MR_QP: A Scalable Approach To Query Processing on Arbitrary-Size Graphs Using The Map/Reduce Framework

by

HARSHIT MODI

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 2020

ACKNOWLEDGEMENTS

I express my sincere gratitude and thankfulness to Dr. Sharma Chakravarthy for providing me an opportunity to work on this project and 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. Leonidas Fegaras and Mr. David Levine for taking their time and serving on my committee.

I would like to extend my gratitude towards Abhishek Santra for constant guidance, support and kind words. Also, I would like to thank my family and friends for their constant support and encouragement throughout my academic career.

May 14, 2020

ABSTRACT

MR_QP: A Scalable Approach To Query Processing on Arbitrary-Size Graphs Using The Map/Reduce Framework

Harshit Modi, M.S.

The University of Texas at Arlington, 2020

Supervising Professor: Dr. Sharma Chakravarthy

The utility and widespread use of Relational Database Management Systems (RDBMSs) comes not only from its simple, easy-to-understand data model (a relation or a set) but mainly from the ability to write non-procedural queries and their optimization by the system. Queries produce *exact answers* that match the contents of the database. Query processing of RDBMSs has been researched for more than 4 decades and includes extensions to more complex analysis on data warehouses. In contrast, search has not been addressed by RDBMSs.

As the use of other other data types (key-value store, column-store, and graphs to name a few) are becoming popular for modeling to match the data set characteristics, query processing and optimization are becoming important again. The approaches used in RDBMSs, such as cost-based, I/O focused may not be applicable in the same way to new models and queries. Hence, new approaches need to be developed that are suited for the data model used and the expressiveness of the queries to be supported.

This thesis addresses query processing of large graphs (or forest) and develops algorithms for query processing as well as develops heuristics for improving the response time using graph characteristics. Although search (unlike RDBMS) has received a lot of attention for graphs, query processing, in contrast, has received very little attention. With the advent of large social networks and other large graphs (e.g., freebase, knowledge and entity graphs), querying to understand the data set and retrieve relevant/exact information becoming critical.

This thesis builds on the previous work at the Information Technology laboratory at UTA (IT Lab) to scale query processing to arbitrary-size graphs (or forests) and to exploit parallelism as much as possible. Partitioning (a form of divide and conquer) and Map/Reduce (for parallel processing) are used as basic ingredients for scalability. Partitioning a graph for query processing and computing all answers poses a number of challenges: i) partitioning schemes, ii) scheduling or choosing which partition or partitions to schedule for processing, iii) developing heuristics for reducing the total response time exploiting query and graph characteristics, and iv) importantly, correctness of results.

This thesis address all of the above challenges using the map/reduce framework. The choice of map/reduce framework allows us to make partitions based on available resources and optimize parallelism based on the number of partitions to schedule at a time. We use a partitioning strategy that has been shown to be good for substructure discovery. We develop a number of heuristics that are based on query and graph characteristics. The query itself is expressed as a graph without having to cast in some other language. Relational comparison operators, Boolean operators, wild cards, and union queries are supported. There is no restriction on node and edge labels, and uniquely labeled multiple edges are supported. Extensive experimental analysis of the approach (partitioning sizes, algorithm, and heuristics) using large data sets (real-world and synthetic) are shown for speedup, scalability, and efficacy of the heuristics proposed.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Relational Database Management Systems have been dominating the computer industry for more than 4 decades mainly for storing and retrieving data and continue to do so today. They are one of the best tools for storing and organizing the structured data in tabular form. Moreover, the importance of relational database also comes from its efficient execution of the given query and optimizations by the system. Retrievals are usually accomplished using SQL, a non-procedural query language. A query optimizer is responsible for finding best cost plan to execute the given query. On the other hand, search capability where the user gives an arbitrary keyword(s) to find the matches has not been addressed by RDBMSs.

Today, numerous real-world applications generate massive amounts of data which entails the need to store the data using the chosen model and query/analyze the data efficiently. Many organizations are turning to novel data stores (key-value store, column-store, graphs, etc) that do not use relational model exclusively. Motivations includes: simplicity of design, distributed [1], horizontally scalable [2], etc. For example, Google's BigTable [3] uses column-oriented databases; Dynamo [4], a key-value storage system, is used extensively by Amazon; Neo4J [5] is a high performance and scalable graph database. Proliferation of these data types generates the need for retrieving useful information from these data. Hence, query processing and its optimizations are crucial for these new models.

1.1 Querying in Graphs

Our goal is to address the problem of query processing on graphs. Graphs are useful for structured data where relationships among entities are important. For instance, the link structure of the World Wide Web, group of friends in social networks, collaboration among authors, all have relationships that can be captured and represented in the form of a graph. Unlike RDBMSs, search has been given lots of attention in graph querying [6–8]. Searching for a subgraph generates approximate matches/answers to keep it efficient and due to not searching the entire graph. Also, search, many a times, is not processed on the graph itself. Indexing and other preprocessing is done on original graph for speeding up search. Moreover, searching has limitation for complex queries where one or more nodes are not known. We address the graph querying in a focused manner to find exact and all matches of query. We accommodate expressive query which can include range specifications, comparisons, unknown patterns with wild card specifications, AND queries, OR queries. We propose heuristics to search only the required portions of the graph.

As the amount of data is increasing, graphs are becoming larger. For example, Facebook has around 2.3 billion active users [9]. Computer science bibliography website, DBLP has over 5 million records of conferences/workshop papers and journal articles [10]. Internet Movie Database (IMDb) contains over 360 million data items with information of 123 million people and 6.5 million titles [11]. This brings us significant challenges in querying large graphs. One way to address the problem is to represent the graph as a relational database and then query it using using relational language. The drawback of this approach is that many graph queries cannot be easily expressed by relational languages such as, SQL. Moreover, when the query contains results from many tables, it will require many join operations which are expensive. Conversely, in a graph database, query processing does not have to scan the entire graph to find the nodes that meet the search criteria. It looks only at nodes that are directly connected to other nodes. Hence, it is imperative to device techniques for querying graphs in its native form.

QP-Subdue [12] is an attempt to move towards general purpose querying of graphs. It has modified a substructure mining algorithm to use a query plan to do constrained expansion for query evaluation instead of mining. It needs to load entire graph into main memory, which is unrealistic if the graph size is large. PGQP - Partitioned Graph Query Processor [13] builds upon QP-Subdue and extends it to work on graph partitions. It runs in iterations, loading a single partition into memory at a time and final answers are formed at the end after all partitions are processed by aggregating all partial answers. It is the easiest way to achieve scalability with very limited resource. We are interested in processing graph query using arbitrary number of resources to exploit parallelism. Also instead of assembling final answers at the end of processing which brings challenges in terms of keeping track of partial results, we build the answers as we process partitions and obtain complete answers by the end of processing.

1.2 Problem Statement

The problem addressed by this thesis is to process graph queries on very large graphs. The important aspects of this problem are handling arbitrary size of graphs, supporting expressive graph queries, and processing graph queries correctly and efficiently.

We propose a graph querying algorithm that is scalable for any size graph (or forest) and utilize as many resources as we have or we provide to leverage parallel processing. We use graph representation itself as the input and process it rather than using data in specific representation such as RDF [14]. Our query representation accommodates queries of varying characteristics such as comparison operators ($>, >=, <, <=, =, != $), range operators (Between, IN), wild cards (Single character match(_), Multiple character match(%)), Boolean operators in conjunction with previous ones, AND and OR queries.

As we are working with large graphs, which cannot be processed in the memory of a single processor, we partition the graph into smaller size partitions which fits in the memory. But when querying on partitioned graphs, answers may span multiple partitions which brings challenges to compute the final answers correctly. Our approach carries the partial answers to the appropriate partitions and continue from those partitions. But the efficiency of evaluating a query on partitioned graph depends on how we schedule partitions to complete the processing. Ideally, we want to load the minimum number of required partitions to answer a query. Hence, we propose a set of heuristics for choosing partitions after each iteration to minimize the total number of partitions loaded. We also formulate a load factor to assess the efficacy of a heuristic used.

Based on given resources, our approach has the ability to adjust to specified parallel processing. So we can use as many partitions and processors as needed to process in parallel. In the ideal case, we want to match the number of partitions to process with resources available for complete parallelism. But in case of limited resources, our approach still works by processing more than one partition on a single machine.

1.3 Why Map/Reduce?

Map/Reduce is a distributed paradigm that has been used to scale computations horizontally to accommodate very large data sizes. Recently, in graph analysis area, researchers have brought their interest to Map/Reduce framework to cope with vast amount of data. Many companies such as Google, Facebook, Yahoo and Amazon use Map/Reduce to process large amount of data in the order of terabytes every day. In particular, Apache Hadoop, which is an open-source implementation of Map/Reduce, has been used extensively.

We use Map/Reduce programming paradigm to leverage parallel processing. If we don't use Map/Reduce, it becomes challenging to implement our algorithm in a traditional way. Main concerns are i) how do we load and split large graphs across multiple machines, ii) where do we store graph partitions and adjacency list partitions for fast access, iii) how to handle inter process communication, iv) how to aggregate results, and v) book-keeping for correct processing.

Map/Reduce automatically parallelizes and executes the program on a large cluster of commodity machines. The runtime system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing required inter-machine communication [**?**]. Also, one of the key features in hadoop implementation of Map/Reduce is fault tolerance [15].

A Map/Reduce framework is composed of three steps:

- **map phase**: Processes a key/value pair to generate a set of intermediate key/-value pairs
- **shuffle phase**: Distribute the data based on key partitions such that all keys in a partition are sent to the same reducer
- **reduce phase**: Merges all intermediate values associated with the same intermediate key and invokes a user defined reduce function.

We can utilize Map/Reduce framework to create initial graph partitions based on resources available. We use partitioning strategy that has been proposed earlier for substructure discovery in Map/Reduce [16]. We cast our query processing computation to map/reduce. Maps have been used to process partitions in parallel. We exploit shuffle and reduce paradigm to aggregate partial results and update partitions.

1.4 Thesis Contributions

The contributions of this thesis are:

- Scalable approach to graph querying that can handle arbitrary size graphs
- Graph querying that can handle comparison operators ($>, >=, <, <=, =, ! =$), range operators (Between, IN), wild cards(Multiple character match(%), Single character match(_)), logical operators, AND queries, OR queries
- Range partitioning algorithm that takes a single pass to create adjacency list partitions and graph partitions of different sizes using the range partitioning approach
- Efficient approach to query evaluation by identifying useful heuristics and their relationship with query processing
- Extensive experimental analysis on real world and synthetic data sets using broad range of queries that validates proposed heuristics

1.5 Thesis Organization

Rest of the thesis is organized as follows -

- Chapter 2 discusses the related work in the area of graph query processing and Map/Reduce based graph partitioning.
- Chapter 3 gives an overview of graph model, query characteristics and cost based query plan. It also discusses query processing over partitioned graph.
- Chapter 4 presents our partitioned based graph querying technique and algorithm in detail.

- Chapter 5 details the implementation of all the components used for graph querying in the Map/Reduce environment.
- Chapter 6 shows experimental results to validate our approach
- Chapter 7 outlines conclusions and future work.

CHAPTER 2

RELATED WORK

Graph Querying – used to find useful information from a large graph, uses expressive queries formulated using wild cards, range expressions, and Boolean/logical operators. For querying a graph database one needs a query specification that is easy to understand and has low or no learning curve. As mentioned earlier, graph sizes can be very large and hence traditional systems that use main memory approach are not useful. The alternate querying approaches keep the graph on disk to be staged into memory as needed. Disk-based approaches [17] are good to deal with graph sizes larger than possible in memory, but introduce difficulties for providing customized buffer management and introduces I/O latency. As the scale of the data concerned grows, traditional systems become obsolete, and it is imperative to adopt a distributed approach to data handling and processing. There are distributed graph databases available for large scale processing - for example Neo4J [5], Titan [18], Infinitegraph [19] - but they are aimed at developing the storage side of graph data and not seek to improve query processing in graph database. Also they are not directly relevant to the problem being addressed here (partitioned graph query processing and optimization) in this thesis as we have to write programs for computing results. This chapter provides brief overview of some of the widely used approaches for graph query processing.

2.1 Existing techniques for searching graphs

Searching for a particular subgraph matching the query means we are interested in finding exact/similar patterns in the graph. Subgraph search is well explored area and given lots of attention. It employs the mechanism of graph indexing and query processing techniques [20]. Several mechanisms have been proposed for different graph query types in a general mode.

2.1.1 Graph-grep

The Graph-grep [6] is a variable path index approach. It enumerates all paths up to a certain maximum length and records the number of occurrences of each path. In the query processing, the path indexes is used to find a set of candidate graphs which contains the paths in the query structure and to check if the counts of such paths are beyond the threshold specified in the query. In the verification step, each candidate graph is examined by subgraph isomorphism to obtain the final results. Because of all the paths are hashed, this approach provides the fast results. However, the size of the indexed paths could drastically increase in line with the size of graph database.

2.1.2 G-Index

The G-Index technique [7] makes use of frequent subgraphs as the basic indexing unit. The main observation of this approach is that graph-based index can significantly improve query performance over a path-based one. Any subgraph is considered as being frequent if its support is greater than a minimum support threshold. Given a query graph q, if q is a frequent subgraph, the exact set of query answers containing q can be retrieved directly since q is indexed. Otherwise, a candidate answer set of query q is retrieved and verified. 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.

2.1.3 Fg-index and Fg*-index

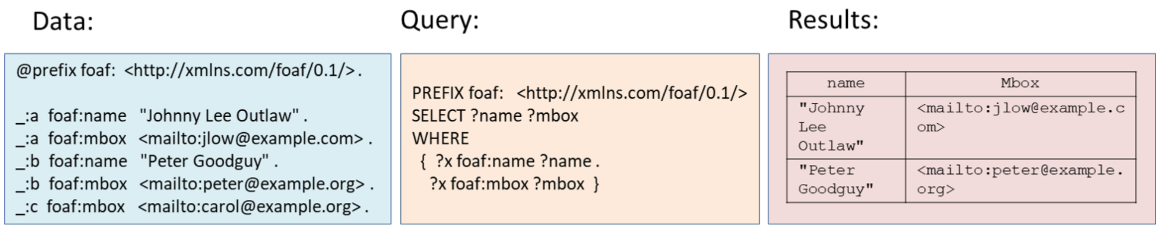
Cheng et al. [21] have extended the ideas of G-Index by using nested inverted-index in a new graph index structure named Fg-Index. In this index structure, a memory-resident inverted-index is built using the set of frequent subgraphs. A disk-resident inverted-index is built on the closure of the frequent graphs. If the closure is too large, a local set of closure frequent subgraphs can be computed from the set of frequent graphs and a further nested inverted-index can be constructed. Another edge-index can be also built on the set of infrequent distinct edges in the graph database. Cheng et al. [22] have extended their work further where they introduced the Fg*-index which consists of three components: the Fg-index, feature-index and the FAQ-index. The Fg-index allows the set of queries that represent frequent graphs to be answered without candidate verification. The feature-index is used to reduce the index probing cost by using the features to filter false results that are matched in the Fg-index. The FAQ-index is dynamically constructed from the set of frequently asked non-frequent subgraphs. Hence, verification is not required for processing frequently asked queries and only a small number of candidates need to be verified for processing non-frequent graph queries that are not frequently asked. However, it requires computationally heavy index construction phase before processing queries. In our approach, instead of indexing we use an exploration approach. A node is fully explored and, with the exception of required nodes every other node is discarded.

2.1.4 G-Ray

Another approach presented by Gallagher, Faloutsos and Eliasi-Rad [8] called G-Ray finds both exact and inexact matches. This approach first finds a seed node and then expands the seed node by finding a matching node followed by bridging both nodes by the best possible path. G-Ray proposes a goodness score which is a measure of proximity between two nodes. Based on this goodness score, it ranks the results. In this approach each vertex stores the information of remaining vertices. Therefore, space requirement is significant and it also does not differentiate between two results having the same goodness score. This approach finds inexact matches whereas our interest is to explore graph only for exact matches of query in focused manner.

However, due to the diversity of graph models as well as the complexity of graph processing, some of the existing graph indexing and query processing techniques are designed solely for specific application domains. Also subgraph search has limitations in finding the complex graph queries where one or more nodes are not-known (i.e. queries containing wild cards) and also do not support queries containing operators $(<, >, =, ! =)$. It doesn't support query in its generality.

Most of the graph indexing strategies requires computationally expensive index construction phase. Also the number of indexing features should be as small as possible to keep the whole index structure compact so that it is possible to be held in the main memory for efficient access and retrieval. However, the graph database can be very large making it inefficient in index construction and index structure can become large enough not to fit in memory. Thus, finding an efficient search technique is immensely important.



Data:

```
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name   "Johnny Lee Outlaw" .
_:a  foaf:mbox   <mailto:jlow@example.com> .
_:b  foaf:name   "Peter Goodguy" .
_:b  foaf:mbox   <mailto:peter@example.org> .
_:c  foaf:mbox   <mailto:carol@example.org> .
```

Query:

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE
  {  ?x foaf:name ?name .
    ?x foaf:mbox ?mbox  }
```

Results:

name	Mbox
"Johnny Lee Outlaw"	<mailto:jlow@example.com>
"Peter Goodguy"	<mailto:peter@example.org>

Figure 2.1: SPARQL query pattern

2.2 Query languages for graphs

Graph querying has been studied for specific graph data models by formulating different query languages [23].

2.2.1 SPARQL

SPARQL query language [14] is a W3C recommendation for querying RDF (Resource Description Framework) graph data. RDF is a directed, labeled graph data format for representing information in the Web. It describes a graph by a set of triples, each of which describes a (attribute and value) pair or an interconnection between two nodes. The SPARQL query language works primarily through a primitive triple pattern matching techniques with simple constraints on query nodes and edges.

Figure 2.1 shows the use of SPARQL for matching patterns. In this example, the data in RDF form and the SPARQL query is given. As shown in query, SELECT clause specify the variables we wish to project as output. The WHERE clause then captures two triple patterns (delimited by periods) that correspond to the edges of the graph. The results for this query are shown in figure.

2.2.2 Cypher

Cypher [24] is Neo4j's [5] graph query language, that allows for expressive and efficient data querying in a property graph. Patterns are expressed syntactically following a "pictorial" intuition to encode nodes and edges with arrows between them.

The query "Find friend(s) of bob" would be written in cypher as:

