EFFECTIVE KEYWORD SEARCH ON LARGE SCALE GRAPHS IN A DISTRIBUTED SYSTEM

by

MENGYU LI

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Master of Philosophy
in Computer Science and Engineering

June 2013, Hong Kong
Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

MENGYU LI
18 June 2013
EFFECTIVE KEYWORD SEARCH ON LARGE SCALE GRAPHS IN A DISTRIBUTED SYSTEM

by

MENGYU LI

This is to certify that I have examined the above M.Phil. thesis and have found that it is complete and satisfactory in all respects, and that any and all revisions required by the thesis examination committee have been made.

PROF. LEI CHEN, THESIS SUPERVISOR

PROF. MOUNIR HAMDI, HEAD OF DEPARTMENT

Department of Computer Science and Engineering

18 June 2013
ACKNOWLEDGMENTS

I would never have completed this work without the help from many people. First of all, I would like to thank my supervisor, Professor Lei CHEN, for his years of supervision, advice, and encouragement. I have learned from him how to develop, evaluate, express, and defend my ideas. He also taught me much knowledge in life. These skills are important for my later study and work.

I would like to thank the members of my thesis committee, Professor Ke YI and Professor WONG, Raymond Chi-Wing, for their insightful comments on improving this work.

I also would like to thank my colleagues in HKUST – Dr.ZHANG Xiaofei, Mr. TONG Yongxin, Mr.CAO Chen, Mr. ZHANG Chen, and many others. We have talked a lot about research and life. In daily life, we have been good friends. Without them, my graduate study in HKUST would not be so colorful.

Last but not least, I would like to thank my parents, my friends, for their understanding and support.
# TABLE OF CONTENTS

Title Page i
Authorization Page ii
Signature Page iii
Acknowledgments iv
Table of Contents v
List of Figures vii
List of Tables viii
Abstract ix

Chapter 1 Introduction 1
  1.1 Our Concern 2
  1.2 Thesis Outline 5

Chapter 2 Related Work 6
  2.1 Keyword Search 6
    2.1.1 Keyword Search on XML Data 7
    2.1.2 Keyword Search on Relational Databases 8
    2.1.3 Keyword Search on Graphs 9
  2.2 Query Process in the Distributed System 12
    2.2.1 Structured Query Process in the Distributed System 12
    2.2.2 Other Query Process in the Distributed System 13
    2.2.3 Graph Partition 13

Chapter 3 Problem Definition 15

Chapter 4 Naive Search Solution 17
  4.1 General Framework 17
  4.2 Naive Algorithm 18
  4.3 Analysis 20
Chapter 5  Index Construction  23
  5.1 Local Index  24
  5.2 Global Index  26
  5.3 Index Optimization  27
  5.4 Index Implementation and Maintenance  29
  5.5 Hardness of the Index Optimization  32

Chapter 6  Advanced Search Solution  34
  6.1 General Framework  34
  6.2 Advanced Algorithm  35

Chapter 7  Experiments  40
  7.1 Index Construction and Maintenance  40
    7.1.1 Correctness  40
    7.1.2 Construction Efficiency  41
    7.1.3 Updating Efficiency  42
  7.2 Search Efficiency  43
    7.2.1 Non-distributed VS. Distributed Search  43
    7.2.2 advanced VS. Other Solutions  44
    7.2.3 advanced with Varied $K_0$  46
    7.2.4 advanced with Varied Graph Size  46
  7.3 Data Shipment  47
    7.3.1 advanced VS. Other Solutions  47
    7.3.2 advanced with Varied $K_0$  48
    7.3.3 advanced with Varied Graph Size  49

