PROCESSING QUERIES OVER PARTITIONED GRAPH DATABASES: AN APPROACH AND IT'S EVALUATION

by

JAY DILIPBHAI BODRA

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

May 2016

To the almighty God, my father Dilip and my mother Geeta

ACKNOWLEDGEMENTS

I express my sincere gratitude and thankfulness to Dr. Sharma Chakravarthy for his constant motivation, support and guidance throughout this research work. Without his persistent help and advice, this work would not have been complete. I would also like to thank Dr. Ramez Elmasri and Mr. David Levine for taking their time and serving on my committee.

I would like to thank the administrative staff, specifically 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 extremely grateful to Ankur Goyal, Soumvaya Das and Abhishek Santra for continually and convincingly conveying the spirit of adventure in regard to research.

My heartfelt thanks to my family for continuous support and inspiration. I would like to express my appreciation to my friends Manish kumar Annappa, Anisa Gilani, Nandan Prakash, Valay Shah and the ITLABians. I also greatly appreciate my numerous other friends for their love and support. I thank the Almighty for the infinite grace and benevolence.

April 21, 2016

ABSTRACT

PROCESSING QUERIES OVER PARTITIONED GRAPH DATABASES: AN APPROACH AND IT'S EVALUATION

JAY DILIPBHAI BODRA, M.S.

The University of Texas at Arlington, 2016

Supervising Professor: Dr. Sharma Chakravarthy

Representation of structured data using graphs is meaningful for applications such as road and social networks. With the increase in the size of graph databases, querying them to retrieve desired information poses challenges in terms of query representation and scalability. Independently, querying and graph partitioning have been researched in the literature. However, to the best of our knowledge, there is no effective scalable approach for querying graph databases using partitioning schemes. Also, it will be useful to analyze the quality of partitioning schemes from the query processing perspective.

In this thesis, we propose a divide and conquer approach to process queries over very large graph database using available partitioning schemes. We also identify a set of metrics to evaluate the effect of partitioning schemes on query processing. Querying over partitions requires handling answers that: i) are within the same partition, ii) span multiple partitions, and iii) requires the same partition to be used multiple times. Number of connected components in partitions and number of starting nodes of a plan in a partition may be useful for determining the starting partition and the sequence in which partitions need to be processed. Experiments on processing queries over three different graph databases (DBLP, IMDB, and Synthetic), partitioned using different partitioning schemes have been performed. Our experimental results show the correctness of the approach and provide some insights into the metrics gleaned from partitioning schemes on query processing. QP-Subdue a graph querying system developed at UTA, has been modified to process queries over partitions of a graph database.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

Today, massive amounts of data are being generated. HTTP links connect billions of documents on the Web. The Linked Open Data project has published more than 20 billion RDF triples [1]. Digital Bibliography and Library Project (DBLP) [2], which is a computer science bibliography, has around 2.8 million records of conference/workshop papers, and journal articles. Freebase [3], an online collection of structured data, harvested from various sources, has around 47 million topics and 2.7 billion facts. Internet Movie Database (IMDB) [4] contains information of 3.3 million titles and 6.6 million people about movies, TV-series etc.

Most of the above data have relationships that can be captured and represented in the form of a graph. Forcing a structure on the data may not only lose the intuitive understanding but also requires the design of a schema in order to store it in traditional Database Management Systems (DBMS). Freebase and others already exist as a graph and in order to query them in their native form, a query processing approach over graph databases is needed. Furthermore, if the size of the graph is massive, it cannot be loaded and stored as a single graph for processing queries. Given the growth in the sizes of data sets, the ability to process queries on partitioned graph databases is becoming important. Several, techniques and approaches for partitioning graphs already exist (developed for other purposes). However, processing queries over graphs have received attention only recently and to the best of our knowledge query processing on partitions have not been explored systematically. Since partitioning may affect query processing efficiency, there is a need for evaluating the partitioning schemes as related to query processing.

Answers can be obtained from this data by posing queries and evaluating them which can be useful for understanding and exploring the data sets. For example, people use Freebase to find information, such as “find Vietnamese restaurant in Palo Alto, California” [5]. As another example using DBLP which has the information of authors, journals, and conference paper queries can be posed to infer co-authors of a topic over specific time periods as well as publishing venues. If a research and development division wants to hire people who have published papers on a particular topic, querying the DBLP data will provide useful information. In addition, nowadays social networking has also become very popular which gives a platform to build social relationships 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 in these domains have explicit relationships (e.g., works for, founder of) unlike traditional applications such as payroll, airline reservation, etc. Despite the richness of data sets in the form of structural relationships, currently they cannot be queried the way user wants to. Queries are limited and decided by the vendors or corporate stakeholders. For example, DBLP provides an option to find papers of a particular author, but instead if a user is interested in finding a paper written in a certain period of time, then these types of queries are not supported. IMDB -, a movie database can be queried to find information about movies, genres, actors, etc. But again, complex queries such as “find movies where “ ‘person1’ and ‘person2’ has 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. QP-Subdue [6] developed at UT Arlington, supports expressive 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.

QP-Subdue [6] is a system which processes queries over databases expressed in the form of a graph, where an entity is represented as a node and a relationship between entities is represented as an edge. Nodes and edges can have labels. Labels are not assumed to be unique. Queries are smaller graph patterns(relative to the size of the database graph) given by a user (may contain conditions and wildcards) which the system accepts and retrieves patterns from the data graph that match the query. Typically, a query is in the form of a connected sub graph which is small as compared to the size of the data graph. The major drawback of this system is that it uses Subdue [7], which requires the entire graph to be loaded into main memory for computing answers. This is unrealistic if the graph size is large and available memory is not sufficient for holding the entire graph. We are interested in answering queries on any size graph and memory availability. This is possible if we can partition the graph based on memory availability and modify the query processing engine to work on partitions to compute answers. This work assumes and uses existing graph partitioning approaches as there is considerable work in the literature where the graph is partitioned into smaller parts in many ways by providing user requirements as parameters.

Partitioning a graph is an NP-complete problem. But, many heuristics-based algorithms have been developed to partition the graph to create reasonably good partitions. There are spectral partitioning methods [8, 9] which produce good quality of partitions but require intensive computations because of the eigenvectors.

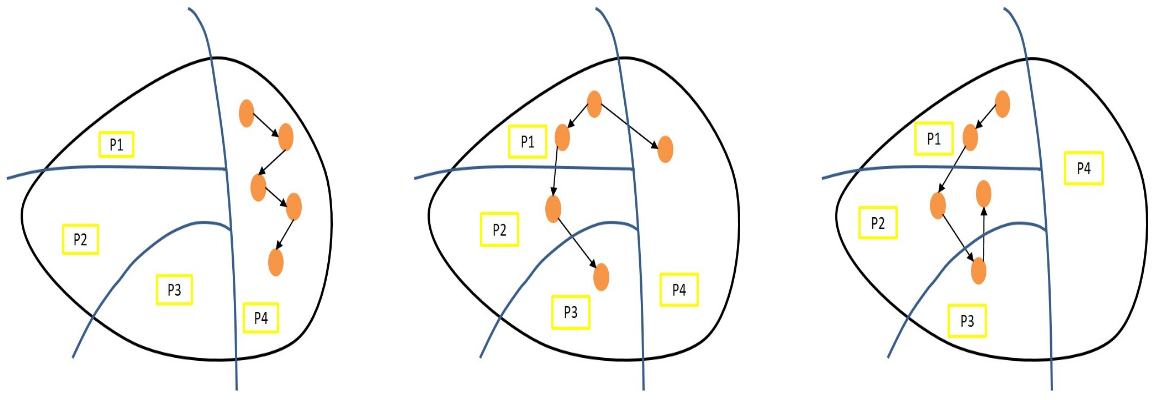
There are geometric partitioning techniques [10, 11] which require less amount of time for partitioning but the quality of partitioning produced is inferior. Due to the random nature of these algorithms they can end up taking multiple runs to find a good partitions. The other drawback of the geometric partitioning scheme is that it requires coordinate information (x,y or x,y,z) to partition the graph. If no coordinate information is available one cannot partition the graph.

Another class of partitioning techniques is termed multilevel graph partitioning [12, 8, 13, 14] where the graph size is reduced by coarsening. Coarsening is done by collapsing the vertices and edges and then partitioning the coarsened graph which is much smaller than the original graph, and finally uncoarsen the partitions to construct the partition of the original graph. This algorithm provides good quality partitions as compared to geometric partitioning schemes and has better time complexity as well. Few multilevel schemes such as METIS [12], KaHIP [14] have been discussed briefly in chapter 2.

Processing queries on a partitioned graph is very different as compared to processing a query on a single graph. When a graph is partitioned it will generate k partitioned small graphs (G_0,G_1,...,G_k) such that all the k small graphs can be combined to form the original graph. Evaluating a query on these partitions requires additional strategies for query processing based on the following observations:

1. Query answers are entirely in a single partition.
2. Query answers are entirely in multiple single partitions.
3. Query answers can span across multiple partitions.

Both case-2 and case-3 entail keeping track of partial answers and continuing them in relevant subgraphs in the same or different partitions. These evaluations may also require visiting the same partition more than once. This again needs to be handled by proper book keeping in order to ensure correctness of results.



(a) Answer within a partition

(b) Answer crossing multiple partitions

(c) Answer using a partition more than once

Figure 1.1: Query evaluation observation

Figure 1.1 graphically depicts different observation of query evaluation. Query answer generated within a partition is shown in Figure 1.1(a). This case needs information of partitions which contain the starting node label of the query to initiate the query computation. While figure 1.1(b) shows the case where a query answer crossing multiple partitions, figure 1.1(c) shows the cases where computation of a query for its answer(s) requires a partition to be loaded more than once. These cases are handled by storing intermediate information regarding the partition and node details in that partition as part of proper book keeping to compute complete and correct answers to a given query.

The efficiency of evaluating a query on partitioned graph database depends on two basic factors:− query access pattern and the quality of the partition. Assessing the quality of partition is important since a graph database can be partitioned in multiple ways using different partitioning schemes. In other words, when a graph database is partitioned using partitioning scheme PS_1 it generates an *edge-cut* set which is defined as number of edges whose incident vertices belong to different partitions. If the same graph database is repartitioned using the other partitioning scheme PS_2 it is possible that the size of the *edge-cut* set generated is smaller. If this process is repeated using k different partitioning schemes, k sets of partitions are generated for the same graph database. How a query expands in different partition schemes can be very different based on the connected components and edge-cuts in that partitioning scheme. We term this the query access pattern. Hence, query access pattern can differ based on the partition characteristics. In order to evaluate query access pattern cost, we consider a set of metrics to evaluate the partition quality for processing queries.

To the best of our knowledge, we are not aware of any work on processing queries over partitioned graph databases. Also, the choice of the starting partition as well as the sequence in which partitions need to be used for query processing is not straight forward. Whether any arbitrary query can be processed using each partition at most once is also not clear. As far as we know, partitioning schemes have not been evaluated for query processing requirements.

This thesis considers a set of metrics to evaluate partitioning schemes specifically for query processing. Also it proposes an approach for extending a query processor that works on a single graph to process the same queries on any number of partitions of that single graph. This is not straightforward as the answer instances are spread across partitions and one or more partitions have to be processed multiple times. This thesis extends and addresses processing of query on a partitioned graph an extension to QPSubdue [6]. The metrics are also analyzed empirically on couple of real world graphs.

The remainder of the thesis is organized as follows. Chapter 2 presents the related work in the areas of graph query processing and multilevel graph partitioning. Chapter 3 discusses an overview of graph mining and querying. Chapter 4 discusses graph representations and specification of query. Chapter 5 discusses query processing over partitioned graph, elaborated design for extending QP-Subdue and different metrics to evaluate the quality of partitions generated by different graph partitioning schemes. Chapter 6 details the query processor architecture and algorithms for all the metrics computation. Chapter 8 describes the implementation of all the metrics, processing of a query plan using modified QP-Subdue over a partitioned graph, experimental setup and results. Conclusions and future work are outlined in Chapter 9.

CHAPTER 2

RELATED WORK

Graph querying is useful for retrieving information from emerging graph databases such as Freebase. Querying a graph database is different from graph mining but useful for retrieving desired information based on the conditions we already know. For example, the query "find all founders of a company who attended Stanford or Harvard university" is more specific than mining where one is looking for patterns that may be of interest because it occurs frequently in the database. For querying a graph database one needs a query specification that is easy to understand and provide. As mentioned earlier, graph database sizes can be very large and hence systems that use main memory approach are not useful. To evaluate a query over any size of graph and memory availability, one feasible solution is to partition the graph database to make each partition fit in main memory. This is an alternative to disk-based approaches where the graph is indexed in some manner and stored on a disk and portions retrieved as appropriate during query processing. Query processing and graph partitioning have been well researched separately and a number of techniques have been proposed for each. This chapter briefly presents an overview of some of the widely used approaches for query processing and graph partitioning systems. Various techniques proposed for query processing are Graph-grap [15], G-index [16], G-ray [17], Subdue [7], and graph partitioning are Chaco [18, 8], METIS [12, 19, 20], KaHIP [21, 14, 22] and SEDGE [23]. The following subsections describe some of the query processing and graph partitioning approaches.

2.1 Query processing on Graph Database

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 researched to some extent. Graph querying is the process of finding exact/similar query patterns in the graph data. In this section we discuss about earlier and related work on graph querying (Graph-grep [15], G-Index [16], and G-Ray [17]).

2.1.1 Graph-grep

Shasha, Wang and Guino [15] propose an algorithm called Graph-grep which is a variable path index approach. This algorithm constructs an index of all possible paths up to length l, from all nodes and stores all possible paths in a hash table. When a query is submitted, the query graph is parsed to build its fingerprint (hashed set of paths). The indexed 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, it loses the structural information which makes it difficult for the chemical compounds that require structural information to be preserved. The proposed approach do not support queries containing comparison or logical operators and it evaluates the query graph from the main graph using indexing techniques. There is no distributed implementation of this approach. Also, there is no mechanism to handle multiple graphs (forest or partitions). In our approach, we use systematic graph expansion instead of indexing, support any graph database size using any partitioning technique and process queries on multiple graphs (or forest).

2.1.2 G-index

Another approach (G-index) proposed by Yan, Yu and Han [16] indexes frequent structures. In this approach the authors define the substructure to be frequent if its threshold is greater than the minimum support threshold (provided by the user) 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.

This approach does not answer infrequent queries because it only indexes the frequent substructures, and if graph is large then index size becomes large as well. Also there is no parallel implementation of the algorithm to handle graphs of any size. Our approach use graph expansion rather than an index. Also, our system is capable of processing any query whether frequent or not.

2.1.3 G-Ray

Another approach presented by Gallagher, Faloutsos and Eliasi-Rad [17] called G-Ray- finds both exact and inexact matches. This approach first ends 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.

This approach provides inexact matches of the query while on the other hand our system does not deal with inexact results. Also, these approaches do not handle large size graphs but our system can handle graph database of any size and generates exact result of a query.

2.2 Graph partitioning schemes

Graph partitioning is an important problem with extensive applications in many areas, including web graphs, social graphs, circuit placement, parallel computing and scientific simulation. The problem is to partition the vertices of a graph in p roughly equal parts, such that the number of edges connecting vertices in different parts is minimized. Many large scale graph partitioning tools are available. For this work, we use some of partitioning schemes for partitioning the graph on which queries are processed. This section provides a comprehensive overview of METIS, KaHIP and SEDGE.

2.2.1 METIS

George Karypis and Vipin Kumar presented METIS [12], one of the earliest graph partitioning schemes that uses multilevel algorithm proposed by Chaco [18, 8]. It supports different heuristics in each phase to partition the graph. It also presents a new variation of the Kernighan-Lin (KL) [24] algorithm for refining the partition during the uncoarsening phase. And the approach has a parallel implementation which reduces overall partitioning time.

A graph G can be bisected using a multilevel algorithm. The working of a multilevel algorithm is very simple, first the input graph G is coarsened to few hundred vertices, a bisection of this coarsest graph is computed and finally this partition is projected back to the original graph by refining the partitions to decrease the edge-cut. In the coarsening phase METIS provides four different heuristics- Random Matching (RM), Heavy Edge Matching (HEM), Light Edge Matching (LEM)and Heavy Clique Matching (HCM). For computing the initial partitioning, spectral partitioning algorithm [25] is used. And for the uncoarsening phase, to minimize the *edge-cut* set, Boundary Kernighan-Lin Refinement (BKLR) [26] is used.

METIS is a system that is widely used. It provides high quality partitions and supports many different heuristics to accommodate different types of graphs. But METIS consumes a fair amount of time in the coarsening and uncoarsening of the graph. Also, if the structure of the original graph changes frequently as in the case of web graphs and social network graphs where vertices and edges are added or deleted frequently, partitioning needs to be done again as METIS partitions graphs statically.

2.2.2 Karlsruhe High Quality Partitioning (KaHIP)

Another recently developed technique KaHIP presented by Peter Sanders and Christian Schulz [21], is a family of graph partitioning programs. It includes KaFFPa (Karlsruhe Fast Flow Partitioner), which is a multilevel graph partitioning algorithm. KaFFPa is a classical matching based graph partitioning algorithm with focus on local improvement methods and overall search strategies. It is a system that can be configured to either achieve the best known partitions for many standard benchmark instances or to be the fastest available system for large graphs while still improving partitioning quality [22]. There are three phases of KaFFPa- coarsening, initial partitioning and uncoarsening. KaFFPa employs the Global Path Algorithm (GPA) [27] as a matching algorithm to coarse the graph. In the initial partitioning phase, KaFFPa employs Scotch [13] as an initial partitioner since it empirically performs better than METIS [19]. KaFFPa also has its own initial partitioning algorithm which uses the multilevel recursive bisection scheme [28]. In the uncoarsening phase, KaFFPa iteratively uncontracts the matchings contracted during the contraction phase. After a matching is uncontracted, local search based refinement algorithms move nodes between block boundaries in order to reduce the *edge-cut* while maintaining the balancing constraint.

KaHIP looks at various local and global search techniques, different coarsening strategies as well as several meta-heuristics to tackle the graph partitioning problem. KaHIP provides very high quality partitions by using KaFFPa as the initial partitioner to make a good trade-off between quality and running time, or to be the fastest system on some graphs while still improving partitioning quality compared to the previous fastest system, METIS [12] and Scotch [13]. KaHIP implements novel local improvement schemes to fit most kind of graphs such as continental-sized road networks as well as large social networks and web graphs compared to METIS.

2.2.3 SEDGE

Unlike METIS and KaHIP, SEDGE (Self Evolving Distributed Graph Management Environment) [23] proposed by Shengqi Yang and Xifeng Yan, is not a static partitioning scheme but based on the workload it manages partitions in large graphs. It also tries to minimize the inter-machine communication during graph query processing such as breadth first search, random walk, and SPARQL queries across multiple machines. SEDGE is based on Pregel [29] and uses two level partitioning i.e. primary partitioning and secondary partitioning which are able to adapt to real time changes in query workload and uses a workload analyzing algorithm for workload change. Mainly SEDGE focuses on three types of query patterns:

1. Random query, a query that touches most part of the graph.
2. Internal query, the query that touches the node inside the partition.
3. Cross-partition query, the query that touches the node inside the partition as well as the nodes outside the partition.

The three techniques used by sedge to make it adapt to real-time changes in query workload are:

1. Complementary partition, it is a technique to find different partition schemes which have different edge cuts.
2. Partition replication, it is a technique to replicate the partition which has a high workload on a different machine to balance the workload of that partition.
3. Dynamic partitioning, it is a technique to construct a new partition to serve the cross partition queries locally using color blocks and envelope.

The Complementary partition is termed as primary partition while partition replication and dynamic partition together form the on demand partitioning which will generate secondary partitions.

Sedge uses partition management techniques to make new partitions which result in faster query processing. But it may require more space to store partitions due to creation of multiple complementary partitions. Our work is different for sedge in many ways. We do not create extra partitions. Neither do we keep track of the queries to replicate them into a new partition. Our system solves the queries on multiple partition and evaluates the partitioning scheme based on a set of metric discussed in a later chapter.