```
MATCH (you {name:\Bob"})-[:FRIEND]->(yourFriends)
RETURN you, yourFriends
```

MATCH clause specifies the pattern. Nodes are written inside "( )" brackets and edges inside "[ ]" brackets. The RETURN clause can be used to project the output variables.

However, above mentioned applications requires query to be represented in specific language model. Hence writing structured queries requires extensive experiences in query language and data model, as well as a good understanding of the particular dataset.

2.3 Previous work at IT Lab

Retrieving useful information from graph data by using expressive query has been studied recently and some efforts has been done to scale the querying to large graph by using partitioned approach.

2.3.1 QP-Subdue

QP-Subdue [12] is a main memory based approach to graph query evaluation. It is an attempt to move towards general purpose querying of graphs. It uses cost based plans that are tailored to the characteristics of the graph database for efficient execution of the given query. Metadata is collected *once* pertaining to the graph database and cost estimation is done to evaluate the cost of execution of each plan.

Extensive experiments on different types of queries over two graph databases (IMDb and DBLP) are performed to validate the approach. Subdue a graph mining algorithm has been modified to process a query plan instead of performing mining. It has proposed a query representation scheme for various types of queries which can be used to retrieve meaningful information from a graph databases. It can handle queries with all comparison operators ($<, <=, >, >=, =, ! =$), logical operators including AND and OR as well as wild cards.

Query processor here 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. 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. Since, QP-Subdue uses main memory to construct the graph, it can process queries on small size graph database restricted by available main memory.

2.3.2 PGQP (Partitioned Graph Query Processor)

PGQP [13] extends query processing to partitioned graph. It can process queries on any graph size by partitioning the graph database into smaller size which fits into main memory by using existing partitioning techniques (METIS, KaHIP) and process queries on these partitions. PGQP modify existing QP-Subdue to make it work on graph partitions by loading one partition at a time. 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. It has 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. Efficient query evaluation is done by choosing the partitions as well as the order judiciously, hence minimizing the total number of partitions that are needed for processing a query or a set of queries. Experiments over three different graph databases (DBLP, IMDb, and Synthetic) were carried out to validate the approach and provide some insights into the metrics gleaned from partitioning schemes on query processing.

PGQP has established scalability through independent processing of partitions. To sequence the loading of partitions, query processor maintains Start Node Information (SNI) file. SNI file contains information about partition id, occurrences, start label and vertex ids of start node of the query plan. Correctness is ensured by updating SNI file correctly when answers span into new partition. Heuristics that are formulated based on number of start/ continuation nodes in each partition from SNI file are: i) MAX-SN-heuristic, ii) MIN-SN-heuristic and iii) RANDOM-SN-heuristic. It also evaluated effect of connected components in each partition and proposed heuristics: i) MAC-CC-heuristic, ii) MIN-CC-heuristic and iii) RANDOM-CC-heuristic. Experiments performed on synthetic graphs to evaluate proposed heuristic using different queries and partitioning schemes and empirically validated that MAX-SN and MIN-CC independently are better than other heuristics.

2.4 Graph partitioning using MapReduce

Recent advancements in graph mining [16] have used distributed processing for large graph data. The traditional main memory based approaches are extended to work on graph partitions for parallel processing. MapReduce distributed framework is used to scale the processing to cluster of commodity machines.

Substructure discovery using MapReduce: Process of substructure discovery in distributed environment starts by dividing the input graph into smaller partitions and then combining the results across partitions effectively. An iterative algorithm is used for substructure discovery that: generates all substructures of increasing sizes (starting from substructure of size one that has one edge), eliminates duplicates if necessary, counts the number of identical (or similar) substructures, applies a metric (e.g. frequency) to rank the substructures. Adjacency list is used for expanding substructures by one edge in each iteration. This process is repeated until a given substructure size is reached or there are no more substructures to generate. In each iteration, either all substructures or a subset of substructures (using the rank) is carried forward updating substructure partitions.

As the Mapper process each input records individually, input graph is represented in the form of sequence of graph edges containing edge label, source and destination vertices. Initial substructure partitions are disjoint (same edge is not repeated across multiple partitions). Adjacency list is also partitioned according to substructure partition. After the first iteration, new edges are added to the existing substructure partitions. It presented two algorithms based on two substructures partitioning techniques: i) Arbitrary Partitioning, ii) Range Partitioning.

Arbitrary Partitioning:

Initial substructure partitions are generated based on random grouping of graph edges. The initial substructure partitions are disjoint but adjacency list partitions are not. Each Mapper expands substructures in parallel using corresponding substructure and adjacency list partition. Counting is done in reducer by grouping isomorphic substructures. Newly generated substructures are written to same partition and sent for further iterations. As new substructure partitions can include edge that spans multiple partitions, adjacency list is updated to include new vertices with their adjacency list. The quality of partitioning can be improved by reducing edge cut set.

Range Partitioning:

It employs the mechanism of partitioning the global adjacency list by dividing vertex ids in equal range for creating adjacency partitions. Unlike the previous approach, the adjacency partitions are disjoint here. Two chained MapReduce jobs are used where first one is used for substructure partitioning and expansion and second job does substructure calculations. Adjacency list partitions are kept fixed across iterations needing to route substructure to processor holding the corresponding adjacency list partition for expansion. Range info used help in quickly determine to which partition the substructure belongs.

The experimental analysis of both partitioning tells that although Range partitioning involves shuffle cost, it performs better than the Arbitrary partitioning which includes I/O intensive update cost. Although traditional partitioning scheme such as METIS reduces the edge crossover(cut-set) between partitions, they make multiple passes over the graph to create initial partitions which are not suitable for large graphs. Moreover, these partitioning schemes don't support adjacency list partitions. Also they focus on creating equal sized connected partitions and hence can create different workloads across the heterogeneous machines hampering speedup. Note that the initial partitioning cost for above partitioning are less then traditional ones as it needs a single pass over the graph. Moreover Range partitioning is also suited for a heterogeneous environment by correlating the ranges to the capacity of each machine in the cluster.

Taking inspiration from this earlier work on graph mining, we propose graph querying algorithm that works on MapReduce distributed environment. We employ range partitioning for creating initial partitions with different range values for the purpose of heterogeneous environment by making one pass over the graph. Unlike the above work, partitioning is also done using a single MapReduce iteration. In addition to partitioning, adjacency list generation, some graph characteristics are also generated (e.g., number of connected components in each partition, number of strart nodes of known queries, etc.) As graph querying means finding only specific nodes in whole graph according to query, constrained expansion is done according to query plan in second MapReduce job in iterations.

CHAPTER 3

PRELIMINARIES

3.1 Graph Model

Graph is a powerful data structure for explicitly capturing relationships between entities in various application domains. Data is modeled as graph by representing entities as vertices and relationships as edges. A vertex typically has a vertex label and a unique vertex identifier(vertex id). An edge is a connection between two vertices which can be either labeled or unlabeled, directed, or undirected. Here we focus on the labeled graphs where node and edge labels are not assumed to be unique. Figure 3.1 shows an example of IMDb graph with node and edge labels.

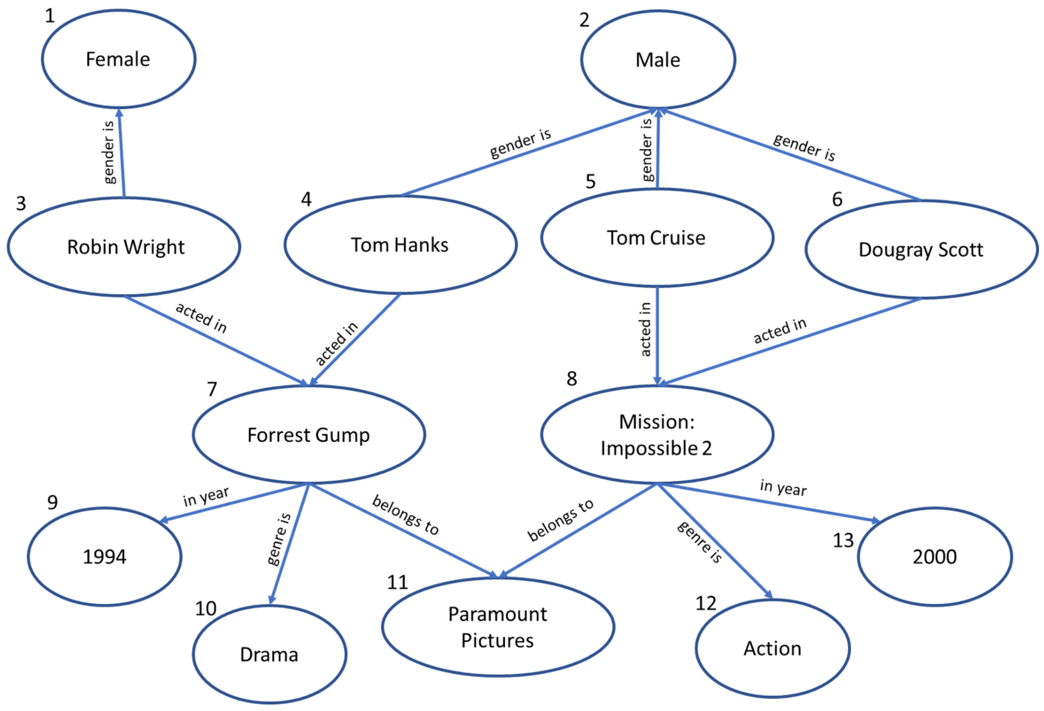


Figure 3.1: Portion of movie information graph

3.2 Graph Query and Query Plan

A query is a subgraph or a pattern (typically small compared to the graph) which may have multiple matches in the data graph. A query is input to the graph query processor and all the matching patterns are returned. In the graph database model, several types of queries can be specified. We support the following operators as part of queries.

1. Comparison operators (=, <, >, >=, <=, !=)
2. Union queries (using OR)
3. Boolean operators AND, OR, and NOT along with comparison operators
4. Wild card match (LIKE): Multiple character match (%), Single character match (_)
5. Range queries (BETWEEN, IN operator)

A general graph query processing system should be able to answer queries that contain the above operators. Figure 3.2 shows our representation of a graph query. In this representation, all nodes in a query follows the form **attr op value** where op is one of the above mentioned operators and attr is either a type or an instance. Depending on what information we need to find, value can be single match, list of matches, range specifying lower and upper bound, pattern containing wild card character or question mark ? (to find all matches).

A query answering system can start from any node in the query and expand according to the sequence of edges specified in the query graph until all matches are found. In previous work, cost-based query plan is used which gives the start nodes of query and an ordering of nodes in which each node is expanded with the desired label. However, to generate query plan it needs catalog information such as type nodes and their instances. But in real world, we don't have those information for all datasets. Hence, we have a used simple approach to generate a query plan avoiding

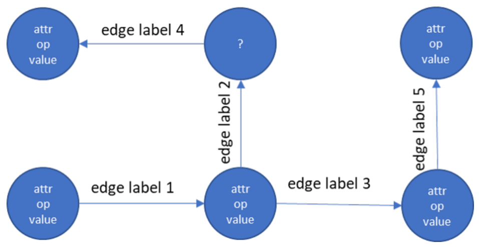


Figure 3.2: A General Graph Query and its representation

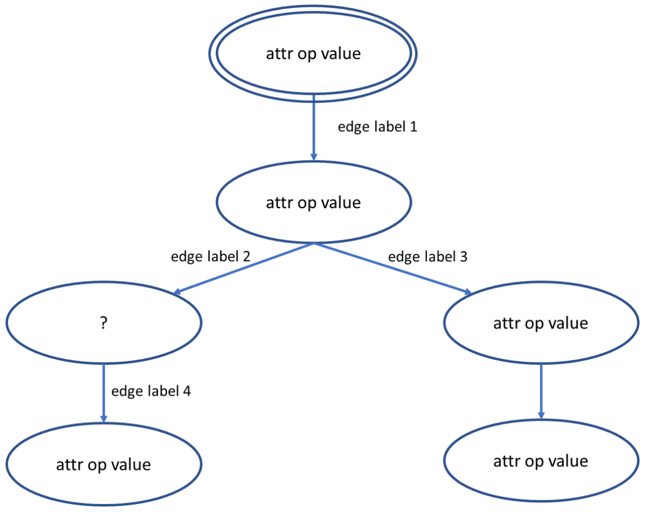


Figure 3.3: A Query plan for the Query Shown

the question marks (?) as start nodes. General query plan can be represented as shown in Figure 3.3.

This general query and plan representation can support all above mentioned operators. Also, there is no need to learn a query language making it easier to construct queries and understand what results it should produce. A GUI can be developed for entering a query similar to QEB (or Query-By-Example [25]). We now discuss how we address each type of operators in our query evaluation approach.

Comparison operator: The graph nodes are compared based on operator present in query plan (either =, <, >, >=, <= or !=). In the cases where query node has '?', all the graph nodes are taken into consideration. Moreover, in case of IN operator, the comparisons are done using the list of values given. For instance, figure 3.4 is graph representation of query "Find movies by company American Broadcasting Company (ABC) in 1959 where the genre should not be Drama.". One of the query plans for this query is shown in Figure 3.5. Here, the query plan node with question mark (?)

will find all the movies and node with (! = Drama) condition will find only those movies which doesn't belong to "Drama" genre.

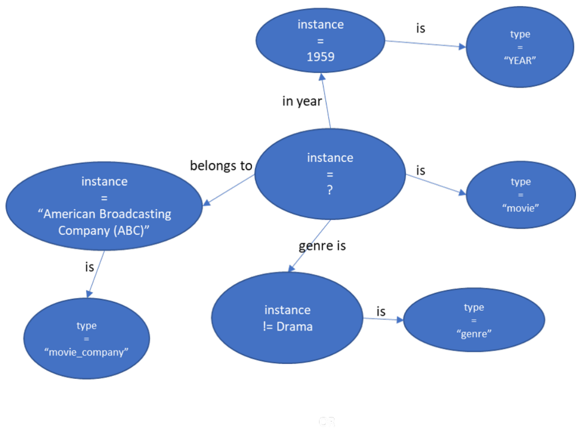


Figure 3.4: A Graph Query With comparison operator

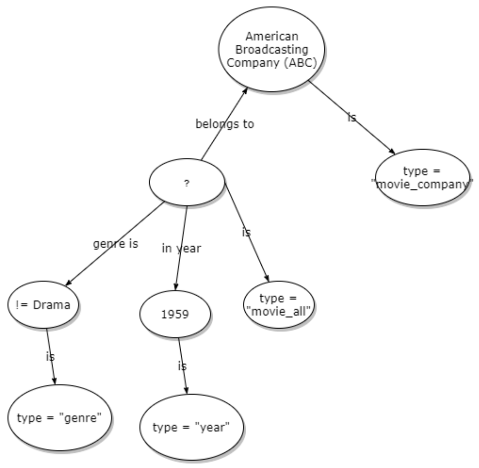


Figure 3.5: One of Its Query plans

Wild card match (LIKE): Graph node is compared using wild card pattern present in query plan.

- **Multiple Character Match (%):** The % symbol as part of a string constant will match zero or more characters. For example, 'John %' can match all author names 'John S. Edwards', 'John R. Durrett', 'John W. Priest', etc.
- **Single character match (_):** The _ symbol matches exactly one character. For example, 'Kill Bill: Vol. _' pattern matches with all movie titles 'Kill Bill: Vol. 1' and 'Kill Bill: Vol. 2'

As an example, "Find all conference papers along with the conference names that are published in 2005 by the author whose first name starts with 'John'.". The graph representation of this query shows the wild card (%) in Figure 3.6. The query plan is also shown in figure 3.7.

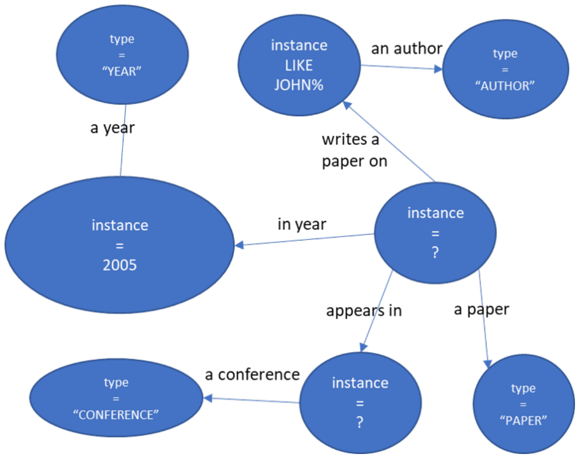


Figure 3.6: Graph query (wild card)

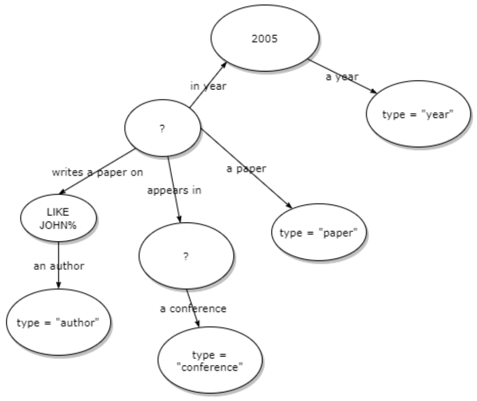


Figure 3.7: Query plan (wild card)

AND query: When the query graph has multiple edges (or paths) from a node, it corresponds to an AND operator. All conditions that are in conjunction with AND translates to different query edges from the same node. When we evaluate this query, the results should match all the query paths. For example, "Find movies produced by "Adam Sandler" where the genres should be comedy AND Sci-Fi (a movie can belong to multiple genres) but the year should not equal to 2000". As shown in figure 3.8, the query graph has AND operator comparing genre Comedy and Sci-Fi. The query plan (figure 3.9) will have multiple paths for both different genres.

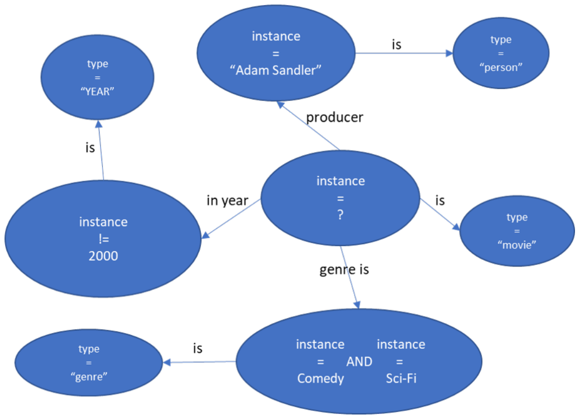


Figure 3.8: Graph query (AND query)

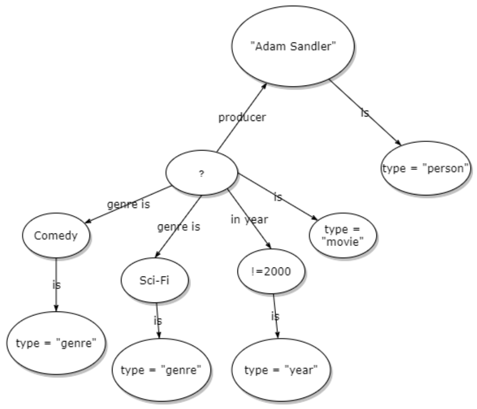


Figure 3.9: Query plan (AND query)

OR query: As OR operator finds answers which satisfies either one of the given conditions, the result would be the union of all the OR conditions. So the query gets translated to individuals plans connected by a UNION operator. For example, Find all actors who have worked in a movie owned by company Paramount Pictures where the genre of the movie is either Drama or Action.". The query graph (figure 3.10) shows the OR operator between Drama and Action genre. In this case, both the genre conditions in conjunction with OR operator are denoted in two different query plans. In this query, results for both the genre would be included in the answer set.

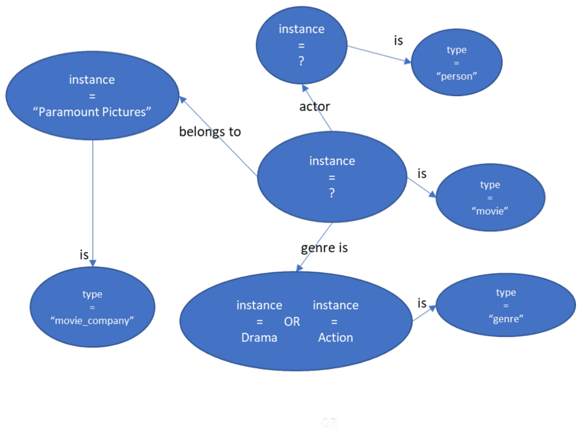


Figure 3.10: Graph query (OR query)

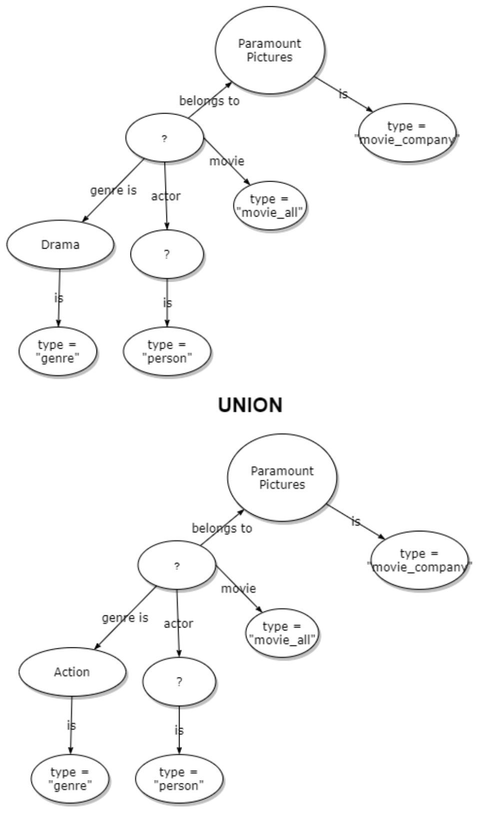


Figure 3.11: Query plan (OR query)

BETWEEN operator: Given two bounds in query, we find graph nodes whose value is within the given range. For example, "Find papers which are published between 1990 and 2015 and the paper name includes the term "Transaction Management in Multidatabase Systems"". The range with BETWEEN operator is shown in graph query figure 3.12. Here, the query plan (figure 3.13) will find all years that are within 1990 and 2015.

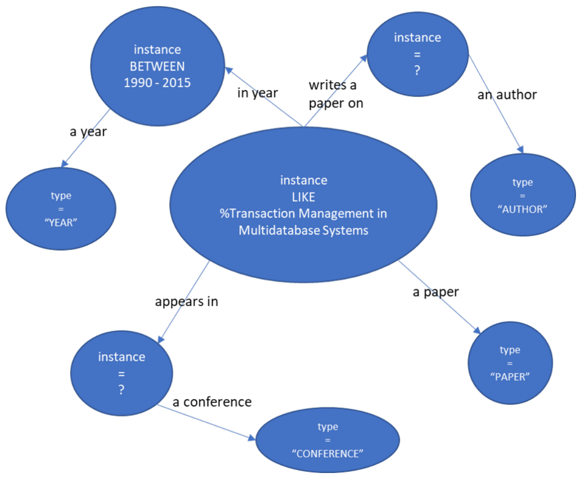


Figure 3.12: Graph query (BETWEEN operator)

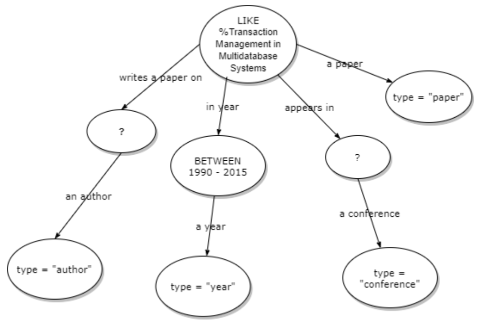


Figure 3.13: Query plan (BETWEEN operator)

3.3 Adjacency List

The adjacency list of any vertex in the graph gives a set of neighbors for that vertex. Table 3.1 shows the adjacency list for all the vertices present in the graph in Figure 3.1. The adjacency list representation allows us to traverse all the neighbor nodes through edges from a particular node and find the matches. We use the adjacency list representation to expand the substructures to the desired node according to query plan.

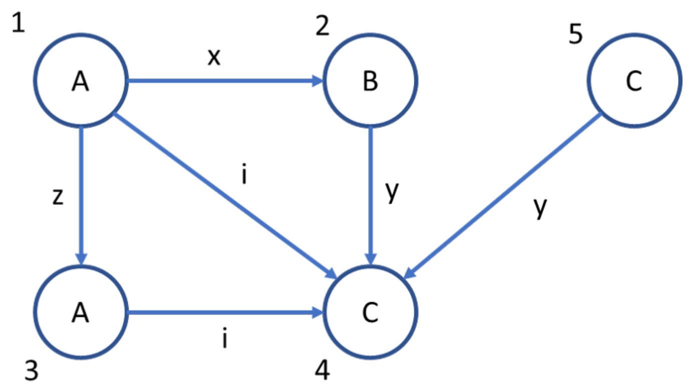


Figure 3.14: Input graph example

Vid	**Adj List**
1	(x,1,A,2,B), (z,1,A,3,A), (i,1,A,4,C)
2	(x,1,A,2,B), (y,2,B,4,C)
3	(z,1,A,3,A), (i,3,A,4,C)
4	(i,1,A,4,C), (y,2,B,4,C), (i,3,A,4,C), (y,5,c,4,c)
5	(y,5,c,4,c)

Table 3.1: Adjacency List

3.4 **Querying Partitioned Graph:**

In our approach, a given graph G is partitioned into p partitions (G1, G2,.., Gp) such that each partition fits into available main memory and all the p partitions can be combined to form the original graph. Partition sizes (number of edges in each partition) need not be same.

Query processing in the presence of the partitions starts from the partition/s containing start node of the query plan. So the first step in query evaluation is to identify all the partitions that contain start label of the query plan. From these eligible partitions, we get all instances of start nodes and continue to desired nodes according to query plan. For a partitioned graph, answer nodes can either be in same partition or from different partitions.

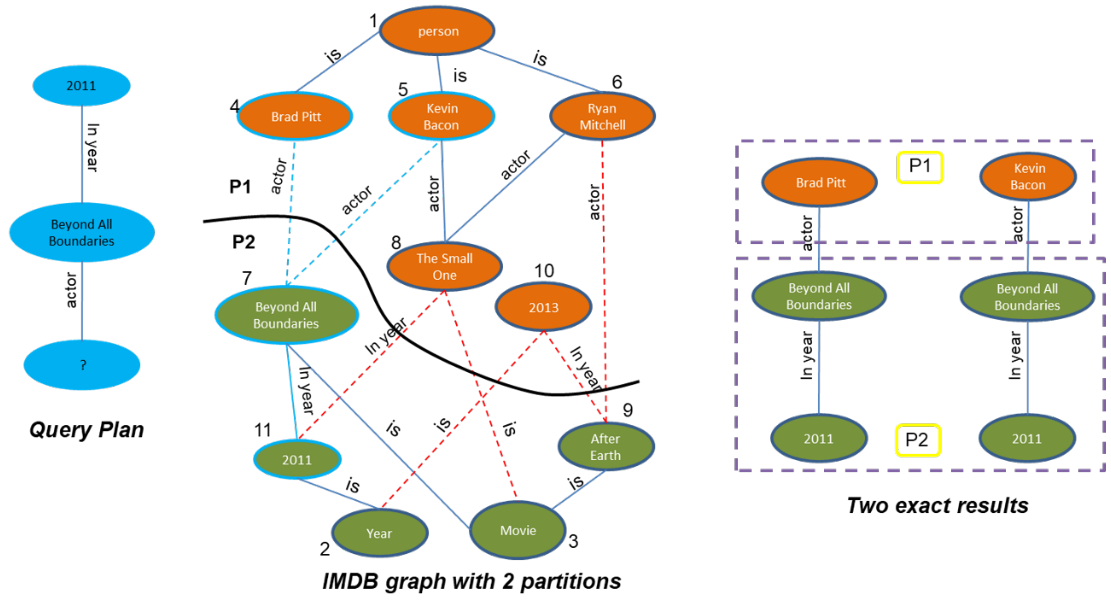


Figure 3.15: Graph querying on partitioned graph

Figure 3.15 shows the IMDB graph partitioned into 2 graph partitions. The query of our interest is “Find persons who have worked in the movie **“Beyond all boundaries” in 2011”**. We use *a* query plan as shown in figure. Here, the start node of the query is 2011. It is present in the graph partition P2. From the start node instance, we find exact matches according to query plan. Here, the query plan nodes are present in 2 different partitions. The two exact results of the query plan will have nodes from both partition P1 and P2.

3.5 Start nodes in each partition

The nodes which matches with start node label of a query plan becomes initial start nodes in that partition. Constrained query expansion in that partition is done only on those start nodes. If the answers continue from one partition to other partitions, the continuing nodes becomes start nodes in those partitions. Start node information allows us to process only partitions that have a plan start node label. Note that this information is obtained during the partitioning of the graph along with adjacency list generation. We will discuss how to process new queries later in the thesis.

3.6 Connected components in each partition

Input to our query processing systems is a graph or a forest. Each graph before partitioning is a single connected component. 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.

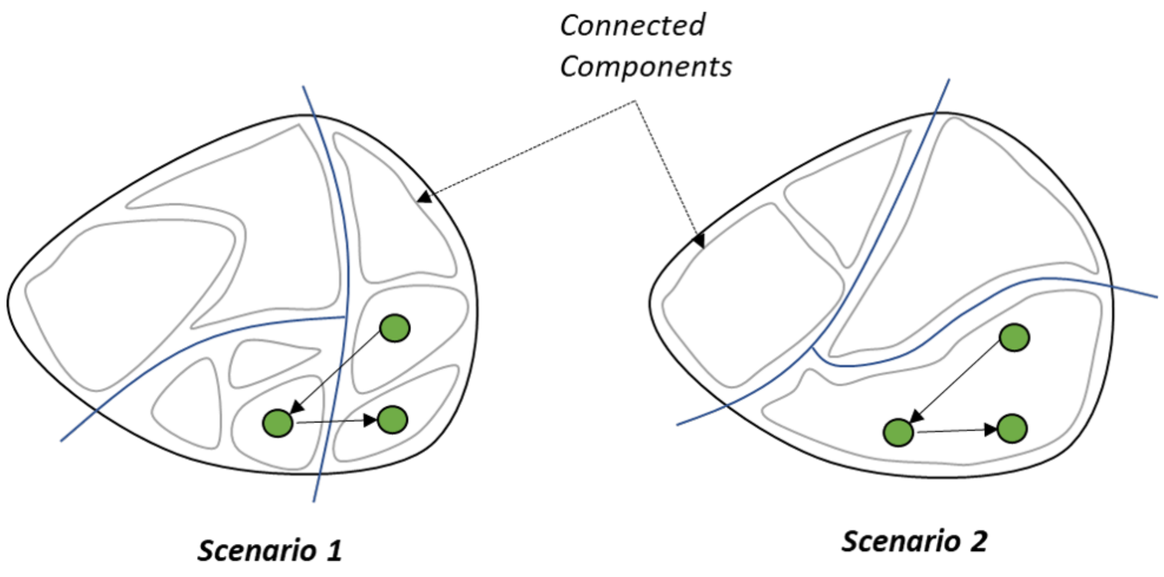


Figure 3.16: Connected components inside graph partitions

Figure 3.16 shows two different scenarios of how partions and connected components inside partitions affect query processing. In scenario 1, graph partitions contains multiple connected components in each partition. On the other hand, in scenario 2, two partitions have one connected component and the other partition has two connected components. So depending on the partitioning strategy, resulting graph partitions can be either more or less disconnected. Disconnected partitions are not suited for query processing as answer nodes within the same partition cannot be reached without going to another partition and coming back. This increases the number of times a partition need to be loaded. We use this information as part of the heuristics to minimize the number of partitions loaded.

If the graph partition is a single connected component, it increases the chances of query answer to stay inside the partition (scenario 2). But as shown in scenario 1, if the graph partition is more disconnected, answers may span multiple partitions.

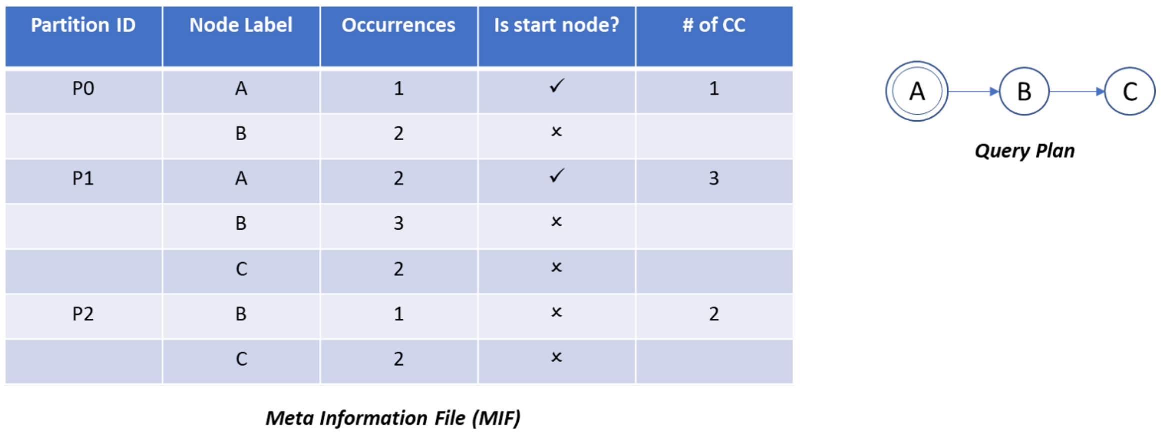
3.7 Meta Information File (MIF)

We collect meta information for each partition that helps us to quickly identify which partitions to choose, in each round, for processing a query. The collected meta information will be used for efficient execution of given query by applying heuristics, which we will discuss in details later in this thesis. Figure 3.17 shows the representation of meta information file(MIF) for a given query plan.

Information collected as part of meta information in each partition are:

Number of start nodes: The count of all the nodes which matches with query plan start node label is recorded.

Number of intermediate nodes: If the graph node label matches with query plan node label other than root, than it is counted as intermediate node. Occurrences of all intermediate nodes are collected in MIF.



Partition ID	Node Label	Occurrences	Is start node?	# of CC
P0	A	1	✓	1
	B	2	×	
P1	A	2	✓	3
	B	3	×	
	C	2	×	
P2	B	1	×	2
	C	2	×	

Meta Information File (MIF)

A
B
C
Query Plan

Figure 3.17: Meta Information File representation

Number of connected components: Connected components are calculated once before query processing during graph partitioning and we store count of connected components in each partition as part of MIF.

CHAPTER 4

DESIGN

A Graph querying system retrieves all or k exact matches of the given query graph on the larger graph (or graph database). It should support expressive queries with a diverse operator choices. In this chapter, we discuss our approach on processing graph queries on large graphs or graph databases. We discuss challenges in our approach and result verification using a controlled setting. Also, heuristics proposed for efficient evaluation of queries are discussed. Finally, we present overall algorithm with detailed explanation for graph query processing.

4.1 Overview of Design

In our design, we use a graph query plan for a given graph query that makes it clear query start nodes and subsequent nodes to expand for obtaining all answers. The important aspects of our design are:

Scalability: We address scalability by employing the "Divide and Conquer" strategy to handle large graphs which can't fit in memory of a single machine. Single input graph (G) is divided into p smaller graph partitions ($\mathrm{G}_0, G_1, \ldots, G_P$).

Resource utilization: To leverage parallel processing on graph partitions, we process queries using k processors at a time (where k can vary from 1 to p). We can increase or decrease the value of k based on resources availability. For example, if we have enough resources, than we can increase k as needed all the way to p.