Chapter 8  Conclusion  50

References  51

List of Publications  56
LIST OF FIGURES

1.1 Distributed Graphs 3
4.1 Naive Search Examples 21
5.1 Local Index Construction 25
5.2 Global Index Construction 27
5.3 Invalidity in Global Index 31
7.1 Compare the Correctness of the Two Measurements 41
7.2 Compare the Building and Updating Efficiency of the Two Measurements 42
7.3 Efficiency of Distributed VS. Non-distributed 44
7.4 Efficiency of advanced VS. Other Solutions 45
7.5 advanced’s Efficiency with Varied $K_0$ 46
7.6 advanced’s Efficiency with Varied Graph Size 47
7.7 Data Shipment of advanced VS. Other Solutions on DBLP 48
7.8 Data Shipment of advanced VS. Other Solutions on Yago 48
7.9 advanced’s Data Shipment with Varied $K_0$ 49
7.10 advanced’s Data Shipment with Varied Graph Size 49
LIST OF TABLES
Keyword query is more user-friendly than the structured query (SQL, XQuery, SPARQL) as it requires no pre-knowledge about the underlying schema of the databases. In the recent decade there has been much work focusing on the keyword search in both the structured databases and unstructured databases. Particularly, since the labeled graph model can generally present both the structured and unstructured databases, graph keyword search has been a hot topic in the database community. Furthermore, with the rapid development of semantic web, the graph data has grown to a huge scale and has to be stored and processed in a distributed environment. However, no existing approach can be directly applied to answer the keyword queries in the distributed graphs. The challenges exist in two-fold, the poor locality exists in the keyword query answers may lead to high communication cost for graph exploration and the huge traffic load for shifting the data among different machines. To address these two challenges, in this thesis, we first propose to optimize the index in order to minimal the distributed query processing cost within the space budget on each machine; Then, based on this indexing scheme we introduce an advanced distributed query processing algorithm, which includes an optimal scheduling problem defined to reduce traffic and the time cost, and we propose a greedy algorithm with a factor-2 approximation to the optimal schedule problem; At last, we verify the effectiveness and efficiency of our solution with large scale real datasets.
With the development of Internet, increasing number of people would like to obtain useful information from the web, however, structured query mechanism such as SQL and SPARQL queries, seems too difficult for non-professional users since they often lack the knowledge of the structure schema of the database. For example, we want to search for a book about “distributed systems” written by Jack. If only the structural query is allowed, we must know some critical database schema information, such as the tables, attributes and keys, which is usually unavailable to the common users. However, with the keyword query mechanism we can simple issue a query like “distributed system, Jack” and it is likely that we can get the desired result. The popularity of search engines such as Google.com and Bing.com also demonstrate that keyword search is well received by users.

Due to the importance of the keyword search, it has drawn much attention from the Database community in the recent decades, and there has been much work discussing keyword search in many different data models, such as on the relational databases, XML data, HTML files and the graphs. Among all the different aspects, keyword search over graphs become more and more popular recently, since graph can present other database models. To be more specific, relational database can be presented by projecting of the table tuples and the primary-foreign-key relationship to vertices and edges in graph respectively; Semi-structured data like XML data can be presented by mapping the XML elements and parent-child relationships to the vertices and edges respectively; Unstructured data like HTML files in the semantic web can be presented by modeling web pages...
to vertices and hyper-links to edges. In a word, studies on the graph keyword search problem play an significant role in the database area.

In terms of the data scale, the data amount grows quite rapidly nowadays. For example, Facebook has more than 900 million users in less than ten years, and Wikipedia has more than 4 million English articles up to now. Consequently, one single machine shows incapacity in storing and processing the data of such large scale, so distributed systems such as Trinity [2], MapReduce [9] and Pregel [25] have emerged to handle large-scale graphs in real-world applications. In a distributed system which is consisted of several machines, each machine only holds part of the data, and all the machines cooperate to accomplish the tasks.

1.1 Our Concern

Observed from the importance of keyword search and distributed graphs, it is a compelling need to solve the keyword search problem on large graphs in the distributed systems. Unfortunately, although there have been many existing methods about how to effectively answer keyword queries over graphs [12] [21] [19] [32] [42], etc. and how to process graph queries in the distributed system [46] [24] [33], etc. (See Chapter 2), they cannot be directly applied to solve the keyword search problem in the distributed graphs. The following example briefly illustrates the problem.