In this chapter, we have presented an overview of related work in the area of query processing and graph partitioning. The discussion on graph mining techniques and overview of graph query processing approach 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 data mining 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 entities), 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 [7], 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.

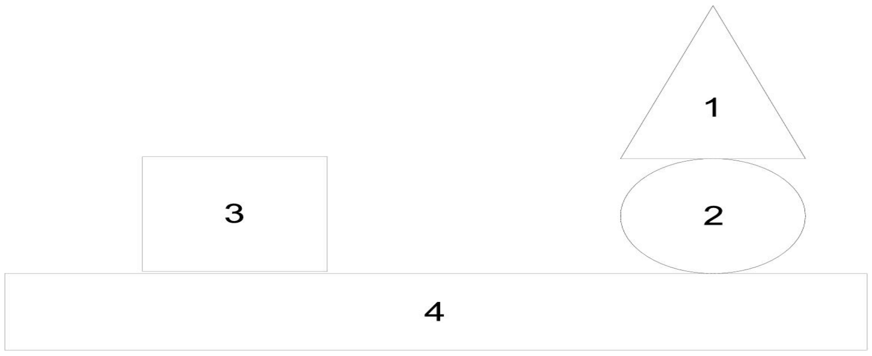


Figure 3.1: High-level view of shapes

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.

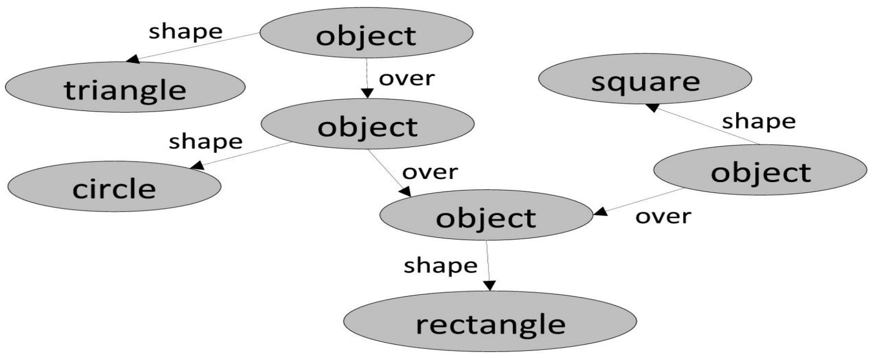
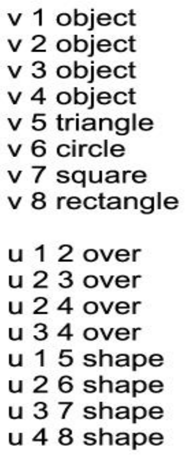


Figure 3.2: Graph representation of shapes example

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.

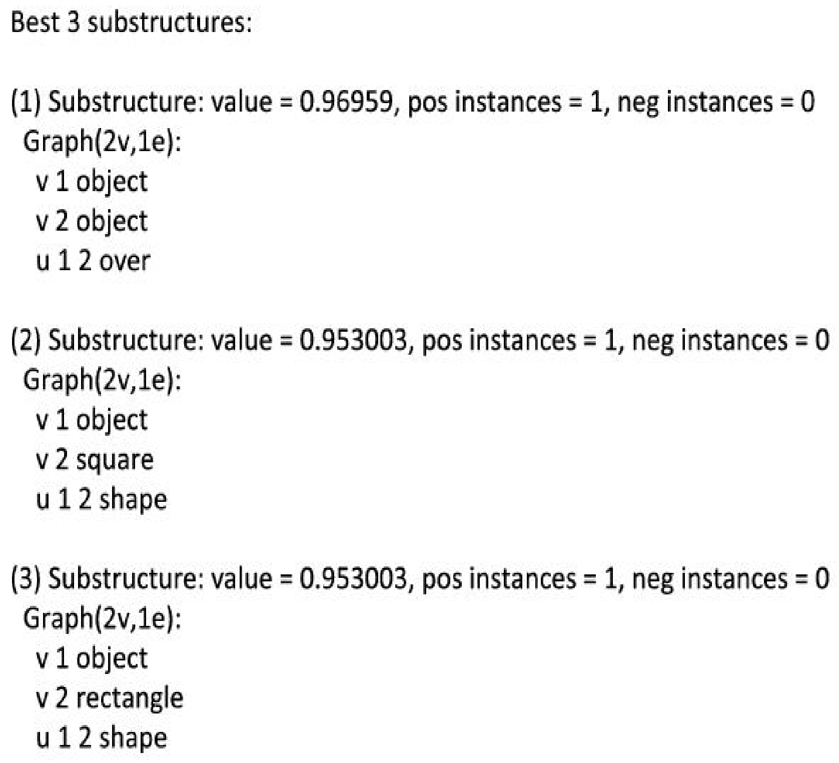


```
v 1 object
v 2 object
v 3 object
v 4 object
v 5 triangle
v 6 circle
v 7 square
v 8 rectangle

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. Th reshold specifies how a different one instance of a substructure can be from the other instance. The instances match if match-cost(sub, inst) $\leq$ 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 exact matches of a query pattern. In the following section we discuss the relationship between graph mining and graph querying and also discuss how a graph mining system can be modified 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, the goal is 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, for a mining system there is no input pattern. Hence, a typical mining algorithm starts from every node in the graph and expands them systematically by using some heuristics to prune the search space. For example, subdue uses MDL, G-ray uses goodness score for each vertex while Graph-grep and G-index uses hash based indexing techniques. However, in querying processing there is always an input pattern that we are trying to match in the larger graph. Hence, following the mining approach for query processing does not make sense.

As mentioned earlier, QP-Subdue is a query processing system developed recently by modifying Subdue, a graph mining system. QP-Subdue accepts an input query plan which restricts the selection of the start node followed by constrained expansion to desired nodes or next nodes in the plan. The intermediate substructures which match the query plan are stored for expansion in the next round. Constrained expansion is carried out for each edge in the plan. Also, QP-Subdue has developed a query optimizer along the lines of Relation Database Management Systems (RDBMS) query optimizers that generates alternate plans, evaluates them using a cost metric, and chooses the best one for execution. The quality of the plan is assessed using cost metric parameters such as cardinality, average connectivity, degree of a node etc. The metric used for determining the quality of a plan is the number of intermediate substructures generated during the evaluation of a query. The more the number of intermediate substructures, more effort is needed to evaluate a query which translates to a costly plan. The query processor computes and maintains a catalog that is similar to a RDMS catalog.

Since, QP-Subdue uses main memory to construct the graph, it can process queries on small size graph database restricted by available main memory. There is no distributed implementation of QP-Subdue to process large size graph as well as no provision to handle multiple graphs (partitions). Thus, we propose an approach which can process queries on any graph size by partitioning the graph database into smaller size which fits into main memory. Our work on processing queries over partitioned graph database requires handling of multiple cases as discussed in chapter 1. We modify existing QP-Subdue which enable switching of partition (multiple small graph) based on the intermediate results by doing proper book keeping which contains information about frontier nodes along with their partition id that needs to be resumed in another partition. The book keeping information helps in loading correct partition to evaluate a query over partitions of graph. Also, processing queries is a function of partition quality and query access pattern. To ensure the efficiency of query processing system over partitioned graph we also define metrics to justify which partitioning scheme better suits for a given set of queries. In upcoming chapters we explain our graph and query representation of our approach.

CHAPTER 4

GRAPH AND QUERY REPRESENTATION

4.1 Graph Representation

In this section, we discuss the representations used by Subdue (also QP-Subdue) for the graph databases. A graph consists of nodes and edges. A node typically has a node label, a unique node identifier(node id) and an edge connecting to the other node. An edge is a connection between two nodes which can be either labeled or unlabeled, directed, or undirected. In a general graph, loops, cycles, and multiple edges are allowed. Consider the graph shown in Figure 4.1.

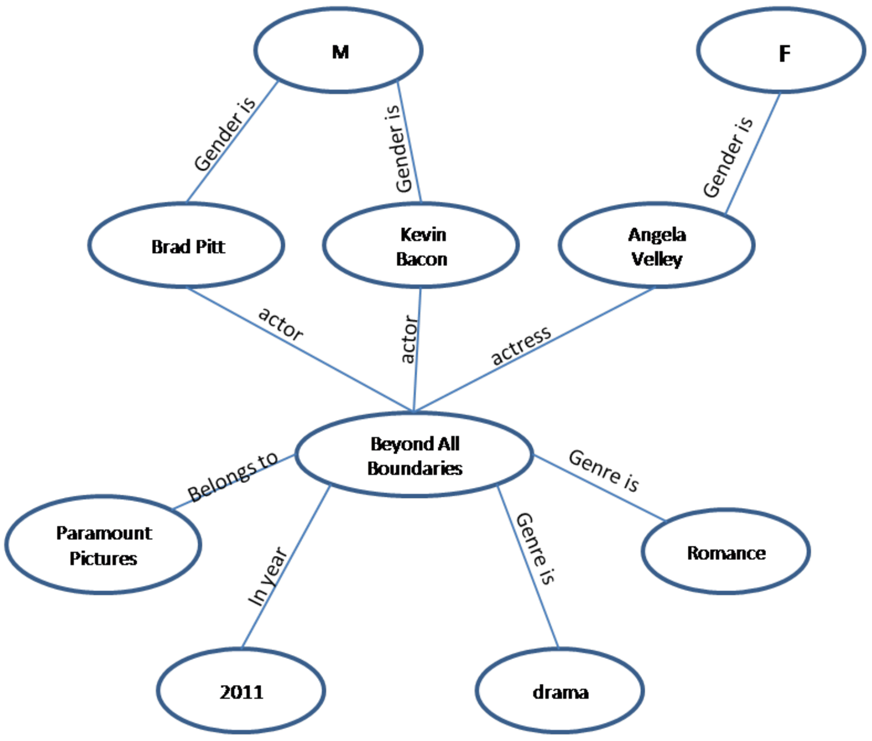


Figure 4.1: An example graph about movie information

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 should be able to query this graph. For example, “Find all actors in the movie ‘Beyond all boundaries’ and year of it production ” is a query on the above graph. This query will answer information about the actor and the year of production. Thus, all the nodes which are connected to the movie ‘Beyond all boundaries’ with the edge labeled as ‘actor’ or year, would be the answer(s).

Nowadays, graphs- IMDB, DBLP, Freebase, Knowledge graph exist with the property information along with the nodes and edges also termed as property graphs [30]. That is, a graph where the edges are labeled and both vertices and edges can have any number of properties associated with them. In other words, each node belongs to some category. Therefore, to group node labels with the same attribute or 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 an attribute name/type and instance nodes belong to a type node. 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.

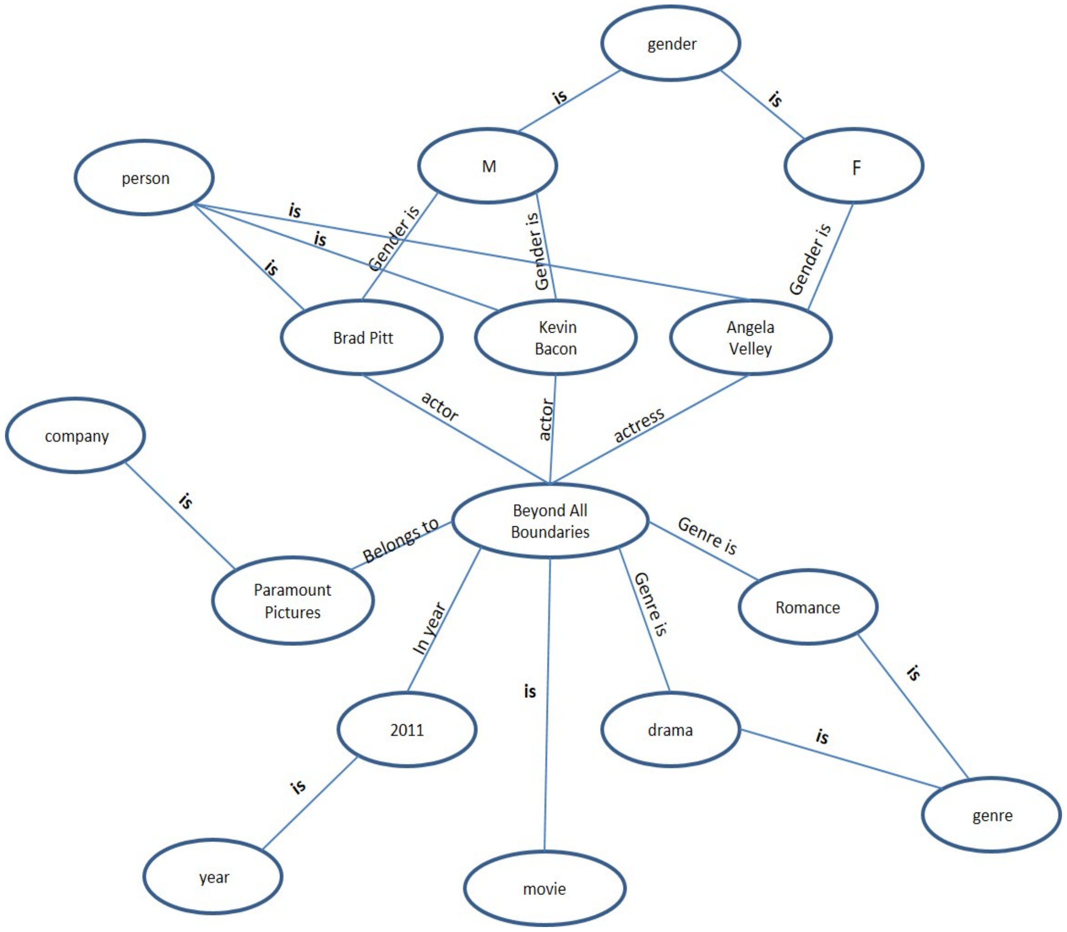


Figure 4.2: An example graph with type information

In Figure 4.2 six type nodes are shown. Number of type nodes is application dependent. This number is an indicator of how many different categories of the nodes exist in the graph data. 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.

4.1.1 Partitioned graph representation

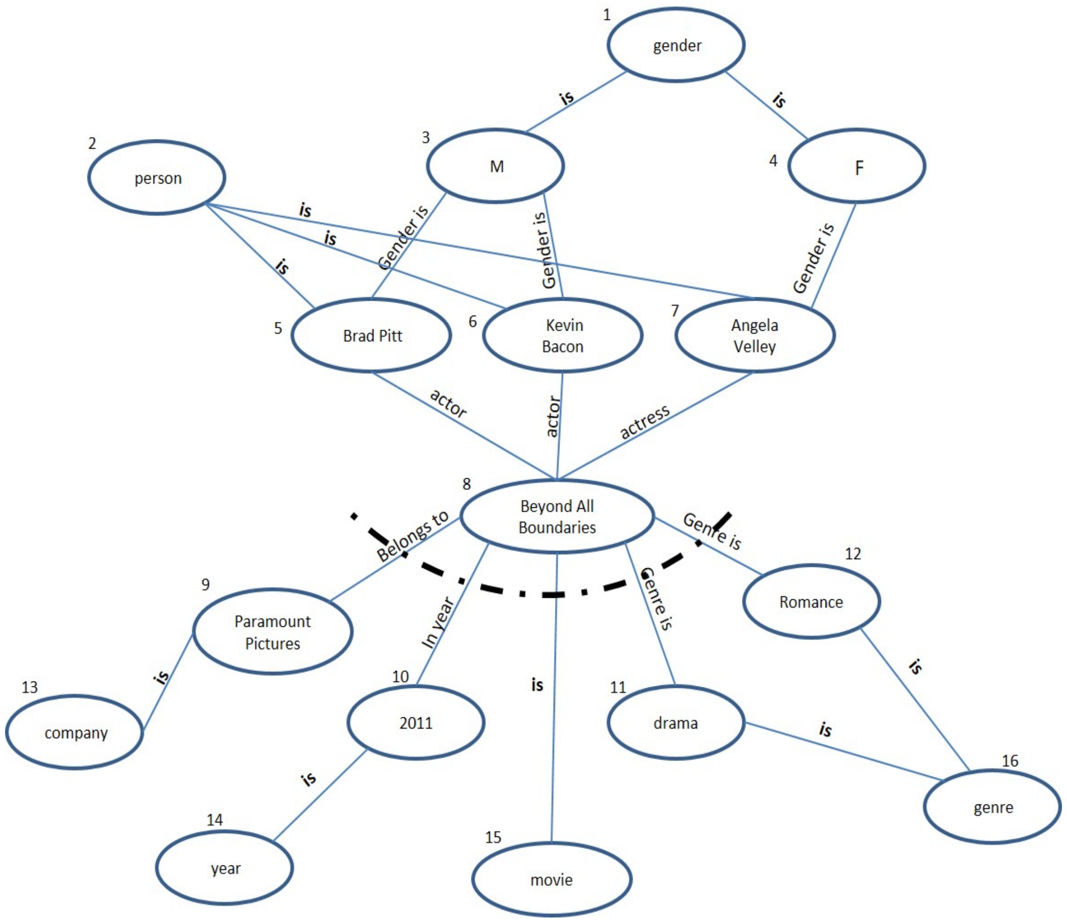
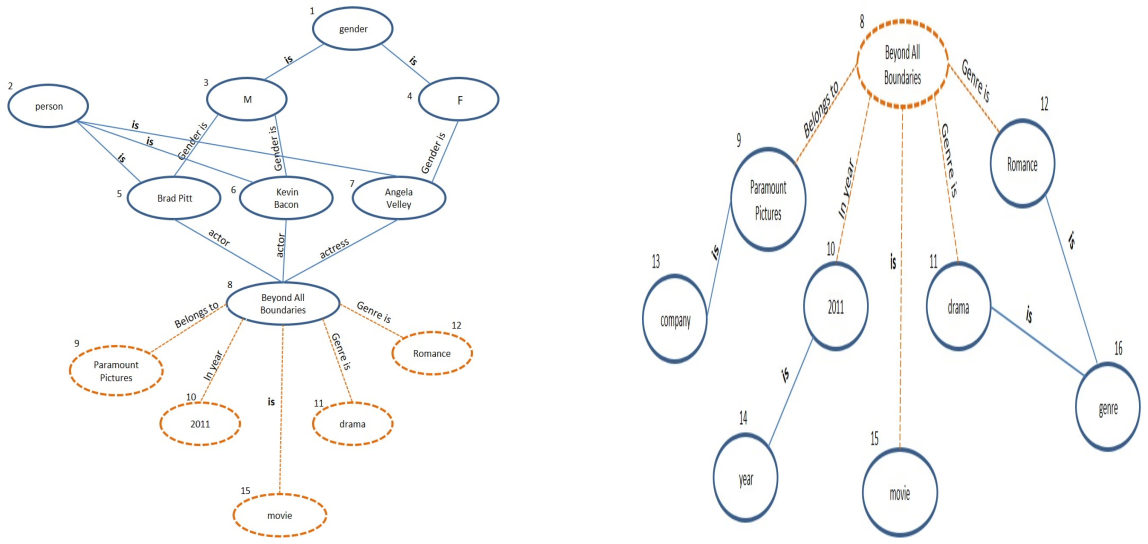


Figure 4.3: An example graph with two partitions

Generally, the size of graphs are massive which can easily overwhelm the main memory. Thus, in our approach we partition the graph. When a graph database is partitioned using any graph partitioning scheme such as METIS [19], KaHIP [14], Chaco [8]. *Edge-cuts* are generated which connects two different partitions. In other words, these edges connects nodes in one partition to a node in another partition. The graph in figure 4.2, when partitioned into two parts using METIS generates the graph shown in figure 4.3 where edges between the nodes ‘Beyond all boundaries’, 'Paramount', '2011', 'drama' and 'Romance' form the *edge-cut* set along with the vertex id.

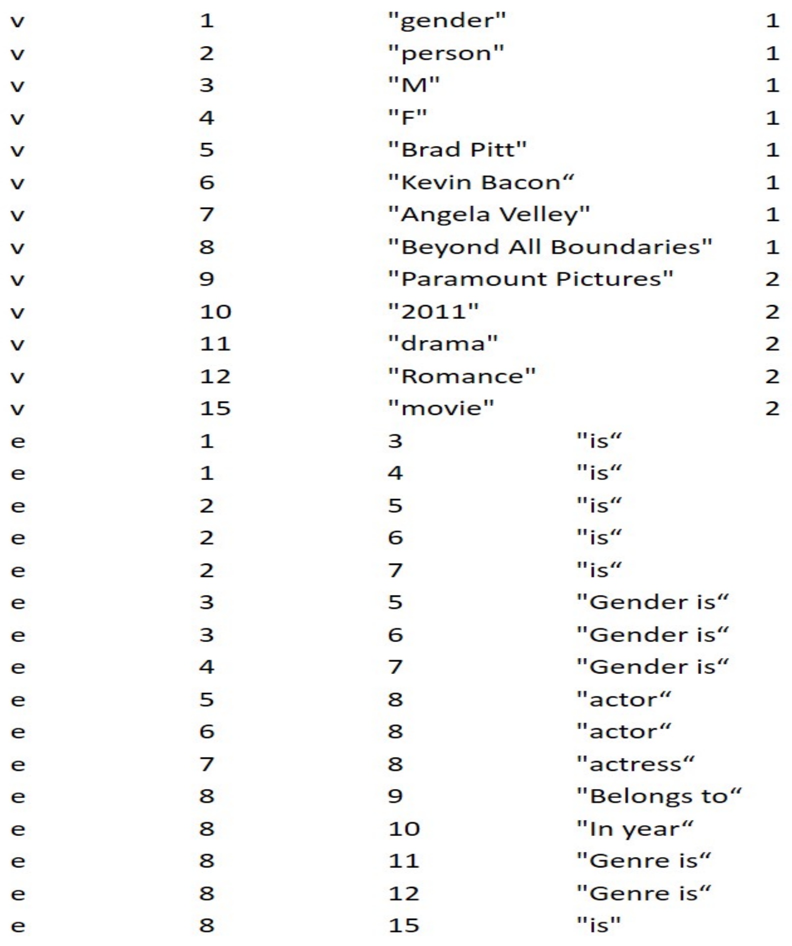


(a) Graph partition P_1 (b) Graph partition P_2

Figure 4.4: Partitions of graph 4.3

In our approach, replication of edge cut along with the node is used for (shown in Figure 4.4(a) and Figure 4.4(b). The two graph database partition P_1 and P_2 after partitioning along with the replicated edges is shown in Figure 4.4(a) and Figure 4.4(b) respectively. The dotted lines show the replicated edges along with nodes in each partition. For example, consider a query "Find actors in the movie 'Beyond all boundaries' in the year '2011' " and the staring node for this query is '2011'. The node '2011' belongs to partition P_2 which is initially loaded then it finds 'Beyond all boundaries' which is a replicated node and belongs to partition P_1. Node label, node id and partition id of node 'Beyond all boundaries' is stored as book-keeping information because the query is crossing the partition. Finally, the system loads partition

P_1 and searches for the all the actors. Details of query processing over partitioned graph is explained in the next chapter.