Efficient query evaluation: Query processing in partitioned graphs require us to schedule partitions to get good response time. This calls for ranking partitions after

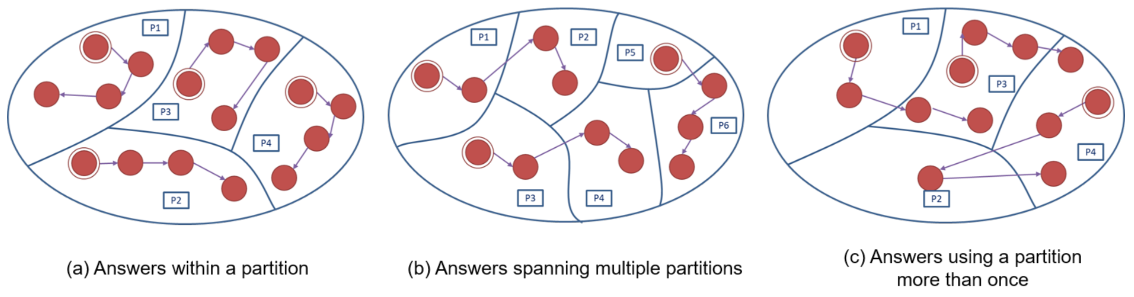

P1
P3
P4
P2
(a) Answers within a partition
P1
P2
P5
P6
P3
P4
(b) Answers spanning multiple partitions
P1
P3
P4
P2
(c) Answers using a partition more than once

Figure 4.1: Challenges for Querying Partitioned Graphs

every iteration to choose top-k partitions to process. We propose several heuristics for efficient query evaluation (by choosing which partitions to schedule) using initial and runtime meta information.

4.2 Challenges

Processing a query on partitioned graph is very different as compared to processing a query on single graph. The three cases that need to be addressed for answering queries correctly are shown in Figure 4.1.

1. All answers are within single partition (Figure 4.1 (a))
2. Answer span more than one partition (Figure 4.1 (b))
3. Answer using a partition more than once (Figure 4.1 (c))

In addition to the above, there are additional challenges to make sure answers are correct in the presence of non-unique node and edge labels. Below, we discuss different scenarios where we have to take measures for correctness of query evaluation.

Handling overlapping answers: Figure 4.2 shows an example of answer instances that overlaps. That is, the vertex or edge is part of more than one answer. Here, our query processing should identify all the instances as separate answers for correct result verification. Note that node B in partition p2 needs to be expanded twice once for each answer to get correct answers.

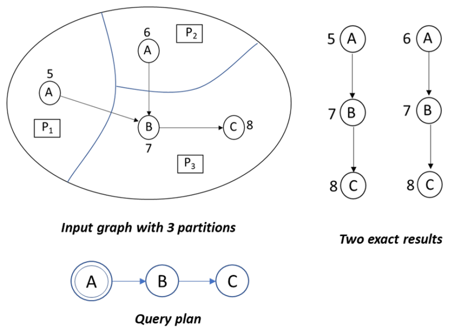


Figure 4.2: Overlapping answers

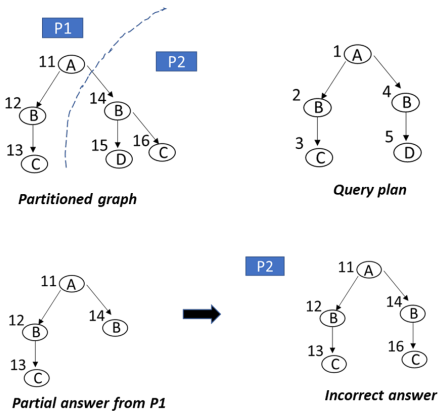


Figure 4.3: Issue with multiple path query plan

Multiple path query plan: Based on query characteristics, query plan can have more than one path from top to bottom. If the plan contains multiple traversal paths, then the complete answers should match the query edges present in all paths. Figure 4.3 shows multiple path query plan and issues with processing it. The partial answer found in partition P1 continues for node $< 14, B >$ in partition P2 and becomes start node in that partition. Then we load and process all start nodes in partition P2 and as we expand from root of query plan, the node $< 14, B >$ may get expanded for edge $B- > C$ from the query plan and the final answer that we get would be incorrect. So without proper mechanism, we may find incorrect results.

Here we need to not only keep track which node to expand, but how it needs to be expanded with respect to query plan.

Duplicates in query expansion: Systematic expansion of query from start nodes may lead to generation of duplicates. Hence, duplicate elimination is needed to reduce the cost of query expansion and to find correct answers. Figure 4.4 shows an example of how duplicates are formed during query expansion. The circled instances highlighted in same color have the same set of vertex ids and edges between them. In the presence of duplicates, query processing not only uses resources, but also gen-

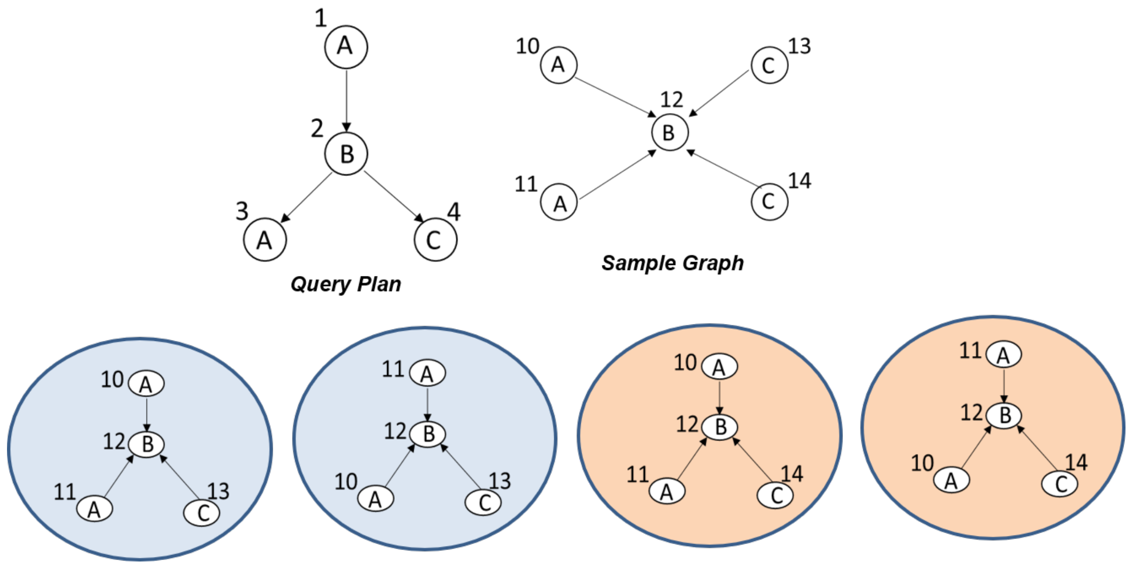


Figure 4.4: Duplicates in query expansion

erates incorrect answers. Therefore we need a technique to identify duplicates and eliminate them as early as possible.

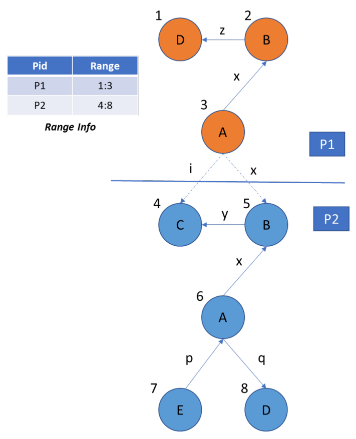
4.3 Overall approach

As we are considering large graphs in this thesis we assume that the entire information needed (graph, adjacency list of the graph) cannot be stored in the memory of a single machine. Hence partitioning the graph among multiple processors is needed where each partition can be held in the memory of a single processor. Previous work on graph query processing (PGQP [13]) has used well known partitioning schemes METIS [26] and KaHIP [27].

We take inspiration from earlier work on MapReduce-based substructure discovery [16] to divide the graph into ranges using vertex ids. This earlier work conducted extensive experiments and cost analysis and established improved response time as compared to other partitioninig schemes including METIS and KaHIP. Moreover, the partitioning can be done in a single pass of the graph. There is no constraint on the number of partitions (as in METIS as a power of 2). Also, the range of partitions or partition sizes need not be same. This allows us to match partitions to available resources.

4.3.1 Range based partitioning

First phase of our approach is range partitioning. We use contiguous range of vertex ids to partition the large graph and create adjacency list for each partition. In the example given in Figure 4.5 , to partition the graph into two partitions using given range information, we assign first 3 nodes (vertex ids 1 to 3) to partition 1 and other 5 nodes (vertex ids 4 to 8) to partition 2. And with this, adjacency list for those two partitions are created as shown in Table 4.1 and 4.2.



Pid	Range
P1	1:3
P2	4:8

Range Info

Figure 4.5: Range Partitioning

Vid	Adj List
1	(z,2,B,1,D)
2	(z,2,B,1,D), (x,3,A,2,B)
3	(x,3,A,2,B), (i,3,A,4,C), (x,3,A.5,B)

Table 4.1: Adjacency Partition 1

Vid	Adj List
4	(i,3,A,4,C), (y,5,B,4,C)
5	(x,3,A.5,B), (y,5,B,4,C), (x,6,A,5,B)
6	(x,6,A,5,B), (p,7,E,6,A), (q,6,A,8,D)
7	(p,7,E,6,A)
8	(q,6,A,8,D)

Table 4.2: Adjacency Partition 2

As the partitioning takes one pass on the graph, we utilize this to generate meta information to establish heuristics to use during query evaluation. The information collected include, query plan start nodes in each partition and intermediate query plan nodes in each partition, connected components in each partition. Partitions generated after range partitioning contains initial substructures in that partition and separately its adjacency list. As query expansion on these substructures needs corresponding adjacency list partition, we need to direct this partition to the processor holding the appropriate adjacency list. Without an order among vertex ids across the adjacency partitions, this operation requires the complete information of all vertices and their adjacency partitions. This information of all vertices will be large and can not be stored in the memory of a single processor. Range information can help us direct a partial answer to the correct partition in which the adjacency list of node is stored. Also, as we use mapping between partition id and vertex ids range as range information, this gives us fine control over size of the partitions for the heterogeneous processing environments.

4.3.2 Query Processing

Based on given k value, we choose top-k partitions using heuristics and evaluate a given query (actually its query plan) in parallel.

Constrained Query Plan Expansion: In our approach, a query plan is used to determine the order in which nodes should be expanded to obtain the answer instances that match the given query. Constrained query expansion in a partition starts from initial instances of the start nodes and then we expand all instances one edge at a time according to query plan. As we expand, if the query edge goes out of current partition, we update the corresponding partition with the partial answer instance.

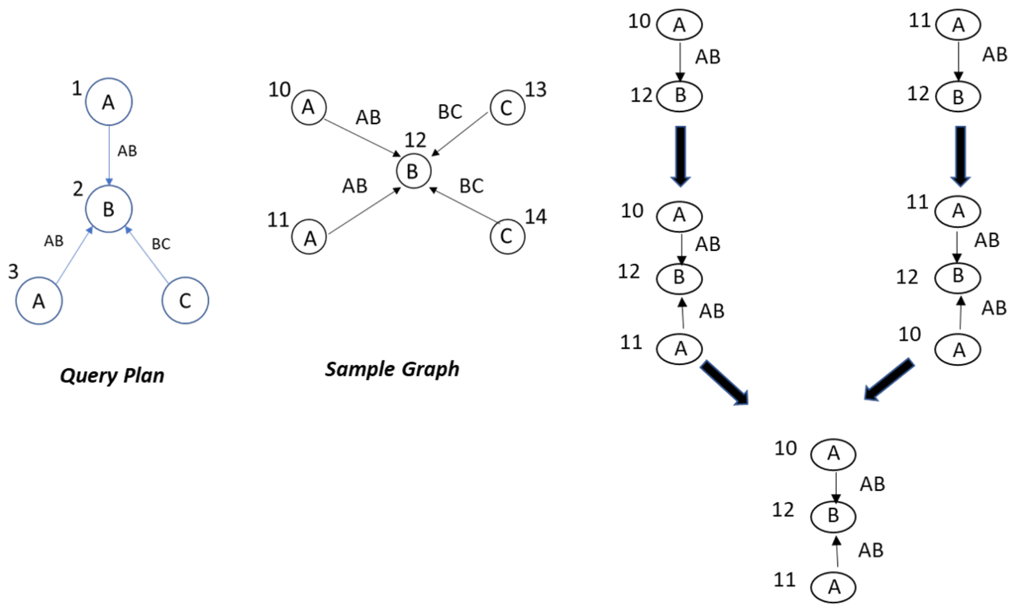


Figure 4.6: Canonical ordering of instances

We do so until no more edges to visit in the query plan or no more answers to process in that partition.

Removing duplicates: Duplicates have the same vertex ids and same connectivity among those vertex ids. To remove duplicates we convert the expanded instance to its canonical form. We employ a lexicographic ordering on edge label for canonical form. If there are multiple edges with the same edge label in a substructure, they are ordered on the source vertex label. If source vertex label is also same, they are further ordered on the destination vertex label. If edge label, and vertex labels are also identical, then source and destination vertex ids are used for ordering.

Figure 4.6 shows an example of how lexicographical ordering during expansion handles duplicate elimination. The start node instances (10, A) and (11, A) are expanded according to query plan and as we get duplicate instances (10, 12, 11) and (11, 12, 10), we merge them by canonical ordering.

Routing partial answers: When all query results are completely inside a single partition, complete answers are computed and stored while processing that partition. When query results span multiple partitions, we compute partial answer in one partition and carry that instance to appropriate partition which has further query edges. We keep constructing the answers by routing it to appropriate partition and loading additional partitions. The complete answers are collected when the answer instance have all the edges according to query plan. When an answer instance comes back to a partition that has been already processed, it is special case of above. As we carry instances and process additional partitions, we ensure correctness by loading those partitions multiple times.

Addressing overlapping answers: Our system makes sure that our final results set contains all overlapping answers as *separate instances.* We compute query on each initial substructure systematically and if any answer goes out of partition, we carry them as *separate instances* and systematically keep finding complete answers. For the example discussed in Figure 4.2, the initial instances (5, A) and (6, A) will be carried to partition P3 and when we process partition P3, node (7, B) will be replicated for both the instances. Hence, all the overlapping answers comes out correctly.

Addressing multiple path query plan: Query plan nodes are numbered for this purpose. The issue with multiple path query plan is handled by employing the mechanism of marking the node that continues to different partition with node id of the query plan node. Figure 4.7 shows the relative marking of node 14 for our previous example (Figure 4.3). Once this partial answer continue to partition P_2 , we would know which query edge to expand further from the relative NODE id of the node and find correct results. In our case node $< B : 14 >$ will be further expanded for query edge $B- > D$.

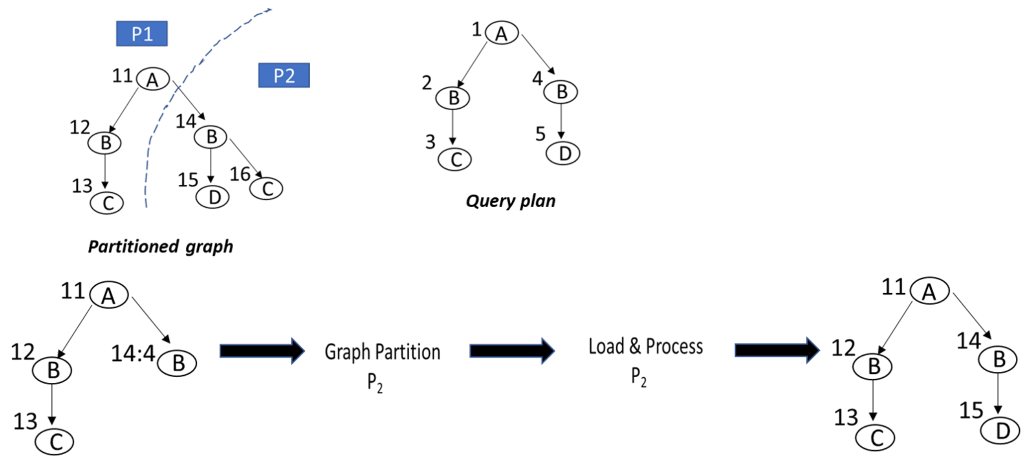


Figure 4.7: Marking node for correctness

4.4 Correctness of querying approach

For the correctness of query evaluation, we need to find all the exact matches of query graph from the whole graph. As we have partitions of the graph, the results set should have answers from within a partition or across partitions. In our approach, we have addressed each case of answers spanning partitions. Also depending on query characteristic, we have addressed both single path or multiple path query plans.

Overall the correctness of the query processing is operationally ensured by i) Use of Meta Information file to correctly identify partitions which has start nodes (whether initial or continuation) ii) Update of Meta Information File to include partitions which has continuation or to remove partitions which has been processed iii) Carrying intermediate results to the correct graph partition which will then load respective adjacency list for further expansion in next iterations. iv) Marking the partial answers and continuing expansion systematically from the last visited node.

We have also tested the correctness empirically on embedded graphs with known frequency and evaluation complete or partial queries of embedded graphs. This is discussed in Section 6.

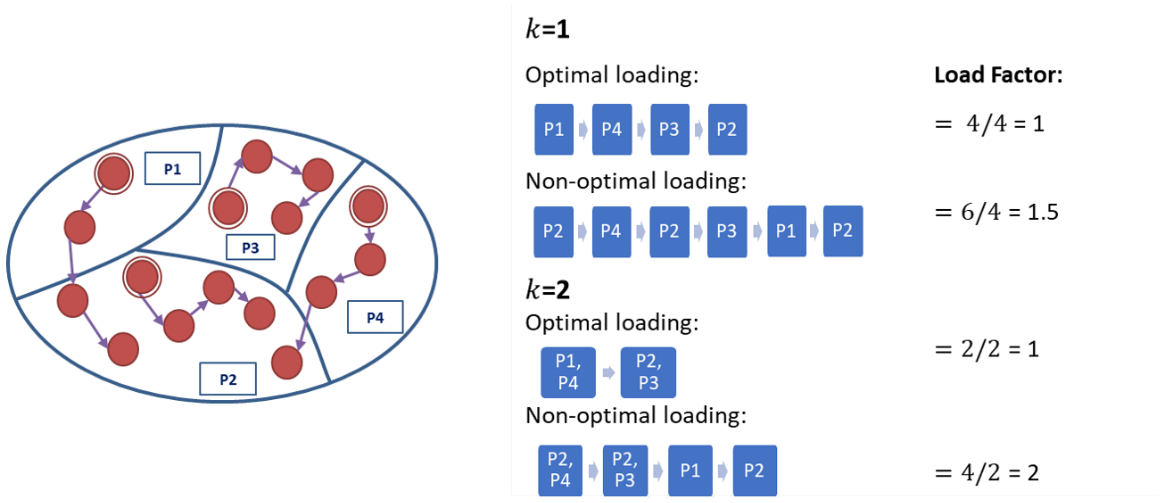


Figure 4.8: Optimal vs Non-optimal partitions loading and Load Factor

4.5 Heuristics for scheduling partitions

With out meta information or heuristics, query evaluation becomes an exhaustive search process. Partitions have to be loaded randomly and hope one finds all answers. This will not eliminate keeping track of query results crossing partitions. Hence, both meta information (collected in each iteration and at the beginning) as well as heuristics based on them play a key role in reducing the response time of query evaluation. This is also borne out experimentally.

Optimal vs Non-optimal Loading of partitions: Based on chosen k value, we need to identify which k partitions need to be loaded in each iteration. In the ideal case, we want to load minimum number of required partitions to answer a query. A required partition is one in which one or more number of the query plan nodes exists.

In processing a query over a partitioned graph, minimizing the total number of partitions loaded over all the iterations can be used as an efficiency measure. The optimal case is loading at most the number of required partitions only once.

Load Factor: Apart from number of partitions loaded, the query processing can take more iterations or less iterations depending upon chosen value of k. The ideal number of required iterations are when we load only required partitions. We define 'Load Factor' which measures efficiency of query processing by comparing number of iterations needed for the ideal case. Formula for calculating load factor is as follows:

$$\text{Load Factor} = \frac{\text{Total iterations taken for query processing}}{\text{Required number of iterations}}$$

where

$$\text{Required number of iterations} = \lceil \frac{\text{unique partitions that has query answers}}{k} \rceil$$

For optimal loading as shown in figure 4.8, load factor is 1. This example also shows Non optimal sequencing, where some partitions are loaded more than once. As number of iterations also increases here, load factor for non-optimal loading increases. For k = 1, load factor is 1.5 whereas for k = 2, load factor is 2. The closer the value of load factor is to 1 the better the scheduling based on the chosen heuristic.

Scheduling Based on Heuristics: We need heuristics to load top-k partitions in each iteration to reduce the number of total partition loads as well as the number of iterations. We propose a suite of heuristics to identify which is better according to graph or query characteristics. If there are $p^{'}$ required number of partitions and for given k value, H(a) loads $p^{'}$+a partitions and H(b) loads $p^{'}$+b partitions, then we can say H(a) is better than H(b) if a < b. In other words, less the number of repeated partitions (or load factor), better the heuristic.

4.5.1 **Existing heuristics:**

The previous work on query processing in partitioned graph [13] has proposed the following heuristics. Remember that the value of k used in the previous work was 1.

Number of Query Plan Start/Continuation Nodes in a Partition

1. **MAX_SN:**

   This heuristic is determined by number of start nodes in each partition. While processing, top-k partitions which has maximum number of start nodes are chosen. This is a greedy strategy for choosing partitions for processing. The intuition behind this is that highest number of query answers will be explored in the beginning and is likely to help reduce the number of partitions loaded if the answers span into other partitions.

2. **MIN_SN:**

   For this heuristic, top-k partitions which has least number of start nodes are chosen in every iterations. The intuition is we accumulate continuation from partitions containing less number of start nodes and process them later once.

Total Number of Connected Components

3. **MIN_CC:**

   MIN_CC chooses top-k partitions which has least number of connected components. The intuition is this will likely to make an answer stay inside a partition more often and thereby reduce the total number of partitions used.

Comparing heuristics: MAX_SN, MIN_SN & MIN_CC

Figure 4.9 shows 5 graph partitions with meta information such as start nodes, connected components, etc inside each partition. The arrow shows the continuation of answers from one partition to another in that direction and count of it. When we apply above 3 heuristics for evaluating partitions in this example, the number of

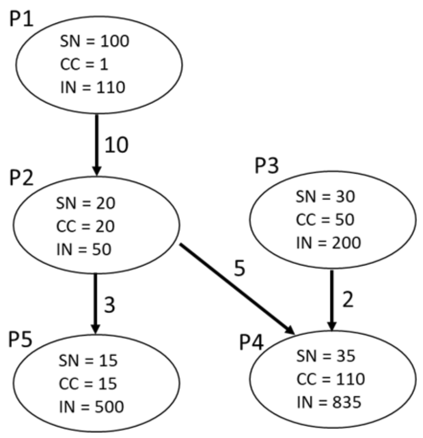


Figure 4.9: Graph partitions with start nodes, intermediate nodes and connected components information

partitions loaded and total number of iterations required for each heuristics are shown in Table 4.3 for k = 1 and k = 2. Load Factor is also calculated for each case.

From the table, we can say that MAX_SN and MIN_CC are better than MIN_SN because they load less number of partitions and required less iterations. Load Factor for both MAX_SN and MIN_CC is less than MIN_SN. This was also borne out empirically in the previous work with k as 1.

Heuristic	**k**	**partitions loaded**	**# of iterations**	**Load Factor**
MAX_SN	1	6	6	1.2
MAX_SN	2	6	3	1
MIN_SN	1	7	7	1.4
MIN_SN	2	7	4	1.33
MIN_CC	1	6	6	1.2
MIN_CC	2	6	3	1

Table 4.3: Heuristics comparison: MAX_SN, MIN_SN & MIN_CC

4.5.2 **Heuristics using Combinations of Meta Information:**

Instead of using the meta information individually, combining them is likely to yield a better strategy for loading partitions when the k value is greater than 1. Hence, we have explored several combination heuristics which are discussed below.

We further investigate combining each individual heuristics and validate the use of them to improve querying performance. The intuition behind this is that combining heuristics may yield better results than individual. We have formulated following 3 different heuristics:

4. **SN_CC:**

   The intuition behind this is to not only use start nodes in each partition but also use connected components to choose top-k partitions. The partitions containing more start nodes and less connected components should be given more priority and loaded first. Formula for calculating SN_CC heuristic is as follows:

$$H_{SN\_CC} = (e^{-0.04*C}) * S$$

   where S is number of start nodes in each partition and C is number of connected components in each partition.

   Table 4.4 shows the values of heuristic SN_CC for our previous example in figure 4.9. If we analyze these values, the partition containing more start nodes and less connected components are given more heuristic value. But for partitions which has more connected components, heuristic values are less.

   For the graph partitions in figure 4.9, if we apply SN_CC heuristic to choose top-k partitions in every iteration, then number of partitions loaded and iterations required are shown in 4.5. As we can see, SN_CC performs better than above 3 discussed heuristics because it loads less partitions and takes less iterations. The load factor is also 1 for both values k = 1 and k = 2.

partition	Start Nodes	Connected Components	H_{SN_CC}
P1	100	1	96.08
P2	20	20	8.99
P3	30	50	4.06
P4	35	110	0.43
P5	15	15	8.23

Table 4.4: SN_CC heuristic values

k	partitions loaded	# of iterations	Load Factor
1	5	5	1
2	6	3	1

Table 4.5: SN_CC heuristic performance

5. **SN_CC$_{2k}$:**

   This also uses start nodes and connected components in each partition. It chooses top 2k partitions first from the set of eligible partitions according to MAX_SN heuristic and from that, it chooses top k partitions according to MIN_CC. The intuition is that it utilizes both start nodes and connected components in each partition in choosing top k partitions.

k	partitions loaded	# of iterations	Load Factor
1	5	5	1
2	5	3	1

Table 4.6: SN_CC_{2k} heuristic performance

   Table 4.6 gives this heuristic performance for the example given in Figure 4.9. It also performs better than individual heuristics as for both k = 1 and k = 2, it loads only required partitions. Hence, load factor in this case is also 1.

6. **SN_IN:**

   The intuition behind this is in addition to start nodes in each partition, take advantage of query plan intermediate nodes to choose top-k partitions. The partitions having huge number of intermediate nodes compared to start nodes will have potentially more answers (though partial) and will be needed multiple times during processing. Following is the formula used for this heuristic:

$$H_{SN\_IN} = ln(S+1) * \frac{S}{S+\frac{I}{2}}$$

where S is Number of start nodes in each partition and I is Number of intermediate nodes in each partition.

In Table 4.7, we compare values for this heuristic for different partitions present in figure 4.9. From the table, we can see if the partition has huge number of intermediate nodes, than those partitions given very less heuristic value (for example partition P4 and P5).

partition	**Start Nodes**	**Intermediate Nodes**	H_{SN_IN}
P1	100	110	2.97
P2	20	50	1.35
P3	30	200	0.79
P4	35	500	0.44
P5	15	835	0.09

Table 4.7: SN_IN heuristic values

As we apply SN_IN heuristic to choose partitons in our previous example (figure 4.9), the partitions needed for processing and iterations required are shown in Table 4.8. We can see that this heuristic also performs better than individual heuristics (Table 4.3) as it loads less partitions and requires less iterations. Hence, the load factor for both the cases k = 1 and k = 2 is 1.

k	**partitions loaded**	**# of iterations**	**Load Factor**
1	5	5	1
2	5	3	1

Table 4.8: SN_IN heuristic performance

Our experimental analysis compares all of the above heuristics to establish pattern with respect to the choice of heuristic.

4.6 Algorithm

Algorithm 1 details our query processing approach on large graphs. As discussed earlier, input graph is divided into p partitions using range information and adjacency list for each graph partition is created once. While partitioning, we collect meta information (MIF) such as start nodes, intermediate nodes and connected components for each partition to be use for heuristics during query evaluation. As not all partitions has query answers, MIF is crucial for determining the eligible partitions and eliminate loading those partitions which does not have query plan nodes.

The algorithm runs in iterations (line 2 to 14) until there are no more partitions to load. MIF contains eligible partitions information for each iteration. We compute heuristics using MIF to choose top-k partitions (line 2). Systematic query expansion is done for all initial start node substructures or partial answer substructures in this partitions according to query plan (line 4 to line 11). Adjacency list is used to expand every substructure by one edge at a time which matches query plan. During expansion itself, we identify which expansion goes out of partition and we route those instances to continuing partition and update MIF accordingly (line 9). Duplicate elimination are done after every edge expansion (line 10). Complete answers are constructed as we process partitions during runtime (line 12). The processing continues with another set of partitions using updated heuristics (line 13 and 14).

Algorithm 1 Graph Query Processing Algorithm

