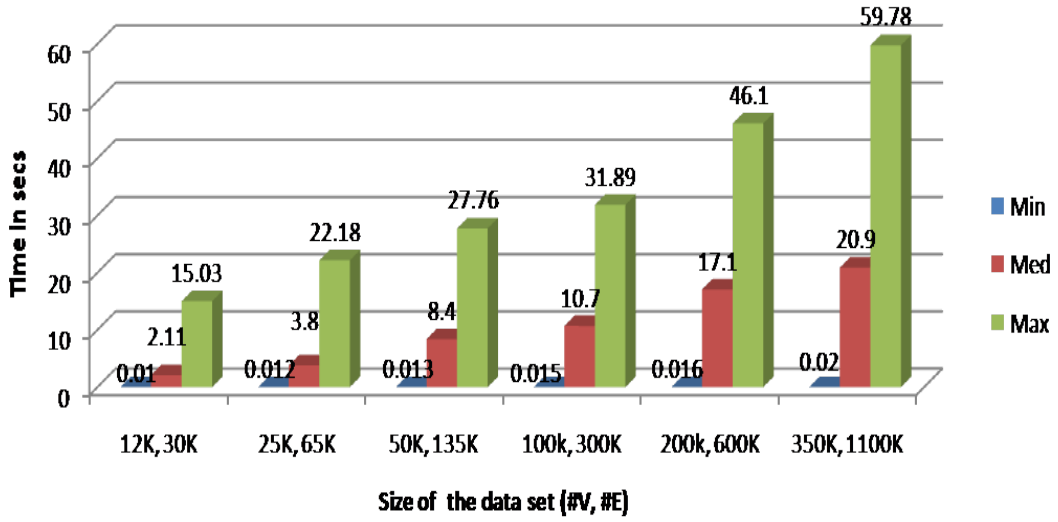

Figure 7.8. Execution time (in secs) of an OR query on different sizes of databases for min, med, and max cost plan .

Again, the performance of the minimum cost plan by the plan generator is exceptional compared to the other plans. On the largest data set the minimum cost plan performs approximately 300 times faster than the maximum cost plan. If we compare the execution time of this *OR* query with the execution time of above mentioned *AND* and *COMB* query, the time taken in the case of *OR* is more. One of the reason for it is because of multiple sub plans. The time for the minimum cost plan on different data sets varies from 10 milliseconds to 20 milliseconds, while for the maximum cost plan it varies approximately from 15 seconds to 59 seconds.

- Comparison operator (<): "Find papers published by the author *Eric Hanson* ***before*** the year *2009*" is an example of query which contains < operator, and this query is run on DBLP graph database. In this query, user is looking for papers by a particular author before the specified year. We run this query on DBLP graph databases of different sizes, following are the results of executing different plans on these data set.

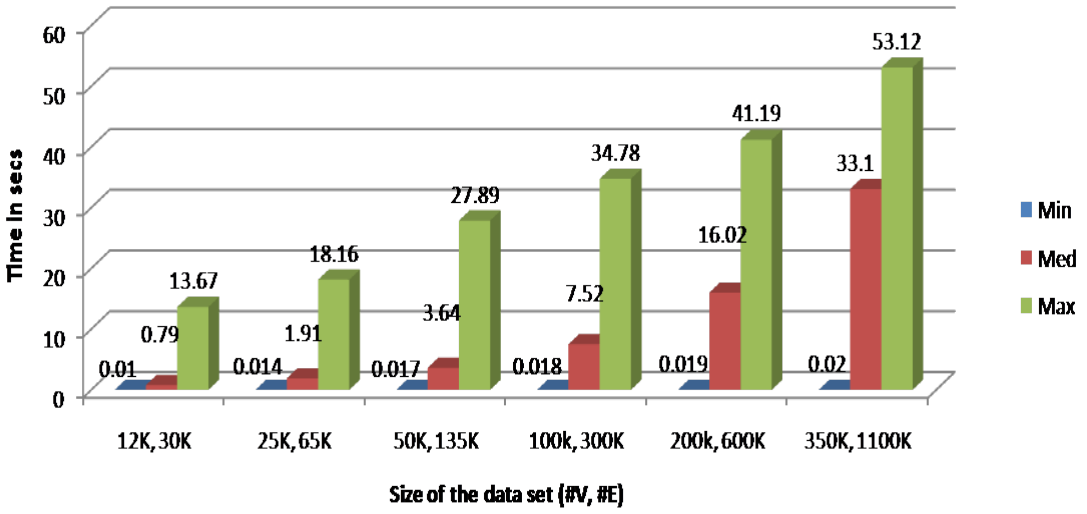

Figure 7.9. Execution time (in secs) of a query containing comparison operator on different sizes of databases for min, med, and max cost plan .

In the figure shown above, for each data set, three plans are executed for the same query. The minimum cost plan executes in considerably less amount of time compared to other plans. This shows the effectiveness of the plan generator.

- COMB Query: "Find papers where *Yuri Breitbart* ***AND*** *Abraham Silberschatz* have collaborated together ***after*** the year 1980" shows an example of a combination of a query with comparison and logical operator. In this query the ***AND*** operation is in between two authors and all the papers ***after*** 1980 would be in the result set. This query on DBLP data set performs as shown in the following figure.