```
v     1     "gender"                   1
v     2     "person"                   1
v     3     "M"                        1
v     4     "F"                        1
v     5     "Brad Pitt"                1
v     6     "Kevin Bacon“              1
v     7     "Angela Velley"            1
v     8     "Beyond All Boundaries"    1
v     9     "Paramount Pictures"       2
v     10    "2011"                     2
v     11    "drama"                    2
v     12    "Romance"                  2
v     15    "movie"                    2
e     1     3          "is“
e     1     4          "is“
e     2     5          "is“
e     2     6          "is“
e     2     7          "is“
e     3     5          "Gender is“
e     3     6          "Gender is“
e     4     7          "Gender is“
e     5     8          "actor“
e     6     8          "actor“
e     7     8          "actress“
e     8     9          "Belongs to“
e     8     10         "In year“
e     8     11         "Genre is“
e     8     12         "Genre is“
e     8     15         "is"
```

Figure 4.5: Input to modified QP-Subdue of a graph Partition P_1

The input file to the modified QP-Subdue consists of vertex and edge entries corresponding to the graph. Each entry corresponding to a vertex is represented as unique vertex id with every vertex label and partition id. Each entry corresponding to an edge is represented as an undirected edge between a pair of vertices and the corresponding edge label. The input file to the modified QP-Subdue system corresponding to the representation in Figure 4.4(a) is shown in Figure 4.5

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

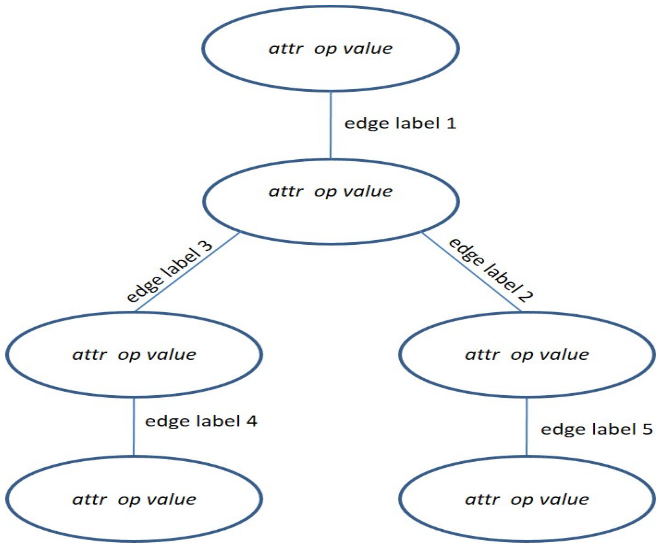


Figure 4.6: General query graph

Figure 4.6 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 the query representation used in our approach.

Consider a query containing combination of more than one logical operator and range operators- " Find movies with all the persons working as actors where genre should be romance **and** the production year should be **after** '2009'. Movie should **not belong** to the company 'American Broadcasting Company(ABC)' ". In this case each single query operator- "=", logical operator- "*AND, NOT*" and range operator- ">" is represented as

1. Single query operator: In the query, for each instance node only the equal operator present.
2. Logical operator: AND- This operator is used in queries where all the conditions have to be true. NOT- is used to get all the results except for a particular condition. Our approach also support OR- This operator is used if any one out of all given conditions is true.
3. Range operator: These operators(=<,>=,<,>) are used to get all the result(s) which qualify the given range.

and the result(s) would include the answers satisfying all the conditions specified in the query. Following is the representation of this query.

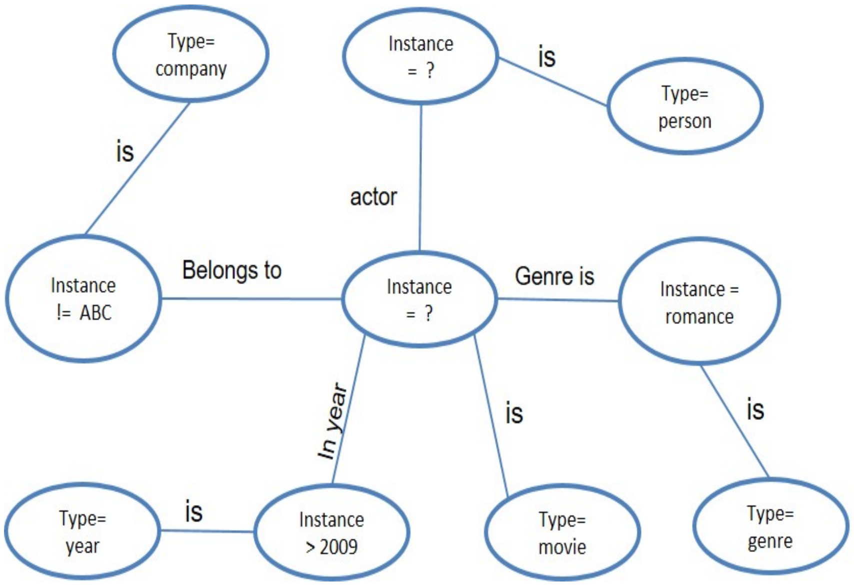


Figure 4.7: Representation of Query with Logical and Range Operator

Figure 4.7 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 graph and query representation. A graph query generally consists of a small number of nodes and edges compared 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 processing 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 comprehensive description of the query categories, in the next chapter we discuss about processing queries over a partitioned graph, system architecture of our approach, metrics to evaluate the quality of partitions generated by different graph partitioning schemes and algorithms for their computation.

CHAPTER 5

NEED FOR QUERYING PARTITIONED GRAPH

Recent work on querying a graph database (QP-Subdue [6]) generates a query plan using cost estimates that is appropriate for a graph database and uses a modified version of the Subdue [7] mining algorithm to evaluate a query plan. Subdue, a main memory mining algorithm has been modified minimally for evaluating a plan. The modifications use a constrained expansion (instead of the unconstrained expansion needed for mining) for a query plan which indicates the order in which nodes should be expanded to obtain answer instances that match the given query. QP-Subdue being a main memory algorithm loads the entire graph in main memory before processing queries. This makes sense for mining as substructures need to be generated starting from all nodes, some of the generated substructures pruned after applying a metric (such as MDL or frequency) and the process repeated again. However, for query evaluation the expansion is constrained and the nodes to be used for expansion are also known from the generated plan. Clearly, loading the entire graph into memory is not needed as only small portions of the graph need to be expanded to answer the query. Subdue approach makes it unrealistic for querying (or even mining) large graphs that do not fit in main memory. The question this thesis is addressing is "how to process queries if the graph database size is large and cannot be held in main memory still using the Subdue approach?".

Divide and conquer has been around for a long time as a potential solution to reduce a larger problem into a number of smaller problems and solve each one individually and combine the results to solve the larger problem. This has been successfully applied in sorting (e.g.,- quicksort [31], mergesort [32]) and also in processing large data sets. Map/reduce has scaled this approach to arbitrary number of partitions to deal with analytics needed for big data.

Partitioning of graphs has also been addressed in the literature to deal with limitation of main memory when dealing with large graphs. However, to the best of our knowledge, this has not been applied in the context of query processing which is the main focus of this thesis. For our work, we will use partitioning to reduce the size of the graph used for query processing on which we will use QP-Subdue for processing queries. Many partitioning schemes such as - METIS [12], KaHIP [21], Scotch [13] and Chaco [18] are available in the literature. Partitioned approach to query processing poses a number of issues that need to be addressed. The scope of this thesis is to identify these challenges and propose viable solutions. When a query is evaluated over a partitioned graph additional cases need to be handled properly. The problems that need to be handled are:

1. Dealing with many answer instances in a partition (similar to single graph approach)
2. Dealing with answer instances that span multiple partitions (not present in the previous approach QP-Subdue)
3. Dealing with answer instances that require the same partition to be used multiple times (not present in the previous approach QP-Subdue)
4. Processing of a query on a partition and continuing it in subsequent partitions (not handled by QP-Subdue [6])

In our approach, we handle all of the above mentioned cases. In addition, we also propose several metrics to understand the behavior and effect of partitioned approach on the cost of query processing. Case-1 above does not require any change except keeping track of the answers. Case-2 and 3 require that a query be processed over multiple partitions in a sequence and the results collected to form answers. This is done by storing some continuation information when a partition is used and use that for determining the next partition to be used until the query is completely processed. The results from multiple partitions need to be combined to generate query results. Finally, QP-Subdue has been modified to accept graph partitions along with minimal partition information to determine whether an answer continues in another partition and if so in which partition. This has to be determined at run time and may lead to inefficiency if the order if partitions are not chosen properly. For case-1, a query may have multiple starting nodes in different partitions. The number of starting nodes in each partition for a query may vary based on the partitioning scheme used. Also, if there is no start node in a partition that does not necessarily mean that partition is not required; an answer may grow into that partition which can only be figured out at runtime. Currently, the number of start nodes in each partition for a given query is computed at the end of the partitioning phase. For answers that are entirely contained in a single partition, there is not any difference between QP-Subdue and this approach except for handling the partition instead of the entire graph. In contrast, case-2 requires additional book keeping information to be maintained to resume the computation of an answer in a different partition. Note that many answer instances may grow into different partitions from a given partition. To handle this, each answer instance maintains the node to be resumed and the partition info of that node so that this answer instance can be resumed when the corresponding partition is loaded. Note that we load the partitions sequentially and the order of this loading is also important for efficient computation of answers as will be illustrated later in this thesis. Case-3, is a special case of case-2 in which the answer grows back into the partition which was used earlier. The book keeping information needed is that same as that of case-2. A query need to be processed using partitions as needed (even if it is the same partition a number of times) until there are no more answers. Note that intermediate result(s) generated in case-2 and 3 provide starting point(s) for continuing in the relevant partition, in addition to the starting points in that partition. The correctness of partitioned approach to query processing is to produce the same results as if the entire graph was processed as a single graph using the earlier approach.

In the presence of partitions, an additional decision is to determine whether we will be processing all or several partitions in parallel or we will load the partitions sequentially and process each one at a time. Processing partitions sequentially will of course increase the total response time but reduces the complexity of case 2 and 3 discussed above which will require inter-process communication, synchronization, and putting all answers together in some meaningful manner. In this thesis, we have chosen to process partitions sequentially to avoid inter-process communication and the additional complexity associated with synchronization, number of processors needed based on the number of partitions, or combination of both if one were to use less number of processes than the number of partitions.

A graph database can be partitioned in multiple ways using different partitioning schemes. A set of queries (or a query workload) can exhibit different costs in terms of total response time, number of partitions used for processing queries and the number of times a partition has to be used for processing the workload. Some of the above costs are affected by the choices made in terms of the starting partition for processing a query as well as the sequence of partitions. Ideally, the goal is to match a partitioning scheme with a workload for optimizing the cost of processing a query workload. In this thesis, we have proposed and analyzed the metrics for a single query instead of a workload. We have made some preliminary attempts in understanding the effect of some metrics on the cost of processing queries. To understand the effect of partitioning schemes in a meaningful way, we define a few intuitive metrics which we use to evaluate the partitioning schemes for processing queries.

5.1 Metrics for Evaluating Partitioning Schemes

With overview of issues and evaluation of queries in the presence of partitions, it would be useful to understand the effect of partitioning schemes, if any, on processing a single query or workload of queries. The questions to be answered is "is one partitioning scheme is better suited for a given workload than other partitioning schemes and if so, what aspects of partitioning can be used to determine this?" The general problem can be defined as follows. Given a workload (a set of queries) and a graph database, "how do we determine which partitioning scheme is good for that workload?". In other words, are there metrics that can be used to predict (or reason about) the total amount of work done in answering queries in the workload on that graph.

One of the widely used metrics for graph partitioning is the *edge-cut*. An edge cut is defined as the total number of edges that connect the original graph from a partition to any other partition. Partitioning schemes try to minimize the edge-cut to reduce inter-partition book keeping in the presence of partitions. For our problem, edge-cuts are the means by which an answer to a query can span multiple partitions. In general, if the number of edges in an edge-cut is small, the likelihood of an answer crossing to another partition from that partition is likely to be small as well (a single partition is an extreme case). Of course, this depends on the query mix and the characteristics of the *edge-cut* (e.g., labels).

In this thesis, we propose three metrics. Some of them can be computed after the partitioning the graph directly even before a query is evaluated where as others may have to be computed at runtime after partition is used for query evaluation.

For example, the number of connected components in a partition does not change and can be computed before starting query evaluation (static metric). This metric is based on the structure of partitions determined by the partitioning parameters used. Similarly, the number of starting nodes in each partition for a given plan can be initially computed prior to query evaluation. However, the number of continuations of answers from a partition to other partitions (both the number of partitions in which it continues and the number of continuations in each partition) can only be computed after a partition is used for query evaluation (hybrid metric as it is a combination of a static and a dynamic metric). The first metric, termed a static metric, is based on the partitions of the graph database and can be calculated before evaluating any query. The edge-cut is also a static metric as it can be computed after the partition phase and does not change. While the number of start nodes for a plan can be calculated statically, the continuation of answers can only be calculated after using each partition and hence it is termed a dynamic metric (runtime). The number of starting points in a partition after the first partition can be viewed as a hybrid metric as it is a combination of static and dynamic metrics. In the following section, we provide details of all the three metrics used to assess the quality of partitions generated by any partitioning scheme.

5.1.1 Number of starting nodes in each partition

For a partitioned approach to query evaluation, identifying the first partition for query evaluation can benefit from knowing the number of starting nodes in each partition. Obviously, if a partition does not have any starting nodes, it makes sense to use it later (or not at all) if other partitions grow answers into that partition. To determine the initial sequence (or order) of partitions to be used, a partial ordering of partitions can be beneficial. An initial partial order of partition sequence can be determined using the ascending (starting with non-zero), descending, or a random order of the number of starting nodes in partitions. After the first partition is used, this metric needs to be recalculated using the dynamic component and a new partial order is generated. This process will continue until the query is completely evaluated.

It is difficult to speculate or theoretically establish which property (max, min, or random) is better from the perspective of total number of partitions (including partitions that may have to be loaded multiple times) that need to be used for evaluating a query. The goal is find a metric or combination of metrics that can be used to minimize the number of partitions used for answering a query. This needs to be extended for a set of queries in which case an objective function need to be established. Our experimental evaluation using different properties may throw some light on this.

Once the initial partition to be used is identified, the system loads that partition to evaluate the query for its exact matches and continuations. The number of continuations are added to each partition to determine the next partition to be used. During the evaluation of a query using a partition, partial answers of the query as well as continuation node ID (if any) and their partition are written to a file to be used for determining the next partition. In each iteration this book keeping information is appended/updated and this updated book keeping information is used for the next iteration. We want to see whether a property such as max or min will perform better than the random property for determining the partial order. An explanation for selecting a partition having maximum starting nodes in each iteration is to get maximum number of complete result(s) as well as continuations from that partition. Similarly, min property may help accumulate continuations and process them later once. It would be useful if one can establish a property (one or a composition of metrics) that limits the number of a partition may have to be loaded to at most twice for processing any query or a query load.

Another application of this metric is to identify the distribution of vertex labels across all the partitions generated by different partitioning schemes. There are few partitioning schemes- hash based partitioning and random partitioning which creates partitions based on vertex labels. This approaches creates partitions containing higher number of homogeneous labels resulting in high edge-cut which increases communication latency amongst all the partitions [33, 34] and the generated partitions may not be balanced in terms of size. In a distributed environment for processing queries, homogeneous label distribution can result in severe performance issues as query having the same starting (seed) label will have to wait for the partition containing the starting node label. In other words, a query waits on the previous query for the partition.

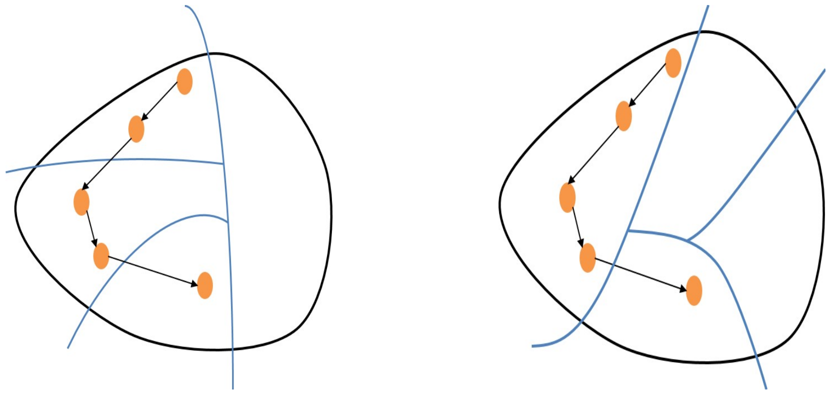
This metric may shed some light on the relative distribution of a node label among the partitions. However, it is not clear as to how it can be leveraged for query processing. Edge-cut information combined with start node information may be more useful in determining the number of continuation of answers from one partition to other partitions.

5.1.2 Number of partitions used to evaluate a query

It is clear that all partitions that have starting nodes of a query plan need to be used (or loaded) at least once for evaluating a query. However, the presence of partial answers in a partition and its continuation in a different partition makes query evaluation of partitioned graphs more challenging. This partial answer flow is affected by the partitioning scheme as well as the number of partitions. It is also affected by the number of connected components within a partition which is further discussed in the next subsection.

Given a partition of a graph and the query, there is no way to guarantee that each partition is needed at most once. In some cases, it may be possible to sequence the partitions to be executed to achieve this. Our goal is to accomplish this wherever possible by understanding and is possible establishing a relationship between query processing and the metrics defined in this section.

Figure 5.1 shows the same graph being partitioned using two different schemes PS_1 and PS_2. For simplicity, a single query answer is shown on these partitions. It is easy to infer that when the graph database is partitioned using different partitioning schemes (PS) the number of partition(s) needed (as well as their order) by the query can differ. For processing a query over graph database partitioned using partitioning scheme PS_1 in figure 5.1(a), three different partitions are needed to get the complete results. On the other hand, processing the same query over graph database partitioned using PS_2 in figure 5.1(b), only two partitions are needed to get all the results. Hence, the number of partitions used by a query changes based on the characteristics of partitioning schemes. This metric provides information on the number of partitions used by a query as well as number of times the same partition is needed for evaluating the query. The metric is calculated during query processing a runtime metric.



(a) Partition using Partitioning Scheme(PS_1)

(b) Partition using Partitioning Scheme(PS_2)

Figure 5.1: Different Partition Quality

Another information provided by this metric is, frequently used partitions to solve the query. Most of the time queries are related to important nodes in the graph. For example, in a social network graph it is often seen that one particular user is followed by a large number of people, making the user an important node and there can be many such important nodes in a graph. But when the graph is partitioned multiple important nodes can belong to same partition. If these important nodes are trending most of the queries will consist of these nodes resulting in frequent usage of the partition to answer the query. Based on this metric, in a distributed environment replication of the whole partition on a different machine can be done to efficiently handle the query workload of a frequently used partition.

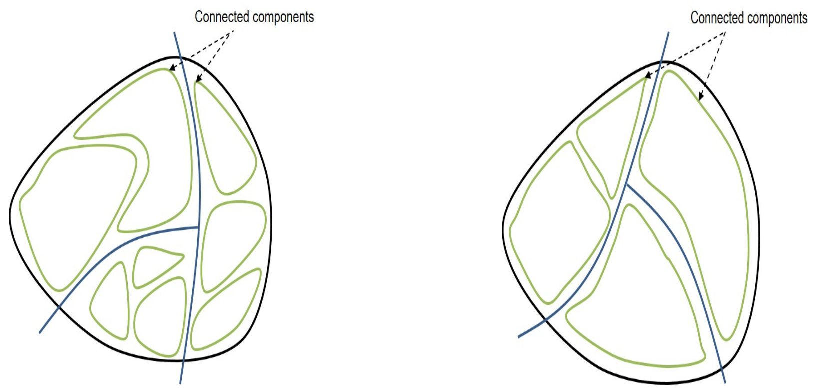
5.1.3 Number of connected components in each partition

This metric is useful to determine the extent to which a partitioning scheme preserves the structure of a graph after it is partitioned into multiple smaller graphs.

A graph before partitioning can be viewed as one single clique of connected nodes. When the graph is partitioned,

1. Each partition can be a completely connected graph.
2. Each partition can contain multiple connected components. In the worst case, the number of connected components can be equal to the number of nodes in that partition.

Figure 5.2 depicts a graph database partitioned using two different partitioning schemes. In figure 5.2(a) the partitioned graph database using partitioning scheme(PS_1) contains multiple connected components in each partition. On the other hand, in figure 5.2(b) the partitions generated by another partitioning scheme(PS_2) has one connected components in two partitions and multiple connected components in one partition.



(a) Partitions using Partitioning Scheme(PS_1)

(b) Partitions using Partitioning Scheme(PS_2)

Figure 5.2: Partitions of a graph database using two different partitioning schemes

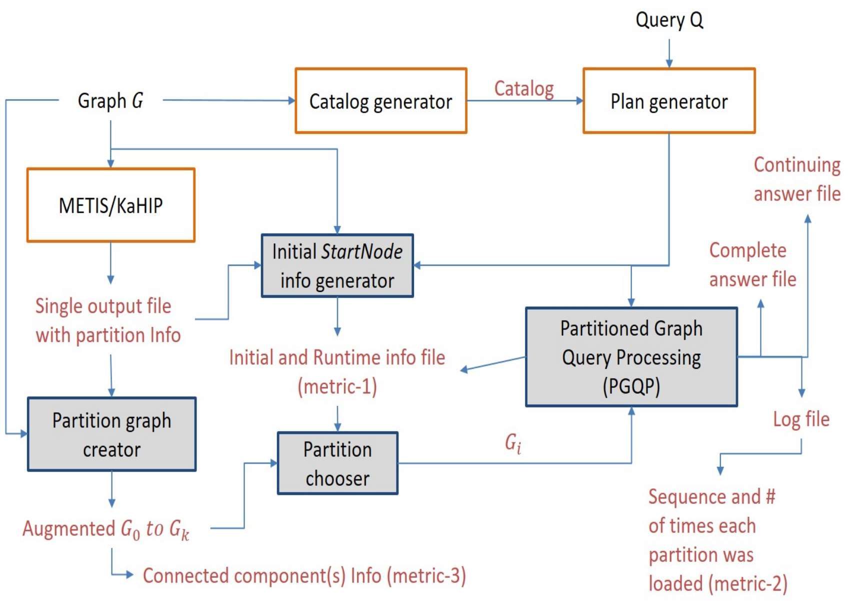
Number of connected components in a partition can adversely affect the total number of partitions needed (or need to be loaded) to answer a query. For example, if an answer were to span more than one connected component within a partition, it has to happen through one or more different partitions. This indicates that using a partitioning scheme that produces less number of connected components in each partition is better than a partitioning scheme that generates more connected components in each partition. This metric can be used to assess the quality of a partitioning scheme. The *edge-cut* discussed earlier can be used in conjunction with this to assess the quality of a partitioning scheme.

Having provided an overview of various problems in processing queries over a partitioned graph database and metrics to assess the quality of partitioning and query evaluation, the next chapter we elaborates on the design along with the algorithms for our approach to process queries over a partitioned graph and evaluation of metrics.

CHAPTER 6

DESIGN ISSUES AND ALGORITHMS

The partitioned graph query processor (PGQP) developed by modifying QP-Subdue as a part of this thesis, aims at processing queries over a partitioned graph database. It uses the book keeping technique and stores partial and complete results generated during query evaluation. In this chapter, we discuss working of our system and algorithms. Following is the architecture of our query processor for partitioned graph databases.



Query Q
Graph G
Catalog generator
Catalog
Plan generator
METIS/KaHIP
Initial *StartNode* info generator
Continuing answer file
Complete answer file
Single output file with partition Info
Partitioned Graph Query Processing (PGQP)
Initial and Runtime info file (metric-1)
Partition graph creator
Partition chooser
G_i
Log file
Augmented G_0 to G_k
Sequence and # of times each partition was loaded (metric-2)
Connected component(s) Info (metric-3)

Figure 6.1: System architecture

As shown in the Figure 6.1, catalog is generated from the graph G (input graph) which is input to the plan generator. A query plan is generated using the catalog and input query. Catalog generator and plan generator were developed as a part of QP-Subdue [6]. Partitions of the graph are generated using a partitioning scheme (METIS, KaHIP) which outputs a single file (partition file) containing information about the vertex id and partition id. This file is further processed by partition creator to generate partitions with the representation described in chapter 4.1. The partition files with PGQP representation is used to calculated number of connected components in each partition. Query plan, graph G and partition file are input to the initial starting partition and outputs start node info (SNI) file which contains information about partition id, occurrence, start label and vertex id of the start node of the query plan. After start node info file is generated, partition chooser selects a partition to initiate query evaluation. Once a partition is selected, PGQP takes query plan, partition of a graph database and start node info file to process the query on the graph database and outputs complete and continuing answer file. PGQP can also generate new start node info file which is interpreted by the partition chooser to select a partition to be loaded in next iteration. This is continued until no more answers need to be processed. At the end, log file is processed to calculate metric for computing Metric-2 (number of times and sequence of partition loading). In the following section we discuss details of each component described in Figure 6.1 along with the algorithm.

6.1 Partitioning Graph Database

The goal of graph partitioning scheme is to divide the graph database in to k smaller parts such that it minimizes the edge-cuts set while preserving the structure in the partitions. Many large scale graph partitioning tools are available. In our approach, to process queries over any size of graph database we use existing graph partitioning schemes such as- METIS and KaHIP. A given graph G is partitioned in to k partitions such that each partition fits into the available main memory. Consider the graph in Figure 6.2 which is partitioned using METIS [12] into two different partitions.

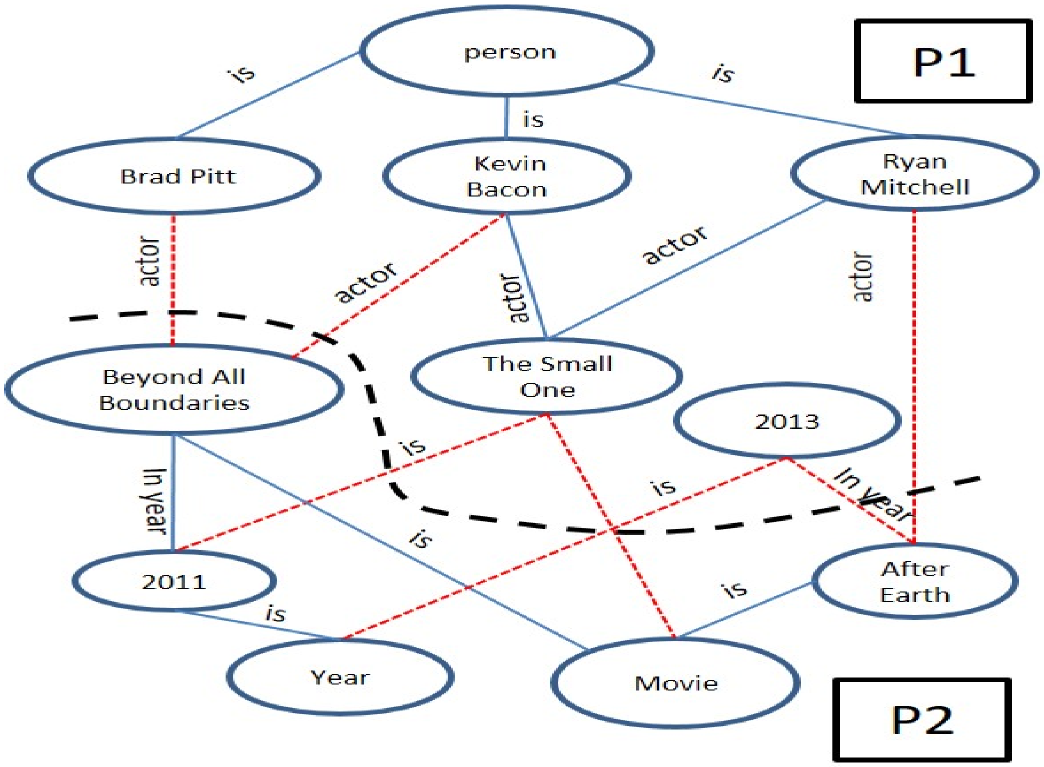
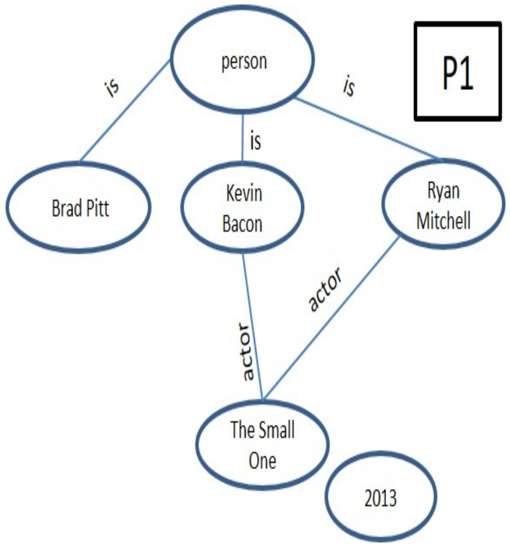


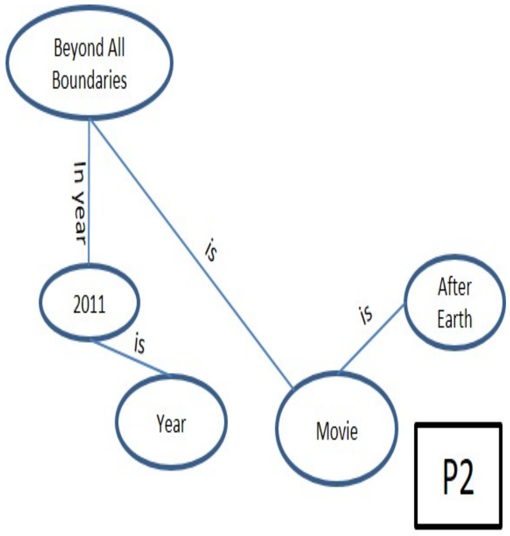
Figure 6.2: Partitioned graph database using METIS

The figure above shows two partitions of the graph database. The dotted edges form the *edge-cut* set. The partitions are shown in figure 6.3 which are generated by METIS. KaHIP generated similar partitions but the format is same. In our approach, the partitions are not used in the way they were generated. But we pre-process this partitions and replication of edge cut along with the node is done which helps in effective query processing for queries spanning multiple partitions. It is also clear that the structure of the original graph changes which can affect query processing.

In partition P_1 there are two different components while in partition P_2 there is only one connected component.



(a) Graph partition P_1



(b) Graph partition P_2

Figure 6.3: Two partitions of graph database 6.2

6.2 Connected components

As discussed earlier, in this work we use a metric for finding connected components in every partition of graph database. As shown in the figure 6.1, this metric is calculated separately (offline). In other word, this metric does not include any query evaluation information. It only requires partitions of the graph database in PGQP representation. This metric can be calculated after query processing or before query processing but not for all the queries. Hence, if the partitioning scheme is changed or graph is repartitioned this metric needs to be recalculated for accurate information. In our approach, we use sets to find connected components. The algorithm for calculating connected components is discussed in Algorithm 1.

Algorithm 1 Connected Components Algorithm