```
INPUT: Input Graph G, Graph Queries
OUTPUT: Exact matches of all queries
1: Use Range based partitioning to generate graph partitions and adjacency list
   partitions. Also collect meta information (MIF) for heuristics
2: For given value of k, apply heuristics and choose top k (or less) partitions
3:     Terminate if no partitions to process
4:     Get partition ids of currently loaded $k$ graph partitions
5:     Load adjacency partition $AL_i$ for all $k$ partitions
6:     Get all initial substructures or partial answer instances for each partition
   that contain query plan start node
7:     Expand substructure to the desired node according to query plan by adding
   one edge at a time using $AL_i$
8:     If partition id of expanded node does not match with current partition id
9:         Write out partial answer substructure to corresponding graph partition
   and update meta information for that partition
10:    Else, create canonical instance of answer substructure, remove duplicates
   and continue for next query edge
11:    Terminate the expansion when all the edges in the query plan are visited
12:    Write out complete answer results
13:    Update the meta information file for each partition being processed
14:    Go to step 2 and continue using updated heuristics
```

4.7 Resource Utilization

Our approach processes k partitions in parallel. The value of k can be set by the user for each query if need be. The number of partitions is fixed and done once to match resources. For a given or chosen k value, our query processing can accommodate as many resources as available. If we have enough resources available, then all k partitions will be loaded and processed by individual processor at a time in parallel. This is best case for effectiveness of parallelism. But in the case when we have limited resources, we can not process all k partitions in parallel at a time. We address it by loading more than one partition on a single processor.

Furthermore, query processing on p graph partitions using k at at a time must deal with what value of k we should choose to utilize maximum resources and to get good response time. If we have only single processor available, then the k value is 1. But if we have more than one processor available, then we can increase k to match with number of processors. In the case, when number of processors are more than p, then we can increase k to maximum value of p. But not all p partitions can contain query answers. So we may not have p partitions to schedule at a time. Hence, blindly choosing k = p may not be good idea. In this case, to choose the best value of k, analysing the type of query and graph characteristics may become helpful.

4.8 Handling queries that are not known at partition time

The query processing on partitioned graph starts by identifying the partitions having start nodes. The initial meta information collected at the time of partitioning is used for this purpose. But for the new queries that were not known at partitioning, we would not have those information. Hence, we would require a different approach for those types of queries. Bellow we discuss two alternatives to handle those queries:

1. One possible solution is to arbitrary choose k partitions from initial set of p partitions. If any of the partitions have start nodes then it will update the MIF, which will then be used to load subsequent partitions. If none of the k partitions have start nodes, a different set of k partitions can be loaded.
2. Another solution is to create an inverted index on node labels. In other words, we create a mapping of each unique node label and list of partitions in which they occurs. As we can construct this index during initial partitioning, there is no extra computation cost for it. From this index we can create MIF for any arbitrary query and process the query. This may even work better than first approach as it quickly identifies the partitions which has start nodes. However, this can become issue if there are many unique node labels in a graph, increasing the size of the index. We can address this by loading the index on disk. We can use B+ tree by using keys as node labels and values as lists of partitions in which those node labels occurs.

In this chapter we have explained the design of the system and also elaborated on the detailed algorithm of query processing on graphs. Having provided this discussion, in the next chapters we elaborate the implementation aspects of our design in the Map/Reduce framework and present our findings of our experimental results.

CHAPTER 5

IMPLEMENTATION

This chapter provides detailed description of the implementation of scalable query processing system based on partitioning and parallel processing. MR_QP developed as part of this thesis uses Map/Reduce framework for a distributed processing to utilize parallel processing. Figure 5.1 shows overall architecture of Map/Reduce based query processing system.

Input Graph Representation: Our input graph is represented as a sequence of unordered edges (or 1-edge substructure). This makes it possible to be distributed among many machines. Both undirected and directed graphs can be used for our approach but we have considered only directed graphs in our system as it has relations explicitly represented using directions. Table 5.1 shows edge input representation of graph in figure 5.2. Each edge is completely represented by a 5 element tuple ⟨edge label, source vertex id, source vertex label, destination vertex id, destination vertex label ⟩.

Both partitioning and query processing are done using Map/Reduce program. The driver program which runs Map/Reduce jobs is implemented in java. The configuration file is passed to the driver program that specifies necessary parameters. Range partitioning is done in a single iteration of Map/Reduce, which generates graph partition files and adjacency list partition files. Also, partitioning generates, if query plans are available, initial Meta Information File (MIF). MIF contains eligible partitions for query evaluation. After MIF is generated, our driver program applies specified heuristic using the contents of MIF and choose top-k partitions. When the chosen

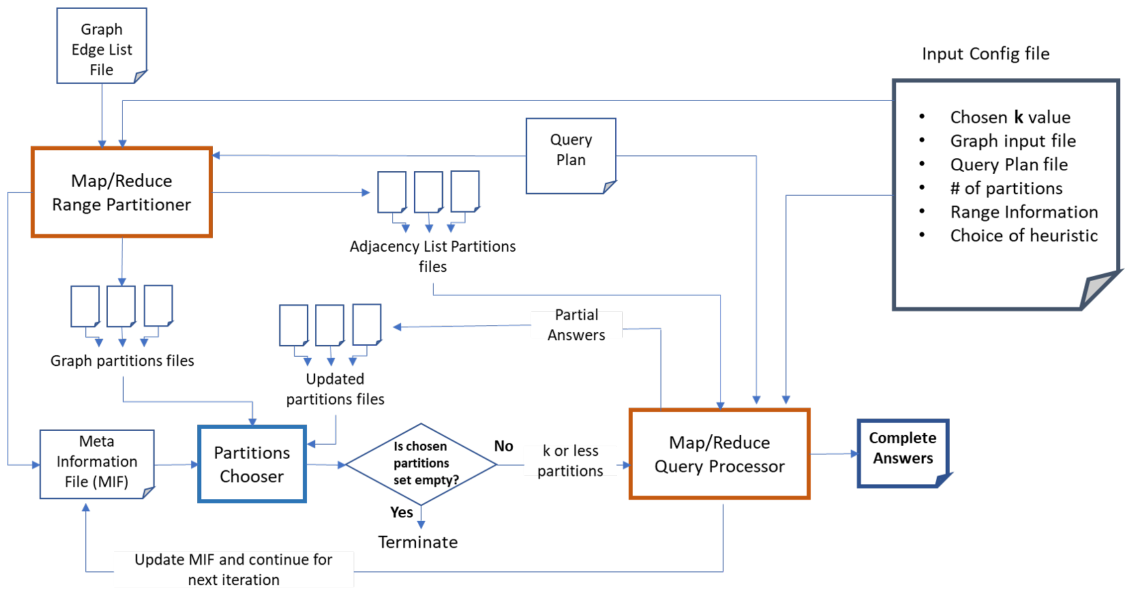


Figure 5.1: MR_QP System Architecture

partitions set becomes empty, then query evaluation has been done and we terminate the computation. Else we would have k or less partitions to process at a time, using which a Map/Reduce based query processing does query expansion on those partitions in parallel and outputs the complete answers to Final All Answers (FAA) file and writes the intermediate answers to the relevant graph partition files for further expansion. During the processing of partitions, MIF is updated and will be used by driver program to select next set of partitions to be loaded in next iteration. This is continued until eligible partitions set in MIF is empty.

5.1 Range Partitioning using Map/Reduce

Figure 5.3 shows the range partitioning in Map/Reduce. Algorithm 2 further expands each mapper, partitioner and reducer class in details.

Split and shuffle edges: Based on number of mappers available, Map/Reduce paradigm divides the input graph edge list into input splits and given as the input to

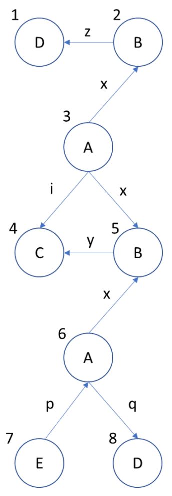


Figure 5.2: Example Input Graph

Edges
(z,2,B,1,D)
(x,3,A,2,B)
(i,3,A,4,C)
(x,3,A,5,B)
(y,5,B,4,C)
(x,6,A,5,B)
(p,7,E,6,A)
(q,8,D,6,A)

Table 5.1: Graph Edge List Representation

the mapper. The map function then processes a single edge at a time and for both the vertex ids in the edge, it outputs vertex id as key and edge instance as the value (line 1 to 5).

Shuffle phase sorts the mapper output based on vertex ids and groups all the records that have same vertex id. As we want to divide output records based on range information, we use the custom partitioner that partition keys with custom hash function based on range information (line 6 to 9). This will route all the output records with vertex id in the same range to the same reducer.

Creating graph partition and adjacency list partition: Once the reducer gets desired set of keys, we create graph partition files that contains initial substructures and adjacency list partitions that contains adjacency list of all the vertices in the same partition. First, inside setup of reducer, we create new graph partition file and local MIF (line 14). After that, the reduce function gets key as vertex id and value list as all adjacent edges from that vertex (line 16). For each vertex, we create initial substructure and output it to corresponding graph partition file (line 25). Also, for each vertex, single adjacency list is created. And finally, reducer emit key as

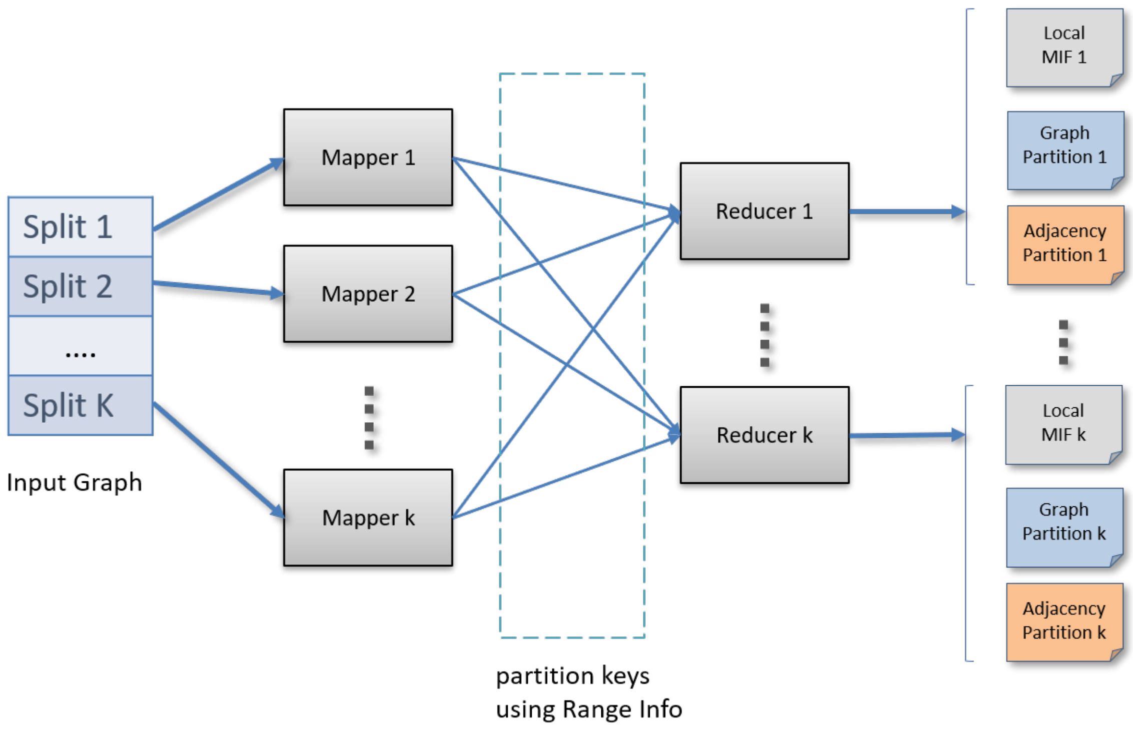


Figure 5.3: Range Partitioning using Map/Reduce

vertex id and value as adjacency list (line 27 and 28). The emitted keys that belongs to same partition is written to single file creating adjacency list partition.

Collecting meta information and creating MIF: As we create initial substructures inside reducer, we compare them to query plan start node or intermediate nodes and if there is match, we update the occurrences of those instances in current partition in MIF (line 18 to 24). Also, as the reducer gets all the edges within single partition, we collect all the edges (line 26) and in reducer cleanup function, using every edge in a partition, we find connected components using find and union algorithm and count of connected components in that partition is written to MIF (line 31 to 38). As each reducer process separately, local MIF for each reducer will be created and then driver program collect them to create global MIF.

Algorithm 2 Range Partitioning