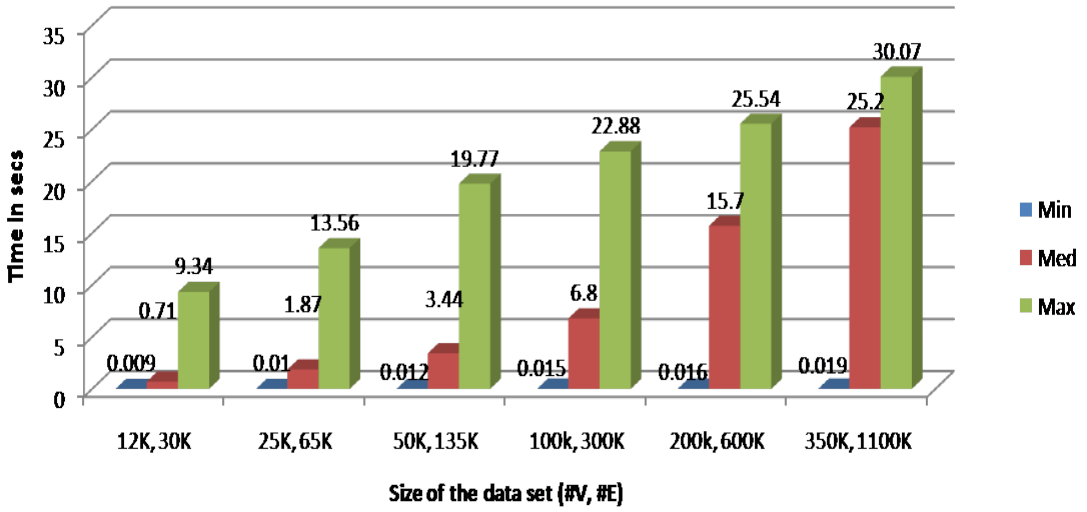

Figure 7.10. Execution time (in secs) of a combination of multiple operator on different sizes of databases for min, med, and max cost plan .

Execution time is shown in the figure for the query with a combination of multiple operators. 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. In each data set, the minimum cost plan performs better than any other plan.

- Query with multiple unknowns: Plan generator is also able to handle queries which have multiple unknown nodes. "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). Clearly if there are multiple unknown nodes that means the expansion would happen to all possible nodes, therefore number of intermediate substructure would be more. The execution time for this query is shown in the following figure.

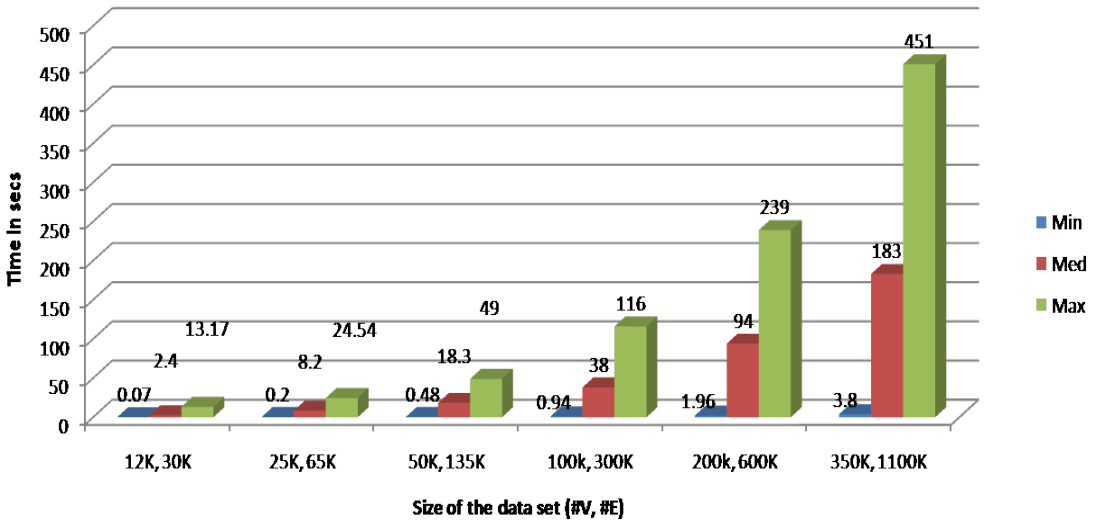

Figure 7.11. Execution time (in secs) on different sizes of databases for min, med, and max cost plan .

It is clear from the above results that execution time of this query is comparatively more than any other previously mentioned query. As explained that it is because of more number of intermediate substructure. In this query the minimum cost plan varies from 70 milliseconds to 3.8 seconds, while the maximum cost plan varies from 13 seconds to 451 seconds. Evidently, the effectiveness of the plan generator is in bringing down the time from 451 seconds to 3.8 seconds in the largest data set. In all the data set this system is capable of generating a plan which is at least 100 times faster than the worst plan.