```
1: Input: PGQP Format Partition Files for each partition i (i = 0 to k)
2: Output: # of connected components in each partition
3: initialize totalSets=0, Map<vertexId, setId> M = null, vertexList = null
4: for each edge e in partition i do
5:     Get the source vertex id(svid) and destination vertex id(dvid)
6:     if findSet(svid) = -1 and findSet(dvid) = -1 then
7:         Increase totalSets by 1
8:         M ← add svid and dvid with value = totalSets and update vertexList[findSet(svid)] ← add svid and dvid
9:     end if
10:    if findSet(svid) ≠ -1 and findSet(dvid) = -1 then
11:        M ← add dvid with value = findSet(svid) , vertexIdList[findSet(svid)] ← add svid and dvid
12:    end if
13:    if findSet(svid) = -1 and findSet(dvid) ≠ -1 then
14:        M ← add svid with value = findSet(dvid) , vertexIdList[findSet(dvid)] ← add svid and dvid
15:    end if
16:    if findSet(svid) ≠ -1 and findSet(dvid) ≠ -1 then
17:        if findSet(svid) = findSet(dvid) then
18:            vertexIdList[findSet(svid)] ← add svid and dvid
19:        else
20:            Merge(findSet(svid), findSet(dvid)); totalSets = totalSets - 1
21:        end if
22:    end if
```

```
end for
function findSet(vertexId)
    for each vId in M do
        if vId = vertexId then
            return SetId
        end if
    end for
    return $-1$
end function
function Merge(set1 , set2)
    if vertexList[set1].size < vertexList[set2]. size then
        vertexList[set1] $\leftarrow$ add vertexList[set2]
        for each vid in vertexList[set2] do
            update M $\leftarrow$ vid with value=set1
            vertexList[set2] = null
        end for
    else
        vertexList[set2] $\leftarrow$ add vertexList[set1]
        for each vid in vertexList[set1] do
            update M $\leftarrow$ vid with value=set2
            vertexList[set1] = null
        end for
    end if
    return
end function
```

The algorithm for connected components takes the PGQP format partition file (each partition separately) as input and outputs number of connected components in each partition (line 1 to line 2). The algorithm uses totalSets to count number of connected components, map M which maintains vertex id and set to which it belongs and vertexList array for tracking of all the vertex in i^{th} set (line 3). For the source and destination vertex in the edge, algorithm finds the set to which it belongs and assign appropriate set, merging of set can occur if both the vertex id belongs to different set (line 6 to line 22). The algorithm continues for all the edges in partition file (line 4 to line 23).

6.3 Initial Starting Partition

For a given query Q, a plan is generated using the query optimizer (catalog generator and plan generator) developed in QP-Subdue [6]. In other words, query plan(QP) is a sequence in which the query evaluation must be done. The metric used for determining the quality of a plan is the number of intermediate substructures generated during the evaluation of a query. The more the number of intermediate substructure, more effort is needed to evaluate a query which translates to a costly plan. Figure 6.4 shows the graphical representation of the query "Find persons who have worked as actors in the movie Beyond all boundaries in 2011" and figure 6.5 shows the query plan.

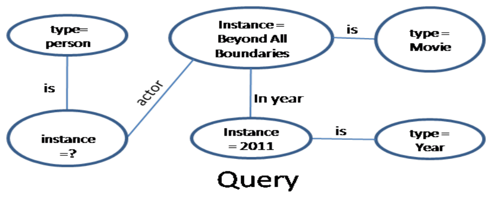


Figure 6.4: Graphical representation of query

“2011”	“year”	“=”	“is”	
“2011”	“Beyond all boundaries”		“=”	“in year”
“Beyond all boundaries”		“movie”	“=”	“is”
“Beyond all boundaries”		“?”	“=”	“actor”
“?”	“person”	“=”	“is”	

Figure 6.5: Input query plan to PGQP

For a given query Q and partitions (G_0,G_1,...,G_k) of graph database, the start node label of query can be present in single partition or multiple partitions. The partitioned graph query processor (PGQP) requires to load a relevant partition in order to evaluate the query for its exact results. The first step for evaluating a query on partitions of the graph database is to identify all partitions containing the starting label of the query plan. The algorithm for finding the initial starting partition (ISP) is discussed in Algorithm 2 which generates a Start Node Info (SNI) file shown in Figure 6.6(a) for the query plan of Figure 6.5. During evaluation of the query, if another partition is required for further computation the partitioned graph query processor writes the continuing start label (frontier node) to a new start node info file (book keeping) as the starting node(s) in that partition shown in Figure 6.6(b). In our approach, if there are multiple partitions containing the starting label of the query, selection of partition is done based on the number of occurrence of starting nodes. In other words, a partition can be selected based on maximum or minimum number of occurrence of start nodes.

partitionId	occurrence	startlabel	vertexid	edgelabel
2	1	2011	11	null

(a) Computed starting node by metric-1 (ISP)

partitionId	occurrence	startlabel	vertexid	edgelabel
1	null	Brad Pitt	4	null
1	null	Kevin Bacon	5	null

(b) Continuation info written by PGQP

Figure 6.6: Start Node Info File (Book Keeping)

Algorithm 2 Initial Starting Partition Algorithm