```
INPUT: Graph edge list, Range info, Query Plan
OUTPUT: Graph partitions, Adjacency list partitions, MIF
Class Mapper
 1: function MAP(key = line no, value = (one) edge instance)
 2:     get the source vertex id($sVid$) and destination vertex id($dVid$) from edge
 3:     emit(newKey = $sVid$, newValue = edge instance)
 4:     emit(newKey = $dVid$, newValue = edge instance)
 5: end function
Class Partitioner
 6: function getPartition(key = vertex id, value = one edge instance)
 7:     reducerId = get partition id of vertex from Range Info
 8:     return reducerId
 9: end function
Class Reducer
10: function setup
11:     $pId$ = get id of current reducer
12:     partitionRange = get range of partition $pId$ from Range Info
13:     edgeList = set to hold unique edges in current partition
14:     create new files for graph partition($pId$) and MIF
15: end function
16: function REDUCE(key = vertex id, valueList = list of adjacent edges)
17:     get vertex id($vId$) from key and vertex label($vLabel$) from valueList
18:     if $vLabel$ is Query Plan start node then
```

```
19:         update MIF with ⟨ partition id = $pId$, label = $vLabel$, count = 1,
20:             is start node = True ⟩
21:     else if $vLabel$ is Query Plan intermediate node then
22:         update MIF with ⟨ partition id = $pId$, label = $vLabel$, count = 1,
23:             is start node = False ⟩
24:     end if
25:     create initial substructure $\langle vId, vLabel \rangle$ and write out to graph partition($pId$)
26:     append every edge from $valueList$ to edgeList
27:     create adjacency list ($AL$) for vertex ($vId$)
28:     emit(newKey = $vId$, newValue = $AL$)
29: end function
30: function CLEANUP
31:     S = make sets for all vertices in current partition($pId$)
32:     for each edge $< sVid, dVid >$ do in edgeList
33:         if sVid and dVid not in same set then
34:             union both sets of sVid and dVid
35:         end if
36:     end for
37:     count of connected components($CC$) = total unique sets in S
38:     update MIF with ⟨ partition id = $pId$, connected components = $CC$ ⟩
39: end function
```

5.2 Query Processing using Map/Reduce

Figure 5.4 shows the single iteration of Map/Reduce based query processing. The complete algorithm for mapper and reducer class is presented in Algorithm 3. Top

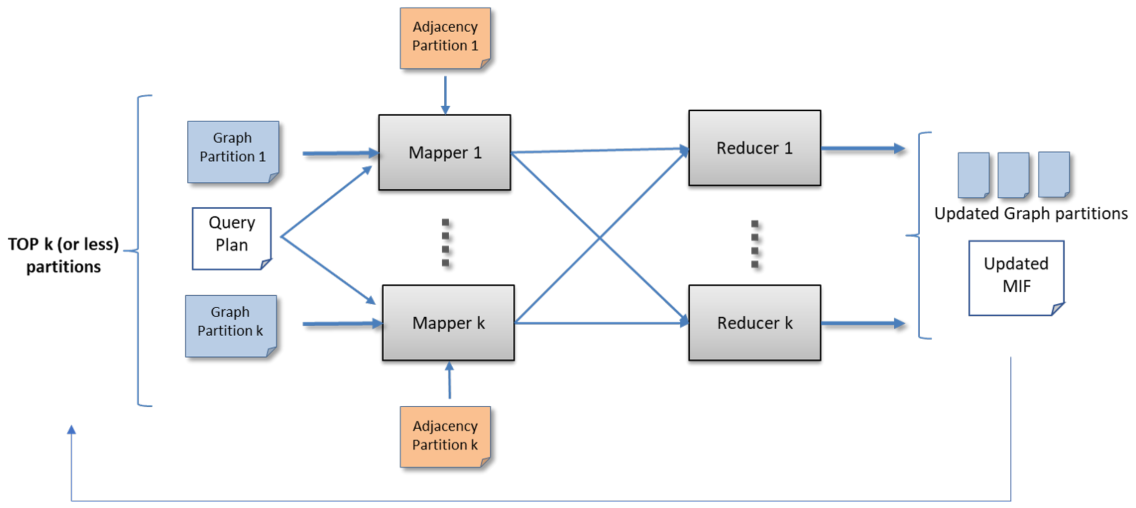


Figure 5.4: Query Processing using Map/Reduce

k or less graph partition files chosen from eligible partitions after applying heuristics are taken as input in each query processing iteration.

Query expansion inside mapper: Input to the mapper is the graph partition that contains initial substructures or partial answer instances. In setup function, mapper loads respective adjacency list partition and query plan (line 3 and 4). We then create a set to hold unique completed answers from current partition (line 5).

The map function takes the key as line number and value as initial substructure or partial answer instance (line 7). Then we start query expansion for those instances which has query plan start node (line 8) and do systematic expansion for every query edge present in query plan and store the intermediate substructures in a set (line 9 to 28). In the case of partial answer instance, we will continue expansion only if the last visited node marked with relative id of current query node (line 15). Once we get node to be expanded, we load adjacency list of that node and according to query plan, find all edges that match with current query edge and expand the substructure with that edge (line 17 and 18). If the newly expanded node belongs to different partition, then we mark that node relative to last visited query node in query plan and the mapper emits new partition id as key and marked partial answer instance as the value (line 24 and 25). Else, if the expanded node is in same partition, we create canonical instance of the answer and store them in a answers set to remove duplicates (line 20 to 22). We then continue for next query edge from query plan. When there is no more edge to expand from the query plan, our answers set contains all complete answers. We collect them to global answers set for that mapper (line 30). In the cleanup function, the complete answers are written to the Final All Answers (FAA) file (line 34).

Aggregating partial answers and updating graph partitions in reducer: After mappers have been processed, Shuffle phase then groups the output of mappers on the partition id and sends them to the corresponding reducer. The reducer processes key-value lists generated for each partition. The reduce function gets key as partition id and value list containing partial answers in that partition (line 36). For each partial answer substructure present in the list, it updates the MIF to include continuing nodes in appropriate partitions (line 39). According to partition id, it then loads corresponding graph partition file from HDFS and write out partial answers in that file (line 41).

Algorithm 3 Map/Reduce based Query Processing

INPUT: Top k graph partitions and adjacency list partitions, Query Plan

OUTPUT: Updated graph partitions, Final answers file(FAA), Updated MIF **Class Mapper**

```
 1: function SETUP
 2:     $currentPartitionId$ = partition Id from currently loaded graph partition
 3:     adjMap = load corresponding adjacency partition
 4:     load Query Plan file
 5:     finalAnswers = Set to hold completed answers
 6: end function
 7: function MAP(key = line no, value = initial or partial answer instance)
 8:     if value contains Query Plan start node then
 9:         answerSet = Set to hold unique substructures after each query edge
10:         expansion
11:         add initial value substructure to answerSet
12:         while all the nodes and edges in the plan get visited do
13:             for each $ans$ substructure in answerSet do
14:                 get last visited node (v) from $ans$
15:                 if node (v) has been marked and doen't match with Query node
16:                      then continue for next query edge
17:                 aL = get adjacency list of v from adjMap
18:                 expand the current node to the desired node by adding edge e
19:                      from aL based on the plan
20:                 if $pId$ of expanded node matches currentPartitionId then
21:                     create canonical instance newAns
22:                     remove $ans$ from answerSet and add $newAns$ to answerSet
```