Example 1.1 (Distributed Keyword Search) In the non-distributed system, a classical idea to solving the keyword queries is to do the backward graph exploration [6] with the help of index. Specifically, the exploration traces backwardly from the keyword matching nodes to find the query answers, and the index provides guidance for efficient exploration. However, in the distributed system this idea does not fit because 1) the entire graph has been partitioned into many distributed graphs, and it is no longer feasible to build the
index for the entire graph using the existing methods; 2) we need additional handling of the inter-machine search in the distributed system. Therefore, the graph keyword search problem in the distributed system calls for new solution.

Therefore, in this work we study how to effectively answer the keyword queries in the distributed graphs, and this problem is non-trivial since there exist the following two difficulties.

**Difficulty 1**: How to make the keyword query processing efficient in the distributed system?

According to the previous works [12] [21] [19] [32] [42], etc., keyword search is in fact a graph exploration procedure. As for the distributed system, the exploration happens both within machine and across machines, and if we explore the graph without any useful guidance, it would seriously reduce the efficiency, the following example illustrates this case.

**Example 1.2 (Inter-machine Search)** As shown in Figure 1.1, the entire graph is partitioned into three machines: $M_1$, $M_2$ and $M_3$, in a distributed system (the dashed ellipses show the partitions of the whole graph). A query answer may cover multiple machines. Suppose a query answer is the subgraph containing $a_2$, $p_1$ and $p_2$, and $p_1$ and $p_2$ are the keyword matchings (backward exploration starts from keyword matchings). To find this
answer; firstly we discover the keyword matching nodes $p_1$ and $p_2$ in the machine $M_3$ and $M_2$, resp., and then we need to trace backward along all the neighbor nodes of $p_1$ and $p_2$, resp., until we find the node $a_2$ in $M_1$ which connects to both $p_1$ and $p_2$. We can see that this aimless exploration wastes much time since all the related neighbor nodes are scanned even if they are not in the query answers.

To accelerate the graph exploration, the effective solution is to build an index which can guide the exploration to avoid useless tracing. For example, the BLINKS [12] index pre-computed the shortest path information between the nodes. However, index construction becomes more tricky in the distributed graphs. This is mainly because, usually the graph stored in each machine is already quite large and there is not much space left to construct index as complex as BLINKS. Therefore, it is difficult to construct an effective index for the distributed graphs which can ensure the efficiency of the query processing.

**Difficulty 2**: How to schedule the machines in the distributed system so that the query can be answered with the minimal time and data shipment?

Known that the query processing requires the exploration across different machines, and all the machines in the distributed system should cooperate nicely to avoid cases which undermine the functioning of entire system, such as too much message exchange in the system or certain machine being excessively overloaded. So, it is important to schedule the system optimally so that the query processing can be accomplished with the minimal time, as the makespan problem requires, and also with the minimal traffic cost. This is not trivial work because the makespan problem alone is an NP-Complete problem, and to achieve minimal time cost and data shipment of the system processing at the same time is undoubtedly hard.

In this thesis, we overcome these two important problems by proposing two approaches as below:
1. We propose an adaptive indexing scheme which can provide the most guidance information for the search within the space budget of each machine in the system. And, in order to offer effective guidance during the query processing, indexes are delicately designed to fit different roles of the machines in the system.

2. We propose the scheduling optimization problem which aims to finish the query processing with the minimal time cost and data shipment. We prove the hardness of the problem, and propose a greedy algorithm which is a factor-2 approximation to the optimal makespan, and we propose a set of heuristics which ensure the data shipment is saved as much as possible.

1.2 Thesis Outline

The rest of the thesis is organized as follows.

In Chapter 2 we review the literature and formally define our problem in Chapter 3.

In Chapter 4, we propose a naive search algorithm and analyze its drawbacks in two aspects, and then in order to improve its performance, we propose an optimal indexing cost model in Chapter 5.

In Chapter 6, we propose the advanced search algorithm based on the optimal indexing scheme and also introduce the optimal scheduling problem which aims to minimize the makespan and data shipment of the system’s operation.

Our work is verified in Chapter 7 with extensive experiment results.