```
1:  Input: Subdue Format Graph (Input Graph $G$), Partition File (Partitioning
    Scheme Output), Query Plan
2:  Output: Start Node Info (SNI) File
3:  Get the starting label $S$ from the query plan
4:  Create a Map<Partition #, Count> M ← null
5:  for each vertex $v$ in the subdue format graph do
6:      Match the vertex label $VL$ with $S$
7:      if $VL$ matches get the vertexId(vid) then
8:          Get the partition id from Partition File
9:          if partition # exist in M  then
10:             count++
11:         else
12:             Add the partition # and initialize count ← 1
13:         end if
14:     end if
15: end for
16: for each partition # in M do
17:     Write out partition Id, count, label, vid ← null and edgeLabel ← null to
    Start_Node_Info                                                ▷ SNI File
18: end for
```

The algorithm for initial starting partition takes subdue format graph (input graph G), partition file (partitioning scheme output) generated by METIS or KaHIP and query plan as input and outputs start node info file which consists of information about partition(s) containing the starting node label of the query plan (line 1 to line 2). The algorithm starts by getting the starting node label from the given query plan and creates a map which maintains the partition id and the occurrence of the starting node label in that partition (line 3 to line 4). The algorithm now matches the starting node label of the query plan with the label of each vertex in the original graph (line 6). For every matching label, the algorithm takes the vertex id and gets the information about partition id from Partition File (Partitioning Scheme Output file) (line 8). The occurrence for corresponding partition id in the map is updated accordingly (line 9 to line 14). Algorithm continues till all the vertices of the subdue format graph are visited at least once (line 5 to line 15).

6.4 Partition Chooser

Once the initial starting partition(s) are identified, the partition chooser selects a partition which has maximum or minimum number of starting nodes. In each iteration the partition chooser maintains the sequence in which the partition(s) are loaded to evaluate the query for its complete answers. Also, partition chooser is responsible for making calls to partitioned graph query processor (PGQP) and providing relevant information about the graph partition, query plan and the start node info file which contains information of all the nodes starting in that partition. In every iteration, PGQP can generate new start node info file which contains the starting nodes for intermediate results. Hence, in each iteration the partition chooser computes for partition containing maximum or minimum occurrence of starting nodes.

Since, the partition chooser computes the sequence in which partitions should be loaded during evaluation of query. Partition usage information is generated as a by product. This information need simple processing for counting which partition was used how many times. The sequence provides information about the way in which partition(s) were loaded. In section 5.1.2, importance of partition usage information has been discussed. The algorithm for partition chooser is discussed below.

Algorithm 3 Partition Chooser algorithm

```
1: Input: Start Node Info (SNI) File, Query Graph, Selection
2: Output: Query Results
3: initialize currentSnifile with sniFile
4: while true do
5:     if currentSniFile $\neq$ null then
6:         if Selection = max then
7:             pid $\leftarrow$ PartitionSelection(currentSnifile, Selection)
8:         else
9:             pid $\leftarrow$ PartitionSelection(currentSnifile, Selection)
10:        end if
11:        PGQP(Graph_pid, query graph, currentSniFile, newSniFile)
12:        currentSniFile $\leftarrow$ newSniFile
13:    else
14:        break
15:    end if
16: end while
17: function PartitionSelection(sniFile , selectBy)
18:    create a map<PartitionId, Count> M $\leftarrow$ null
19:    while EOF do
20:        get the partitionId and occurrence from sniFile
21:        if M contains partitionId then
22:            get the Count
```

```
23:            if occurrence $\neq$ null then
24:                Count $\leftarrow$ Count + occurrence
25:            else
26:                Count $\leftarrow$ Count++
27:            end if
28:        end if
29:    end while
30:    inititalize i $leftarrow$ 0 and partitionArray $\leftarrow$ null
31:    if selectBy = random then
32:        for each partitionId in M do
33:            initialize the partitionArray[i++] $\leftarrow$ partitionId
34:        end for
35:        randomPartition $\leftarrow$ random.nextInt(partitionArray.length-1)
36:        partitionId $\leftarrow$ partitionArray[randomPartition]
37:    else
38:        for each entry in M do
39:            get the partitionId containing count based on selectBy
40:        end for
41:    end if
42:    return partitionId
43: end function
```

Above is the algorithm for the Partition Chooser. This algorithm takes the Start Node Info (SNI) file, query graph and selection (min or max) as input and generates query result(s) as an output (line 1 to line 2). The sequence of loading a partition can be decided based on partition containing maximum or minimum number of starting nodes (line 6 to line 10). Partition chooser calls the Partitioned Graph Query Processor (PGQP) with relevant partition of the graph database, SNI file and query graph and the new SNI file which is empty and acts as book-keeping for next iteration (line 11). After each call to PGQP, a new SNI file may be generated which is used the partition chooser to select a relevant partition for processing query in the next iteration (line 12). Algorithm continues until the new SNI file is empty which means no answers needs to be processed (line 4 to line 16).

6.5 Partitioned Graph Query Processor (PGQP)

In this section, we discuss the modifications in the QP-Subdue to enable it to process queries over partitions of a graph database. As discussed earlier in previous chapters that QP-Subdue is a graph querying system which takes the graph database and a query plan as an input to find all the exact matches of the query. For processing queries over partitions of graph database, the system needs to keep track of partition id of the vertex. This information helps the system to track continuing answers (intermediate result) in other partitions as well as storing the frontier vertex along with its vertex id and label in the start node info file as book keeping information. Since, we process the query by sequentially loading a partition in each iteration a new start node info file is written. Another modification we did to QP-Subdue is, creation of substructures from the start node info file. When a graph partition is loaded, substructures are created from a frontier node of start node info file only if the partition id of the loaded partition and partition id of the frontier node is same otherwise the frontier node is written to a new start node info file.

Processing queries over partitioned graph database is different from single graph database. In our approach, to process queries over partitioned graph database, the system needs a relevant partition to compute the query results and query plan which restricts the expansion to the particular node with specified edge label. We also support multiple operators <, <=, >, >=, ! =, and = in a query. 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. Also if the expanded node meets the criteria but belongs to different partition, the intermediate result is stored and book-keeping is done in the form of new start node info file. We continue until all the nodes and edges in the plan get covered. Following is the algorithm for partitioned graph query processor (PGQP).

Algorithm 4 Partitioned Graph Query Processor (PGQP) algorithm

```
1: Input: Partition of graph database, Query graph, Current Start Node Info (SNI)
   File, New Start Node Info File
2: Output: Complete and Continuing Answer File, New Start Node Info File
3: Get the currentPartitionId from currently loaded graph partition
4: if partitionId of starting node label of the currentSniFile = currentPartitionId
   then
5:     Create substructure for starting node label of the order file
6: else
7:     Create a new start node info file(book-keep) and write the label
8: end if
9: while all the nodes and edges in the plan get visited do
10:    if partitionId of expanded node matches currentPartitionId then
11:        Expand the current node of substructure to the desired node with specific
   edge based on the plan
```

```
12:     else
13:         write out partitionId, label, vertexId and edgeLabel of expanded node to
        new start node info file; intermediateList ← add newly expanded information
14:     end if
15: end while
16: Write out complete results and intermediate results; Done
```

The algorithm for partitioned graph query processor (PGQP) takes a partition of the graph database, query graph and start node info file as input and outputs all the exact matches which can be complete and/or intermediate of the query, new start node info file containing the starting nodes for continuing (intermediate) results in relevant partition (line 1 to line 2). Each plan is a sequence of nodes to be traversed. This algorithm starts by getting the partition id of the currently loaded graph partition from its file name (line 3). In the next step, substructures are created for the starting node label of the start node info file present in currently loaded partition of the graph database (line 4). The other starting points not relevant to the currently loaded partition are written out to new start node info file (line 7). The substructures are expanded to next node as specified in the plan if next node belongs to same partition otherwise the next node is added to new start node info file generated in line 7 and the intermediate result is stored. The algorithm keeps on expanding the nodes to the desired nodes according to the plan until all the nodes and edges get covered (line 9 to line 15).

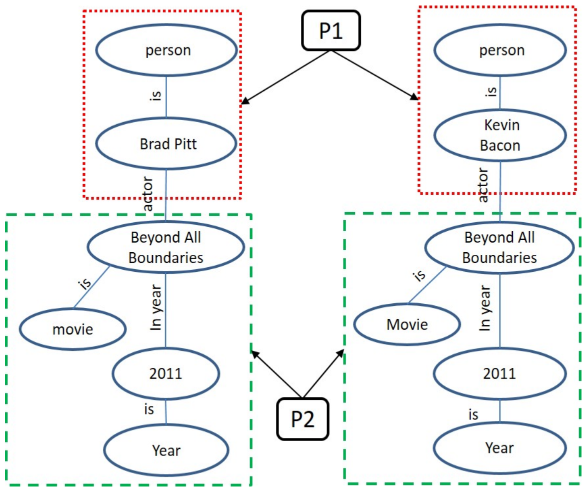


Figure 6.7: Results from the plan executor for the input query plan

After executing the query plan of figure 6.5 in modified QP-Subdue results are shown in the above Figure 6.7 in the form of a graph which are the exact matches. The system utilized both the partitions to compute the given query. Two nodes of the query are found in partition-1 while the rest of the four nodes are found in partition-2. 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 implementation aspects and present our findings for our approach with experimental results.

CHAPTER 7

IMPLEMENTATION DETAILS

This chapter provides a brief overview of the system implementation. The system consists various modules namely- Catalog generation, Plan generation, Partitioning schemes, Partition graph creator, Initial start info file generator (Metric-1), Partition chooser, and Partitioned graph quer processor (PGQP). The catalog generator and plan generation modules has been inherited from QP-Subdue [6]. All the other modules are developed as a part of this work and explained in detail with examples.

The partition chooser is implemented in Bash Script. Bash Script has been chosen as the language of choice, as it provides excellent support for working with files and getting data from one program into another (assuming that data is text). The partitioned graph query processing (PGQP) algorithm is implemented in C as it is a modification of the QP-Subdue system that has been implemented in C. QP-Subdue itself is a modification of Subdue mining algorithm which has been implemented in C and is a main memory algorithm in that the entire graph is loaded and a main memory representation is created before starting mining. The QP-Subdue modification has introduced constrained expansion without modifying the main memory aspect of the algorithm.

The choice of using an interpreted language for developing the various modules of our system is mainly to present the user a single invocation with minimum amount of input in the form of a graph and the queries. Multiple files are written over the course of this execution and multiple programs with appropriate parameters are invoked. A configuration file is input containing the minimum parameters (including partitioning schemes until that time when we can decide it automatically). This provides ease of use and has a very insignificant learning curve. This does not have any penalty on the overall performance. In order to make it easier for a user, the inputs are kept to minimum and a configuration file is used to provide either defaults or user-provided choices. A scripting language is used to accept the configuration file containing all the information. The configuration file contains all the named parameters needed (allows for comments for understanding of the configuration file by others or for later analysis; default values are used if no user-input value is provided for a parameter) for the PGQP system to process a query on a graph database. The partitioned graph query processor system consists of the following modules:

1. Preprocessing of input graph (Subdue format), is done to generate METIS/KaHIP format representation to partition the graph.
2. Graph Partitioning system, which is used to generate k partitions for a given graph database. The partitioning algorithm to be used as well as the partitioning scheme and the number of partitions can be specified.
3. Postprocessing of partitions, is done to generate all the partitions with vertex, its id and label along with edge, its node IDs and label. This step is carried out because METIS/KaHIP generates a single output file in which each line represents a vertex ID and contains the partition ID.
4. Augmenting subdue input with partition information, in this step we add partition number to all the vertices in the graph which is further utilized by PGQP to detect cross-partition edges.
5. Computation of metric 1 and 3, after partitioning and generation of subdue format partitions we compute the the number of connected components (Metric-

3) with each partition using the edges. Metric-1 (Number of starting nodes in each partition) is also generated to initiate the query processing using PGQP.

6. A query plan is generated using QP-Subdue for the query provided. To generate the query plan, the metadata catalog is generated for the input graph database
7. Partition chooser, decides the relevant partition to load, interpret the start node info file to guarantee all the exact matches for the given query
8. Partitioned graph query processor, takes a partition of graph database, a query plan, start node info file and outputs the complete and intermediate answers to that plan and may generate new start node info file for answers crossing the partition.

Using the plan and the output of partitions, the metric "number of starting node in each partition" (Metric-1) and order of partition usage are determined by a program implemented in Java. The order of partition usage is determined using alternative metric values (e.g., Max, Min, random) for understanding their effects. These can also be specified as part of the configuration file. And the number of connected components in each partition (Metric-3) is also computed after the partitioning of the graph database (is a program implemented in Java). Java has been chosen as the language of choice, as it provides excellent support for string processing. 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 substring extraction. The availability of pre-developed classes and methods for many routine tasks and the ability to 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 the set of configuration parameters used.

7.1 Configuration Parameters

The partitioned graph query processor (PGQP) system accepts parameters for different tasks in the form of a configuration file. We have provided options for various parameters such as selection of graph, choice of graph partitioning algorithms, choice of graph partitioning schemes for the chosen algorithm, number of partitions to be generated and so on. Also, values of certain other parameters that are are used by the underlying system for substructure discovery are provided. In the case where certain parameters are absent in the configuration specification, the system uses default values for the same. Listed below are the set of configuration file options for various parameters.

1. Input Graph File: over which queries need to be processed. Specified as a file name with a specific file format.
2. Input Query File: The query which the user wants to evaluate on the above graph database.
3. Log File: The file name to log the results of plan generation, partitioning scheme and various other parameters used by the system to evaluate the query on partitioned graph.
4. Catalog Output: The file to which generated catalog is stored for the given input graph. Catalog is further used for generating query plans.
5. Selection criteria for Plan: Generation of query plan for the given query. Possible values can be min, max or k. If k is large, all plans are generated. A min plan has a lower estimated cost than any other possible plan.
6. Plan Output File: The file to which the generated query plan is written.
7. Partitioning Scheme: This parameter specifies which partitioning schemes should be used. Possible schemes can be METIS and KaHIP. And it takes i number of the partition as input to generate i partitions of the given graph. Also, it

takes different heuristics values to partition the graph. In case of METIS the possible configuration for partitioning and coarsening can be (k-way, recursive bisection) and (random, sorted heavy edge matching) respectively. while for KaHIP the configuration can be eco, ecosocial, fast and fastsocial.

8. Partition Selection: In our system, we allow the user to decide the order of loading the partition based on the number of starting points in the partition(s). The values for this parameter can be min, max, random. In case of min the system load a partition which has minimum number of starting points and vice-versa if max is used. This value is re-computed after each partition usage based on the number of continuations in other partitions. The ordering is re-evaluated after the query is evaluated on each partition. The same partition may have to be used again depending on the answer distribution among the partitions.
9. Output Query Result File: The file to which all the exact result(s) of the query is written. This file contains all the complete and intermediate results generated during query evaluation over partitions of a graph database.

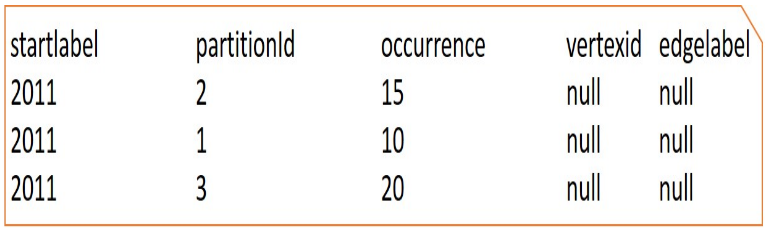
Above are the parameters that can be specified to the partitioned graph query processor system. We think they cover a range of issues we have addressed as part of the thesis, including graphs from different domains, different partitioning schemes, different ways to load relevant partition(s) and so on. With this overview of the configuration settings, we can move onto the details of graph generation.

7.2 Graph Generation

The graph data set used in this work are DBLP, IMDB and synthetically generated graph using SubGen (included along with subdue). The DBLP data set is in the form of XML file which consist of tags to represent author, paper, conference and year. This XML data is parsed to create a subdue format graph which consists of vertices, edges and their corresponding labels. In this work we have used author, paper, conference and year as vertex labels and the edge labels show the relationship between these vertices. Similarly, the IMDB data set is in the form of JSON object which consist of movie information, year, genre, person, gender and movie company. These JSON objects are parsed to create subdue format graph where each vertex represents an instance of either movie information, year, genre, person, gender, movie company and edges represents relationship amongst these instances. The synthetic graph is generated using SubGen which generates a subdue format graph with predefined number of vertices, edges, unique vertex and edge labels. Hence, no preprocessing is required.

7.3 Partition Chooser

Initial starting partition takes a query plan, input graph database and partition file as input. Based on the starting node label of the query plan it finds all the relevant partitions containing the starting node label and generates a start node info file. This file has a line for each partition consisting of the partition number, node label, and a list of vertex ids if available for that node label in that partition. There is a line for each partition if there is a start node in that partition. Figure 7.1 shows the representation of start node info file (SNI file). This information in the start node info file is used to sequence the loading of a partition by the partition chooser using the min, max, or random criterion specified by the user.



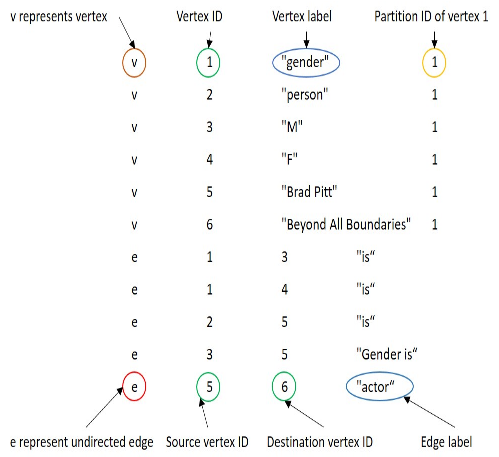
startlabel	partitionId	occurrence	vertexid	edgelabel
2011	2	15	null	null
2011	1	10	null	null
2011	3	20	null	null

Figure 7.1: Start node info file (SNI File)

After choosing a relevant partition, partition chooser calls the partitioned graph query processor (PGQP) by passing partition of the graph database, query plan, start node info file and a new start node info file. While computing the query result if PGQP generates a new start node info file. The partition chooser interprets this newly generated information for loading a partition in the next iteration. Also, if the new start node info file is empty the system will terminate as the query is completely evaluated and all the exact matches have been found and are stored in a file. The partition chooser generates partition usage information using all the files generated and used during the evaluation of a query and is processed separately to find out number of times a partition was used during query evaluation. The sequence of loading a partition is logged in the log file which can be analyzed for the selection of partition in each iteration. With the implementation overview of partition chooser, the details of the plan executor is explained in the following section.

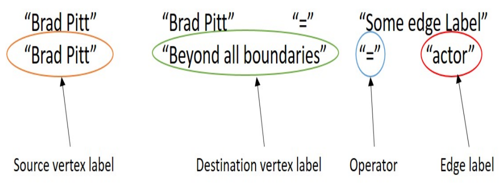
7.4 Partitioned Graph Query Processor (PGQP)

The partitioned graph query processor (PGQP) developed by modifying QP-Subdue system carries out processing of a query over the partitions of a graph database. The partition chooser invokes the partitioned graph query processor iteratively by supplying parameters needed- partition of the graph database, query plan, start node info file and new start node info file. In the previous chapter, we have discussed the algorithm (Algorithm 4) for PGQP. Partitioned graph query processor first processes the partition of the graph database and stores the information of vertices and edges along with the partition identifier to track results crossing the currently loaded partition. Figure 7.2(a) depicts an example of a vertex and an edge of graph in PGQP format. The plan file contains the sequence of nodes with operators and edges with their labels. A sample plan file is as shown in figure 7.2(b).