```
23:                 else
24:                     mark expanded node in $ans$ substructure with relative id of last
    visited query node
25:                     emit(newKey = $pId$ of expanded node, newValue =
26:                       expanded $ans$ substructure)
27:                 end if
28:             end for
29:         end while
30:         finalAnswers ∪ answerSet
31:     end if
32: end function
33: function CLEANUP
34:     write out finalAnswers to FAA file in HDFS
35: end function
Class Reducer
36: function REDUCE(key = partition Id, valueList = intermediate results)
37:     for each value in valueList do
38:         get the last expanded vertex (v) from the value
39:         update MIF file with ⟨ partition id = key, label = null, count = 1,
40:            is start node = True ⟩
41:         write out value to graph partition ($pId$) in HDFS
42:     end for
43: end function
```

5.3 Configuration Parameters

The MR_QP system accepts parameters for different tasks from a configuration file. We have provided options for various parameters such as selection of graph, choice of partitions and range information to control the size of partitions, choice of k, choice of heuristic, choice of query plan and so on. 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: It specifies the path in HDFS where the input graph is loaded.
2. Input Query Plan: The query plan which user wants to evaluate on the above graph database.
3. Number of partitions: It indicates the number of partitions the above input graph will be divided.
4. Range Information: It specifies the vertex ids range for each partition. This is provided in form of mapping between partition id and vertex start and end range (inclusive).
5. k (Parameter for parallelism): Maximum number of partitions that can be processed in parallel in each iteration.
6. Heuristic: The heuristic to choose top k partitions. This can be either MIN_SN, MAX_SN, MIN_CC, SN_CC, SN_CC$_{2k}$, SN_IN or RANDOM
7. Analysis output directory: As we run multiple jobs with different configurations and want to keep the track of all the details of the job, this parameter specifies the path to store the analysis file for each job.
8. Number of Reducers: This parameter states how many reducers are to be used for the Map/Reduce job

5.4 Implementing Analysis

Implementing Job Counters:

To measure the performance of our Map/Reduce jobs, we have used the built in counters and implemented few custom counters (user defined) in our Mapper and Reducer program. Counters in Hadoop Map/Reduce are a useful channel for gathering statistics about the Map/Reduce job: for quality control and to analyse the cost and space. Below is the list of all Job counters used.

- Map Setup time: Implementing a counter for setup time helps to analyse the setup cost for a Mapper/Reducer. In our Map/Reduce job for query processing, main cost of setup is incurred in Mapper as we are loading adjacency list for each graph partition inside mapper in each iteration. As all mappers run in parallel, we take maximum of all map setup times to get overall setup time for that job.
- Map time: The mapper calls map method for every key/value pairs. The map time for any mapper is time taken to process all key/value pairs. The overall map time is calculated by taking maximum of all individual mappers map time.
- Shuffle time: In the shuffle phase in our Map/Reduce query processing job, all the partial answers are grouped and split among reducers according to partition id. Every reducer obtains all partial answers with same partition id. We have implemented a counter here to examine the total time taken by the shuffle phase to transmit the data from the Mapper to Reducer.
- Reduce time: As reduce method in our algorithm carries partial answers to appropriate partition, this counter details the time taken in reducer for routing and updating graph partitions. The reduce time here is significantly less in comparison to map time because our reducer only deals with continuing answers.

- Cleanup time: The cleanup function is called in our mapper program to write completed answers to the FAA file in hdfs. The cleanup time is calculated for each mapper and the maximum of all mappers is stored in this counter to find overall cleanup time.

Controlling the number of Mappers: In Map/Reduce model, response time is optimized by controlling mappers and reducers. The Map/Reduce system divides the entire data into splits and are handled on different machines in a cluster. The default input split size is 128 MB which can be altered. So, for instance let us say we have a cluster of 5 Machines which can be used in parallel and you have a input data of 220 MB. If you go by the default input split size provided by Map/Reduce it would divide your data into 2 partitions which will be processed in parallel on 2 Machines. But if you want to utilize all of your resources (5 Machines) to decrease your response time, then we can change the input split size to 45 MB, and it would create 5 partitions (5 Map tasks) which can be processed in parallel and it will decrease the response time. We used this approach in our implementation to control the Map tasks and eventually the mappers.

On Comet we specify the number of processors used as nodes and each node has 2 processors, so if we want 10 Mappers/Reducers we specify 5 nodes. But the map tasks should be same 10 in this case. If less than 10, then all the processors will not be utilized. So we can alter the size of split size as discussed above to match the number of processors.

In the next chapter, we discuss the experimental analysis of various queries evaluated on partitioned graph databases using different number of partitions, k values, and heuristics to evaluate query processing.

CHAPTER 6

EXPERIMENTS

This chapter presents the results of extensive experimental analysis performed on various synthetic and real-world graphs and queries that exercise expressiveness discussed earlier.

6.1 Experimental Setup

Experimental Environment: All the experiments are performed using Java with Hadoop on Comet cluster at SDSC (San Diego Supercomputer Center). The Comet Cluster has 1944 nodes and each node has 24 cores (built on two 12-core Intel Xeon E5 2.5 GHz processors) with 128 GB memory, and 320GB SSD for local scratch space.

Dataset Description:

Dataset	**# of partitions**	**V**	**E**
DBLP	120	3.3M	13.5M
IMDB	60	0.7M	3.5M
Synthetic	10	400K	1.2M
DBLP_small	3	12K	30K
IMDB_small	3	11K	33K

Table 6.1: Dataset description

As shown in Table 6.1, to perform the experiments, we have used three different datasets- DBLP, IMDB and synthetic graphs to show effectiveness and scalability in our approach. The DBLP data set contains the information about authors, papers, conferences and years as vertex labels and edges showing relationship among these vertex labels. Similarly, 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. And the synthetic graph contains 2000 unique vertex labels and 4000 unique edge labels.

The sizes of the DBLP graph and IMDB graph are very large. The DBLP graph consists of 3.3M vertices and 13.5M edges and it is the largest graph size among all datasets. Our IMDB graph consists of 0.7M vertices and 3.5M edges. To test the correctness of our approach, we need to make sure that our partitioned graph gives the same results as running the query on whole graph. To validate that, we have used already existing system, QP-Subdue, which is a non-partitioned, main-memory query processor for graph databases. However QP-Subdue cannot handle larger graph sizes. Hence we have also used smaller sizes for both DBLP and IMDB graphs to verify the correctness. As shown in table 6.1, DBLP_small has 12K vertices and 30K edges whereas IMDB_small has 11K vertices and 33K edges.

As our DBLP graph has 3.3M vertices, to create smaller size partitions, we chose to divide it into 120 equal size partitions so that each partition has 27K vertices. Similarly, we divided our IMDB graph into 60 partitions using equal range so that each partition has 12K vertices. In the case of synthetic data, as it doesn't have real world information, the size of the graph (adjacency list, etc.) is not very large and hence we can have bigger size partitions of it. So we will divide our synthetic graph using equal range into 10 partitions so that each partition has 40K vertices. For the correctness, we will divide the smaller graphs (IMDB_small and DBLP_small) into 3 partitions and verify the results.

For the above mentioned graph databases, we took queries having different characteristics (Figure 6.1). We have used queries with comparison operator, Wild cards (%), Range queries (BETWEEN), AND query, OR query and combination of logical and comparison operators (>, NOT). For these queries, we have used simple query plans, that avoids using question marks (?) as start nodes. Our query plan representations were discussed in previous chapter (3).

6.2 Empirical Correctness

To establish the correctness of the system, we have first experimented with embedded synthetic graph that contains 100 instances of an embedded substructure (Figure 6.2 (a)) containing 9 vertices and 8 edges. We have formulated three different queries and their query plans from this embedded substructure. Query-1 (Figure 6.2 (b)) represents single path query plan containing 5 vertices and 4 edges. Query-2 (Figure 6.2 (c)) represents the query plan for the embedded substructure itself. We will use this query plan for verifying results for multiple path query plan. And finally Query-3 is query plan for non-embedded substructure. The purpose of this query is to check the correctness for the non-embedded instances. As we run above described queries on the synthetic graph having 10 partitions, we found all 100 instances of Query-1 and Query-2. For Query-3 also we found 1 instance of non-embedded substructure.

We further show the correctness for 2 real world datasets, DBLP and IMDB. To verify the correctness for DBLP, we have used the smaller DBLP graph available (DBLP_small) and tested it for given query DBLP_Q1 (Figure 6.1 (a)). Here, for this smaller dataset, our query has three results. We have verified here two cases of answers spanning partitions. For each case, we have divided the whole graph into 3 partitions using different range information for controlling the size of the graph partitions. Initially, we chose to divide the graph into equal size partitions using equi-range information. In this case, the answers were completely inside first partition. To verify the answers that continues to different partitions, we changed the range information to decrease the size for first partition so that our answers have nodes

Name	English Representation	Feature
DBLP_Q1	Find papers along with their authors, conferences and the publication year, which are published between 1990 and 1995 and the paper name includes the term "Transaction Management in Multidatabase Systems"	Range (BETWEEN)
DBLP_Q2	“Find papers along with their authors and the publication year, which are published after 1995 in the conference VLDB or SIGMOD and the paper name includes the term "Transaction Management in Multidatabase Systems".”	Comparison (>)
DBLP_Q3	“Find all conference papers along with the conference names that are published in 2005 or 2008 or 2009 by the author whose first name starts with 'John’.”	Special character (%)

(a) DBLP Queries

Name	English Representation	Feature
IMDB_Q1	“Find movies produced by "Adam Sandler" where the genres should be comedy and Sci-Fi but the year should not equal to 2000”	AND
IMDB_Q2	“Find all actors (both Male and Female) who have worked in a movie owned by company "Paramount Pictures" where the genre of the movie is either Drama or Action.”	OR
IMDB_Q3	“Find movies along with their actors and producer information by company American Broadcasting Company (ABC) after year 1970 where the genre should not be comedy or horror.”	Combination (>, NOT)

(b) IMDB Queries

Name	English Representation	Feature
Synthetic_Q1	5 vertices, 4 edges - query is part of embedded sub	Single path
Synthetic_Q2	9 vertices, 8 edges - query is embedded sub itself	Multiple path
Synthetic_Q3	5 vertices, 4 edges - query nodes are from (embedded + non-embedded) sub	Multiple path

(c) Syntetic Graph Queries

Figure 6.1: Queries used for experiments

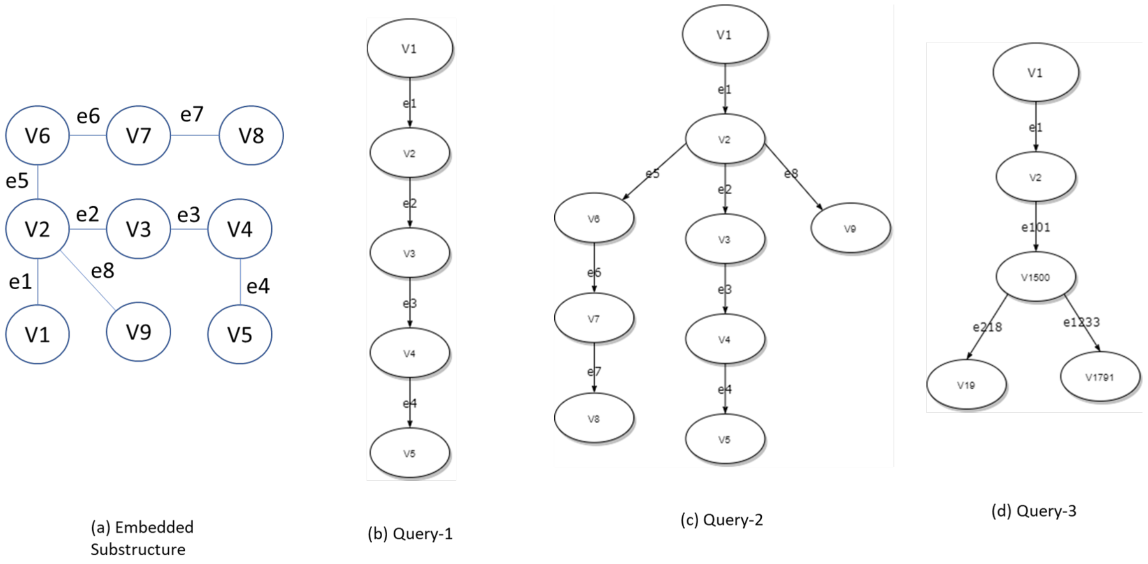


Figure 6.2: Synthetic dataset: Query plans and Embedded substructure