In summary, we have carried out exhaustive experiments across various domains and presented the results of our findings. The consistent performance of the plan generator has validated our expectation about the feasibility of the proposed novel approach for various types of queries over graph databases.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

In this thesis, we have proposed an approach for query optimization in graph databases that uses similar techniques employed in relational databases. We have developed a framework that allows us to generate query plans for various types of queries which include comparison and logical operations. Our premise that time taken to evaluate the query depends on the intermediate number of substructures is justified by the time taken in the execution of different plans which differ in their estimated number of intermediate substructures. The plan which generates less number of substructures takes less time for its execution. The proposed approach overcomes the limitation of conventional techniques in graph databases that evaluate the query without generating any specific plan.

In order to evaluate queries efficiently, we have developed the plan generator for various types of queries and have also demonstrated how a graph mining system can be modified to a graph querying system. The need for a plan generator for generating cost effective plans has been established. Various parameters that affect plan generation have been identified and analyzed in detail. The results of the exhaustive experiments that were carried out validate the effectiveness of our approach.

We have proposed a query representation scheme for various types of queries (containing different relational and logical operators) which can be used to retrieve meaningful information from a graph databases. The plan generator effectively uses the catalog information which is generated just once from the graph database (by making a single pass over the graph database input file and not even materializing

the graph in memory; hence this can be used for a graph database of any size) and used for query processing. With the work presented in this thesis, we have developed the underpinnings of the query optimization techniques for graph databases. Although the performance of query optimizer is good, more work needs to be carried out to include complex conditions that involve both relational and logical operators including the NOT operator. Some of the enhancements that can be carried out are outlined in the following discussion.

The performance of the queries generated by our plan generator for various types of queries is significantly better. However, further improvement can be done in handling a broader range of queries. In the current implementation, we handle queries with all comparison operator ($<$, $<=$, $>$, $>=$, $=$, $!=$) and logical operator including *AND* and *OR*. Relational databases, on the other hand, are capable of handling wide range of query categories (Logical NOT, order by, group by, In, and Conjunctive Normal Form etc.) and aggregation. If we compare our work with the relational databases, it can be significantly improved by incorporating other useful operators. Another scope of improvement is in the catalog information. Currently we are keeping only average values of different type nodes and their instances. In the relational databases, histograms are used to keep more accurate information. Similar techniques can be used in case of graph databases. This aspect needs to be investigated to make this approach more robust as it would provide more accurate estimation of the cost of a plan.

Moreover, although the Subdue system developed for substructure discovery has been modified as a query processor and performs well; additional modifications may further improve the performance of QP-Subdue. The current system expands its instances by either one edge or one node and one edge. Clearly, for query processing, if a node has k edges associated with it, it will take k iterations to obtain k edge

substructure. Instead, it may be more useful to expand all the k edges at once in a single iteration. This will further reduce the intermediate number of substructure and therefore execution time for query processing.

In the current system, user has to input the query in the way a general graph is represented and the result is also returned in the same way. Instead it may be more intuitive if we develop a GUI for the same where user can formulate the query in its graphical representation by dragging and dropping the desired entities. In the GUI, user will be able to see the generated plan from the query and the results in more natural and cognitive way.

In conclusion, we believe the adaptation of query optimization techniques for graph databases are effective and opens up new possibilities and a research direction that is novel and different from contemporary techniques.

REFERENCES

[1] (2015) The DBLP website. [Online]. Available: http://dblp1.uni-trier.de/statistics/recordsindblp.html

[2] (2015) The FREEBASE website. [Online]. Available: www.freebase.com

[3] (2015) The IMDB website. [Online]. Available: http://www.imdb.com/stats

[4] (2015) The FREEBASE website. [Online]. Available: https://www.freebase.com/query

[5] S. Batra and C. Tyagi, "Comparative analysis of relational and graph databases," *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 2, no. 2, 2012.

[6] M. L. Rupley Jr, "Introduction to query processing and optimization," *Indiana University*, 2008.

[7] R. Giugno and D. Shasha, "Graphgrep: A fast and universal method for querying graphs." in *ICPR (2)*. IEEE Computer Society, pp. 112–115.

[8] X. Yan, P. S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," 2004.

[9] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad, "Fast best-effort pattern matching in large attributed graphs." in *KDD*, P. Berkhin, R. Caruana, and X. Wu, Eds. ACM, pp. 737–746.

[10] N. S. Ketkar, L. B. Holder, and D. J. Cook, "Subdue: Compression-based frequent pattern discovery in graph data," in *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*. ACM, 2005, pp. 71–76.

[11] M. Jarke and J. Koch, "Query optimization in database systems," *ACM Computing surveys (CsUR)*, vol. 16, no. 2, pp. 111–152, 1984.

[12] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 3rd ed. New York, NY, USA: McGraw-Hill, Inc., 2003.

BIOGRAPHICAL STATEMENT

Ankur Goyal was born in Bharatpur, India. He received his Bachelors Degree in Computer Science and Engineering from Rajasthan Technical University, India in June 2011. He attained his Master of Science in Computer Science and Engineering from the University of Texas at Arlington in December 2015. His research interests include graph mining, information retrieval and data mining.