```
v   1   "gender"                   1
v   2   "person"                   1
v   3   "M"                        1
v   4   "F"                        1
v   5   "Brad Pitt"                1
v   6   "Beyond All Boundaries"    1
e   1   3   "is"
e   1   4   "is"
e   2   5   "is"
e   3   5   "Gender is"
e   5   6   "actor"
```

(a) PGQP graph representation



(b) Query plan file

Figure 7.2: Input graph and query plan of PGQP

Modified QP-Subdue (PGQP) starts the discovery process by finding the starting node mentioned in the start node info file relevant to the currently loaded partition. If the frontier node in the start node info file does not belong to the currently loaded partition, it is written to the new start node info file. From the plan file, next node along with edge label and operator is looked up.

After the 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. For the substructure meeting the condition but requires a different partition for further evaluation are considered as continuation (intermediate) nodes and are added to the intermediate list. Thus, the node(s) which crosses the partition is treated as starting nodes for that partition which will be added to the new start node info file before the substructure containing continuing result(s) are discarded. This process continues until all the nodes and edges in the plan file get visited. The complete and/or continuing results generated are written out to the output result file. If there are no intermediate results crossing the partition for further computation and new start node info file is empty and the system terminates.

With the detailed overview of our implementation. In the next chapter, we discuss the experimental analysis of various queries evaluated on partitioned graph database using different partitioning schemes and metrics to evaluate partitioning schemes in presence of query processing.

CHAPTER 8

EXPERIMENTAL EVALUATION

This chapter presents the results of extensive experimental analysis performed on evaluating queries with different characteristics on partitions of graph databases generated by different graph partitioning schemes. The experimental results are meant to ascertain two aspects: i) queries are processed correctly over partitioned graph databases irrespective of the approach used for partitioning and ii) understand the effects of partitioning on query evaluation to associate (if possible) partitioning schemes with efficient query evaluation. Also, the purpose of these experiments is to test various metrics identified in the previous section and their impact on query evaluation. The correctness as well as the performance of our system across different types of queries, graph databases and different graph partitioning schemes establishes the applicability of our proposed approach for processing queries over partitions of a graph database. Additional work is needed to establish and validate conclusions.

As we have considered different real world and synthetic graph databases and queries, the performance of query evaluation and different metrics for each partitioned graph database is presented in detail in a separate section. The experimental setup and a brief description of the data sets used are also provided.

8.1 Experimental Analysis

The results of processing queries over partitioned graph database are discussed in this section. 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 behavior of queries on partitions generated using a different graph partitioning scheme. For this work, we have used three different data sets- DBLP, IMDB and synthetic graph to perform our experiments. To test the correctness of our approach, we need to make sure PGQP gives the same results as the QP-Subdue which is a non-partitioned, main-memory query processor for graph databases. The largest graph size we have been able to handle in QP-Subdue on our 16GB machine is 550K nodes and 1700K edges. The 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. And the synthetic graph contains 2000 unique vertex labels and 4000 unique edge labels which provides duplication of labels in the graph resulting in the computation of many possible answers.

For the above mentioned graph databases, we took queries having different characteristics that are relevant to the partitioning problem, such as query answers are completely inside a single partition, query answers spanning multiple partitions for exact results, and and queries that need to use the same partition more than once. We have also used 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. Query plans were generated for all the queries on DBLP and IMDB data sets. For queries on the synthetic graph, instead of generating a query plan, known and embedded substructures were used. This is mainly because of the limitation of QP-Subdue needing type nodes and the synthetic generator does not generate any type nodes. Intuitively, a plan which generates less number of intermediate substructures, should take less amount of time to evaluate. Therefore the minimum cost plan would be a plan which will span partition(s) for minimum number of intermediate substructures among all possible plans. The following sections describes query processing on the set of graph databases used for experiments along with an introduction to the data set used for experimenting along with presenting and discussing the results.

8.2 Querying a partitioned DBLP graph

The DBLP data set contains information about authors, papers, conferences and years as vertex labels and edges showing relationship among these vertex labels. For this data set, we have made three different types of queries. Query-1: "Find authors **and** papers in the conference 'Object Oriented Programming, Systems, Languages and Applications"' show an example of a query containing a logical operator. In this query the results would contain authors and their publications in the given conference by accessing multiple partitions of a graph database. Query-2: "Find authors along with their papers **and** conferences in year **2005**" contains both comparison and logical operators. The results for this query will include authors their publications and conferences particularly in year 2005. This query uses a partition more than once for its evaluation for most of the partitioning schemes used. Finally query-3: "Find author(s) who published 'Transaction Management in Multidatabase Systems' along with the conference **AFTER** the year 1990" which again contains comparison and logical operator but all the results are found inside a partition for most of the partitioning schemes. For these queries detailed discussion of their computation and evaluation of all the metrics is presented in this section.

The DBLP graph consists of 1600K (or 1.6M) vertices and 4800K (or 4.8M) edges. In this thesis, we have used all four configurations of METIS namely-, k-way as partitioning type and random matching (rm) as coarsening type, kway as partitioning type and shorted heavy edge matching (shem) as coarsening type, recursive bisection (rb) as partitioning type and random matching (rm) as coarsening type, recusive bisection (rb) as partitioning type and sorted heavy edge matching (shem) as coarsening type to partition the graph into 4 partitions. The resulting partition contains around 450K vertices and 1300K edges in each partition including replication of the edge-cuts along with its node. For the same data set, we also used all the four configuration of KaHIP to partition the graph into 4 partition. Note that the number of partitions can be tailored to the available memory and machine characteristics. The configurations are fast, eco, fastsocial and ecosocial. When 'fast' is used for partitioning the graph in minimum amount of time while 'eco' is used for a good tradeoff between partition quality and execution time. Configurations with a social in their name are used for social networks and web graphs.

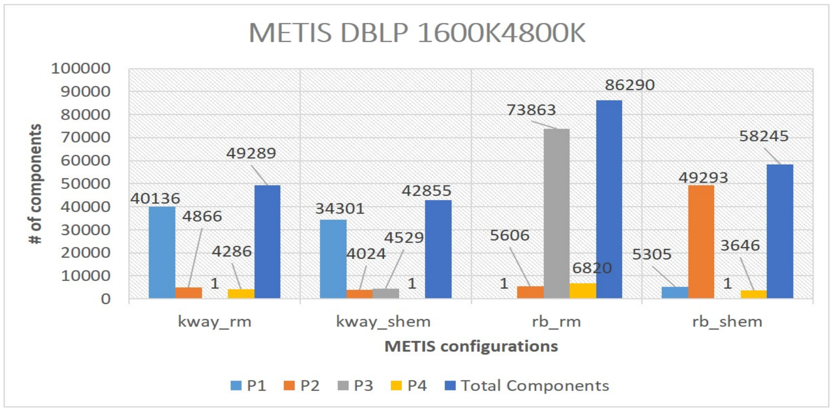


Figure 8.1: Connected components generated by METIS for DBLP graph

After partitioning the graph into four parts. We compute the number of connected components(metric-3) in all the partitions. Figure 8.1 shows connected components generated by different configurations of METIS and Figure 8.2 shows components generated by all four configurations of KaHIP. It is clear that the maximum number of connected components generated by METIS using recursive bisection as

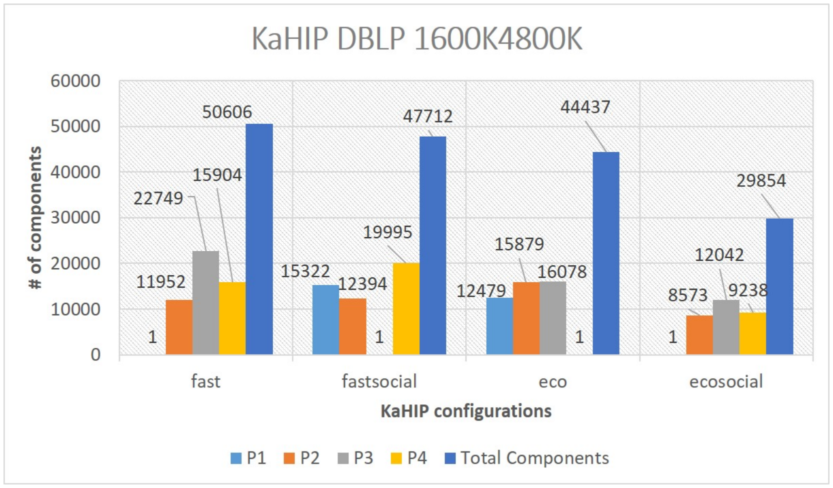


Figure 8.2: Connected components generated by KaHIP for DBLP graph

the partitioning algorithm and random matching as the coarsening algorithm is 73863 while KaHIP has generated maximum of 22749 number of components using fast configuration which is 4 times less than the maximum components generated by METIS. Also, KaHIP with ecosocial configuration has generated 29854 minimum number of connected components amongst all the configuration of KaHIP and METIS. This is as expected, since ecosocial takes huge amount of time as compared to other configuration of KaHIP and METIS for finding good partitions of the graph. The performance of different queries on partitions generated using ecosocial is likely to take less amount of time for their computation due to less number of connected components. To understand the relationship between connected components and queries in the following section we use metric-2, number of times a partition was used to compute the query. It appears that the number of connected components has a bearing on the number of partitions needed as well as the number of times a partition is needed for processing the query.

To process query-1, query-2 and query-3 on DBLP graph, the first step is to identify the starting node labels across all the partitions. Figure 8.3 represents start-

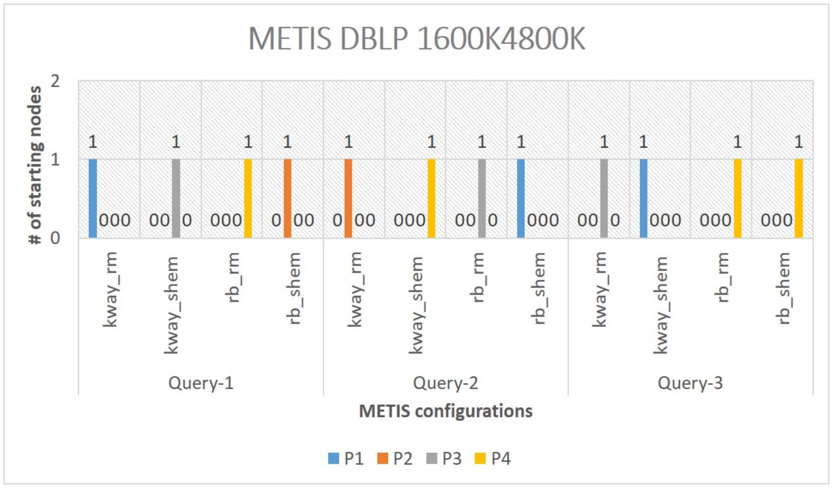


Figure 8.3: Starting node label information for METIS generated partitions of DBLP graph

ing node vertices (or vertex IDs) for the starting node label in partitions generated by different configurations of METIS. From the figure, it is evident that for a given query starting node labels can be present in different partitions. For example, the starting node label of query-1 is present in partition-1 when METIS is used with configuration kway_rm while for kway_shem, rb_rm and rb_shem the starting node label is present in partition-3, partition-4 and partition-2 respectively. Similarly, Figure 8.4 represents starting node label in partitions generated using KaHIP with different configurations. As expected the starting node labels of query-1, query-2 and query-3 belong to different partitions generated using different configurations of KaHIP. From Figure 8.4 another noticeable observation, for query-2 the starting node label belongs to partition-2 for fast, fastsocial and ecosocial configurations of KaHIP. This reinforces that starting node label of the query can belong to same or different partitions which are generated using different partitioning schemes.

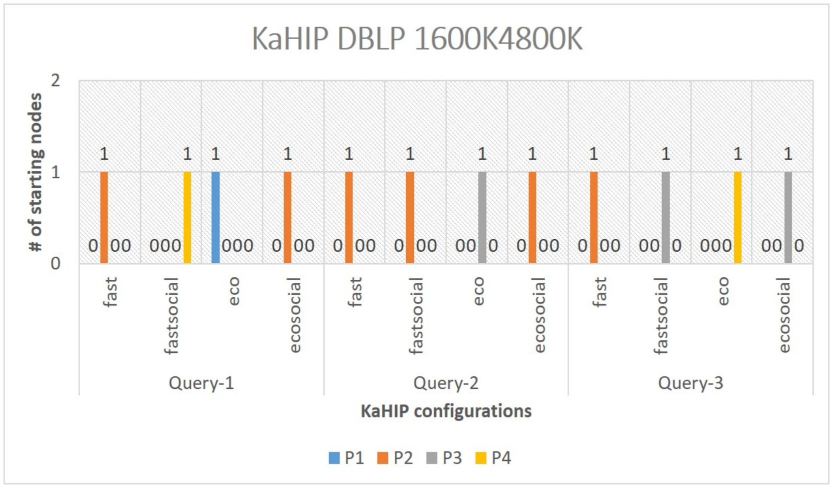


Figure 8.4: Starting node label information for KaHIP generated partitions of DBLP graph

Figure 8.5 and Figure 8.6 provide information about number of times a partition was loaded (metric-2) to compute query-1, query-2, and query-3 for its exact matches on partitions generated using METIS and KaHIP respectively. The partition usage information is based on selecting a partition containing maximum number of starting node labels. Evaluation of query-1 used multiple partitions on different configuration of METIS and KaHIP. For social and ecosocial configuration, query-1 utilized single partition while on partitions generated using rb_shem configuration of METIS query-1 has utilized three different partitions. Computations of query-2 have used multiple partitions where one of the partition is used more than once for most of the partitions generated using different configurations of METIS. On the other hand, query-2 uses multiple partitions on KaHIP but for social configuration partition-3 has been utilized more than once and for ecosocial configuration single partition is used to get the exact matches. Processing of query-3 on partitions generated by different configurations of METIS and KaHIP utilizes single partition for all of its matches. For rb_rm configuration of METIS, computation of query-3 used two different partitions.

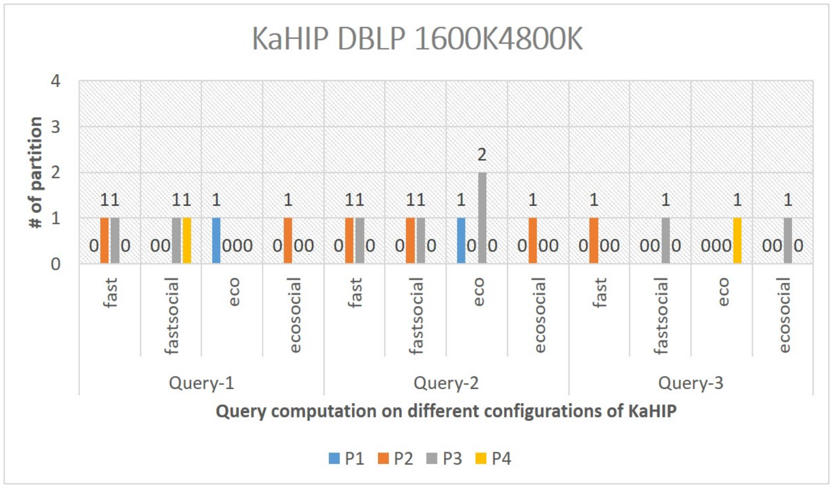


Figure 8.6: Partition usage information for KaHIP generated partitions (selection = max)

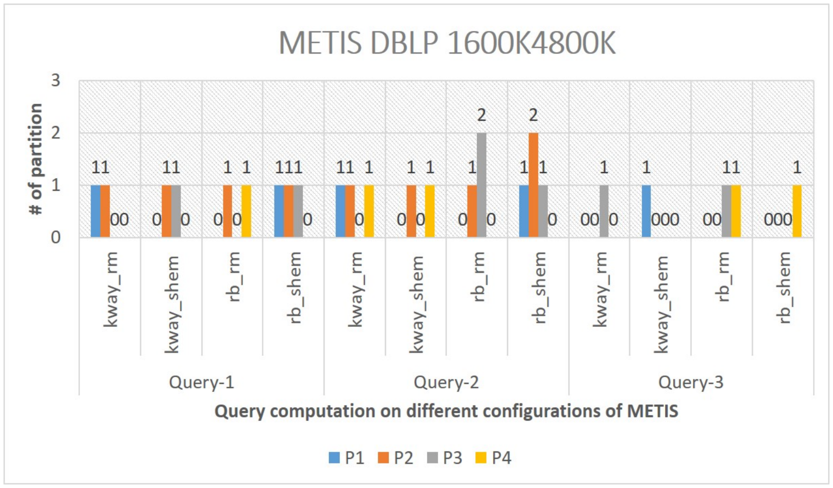


Figure 8.5: Partition usage information for METIS generated partitions (selection = max)

The results of metric-2 reinforces our premise of the partitioning scheme which generates less number of connected components tends to produce exact matches by using minimum number of partitions. The ecosocial configuration of KaHIP has utilized minimum number of partitions for query workload containing query-1, query-2, and query-3. The execution for all the queries on different configurations of METIS and