from 2 different partitions. We verified the results that we got for both cases by matching it with the answers obtained by running QP-Subdue. We chose to do verification of third case with different type of query on IMDB graph as it requires the special case of answers spanning partitions.

The correctness for IMDB graph is also shown using IMDB_small dataset. Here, we were specifically interested for verifying correctness for third case of answers spanning partitions. We have used IMDB_Q3 (Figure 6.1 (b)) for this purpose as it has lots of answers. Hence many answers spanning the partitions and some uses same partition more than once. We used equal range information to create the partitions. The start node of our chosen query plan is "American Broadcasting Company", which was present in the partition 1. In our results, some answers continued to partition 2 or partition 3 and came back to partition 1. We verified all the results by comparing them with answers obtained by QP-Subdue.

6.3 Querying a partitioned DBLP graph

As shown earlier, the DBLP graph has millions of vertices and edges and it is used to verify speedup and the scalability of our approach.

DBLP Queries Description:

For this dataset, we have used three different kind of queries.

Query-1: “Find papers along with their authors, conferences and the publication year, which are published **between 1990 and 1995** and the paper name includes the term “Transaction Management in Multidatabase Systems”.”. This shows an example of range query with BETWEEN operator. The number of results that we get for this query is 18.

Query-2: “Find papers along with their authors and the publication year, which are published **after 1995** in the conference VLDB or SIGMOD and the paper name includes the term “Transaction Management in Multidatabase Systems”.”. This query is of type comparison operator (>). The number of results for this query is 45.

Query-3: “Find all conference papers along with the conference names that are published in 2005 or 2008 or 2009 by the author whose first name starts with ‘John’.”. The query is of type wild card(%). As the wild card present in the query matches with more number of nodes in the graph, this query has lots of answers. The number of results for this query is 7685.

Speedup:

We have done these experiments by increasing the value of k (number of partitions processed in parallel) to match as many resources available to see the effect on response time as we increase k. Hence, we have chosen number of mappers equal to k for this set of experiments. We will go till k = 120 as our DBLP graph will have 120 partitions. As our aim is to see the effect of k on query processing for the given query, we have kept the number of reducers to 2 and chosen RANDOM heuristic for

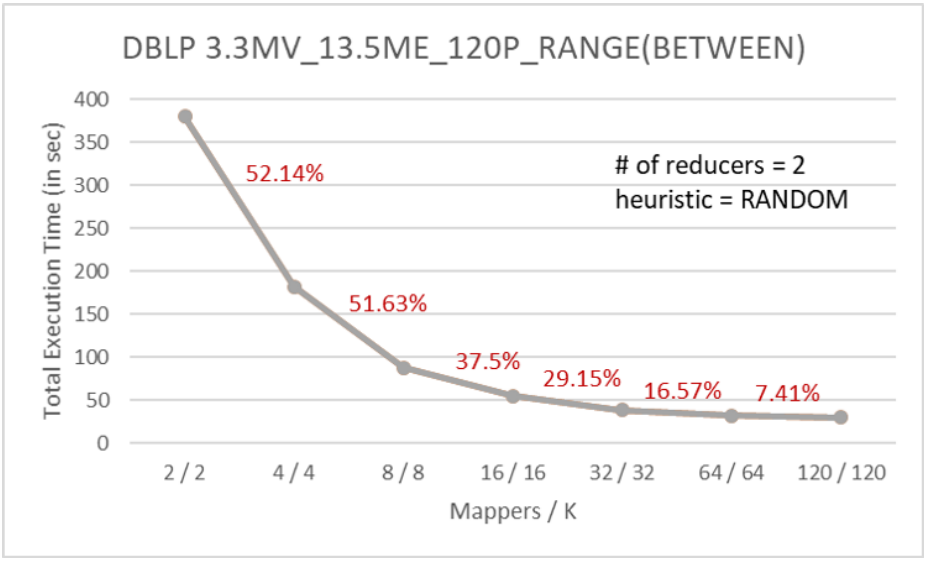


Figure 6.3: Speedup: DBLP (BETWEEN query)

all these set of experiments. Later, we will show different set of experiments with different number of reducers and using other heuristics.

For given BETWEEN query for DBLP graph, Figure 6.3 shows how is the effect on response time as we increase k. As we are increasing k to 2, 4, 8, 16, 32, 64 and 120, we keep seeing the decrease in response time. For increase in k from 2 to 4, we see decrease / speedup of 52.14% and from 4 to 8, speedup is 51.63%. But as we further increase k, although the response time decreases, we don't get same amount of speedup. For example, for increase in k from 64 to 120, we only see speedup of 7.41%.

The reason for decrease in response time as we increase k is we will schedule more partitions to process in parallel using more resources and hence will require less execution time. But for larger k values, we will not have that many partitions to schedule depending on the query type. For example, when required partitions are less than p and in this case increasing k to p will not give good improvement in response time.

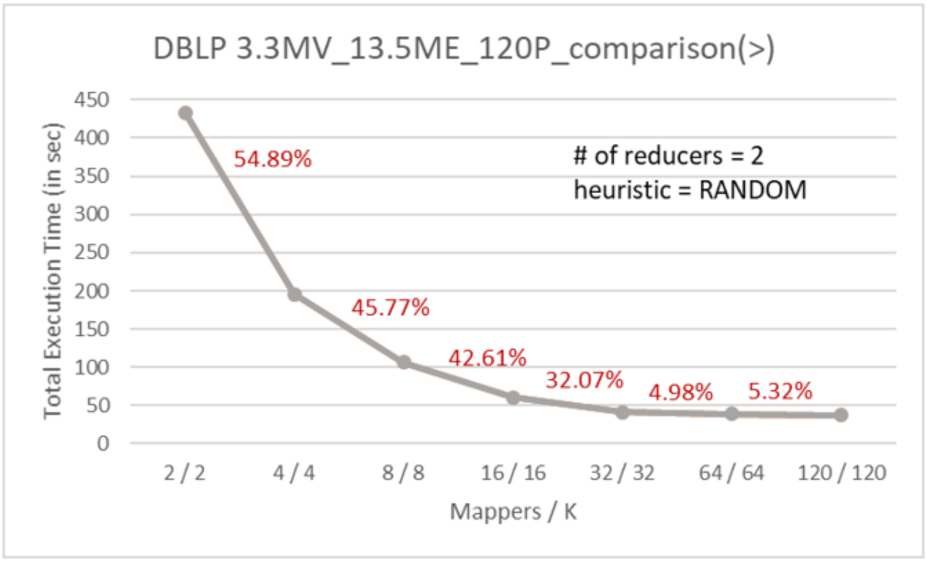


Figure 6.4: Speedup: DBLP (Comparison (> query)

For Comparison operator (>) query (Figure 6.4), for increase in k, the response time decreases. For k = 2, 4, 8, as we double the value of k, we see almost 50% speedup. But for larger k values, there is not much speedup.

For wild card query (Figure 6.5), we see same effect of decrease in response time with increase in k. The speedup for smaller k values are more than speedup for larger k values. For example, speedup after increase from 2 to 4 is 49.8% whereas the speedup for increase from 64 to 120 is 34.77%.

Comparison of above three queries for speedup tells that for smaller k values, as we increase k, we can get good amount of speedup. But for larger k values, the speedup can be more or less depending on the queries. For example, BETWEEN and Comparison (>) queries don't have much speedup for k = 32, 64, 120. On the other hand, Wild Card (%) query continue to get speedup for k = 32, 64, 120. The reason for this is wild card query matches with more instances in the graph and will require more partitions to load. Hence increasing the k will help in achieving better speedup. By analysing the number of results for each query, we can further prove this.

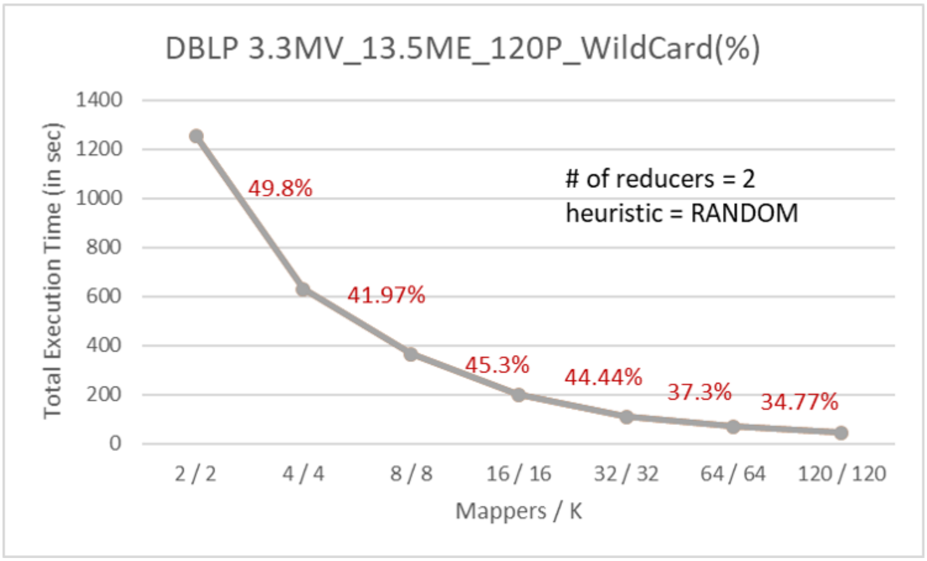


Figure 6.5: Speedup: DBLP (Wild Card (%) query)

From the above analysis, we can infer that provided we have more processors available, then we can increase k to get good response time. But blindly increasing the k may not help in achieving more speedup. For choosing best value for k, investigating query type can become helpful. For example, for the query which has many answers, we can increase k to p given that we have that many resources available.

Effect of changing k values for given mappers:

In this set of experiments, we are interested to see the change in response time as we increase k value for given number of mappers. We have used Wild Card (%) query for doing this analysis.

As shown in figure 6.6, for mappers = 2, as we increase k from 2 to 4 and 8, then the response time slightly increases. In case of mappers = 4, the increase in response time is more as we increase k. We see same pattern for higher values of mappers. The reason for not decrease in response time although we increase k is that we are scheduling more partitions at a time then resources available. And hence single processor will have more than one partition to process. Therefore, response time

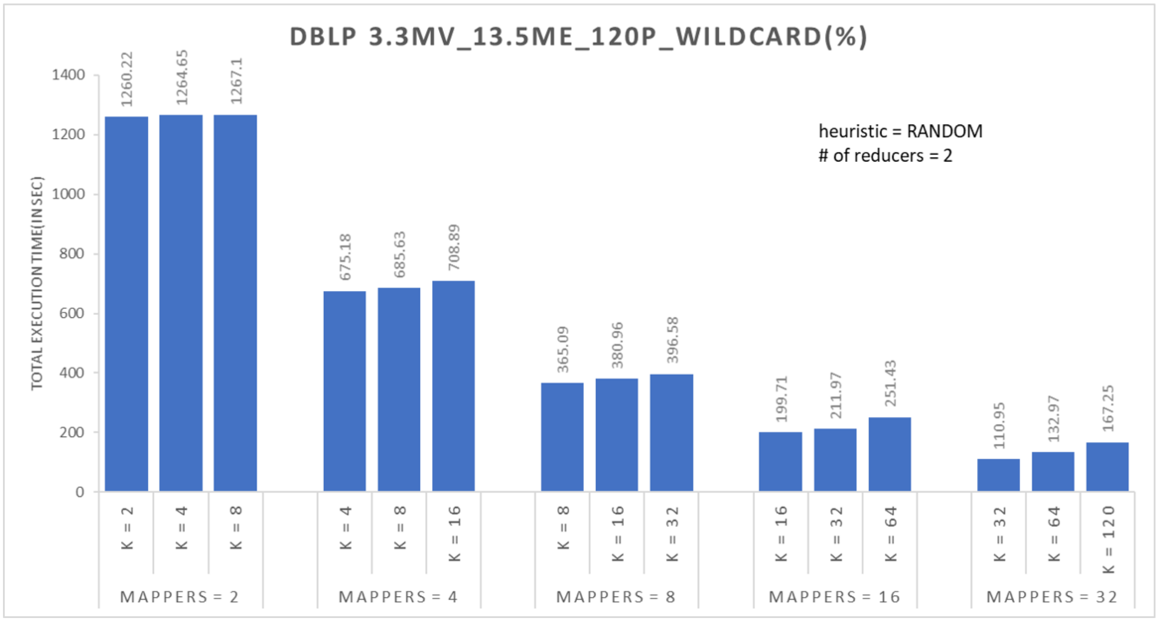


Figure 6.6: Effect of changing k for given mappers

doesn't improve with increase in k. Also scheduling more than required partitions will load some partitions multiple time. For example, if we load all p partitions at a time, this will still require to process additional partitions where answers continues. This case can be seen in above figure for k = 120 where we only have 32 mappers and comparing that with k = 32, we can see the much increase in response time.

Heuristics:

Instead of scheduling partitions for execution randomly, it would be better if we can reduce the total number of partitions loaded using heuristics. We have done these experiments to see the effect of each heuristic (discussed in Chapter 4) on query processing for all queries of DBLP graph. For initial analysis, we chose k = 1 and compared the response time for each heuristic. To see the effect of k, we increased k to 2, 4, 8 and 16 given that we have that many mappers available. Number of reducers for this set of experiments are fixed to 2.

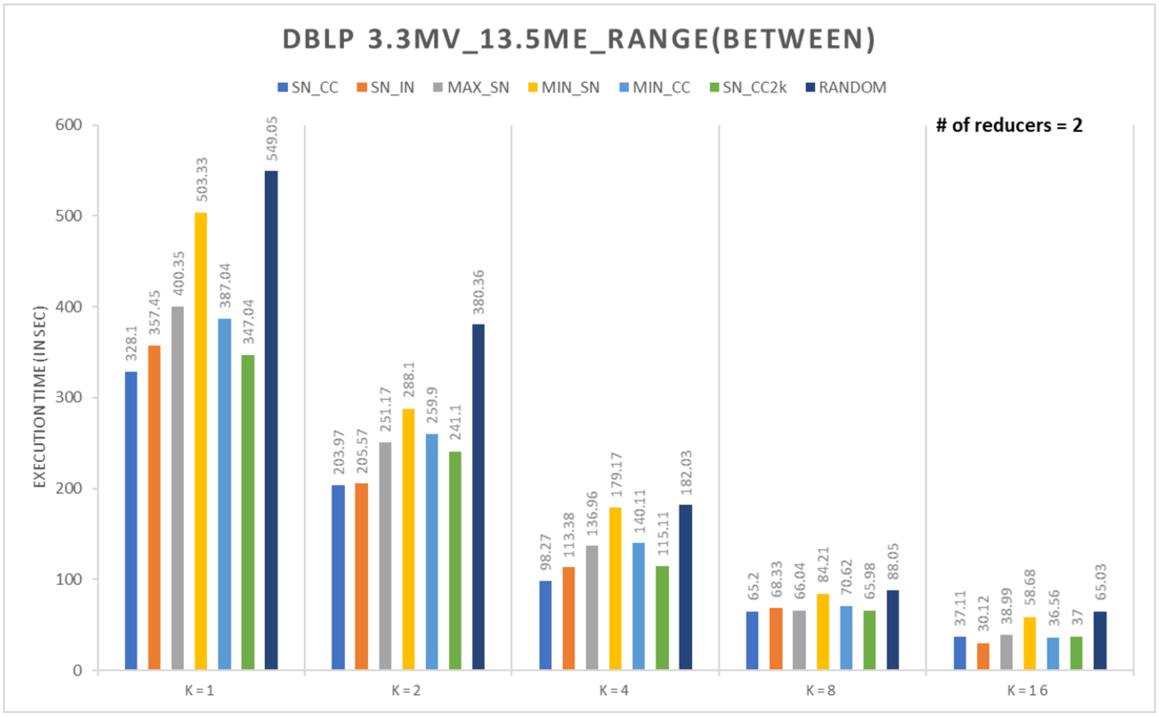


Figure 6.7: Comparison of all Heuristics for DBLP (BETWEEN query)

In case of BETWEEN query (Figure 6.7), for k = 1, SN_CC is better than others as it gives less response time. SN_CC$_{2k}$ and SN_IN are second and third best heuristics respectively. On the other hand, RANDOM takes worst time among all. MIN_SN takes worst time after RANDOM. For other k values, we see almost the same pattern. RANDOM takes worst time in all k values. MIN_SN is second worst heuristic in each case. While SN_CC is consistently better than others for all k values. SN_IN and SN_CC$_{2k}$ are the next top 2 heuristics.

For Comparison (>) operator (Figure 6.8), SN_CC$_{2k}$ is consistently better than other heuristics for all k values. SN_CC and SN_IN comes under top 3 heuristics that gives us good response time. Here also RANDOM has worst performance for any k value. Moreover MIN_SN is next worst heuristic.

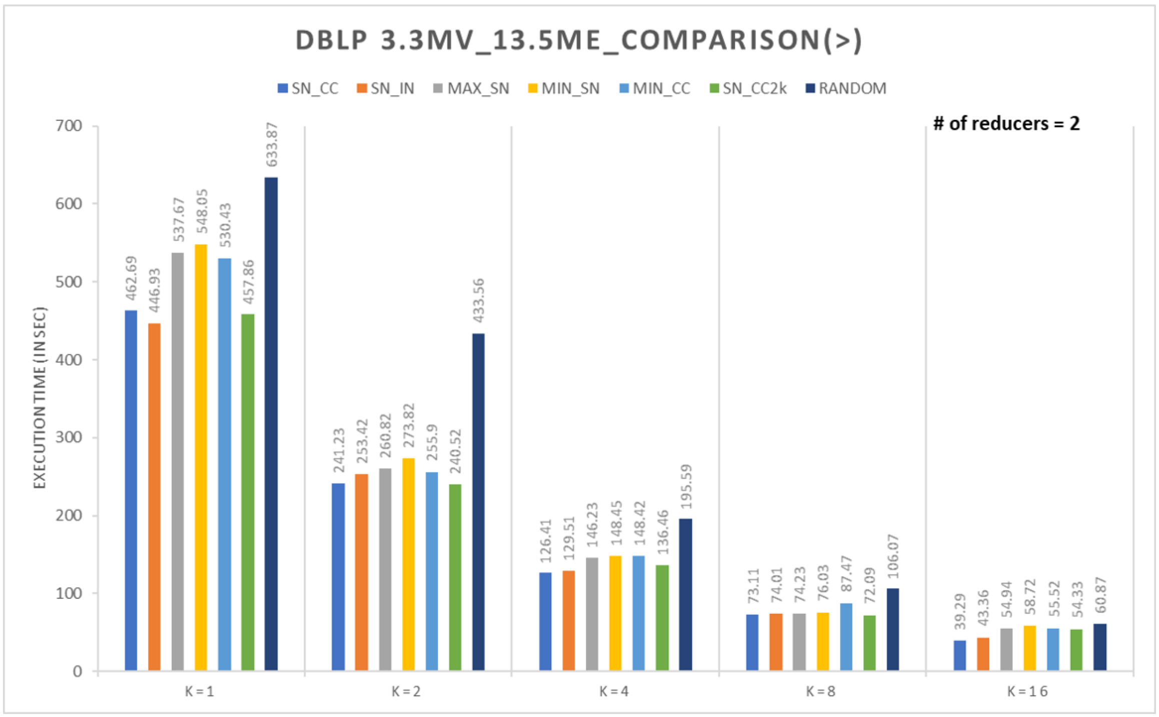


Figure 6.8: Heuristics comparison: DBLP (Comparison > query)

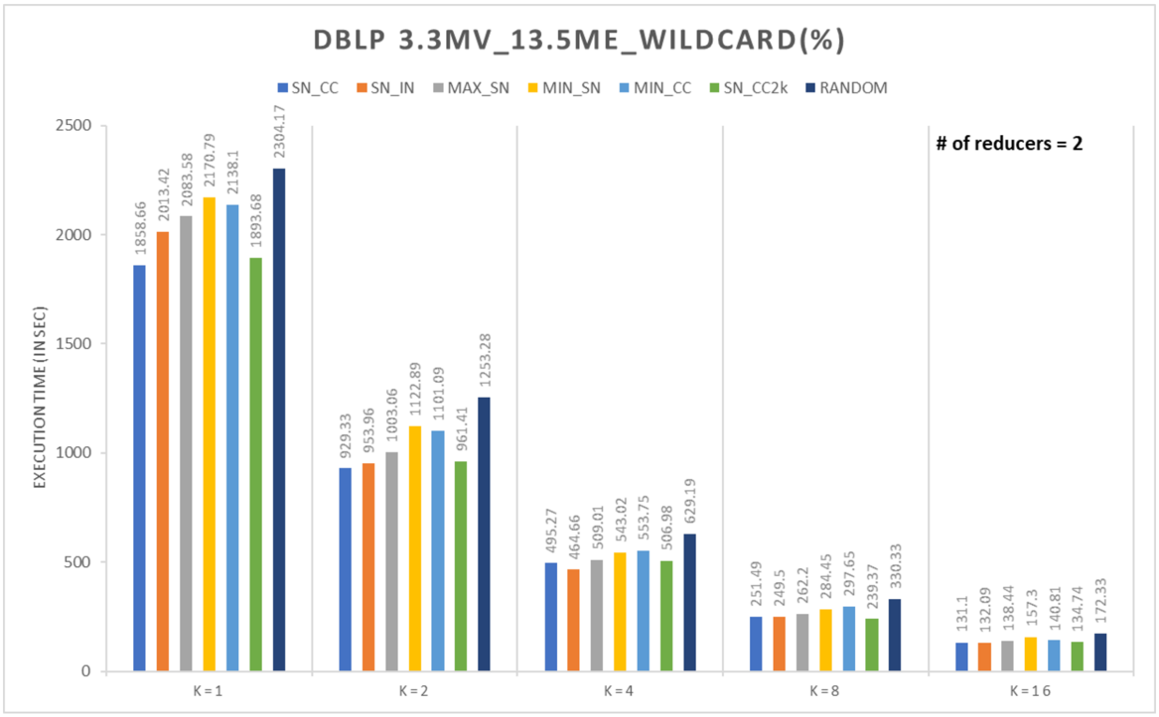


Figure 6.9: Heuristics comparison: DBLP (Wild Card (%) query)

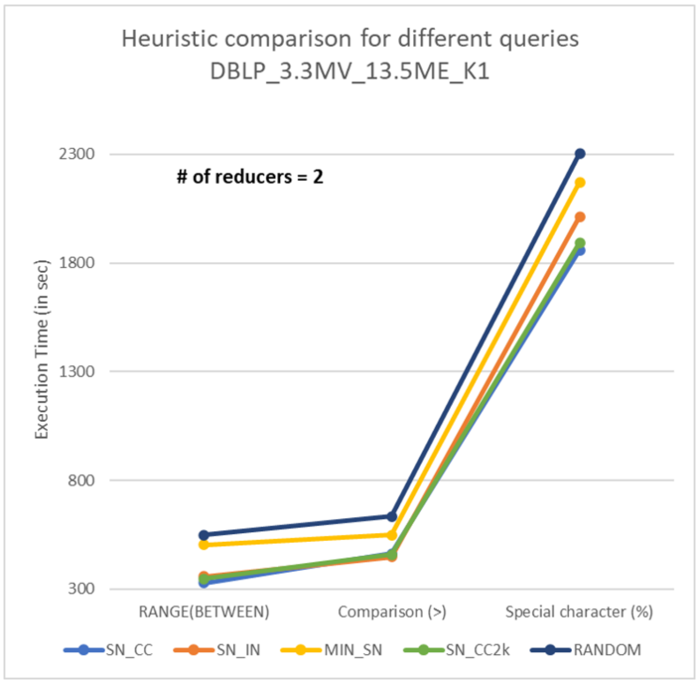


Figure 6.10: Comparing best and worst heuristics: DBLP

For wild card (%) query (Figure 6.9), the worst two heuristics are RANDOM and MIN_SN. The heuristic that gives better performance than almost all others is SN_CC$_{2k}$. The next top two heuristics for this query are SN_CC and SN_IN.

Comparing Heuristics:

To get the better understanding of the heuristics, we will compare top 3 and bottom 2 heuristics (5 in all) for all of the queries (figure 6.10) for k = 1. As analysed above, our combined heuristics (SN_CC, SN_CC$_{2k}$ and SN_IN) are coming in top 3. Further analysing these three heuristics, we can say that for BETWEEN and Comparison (>), we have lots of overlaps. While in case of Wild Card (%), SN_CC and SN_CC$_{2k}$ are better than SN_IN.

Load Factor Analysis:

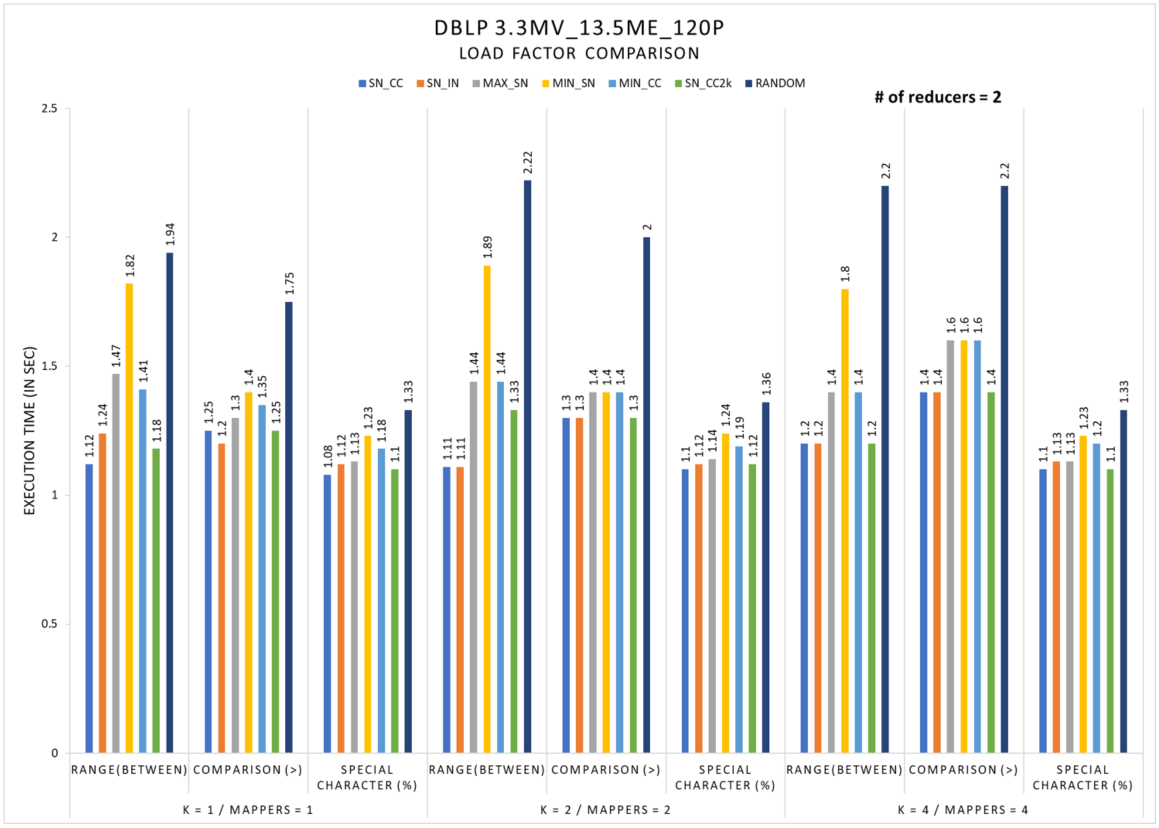


Figure 6.11: Load Factor analysis: DBLP

We can further extend our above heuristics comparison by taking load factor for each heuristic into consideration.

Figure 6.11 shows the load factor values for each heuristic for our given three queries. For k = 1, in case of BETWEEN query, SN_CC has lowest load factor value i.e. it requires less number of iterations and hence, less partitions are loaded for this heuristic. SN_CC$_{2k}$ is second best heuristic and SN_IN is third best heuristic in terms of Load Factor. Here, the load factor for RANDOM heuristic is more than any other. Hence more number of partitions are loaded for this heuristic and so it is not efficient. The second worst is MIN_SN. For all other queries, the top 3 heuristics that has lowest load factor values are SN_CC, SN_IN and SN_CC$_{2k}$. The presented analysis for load factor for all queries also matches with our previous heuristics comparison. We also

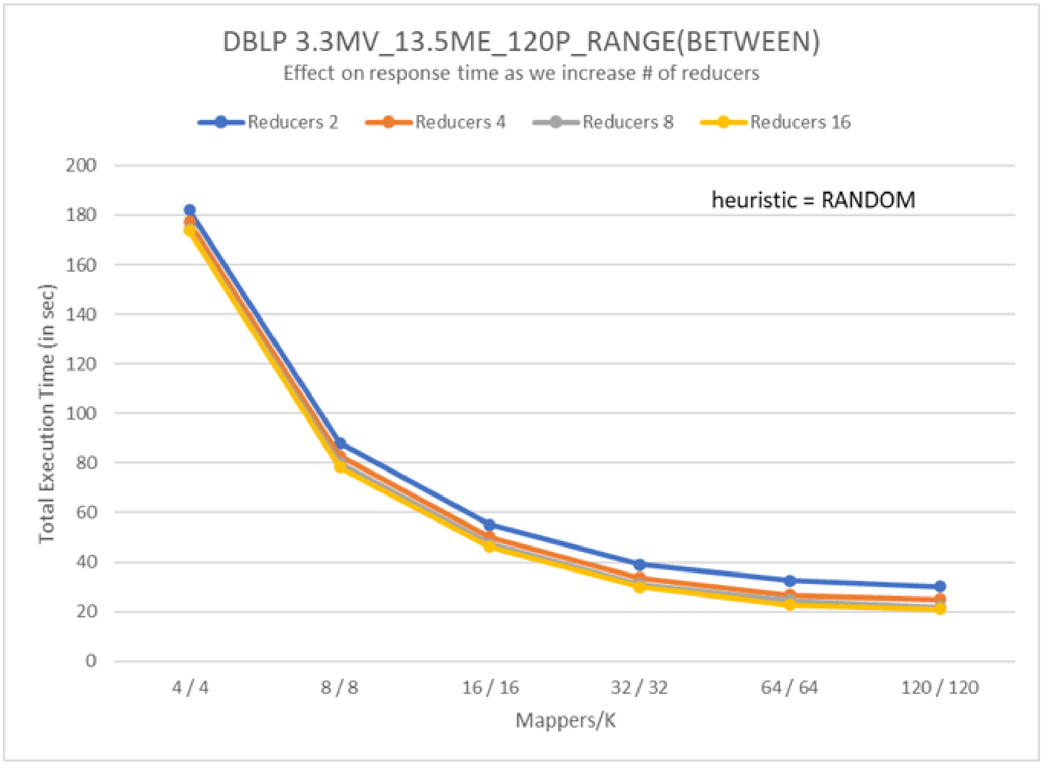


Figure 6.12: Changing the number of reducers (BETWEEN query)

see the similar pattern in load factor values for other values of k. In every scenario, the top 3 heuristics are SN_CC, SN_IN and SN_CC$_{2k}$.

Changing the number of reducers: Here, we have experimented with different number of reducers to see the effect on query processing. We have increased reducers from 2 to 4, 8 and 16.

As shown in figure 6.12, for our given query BETWEEN, for k = 4 as we increase the number of reducers from 2 to 4, the response time decreases, but there is not much gap between them. As we further increases the reducers we see slightly decrease in response time. We see same pattern for all k values. This is because we only use reducers for aggregating the partial answers. Our most of the execution

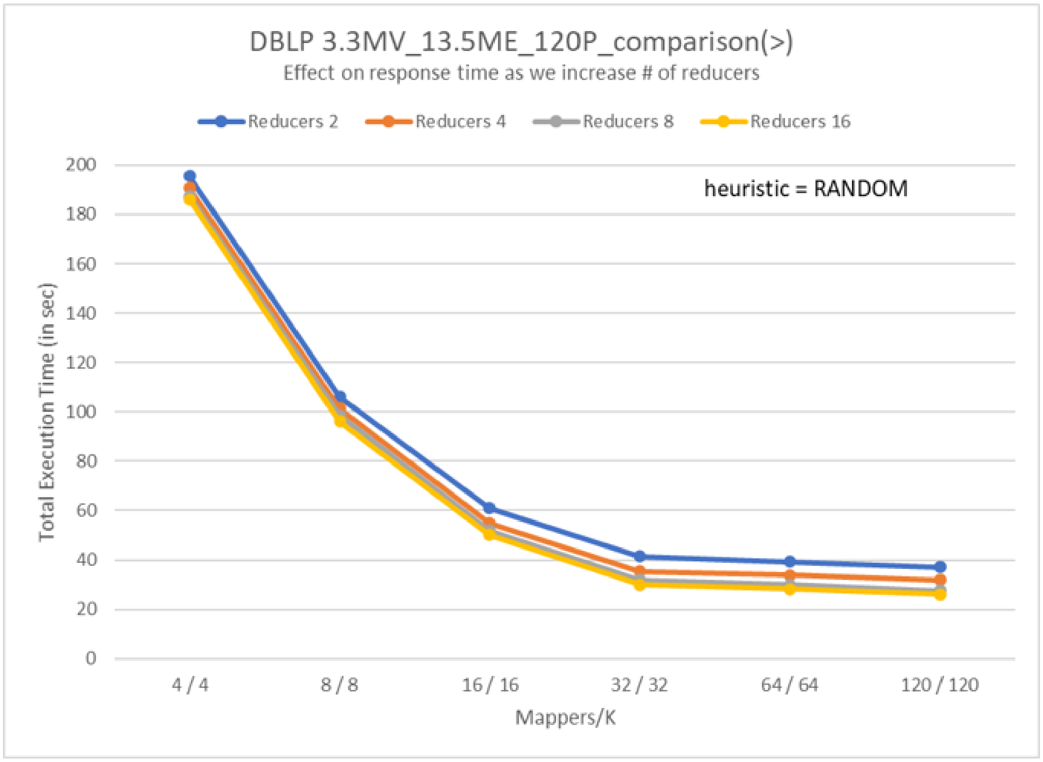


Figure 6.13: Changing the number of reducers (Comparison (>) query)

time happens inside the mapper where we load adjacency list and do query expansion. Hence increasing reducers doesn't have much effect on overall time.

For Comparison operator (>), for any k value, as we increase number of reducers from 2 to 4, the response time decreases. As we further increase the number of reducers, we see very less drop in response time.

For Wild Card query, as we increase the reducers, there is very slight decrease in response time and not much gap between them.

From the above set of experiments, we can infer that changing the number of reducers will not have much effect on the query processing.

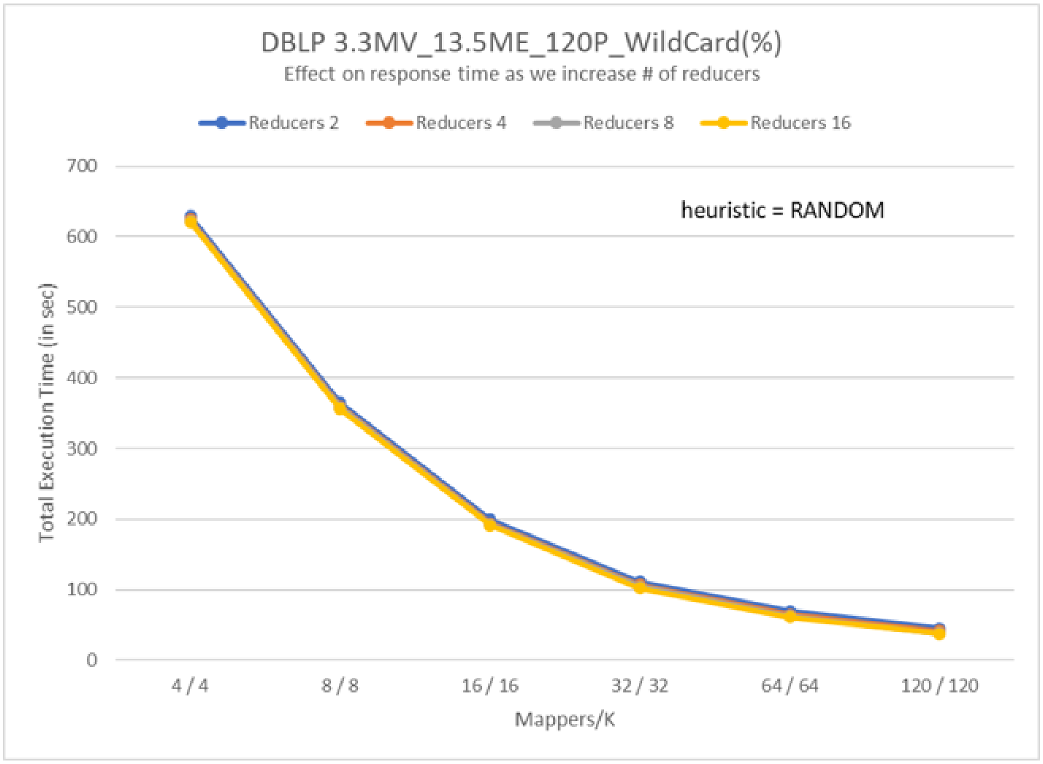


Figure 6.14: Changing the number of reducers (Wild Card (%) query)

6.4 Querying a partitioned IMDB graph

We have used IMDB dataset to show our analysis on different type of graphs. We have used bellow three queries for this dataset:

Query-1: "Find movies produced by "Adam Sandler" where the genres should be comedy **and** Sci-Fi but the year should not equal to 2000". This query features the AND operator. The total number of results are 24.

Query-2: "Find all actors (both Male and Female) who have worked in a movie owned by company "Paramount Pictures" where the genre of the movie is either Drama **or** Action." This query is a union query (OR operator). Because of the OR operator, this query has many results. The query has 6824 results.

Query-3: "Find movies along with their actors and producer information by company American Broadcasting Company (ABC) **after year 1970** where the genre

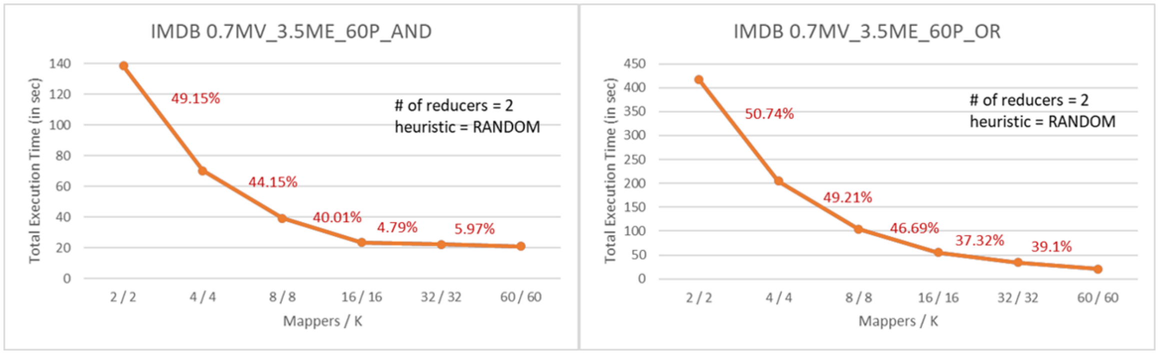


Figure 6.15: Speedup: IMDB (AND and OR queries)

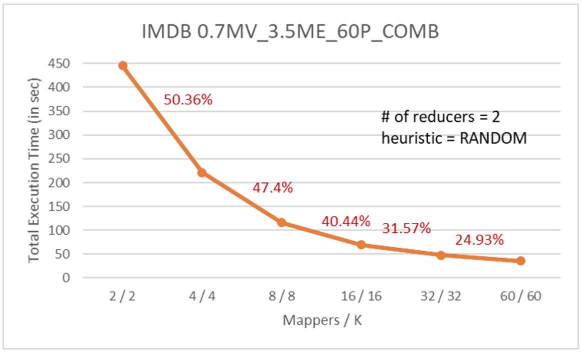


Figure 6.16: Speedup: IMDB (COMB query)

should **not** be comedy or horror.” The query has combination of comparison > and NOT operator. The results for this query are 7252.

Speedup:

We conduct this set of experiments to verify previously presented analysis of speedup on IMDB graph using different query types. Here, we will increase k and go until k = 60 because we have 60 partitions. The reducers are also kept to 2 and chosen heuristics is also RANDOM for performing these experiments.

For AND query (Figure 6.15), the response time is decreasing as we increase k. The speedup that we get when we increase k from 2 to 4 is 49.15%. But when we increase k from 32 to 60, we only get speedup of 5.97%. Best value of k for this query is 16 as we don’t see much speedup after that.

For OR query (Figure 6.15), the response time is decreasing with increase in k. For smaller k values, we are getting more speedup.

For COMB query also (Figure 6.16, we see decrease in response time with increase in k. Here also speedup is more for smaller k values.

Comparing all above three experiments, we can say that for all queries, we see decrease in response time as we increase k. But the decrease is not same in all cases. For smaller k values, we see speedup of almost 50% for all queries. But for larger k values, we see good amount of speedup only for queries OR and COMB. Further analysing the number of results that we get for this queries, we see that AND query has less number of results than other two. Hence, for the queries which will have many results from the graph, we will continue to see speedup as we increase k.

From both DBLP and IMDB experiments, we can say that, in general, for any query types we can increase k to as many processors as available to get good response time. But arbitrarily increasing k may not be good idea because we do not achieve that much of drop in response time for some queries. For the queries that have lots of answers, we can increase k = p for good speedup.

Heuristics:

We have analysed the performance of different heuristics on IMDB data as well for our given IMDB queries. We first analysed it for k = 1 and then increased value of k to 2, 4, 8 and 16.

Figure 6.17 shows the results for given AND query. For k = 1, as we compare the heuristics, we see that SN_IN is giving best response time. The other top two heuristics in this case are SN_CC_{2k} and SN_CC. The RANDOM gives worst performance here and MIN_SN is second worst. As k is increased, we see same pattern among these heuristics. SN_IN is consistently better than all others. SN_CC and

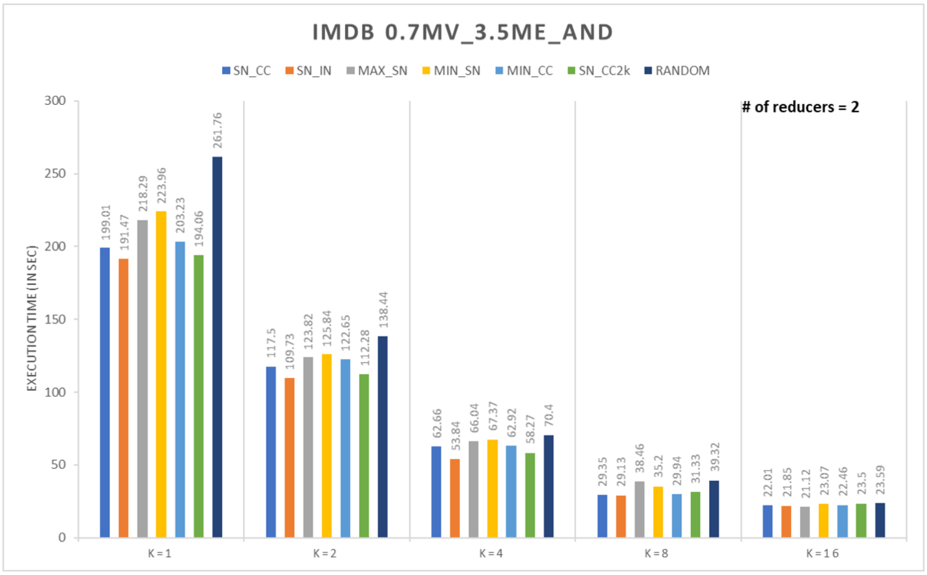


Figure 6.17: Heuristics comparison: IMDB (AND query)

SN_CC$_{2k}$ are among top 3. The RANDOM takes more execution time than any other heuristics. The second worst is MIN_SN.

For OR query, SN_IN is coming best for all k values. Then other two heuristics which are coming in next top 2 positions are SN_CC and SN_CC$_{2k}$. RANDOM is consistently worse than others and MIN_SN is second worst for all k.

In case of COMB query, SN_CC is consistently better than others. The next best heuristics are SN_CC$_{2k}$ and SN_IN. The worst two heuristics are RANDOM and MIN_SN.

Comparing best and worst heuristics:

Figure 6.20 shows comparison of top 3 heuristics and bottom 2 heuristics in terms of performance for all three queries. For all the cases, we see our combined heuristics are in top 3. For AND query, lots of overlaps between top 3 heuristics. For OR query, SN_IN is better than SN_CC and SN_CC$_{2k}$ which almost have same response time. FOR COMB query, SN_CC is best. And SN_CC$_{2k}$ and SN_IN comes in second best and third best heuristic. From this comparison we can say that for

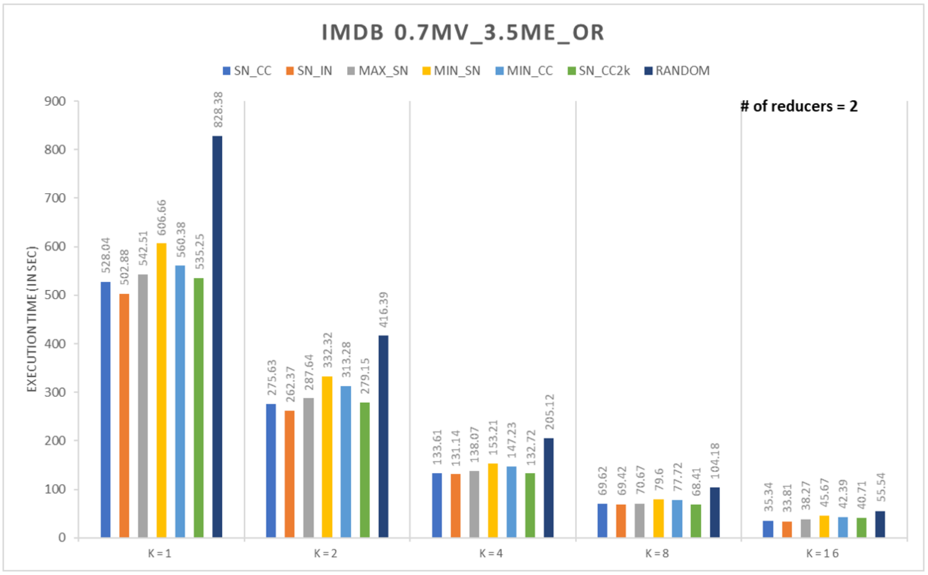


Figure 6.18: Heuristics comparison: IMDB (OR query)

our IMDB data, SN_CC gives better results for any query.

Load Factor Analysis:

We further compare all heuristics in terms of load factor to verify our analysis.

As shown in Figure 6.21, in case of k = 1, for AND query, SN_IN is better because it has lowest load factor value. The other two heuristics that give better load factor values are SN_CC and SN_CC$_{2k}$. The worst load factor value is in case of RANDOM heuristic. For other queries also top 3 heuristics are SN_CC, SN_CC$_{2k}$ and SN_IN. Similar pattern is seen in case of different k values. Moreover, this load factor comparison match with our previous presented analysis for the performance of heuristics.

We can infer from the presented heuristics analysis for both the datasets that for any kind of query, our combined heuristics are better than individual ones. Hence we can always choose those over individual ones to improve querying performance on partitioned graph. Also, as we further analyse these three heuristics on different query

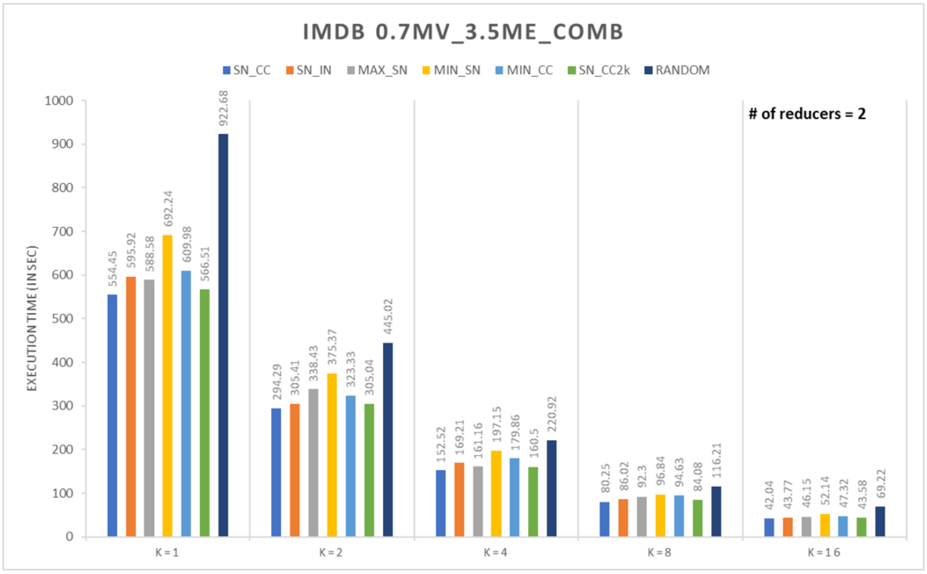


Figure 6.19: Heuristics comparison: IMDB (COMB query)

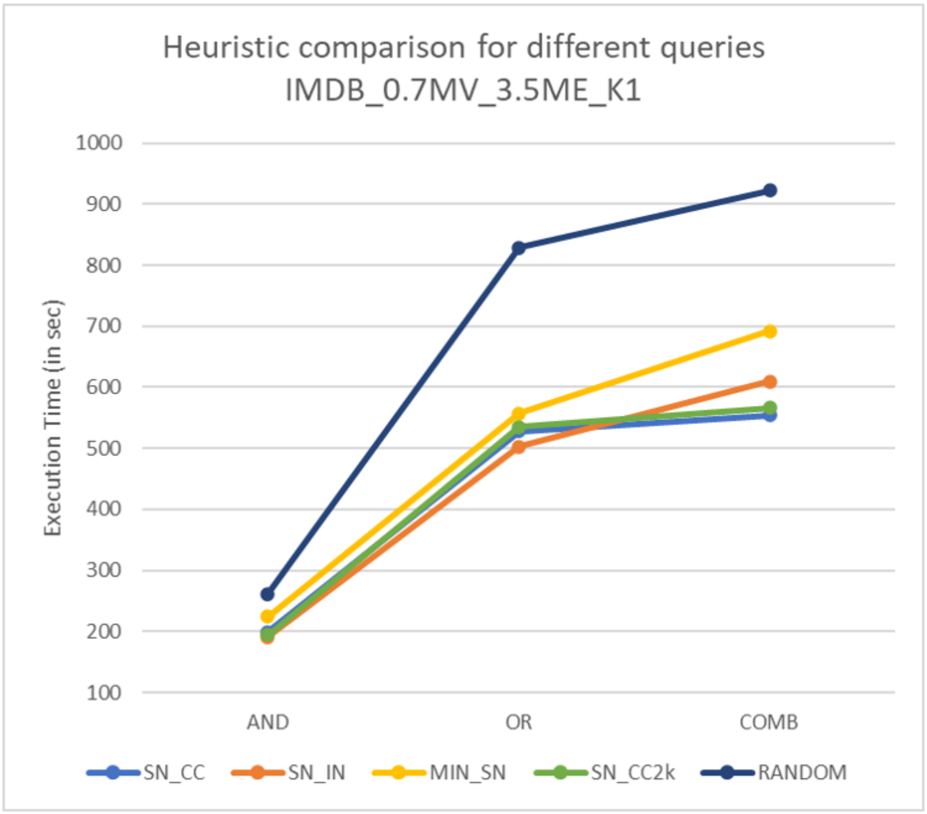


Figure 6.20: Comparing best and worst heuristics: IMDB

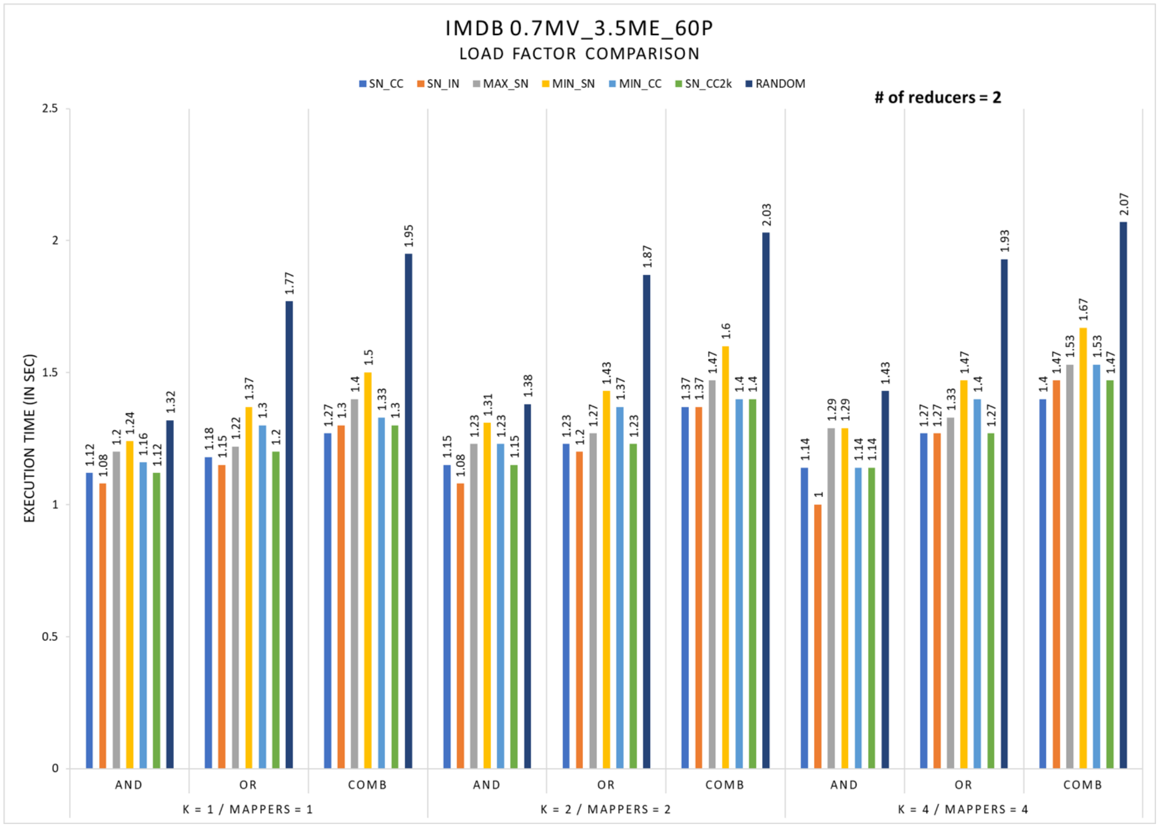


Figure 6.21: Load Factor analysis: IMDB

types, for BETWEEN and Comparison (>), all three are identical. But for Wild card (%), the two heuristics SN_CC and SN_CC$_{2k}$ has better response time than SN_IN. In the case of AND and OR queries, all three are almost identical. While for COMB query, SN_CC and SN_CC$_{2k}$ are very close and they perform better than SN_IN. So we can say, in genreal, SN_CC and SN_CC$_{2k}$ are good for any query.

In summary, we have presented exhaustive experiments on radically different kind of datasets to present the results of our findings. We have used extensive computation environment to show the validity in our approach for huge volume of data. The queries that we validated covers broad range of characteristics that is comparable to relational databases. We have validated our approach for different scale of computation and verified proposed heuristics for various scenarios.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This thesis presents MR_QP, a scalable graph querying system using Map/Reduce distributed framework. The system employs range based partitioning to scale to any size graph by dividing the graph into varying size partitions and process queries on those partitions in distributed environment. The important aspects of this approach is we process queries on graph representation itself rather than using data in any other form such as RDF, support expressiveness of the query, correctly verifying the results over graph partitions, accommodate as many resources as available to exploit parallelism and also heuristics for efficient query evaluation.

To leverage parallel processing, we process k partitions at a time in parallel. Here, we can adjust k to utilize as many partitions and processors as needed to process in parallel. We do systematic constrained expansion for all start nodes in a partition according to query plan and collect the completed answers at the time of processing and if any answer continue to different partition, then we carry partial answers to those partitions to process in next iterations. To identify which k partitions to process in each iteration, we have used meta information such as start nodes, intermediate nodes and connected components in each partition. We have extended earlier proposed heuristics to use this additional meta information to schedule the partitions to minimize the response time. To verify the correctness and efficacy in our approach, we have done exhaustive experiments on two radically different type of graphs having millions of entities. Using every characteristic of query, we have performed our experiments on computations of very large scale and verified our findings. Moreover, we verified the proposed heuristics on every kind of query that signifies the importance of it in query processing.

Future work includes using other partitioning schemes such as arbitrary partitioning to see the effect of it in query processing. Also we can extend our approach for other distributed framework such as Spark to verify the findings.

REFERENCES

[1] Nosql definition: Next generation databases mostly addressing some of the points : being non-relational, distributed, open-source and horizontally scalable. [Online]. Available: http://nosql-database.org/

[2] N. Leavitt, "Will nosql databases live up to their promise?" *Computer*, vol. 43, no. 2, pp. 12–14, 2010.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[5] Neo4j. [Online]. Available: https://neo4j.com

[6] R. Giugno and D. Shasha, "Graphgrep: A fast and universal method for querying graphs," in *Object recognition supported by user interaction for service robots*, vol. 2. IEEE, 2002, pp. 112–115.

[7] 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*, 2004, pp. 335–346.

[8] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad, "Fast best-effort pattern matching in large attributed graphs," in *Proceedings of the 13th ACM SIGKDD*

international conference on Knowledge discovery and data mining, 2007, pp. 737–746.

[9] Facebook active users stats. [Online]. Available: https://www.internetlivestats.com/watch/facebook-users/

[10] Dblp data stats. [Online]. Available: https://dblp.uni-trier.de/statistics/recordsindblp

[11] Imdb data stats. [Online]. Available: https://www.imdb.com/pressroom/stats/

[12] A. Goyal, "Qp-subdue: Processing queries over graph databases," 2015.

[13] J. D. Bodra, "Processing queries over partitioned graph databases: An approach and it's evaluation," 2016.

[14] E. Prud'hommeaux, "Sparql query language for rdf, w3c recommendation," *http://www. w3. org/TR/rdf-sparql-query/*, 2008.

[15] M. Cogorno, J. Rey, and S. Nesmachnow, "Fault tolerance in hadoop mapreduce implementation," 08 2013.

[16] S. Das, "Divide and conquer approach to scalable substructure discovery: Partitioning schemes, algorithms, optimization and performance analysis using map/reduce paradigm," Ph.D. dissertation, 2017.

[17] J. Huang, W. Qin, X. Wang, and W. Chen, "Survey of external memory large-scale graph processing on a multi-core system," *The Journal of Supercomputing*, vol. 76, no. 1, pp. 549–579, 2020.

[18] Titan distributed graph database. [Online]. Available: http://thinkaurelius.github.io/titan/

[19] Infinitegraph. [Online]. Available: http://www.objectivity.com/infinitegraph

[20] S. Sakr and G. Al-Naymat, "Graph indexing and querying: A review," *IJWIS*, vol. 6, pp. 101–120, 06 2010.

[21] J. Cheng, Y. Ke, W. Ng, and A. Lu, "Fg-index: towards verification-free query processing on graph databases," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007, pp. 857–872.

[22] J. Cheng, Y. Ke, and W. Ng, "Efficient query processing on graph databases," *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 1, pp. 1–48, 2009.

[23] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč, "Foundations of modern query languages for graph databases," *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, pp. 1–40, 2017.

[24] The neo4j team. [Online]. Available: https://neo4j.com/docs/

[25] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri, "Towards a query-by-example system for knowledge graphs," in *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, ser. GRADES'14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–6. [Online]. Available: https://doi.org/10.1145/2621934.2621937

[26] G. Karypis and V. Kumar, "Multilevel graph partitioning schemes," in *ICPP (3)*, 1995, pp. 113–122.

[27] P. Sanders and C. Schulz, "Engineering multilevel graph partitioning algorithms," in *European Symposium on Algorithms*. Springer, 2011, pp. 469–480.

BIOGRAPHICAL STATEMENT

Harshit Modi was born in Palanpur, Gujarat, India. He received his Bachelors Degree in Information and Communication Technology from Dhirubhai Ambani Institute of Information and Communication Technology, India in May, 2016. He worked as a software developer at Kuliza Technologies, Bangalore, India from June, 2016 till July, 2018. He started his Masters studies in Computer Science at The University of Texas, Arlington in August, 2018 and received his Masters degree in May 2020. His research interests include graph mining, big data engineering and machine learning.