KaHIP are shown in Figure 8.7 and Figure 8.8 respectively. Thus, all the queries take less execution time on partitions generated using different configurations of KaHIP as show in Figure 8.8.

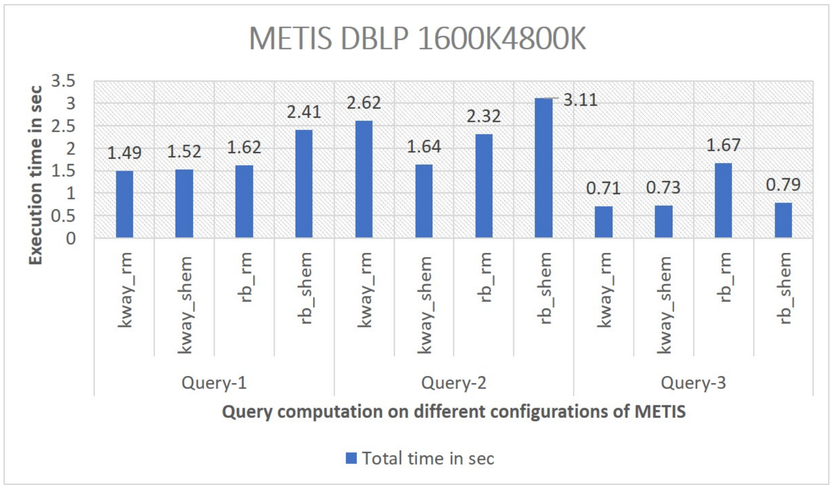


Figure 8.7: Query execution time on partitions generated by METIS for DBLP graph

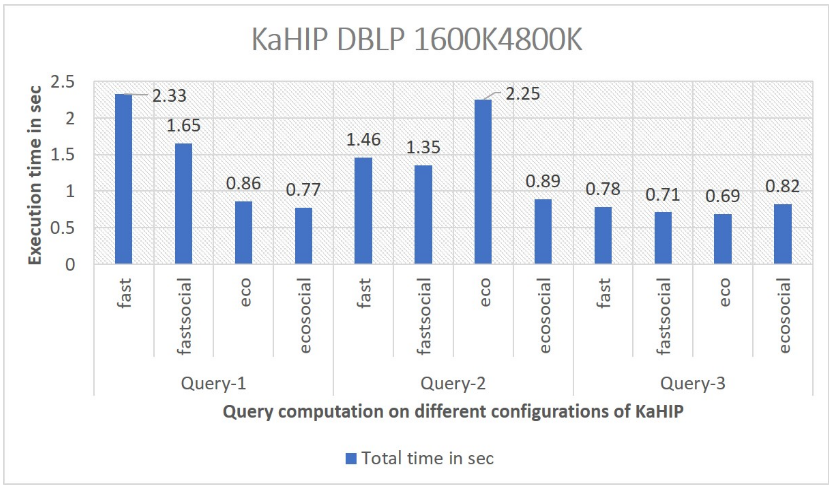


Figure 8.8: Query execution time on partitions generated by KaHIP for DBLP graph

8.3 Querying partitioned IMDB graph

The IMDB data set contains information about movies, year, genre, person, gender, movie company as vertex labels and edges represents relationship among these vertex labels. For this data set, we use three different queries. Query-4, "Find tv-series and its company name by "Kelsey, Wagner" where genre should be animation **AND** comedy" shows an example of a query containing a logical operator. Results for query-4 will contain all the tv series and production companies which have comedy and animation as genre. For most for the partitioning scheme, this query uses a partition more than once for obtaining results. Query-5, "Find movie and its company by "Adam Sandler" where the genres should be comedy **AND** Sci-Fi but the year should be **NOT EQUAL** to 2000" contains both comparison and logical operators. The results for this query will include movies with comedy and Sci-Fi genre and its production company by accessing multiple partitions of a graph database. Query-6, "Find all the companies where "Fred Wolf" has worked as a writer **OR** "Salma Hayek" has worked as an actress" contains the **OR** logical operator. All the results of query-6 are found inside a single partition for most of the partitioning schemes. In this section, we present a detailed discussion of our experimental analysis of these queries with respect to the metrics proposed.

The IMDB graph consists of 1750K (or 1.75M) vertices and 5100K (or 5.1M) edges. In this thesis, we have used four configurations of METIS and four configurations of KaHIP to partition the graph in four partition. The resulting partitions contain around 500K vertices and 1500K edges in each partition including replication of the edge-cuts along with its node (PGQP representation). After partitioning, we compute the number of connected components(metric-3) in all the four partitions. Figure 8.9 shows connected components generated by different configurations of METIS and Figure 8.10 shows components generated by all four configurations of

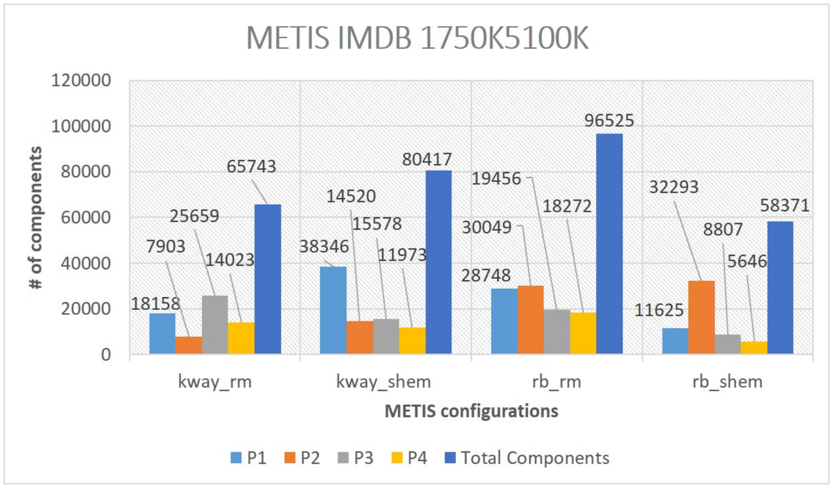


Figure 8.9: Connected components generated by METIS for IMDB Graph

KaHIP. From the figure, it is clear that rb_rm configuration of METIS has generated 96525 number of connected components which is the highest amongst all the configurations. This partitioning scheme also seems to distribute the connected components somewhat evenly among all the partitions. On the other hand, ecosocial configurations of KaHIP has generated minimum number of connected components as expected which is 40975.

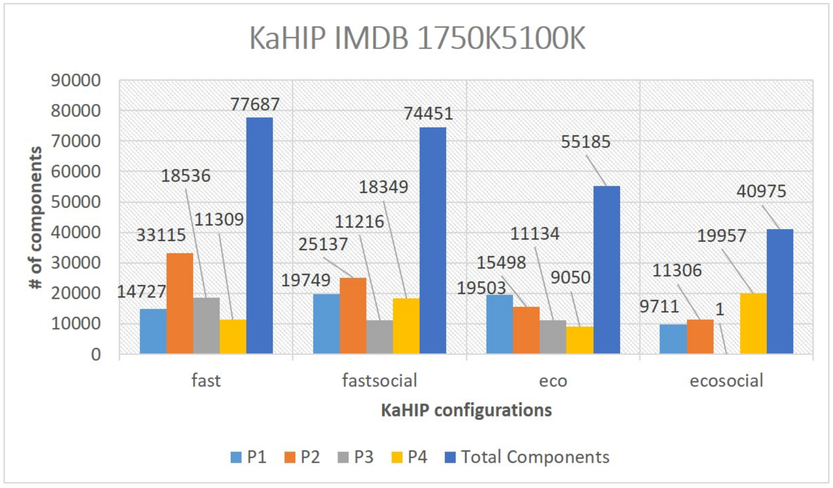


Figure 8.10: Connected components generated by KaHIP for IMDB Graph

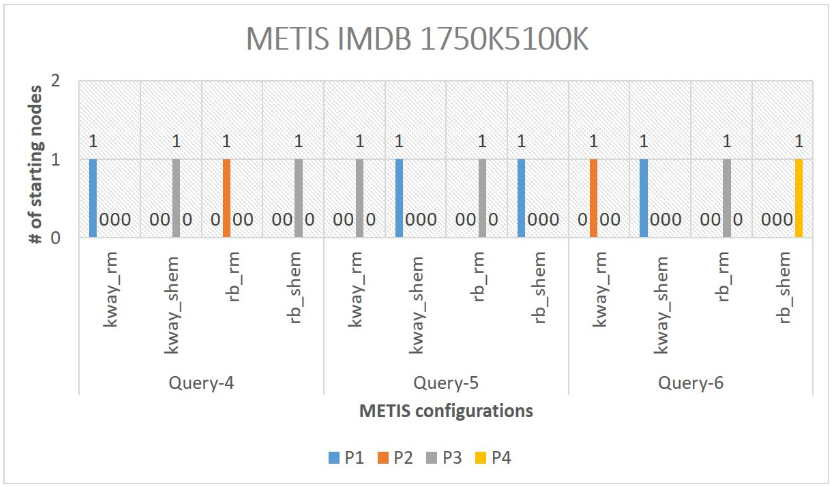


Figure 8.11: Starting node label information for METIS generated partitions of IMDB graph

As discussed earlier, we compute the starting node info file for each query (query-4, query-5 and query-6). Figure 8.11 represents starting node label in partitions generated by a different configurations of METIS. From the figure, it is clear that for query-4 the starting node label is present in partition-1, partition-3, partition-2 and partition-3 when METIS is used with configuration kway_rm, kway_shem, rb_rm and rb_shem. While for query-5 the starting node label is present in partition-1, partition-2, partition-1 and partition-2 for kway_rm, kway_shem, rb_rm and rb_shem configuration of METIS, and for query-6 the starting node label is present in parition-2, partition-1, partition-3 and partition-4 for kway_rm, kway_shem, rb_rm and rb_shem configuration of METIS. Similarly, Figure 8.12 represents starting node label in partitions generated using KaHIP with different configuration. As expected the starting node labels of query-4, query-5 and query-6 belongs to different partitions generated using different configurations of KaHIP. From Figure 8.11 and Figure 8.12 it is evident that starting node label of the query can belong to same or different partitions which are generated using different partitioning schemes.

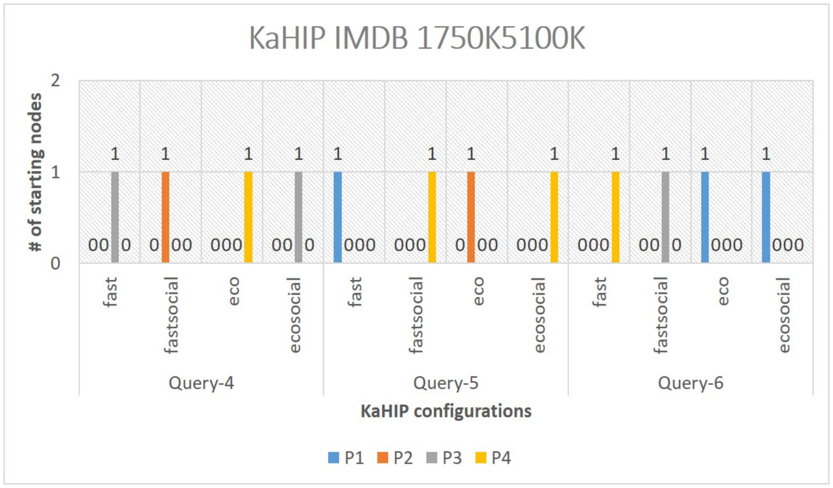


Figure 8.12: Starting node label information for KaHIP generated partitions of IMDB graph

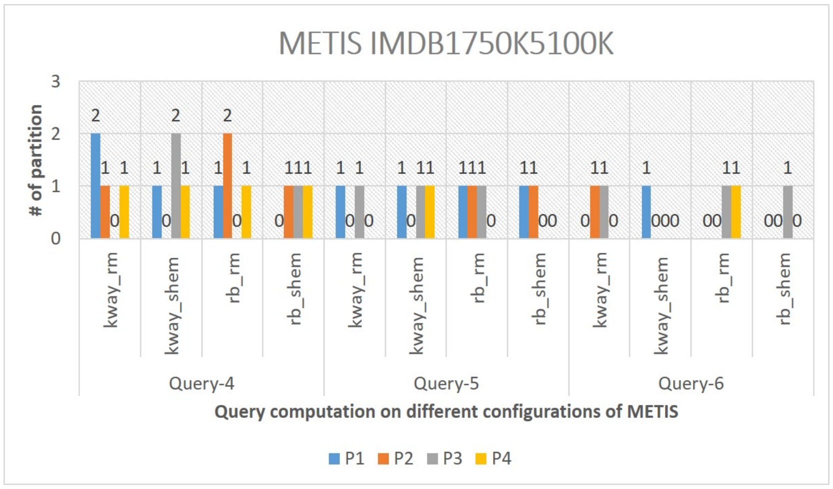


Figure 8.13: Partition usage information for METIS generated partitions of IMDB graph (selection = max)

During evaluation of query partition chooser keeps track of number of times a partition was loaded (metric-2) to find the answers. For computing all the answers for query-4, query-5 and query-6 Figure 8.13 and Figure 8.14 provide information about number of times a partition was loaded which were generated using METIS and KaHIP respectively. The partition usage information is based on selecting a partition containing maximum number of starting nodes in a partition. Computation of Query-4 has used three distinct partitions for one partitioning scheme (rb_shem) and has used four for the rest where one partition has been used twice. Certainly, in the absence of multiple use of the same partition, it would have been three. Hence, rb_shem partitioning scheme is better for this query than the others. It turns out the partitioning scheme rb_shem is also better for this workload of three queries as the number of partitions used is minimum without any partition being used more than once. It is also interesting to note that this partitioning scheme has also the least number of connected components overall as well as a distribution of connected components in all partitions. Also for fast and eco configuration of KaHIP, query-4 shows similar usage pattern. On the other hand, for fastsocial and ecosocial configuration of KaHIP query-4 uses multiple partitions to get the exact matches. Evaluation of query-5 used multiple partitions on different configuration of METIS and KaHIP. For eco configuration, query-5 utilized single partition while for partitions generated using fast configuration query-5 has utilized partition-4 more than once. Processing of query-6 on partitions generated by different configuration of METIS and KaHIP utilizes single partition for its complete matches. For kway_rm and rb_rm configuration of METIS, computation of query-6 used two different partitions.

The execution for all the queries (query-4, query-5, and query-6) on different configurations of METIS and KaHIP are shown in Figure 8.15 and Figure 8.16 respectively. Again, the results reinforce our premise of the partitioning schemes that generate less number of connected components need fewer number of partitions. Thus, translates to minimum execution times.

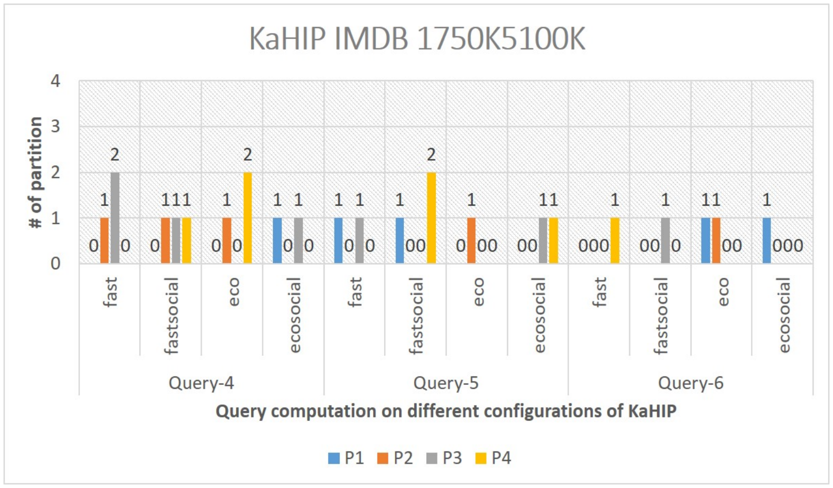


Figure 8.14: Partition usage information for KaHIP generated partitions of IMDB graph (selection = max)

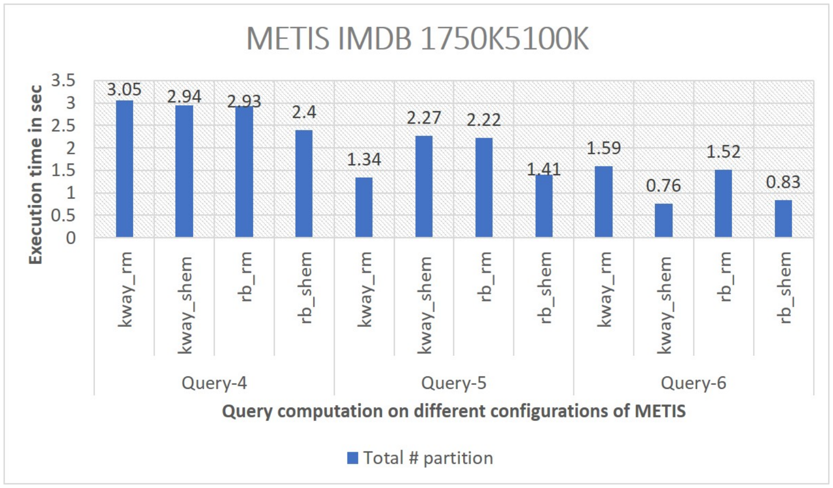


Figure 8.15: Query execution time on partitions generated from METIS for IMDB graph

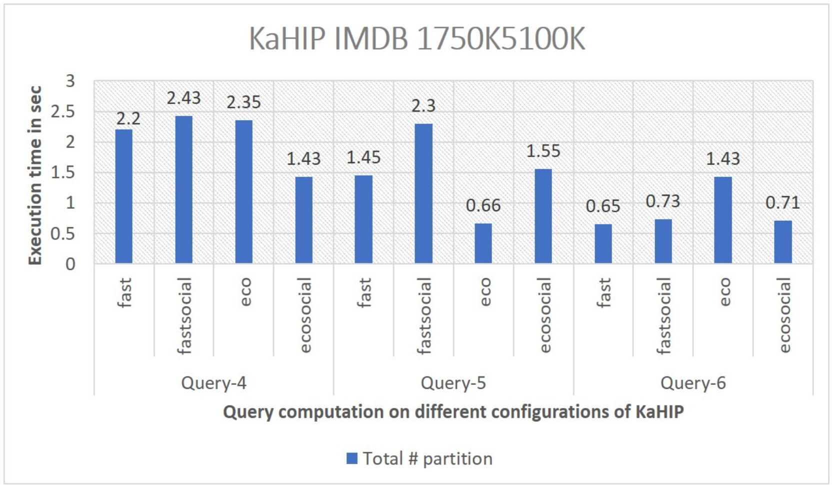
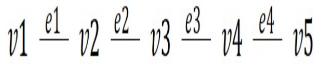


Figure 8.16: Query execution time on partitions generated from KaHIP for IMDB graph

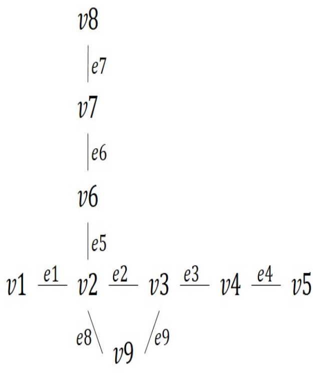
8.4 Querying over a partitioned Synthetic graph

Synthetic graphs are generated using Subgen which is included along with the Subdue system. A synthetic graph allows us to test our premises for characteristics that may not be present in a real world graph database. Hence, this is very useful for analysis and for understanding the effects of various graph types. For this work, we generated a graph containing 400K vertices and 1200K edges, 2000 unique vertex labels and 4000 unique edge labels. Hence, the generated graph contains, on a average, 200 vertices for each vertex label and 300 vertices for each edge label. We also embedded 200 instances of a structure containing 9 vertices and 9 edges. We formulate three different queries from this embedded substructure. Figure 8.17 shows the graphical representation of all the three queries. Query-7 which represents a line shape containing 5 vertices and 4 edges. Query-8 represents the substructure that we embedded which contains a cycle and is a T-shaped query. Finally, query-9 contains 5 vertices and 4 edges and represents a T-shape. This query contains two nodes ($v1$,$v2$) and one edge ($e1$) from the embedded structure while the other three nodes

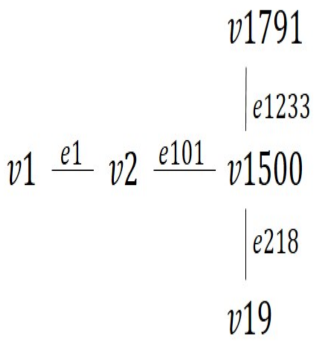
($v19$,$v1500$,$v1791$) and three edges ($e101$,$e218$,$e1233$) are not part of the embedded substructure. The purpose of query-9 was to check the correctness of the system for non embedded structure. The expected answer instances of query-7 and query-8 are 100.



(a) Query−7



(b) Query−8



(c) Query−9

Figure 8.17: Different queries for Synthetic graph

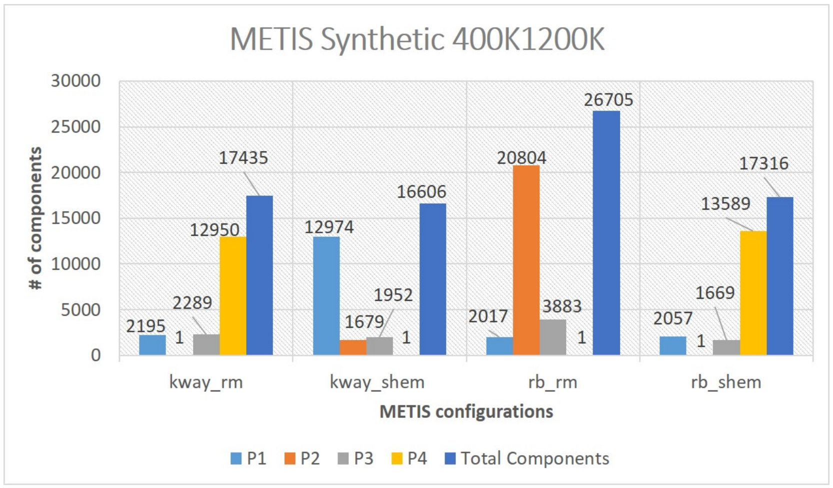


Figure 8.18: Connected components generated by METIS for Synthetic Graph

The synthetic graph was partitioned using different configurations of METIS and KaHIP and we calculated connected components in each partition. The resulting partitions contained around 150K vertices and 470K edges. Figure 8.18 shows connected components generated by different configurations of METIS and Figure 8.19 shows components generated by all four configurations of KaHIP. From the figure, it is clear that ecosocial configuration of KaHIP has generated 5367 connected components which is almost three times less than 16606 connected components generated by kway_shem configuration of METIS.

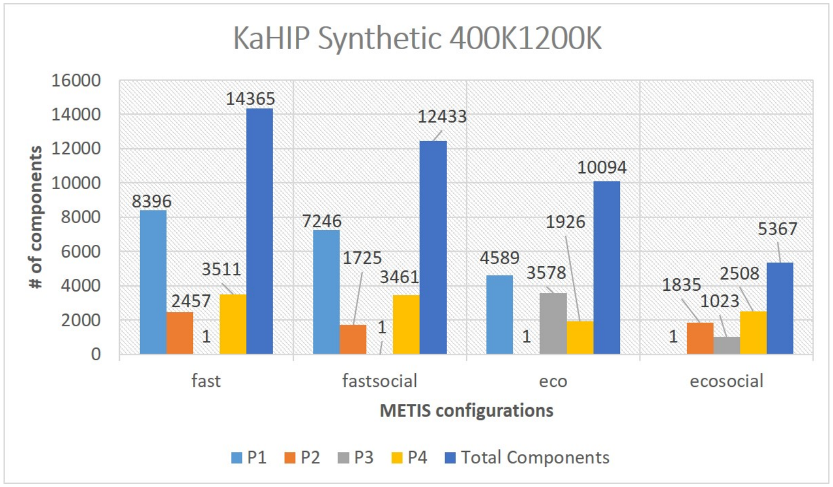


Figure 8.19: Connected components generated by KaHIP for Synthetic Graph

Before processing the query, we compute the starting node information for each query (query-7, query-8, and query-9) which will be same for all queries has the same starting node label is used for all the queries ($v1$). Note that plan generation is not used for queries on the synthetic graph as there are no type information making it not amenable to catalog usage and plan generation. Figure 8.20 represents starting node label for all the queries (query-7, query-8 and query-9) in partitions generated by different configurations of METIS. And Figure 8.21 represents starting node label for all the queries (query-7, query-8 and query-9) in partitions generated using KaHIP with different configurations. From both the figures, it is clear that the start node label of all the queries is present in multiple partitions because the graph contains duplicate vertex labels. The start node info file generated for these queries is different as compared to the start node info file generated for different queries on DBLP and IMDB graph where the vertex labels are unique. This set of queries also signify the importance of partition chooser for selecting a partition since all the partitions contain starting node label and any partition can be loaded first to initiate query processing.

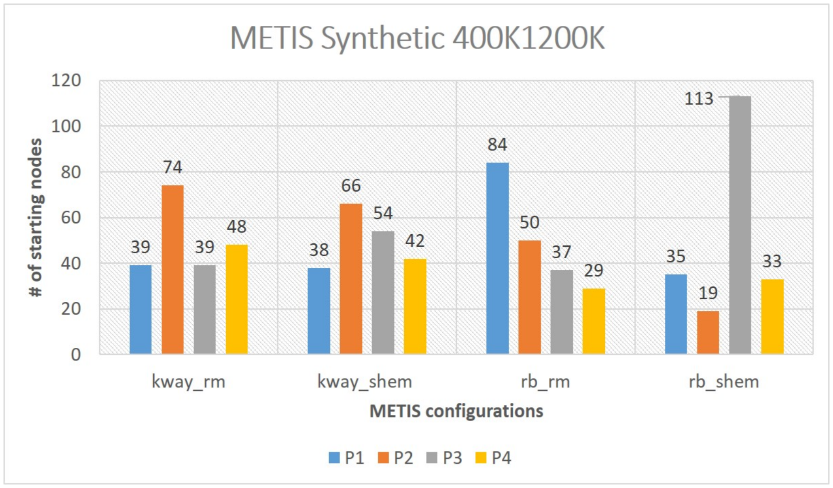


Figure 8.20: Starting node label information for METIS generated partitions of Synthetic graph

To compute answers for query-7, query-8, and query-9 Figure 8.22 shows the loading of partitions and number of times a partition was loaded. The partitions were loaded using MAX and MIN number of occurrence of the start node label in each iteration of query processing. Figure 8.23 provides information about number of times a partition was loaded for partitions generated using KaHIP and the selection of loading a partition was done based on MAX and MIN number of occurrence of

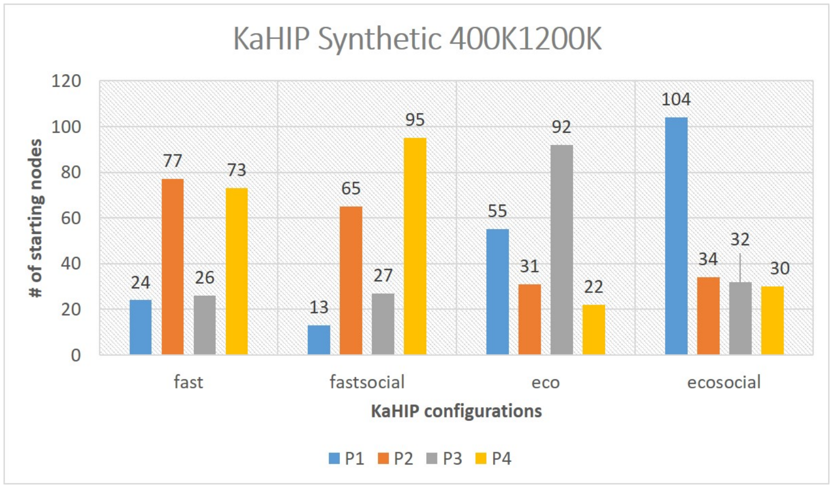


Figure 8.21: Starting node label information for KaHIP generated partitions of Synthetic graph

the start node label. From the figures, it is evident that sequencing the partition by MAX has loaded minimum number of partitions when compared to MIN.

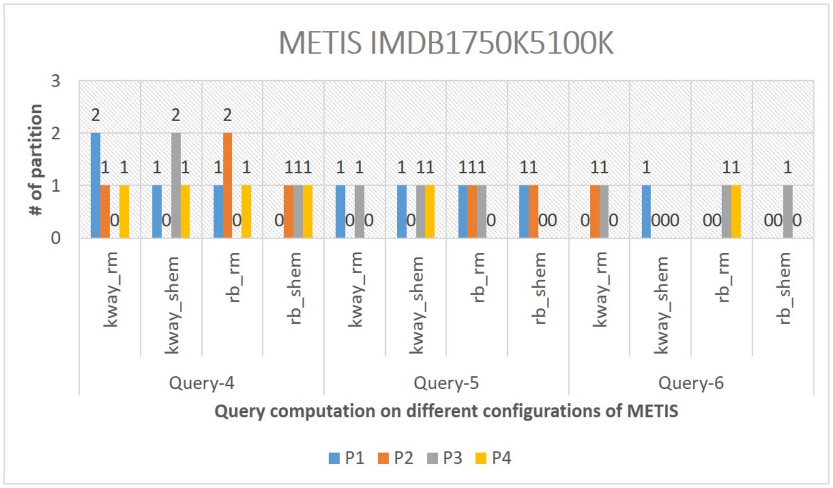


Figure 8.22: Partition usage information for METIS generated partitions of Synthetic graph (selection = max, min)

Figure 8.24 and Figure 8.25 shows the time taken for executing all the queries (query-7,query-8 and query-9) on different configurations of METIS and KaHIP re-

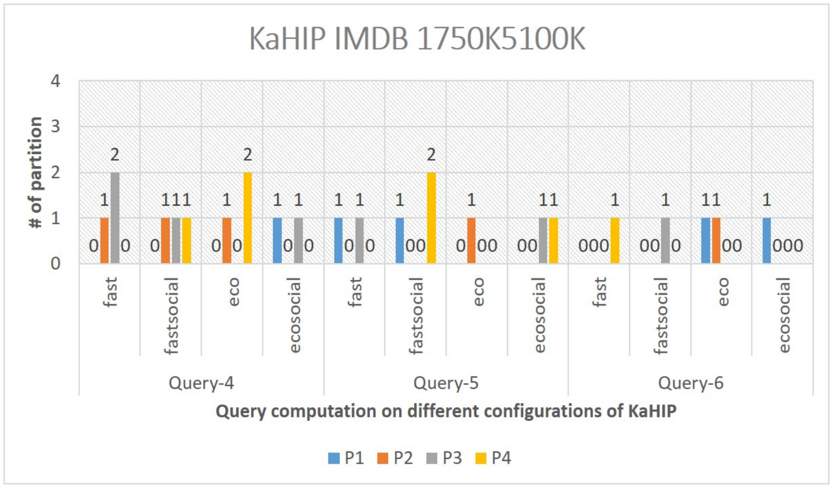


Figure 8.23: Partition usage information for KaHIP generated partitions of Synthetic graph (selection = max, min)

spectively. It can be concluded from the figure that partitioning scheme which generates less number of connected components produces exact matches by using minimum number of partitions. Hence, minimum time is taken for processing the query.

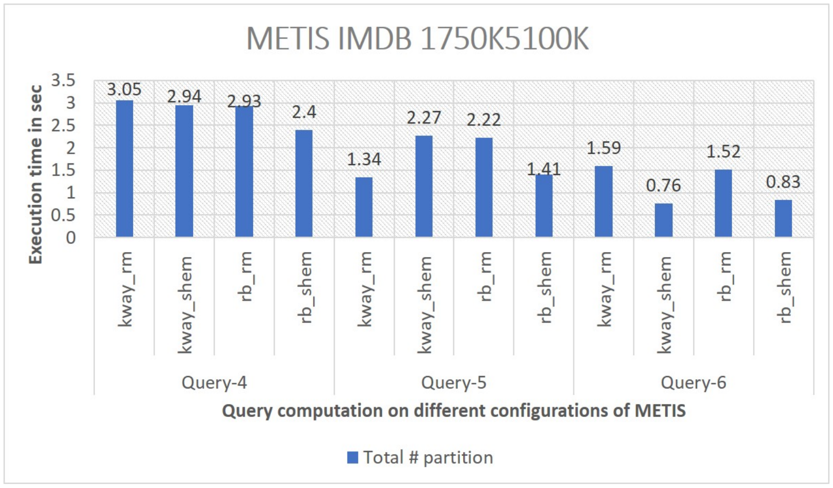


Figure 8.24: Query execution time on partitions generated from METIS for Synthetic graph

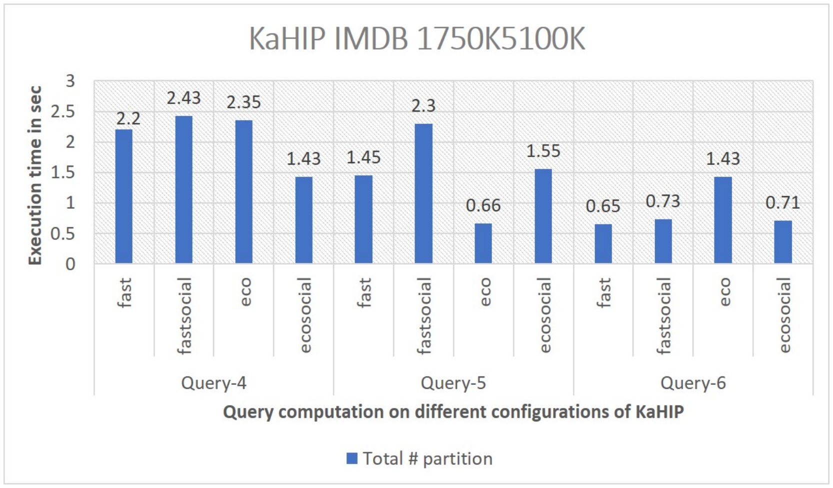


Figure 8.25: Query execution time on partitions generated from KaHIP for Synthetic graph

In summary, we have carried out exhaustive experiments across various domains and presented the results of our findings. The consistent performance of the partitioned graph query processor has validated our expectation about the feasibility of the proposed novel approach for various types of queries over partitioned graph database. And the metrics proposed provides insights into the effects of partitioning schemes on query processing.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

In this thesis, we have proposed an approach for processing queries over a partitions of a graph database. This is one way to ensure scalability of query processing to a graph database of any size. We have developed a framework that allows us to partition a graph database using existing partitioning techniques (METIS, KaHIP) and process queries on these partitions. This allows one to use existing methods as well as choose different properties for making partitions to meet the needs of the graph database as well as queries that are processed often. We have identified information needed for computing the correct result of a query over partitions of a graph database by loading one partition at a time. Although the partitions can be loaded in any order until all results are obtained, by choosing the partitions as well as the order judiciously, one can minimize the total number of partitions that are needed for processing a query or a set of queries. The answers of a query are separately accumulated as the partitions are processed by writing to a file initial start node label and partial results from each partition. In this work, we have defined a set of metrics to analyze the relationship between the properties of partitions (obtained by using a partitioning scheme) such as the number of start nodes in each partition and the number connected components in each partition and their effect on query processing. We also have tried to minimize the total number of partitions used for processing a query (or a query load) using the properties of the partitions and other runtime information such as how many answers continue from a partition to other partitions. It is clear that the total number if partitions required to compute a query depends on a partition characteristics. For example, a partitioning scheme that generates more number of connected component within a partition is likely to require more number of total partitions to compute the query results. This is because a query answer that is distributed over two components with in partition needs to go out to a different partition and come back requiring the same partition to be loaded (used) multiple number of times. Apart from minimizing the effort required to process queries over partitioned graphs, the proposed approach overcomes the limitation of techniques which uses main memory to load the entire graph databases to evaluate queries.

In order to efficiently evaluate queries over partitions of a graph database, we have modified an existing query system QP-Subdue to process partitions along with storing the partition information with each vertex. Also, we preprocess the partitions of a graph to replicate the *edge-cut* information which helps in processing queries spanning more than one partitions. The need for analyzing the partitioning scheme from a query processing perspective has been established. Various parameters such as number of starting and continuing nodes, total number of partition(s) required to compute the query results, and the number of components with in each partition which can affect query processing over partitioned graph have been identified and analyzed in detail. Experiments were carried out to validate our approach and provide some insights into the metrics gleaned from partitioning schemes on query processing. Some of the enhancements that can be carried out are outlined in the following discussion.

The evaluation of queries over a partitioned graph database using the proposed approach results in metric-based sequence of partition loading. However, as we have not used all possible metrics, further improvements can be done by identifying additional metrics for optimizing partition sequence to process a query. For example, *edge-cuts* information combined with the number of starting nodes in a partition may be used instead of the start node information alone. The intuition behind this idea is if the loaded partition has less number of starting partitions and more number of edges going to other partition, the chances for a query spanning to other partition increases because of more *edge-cut* between the currently loaded partition and other partitions. Other improvement that can be made in determining best sequence of partitions for the given query by using the graph and query characteristics. Also catalog information can be utilized to determine the best partitioning quality.

QP-Subdue has modified the Subdue system developed for mining (substructure discovery) for processing a query. This system used the entire graph for processing a query. QP-Subdue was further modified to accommodate for loading a partition and processing a query partially on that partition. Additional modifications were needed to collect information during the execution of one partition to be used during the next partition. Also, complete results needed to be accumulated over the execution of queries over several partitions. The current system expands its instances by either one edge or one node and one edge. For query processing, if a node has k edges associated with it, it will take k iterations to obtain a k edge substructure. Instead, it may be more useful to expand all the k edges in a single iteration which will further improve the execution time for query processing.

In conclusion, we believe the adaptation of partitioning schemes to process queries over large size graph databases are effective and opens up new possibilities and a research direction that is novel and different from contemporary techniques.

REFERENCES

[1] T. Heath and C. Bizer, *Linked data : evolving the web into a global data space*, ser. Synthesis lectures on the semantic web : theory and technology. Morgan & Claypool publishers, 2011.

[2] (2015) The DBLP website. [Online]. Available: http://dblp1.uni-trier.de/statistics/recordsindblp.html

[3] (2015) The FREEBASE website. [Online].

[4] (2015) The IMDB website. [Online]. Available: http://www.imdb.com/stats

[5] (2015) The FREEBASE website. [Online].

[6] A. Goyal. (2015) QP−SUBDUE: PROCESSING QUERIES OVER GRAPH DATABASES. [Online].

[7] 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*, ser. OSDM '05. New York, NY, USA: ACM, 2005, pp. 71–76. [Online]. Available: http://doi.acm.org/10.1145/1133905.1133915

[8] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '95. New York, NY, USA: ACM, 1995. [Online]. Available: http://doi.acm.org/10.1145/224170.224228

[9] A. Pothen, H. D. Simon, and K.-P. Liou, “Partitioning sparse matrices with eigenvectors of graphs,” *SIAM J. Matrix Anal. Appl.*, vol. 11, no. 3, pp. 430–452, May 1990. [Online]. Available: http://dx.doi.org/10.1137/0611030

[10] M. T. Heath and P. Raghavan, “A cartesian parallel nested dissection algorithm,” *SIAM J. Matrix Anal. Appl.*, vol. 16, no. 1, pp. 235–253, Jan. 1995. [Online]. Available: http://dx.doi.org/10.1137/S0895479892238270

[11] G. L. Miller, S. Teng, and S. A. Vavasis, “A unified geometric approach to graph separators.” Puerto Rico: IEEE, Oct 1991, pp. 538–547.

[12] G. Karypis and V. Kumar, “Multilevel graph partitioning schemes,” in *Proceedings of the 1995 International Conference on Parallel Processing, Urbana-Champain, Illinois, USA, August 14-18, 1995. Volume III: Algorithms & Applications.*, 1995, pp. 113–122.

[13] F. Pellegrini and J. Roman, “Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs,” in *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, ser. HPCN Europe 1996. London, UK, UK: Springer-Verlag, 1996, pp. 493–498. [Online]. Available: http://dl.acm.org/citation.cfm?id=645560.658570

[14] P. Sanders and C. Schulz, “High quality graph partitioning,” in *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, 2012, pp. 1–18. [Online]. Available: http://www.ams.org/books/conm/588/11700

[15] R. Giugno and D. Shasha, “Graphgrep: A fast and universal method for querying graphs.” in *ICPR (2)*. IEEE Computer Society, 2002, pp. 112–115. [Online]. Available: http://dblp.uni-trier.de/db/conf/icpr/icpr2002-2.html#GiugnoS02

[16] X. Yan, P. S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04. New York, NY, USA: ACM, 2004, pp. 335–346. [Online]. Available: http://doi.acm.org/10.1145/1007568.1007607

[17] 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, 2007, pp. 737–746. [Online]. Available: http://dblp.uni-trier.de/db/conf/kdd/kdd2007.html#TongFGE07

[18] B. Hendrickson and R. Leland, "The Chaco User's Guide: Version 2.0," Sandia National Lab, Tech. Rep. SAND94–2692, 1994.

[19] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998. [Online]. Available: http://dx.doi.org/10.1137/S1064827595287997

[20] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *J. Parallel Distrib. Comput.*, vol. 48, no. 1, pp. 96–129, Jan. 1998. [Online]. Available: http://dx.doi.org/10.1006/jpdc.1997.1404

[21] P. Sanders and C. Schulz, "Engineering multilevel graph partitioning algorithms," in *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, 2011, pp. 469–480. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23719-5_40

[22] P. Sanders and C. Schulz, "Think locally, act globally: Highly balanced graph partitioning," in *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, 2013, pp. 164–175. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38527-8_16

[23] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *Proceedings of the 2012 ACM SIGMOD*

International Conference on Management of Data, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 517–528. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213895

[24] T. N. Bui and C. Jones, "A heuristic for reducing fill-in in sparse matrix factorization," in *PPSC*, 1993, pp. 445–452.

[25] S. T. Barnard and H. D. Simon, "Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems." *Concurrency - Practice and Experience*, vol. 6, no. 2, pp. 101–117, 1994. [Online]. Available: http://dblp.uni-trier.de/db/journals/concurrency/concurrency6.html#BarnardS94

[26] B. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell Systems Technical Journal*, vol. 49, no. 2, 1970.

[27] J. Maue and P. Sanders, "Engineering algorithms for approximate weighted matching," in *Proceedings of the 6th International Conference on Experimental Algorithms*, ser. WEA'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 242–255. [Online].

[28] C. Schulz. (2013) High quality graph partitioning.phd thesis. [Online]. Available: http://algo2.iti.kit.edu/schulz/dissertation_christian_schulz.pdf

[29] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[30] M. A. Rodriguez and J. Shinavier, "Exposing multi-relational networks to single-relational network analysis algorithms," *CoRR*, vol. abs/0806.2274, 2008. [Online]. Available: http://arxiv.org/abs/0806.2274

[31] J. L. Bentley and M. D. McIlroy, "Engineering a sort function," *Softw. Pract. Exper.*, vol. 23, no. 11, pp. 1249–1265, Nov. 1993. [Online]. Available: http://dx.doi.org/10.1002/spe.4380231105

[32] J. Katajainen, T. Pasanen, and J. Teuhola, "Practical in-place mergesort," *Nordic J. of Computing*, vol. 3, no. 1, pp. 27–40, Mar. 1996. [Online]. Available: http://dl.acm.org/citation.cfm?id=642136.642138

[33] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen, "Hermes: Dynamic partitioning for distributed social network graph databases," in *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, 2015, pp. 25–36. [Online]. Available: http://dx.doi.org/10.5441/002/edbt.2015.04

[34] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, ser. WSDM '14. New York, NY, USA: ACM, 2014, pp. 333–342. [Online]. Available: http://doi.acm.org/10.1145/2556195.2556213

BIOGRAPHICAL STATEMENT

Jay D. Bodra was born in Mevasa, Gujarat, India. He received his Bachelors Degree in Computer Science and Engineering from Narsee Monjee Institute of Management Studies, India in August 2012. His interest in research brought him to University of Texas at Arlington where he later obtained his Masters degree in Computer Science and Engineering in May 2016. His research interests include graph mining, information retrieval and data mining.