Finally, we conclude the work in Chapter 8.
CHAPTER 2

RELATED WORK

In this chapter, we review the related literature of our work. The review can be divided into two categories: One is about the previous graph keyword search approaches in different data models; The other is about the query processing in the distributed system. The two aspects are elaborated in the following two sections, respectively.

2.1 Keyword Search

Since the keyword search provides a simple and user-friendly way to retrieve information from databases of complex structures, there have been much work focusing on the keyword search in the database community. In this section we review the different keyword searching techniques in relational databases, XML data and graphs, since both relational databases and XML data can be presented by the graph model. Generally speaking, the graph keyword search faces three main problems as follows [37] [28] [27] [18]:

1. How to properly define an answer to the keyword query?

On the Web the answer can simply be a web document containing all the keyword, but it is not straightforward to find the counterpart of the notion of “document” in the graphs. Therefore, it is important to define what kind of subgraph can be a query answer.

2. How to rank the query answers?

There can be numerous subgraphs which can be the query answers, so the ranking scheme should mark them according to their relevance to the users’ queries, and the
most relevant answer should be at the top of the query result.

3. How to efficiently search the answers?

This problem is greatly relevant to the first two problems, so different query answer definition and ranking scheme require different searching algorithm. Many searching approaches reduce the searching time cost by reducing the search space, indexing and pruning.

In the following three subsections, we first briefly review keyword search on XML in terms of the three problems, and then on the relational databases, and finally the graph keyword search.

2.1.1 Keyword Search on XML Data

As for the query answer semantics, almost all the XML keyword search work defines the keyword query answers based on the notion of lowest common ancestor (LCA), namely, the answer to the keyword query is a subtree in the XML data which covers all the keywords and the subtree is rooted at the lowest common ancestor of all the keyword matching nodes [23] [37] [34]. Based on the LCA semantics, [14] [39] [40] proposed the SLCA semantics to define the query answers. Specifically, in these work the query answers were defined as a subtree rooted at a LCA node and none of its child nodes can also be an LCA node to the query. The SLCA-based query answers ensured a tight connection within the answer subtree and pruned out many redundant subtrees. But the SLCA-based query answer can be too strict that some qualified query answers are missed, so [11] proposed ELCA i.e. exclusive LCA. The ELCA-based query answer semantics returned a bigger result set compared to the SLCA semantics.

As for the ranking scheme, XRank [11] used a ranking function similar to the PageRank method [7], which evaluated the importance of each element in the XML data and ranked
the query answers according to the importance of the elements. But this ranking method did not consider the notion of keyword proximity, to address this problem, XSEarch [26] used the inter-connection between the XML elements to measure the keyword proximity, this work also used node-level tfidf based method to evaluate the importance of keyword matching elements.

As for the searching algorithms, XRank [11] was based on the ELCA semantics and [22] [14] [39] [40] [16] used the SLCA semantics, and they all used DIL (Dewey Inverted List) in their processing. DIL algorithm encodes ancestor-descendant relationship into IDs stored in the inverted list, so in light of the Dewey IDs the searching can run efficiently via the fast checking of parent-child relation between the XML elements. In addition, XKSearch [39] took the frequency of keywords in the XML files into consideration, and avoided full can of all the inverted lists, which worked well when at least one of the keywords has low frequency.

In sum, the keyword search in XML files have made many contributions such as the notion of SLCA and ELCA, the ranking scheme and DIL-based searching algorithms. However, due to the tree structure of XML data, it is more straightforward to define query answer and search for the answers than in the free-schema graphs. For instances, the DIL method cannot be directly applied to solve the keyword search on graphs.

2.1.2 **Keyword Search on Relational Databases**

There has been much work exploring the keyword search on relational databases [4] [15] [6] [13] [19] [5] [43] [27] [28]. In this subsection, we review these papers in terms of the three problems.

For the query answer semantics, there is no straightforward mapping from the “doc-ument” in Web to the contents in the relational databases since the logical unit in the relational databases often needs to be joined from multiple tables. Therefore, all the work
about keyword search in relational databases [4] [15] [6] [13] [19] defined the query answer as the joined tuples covering all the keywords.

For the ranking scheme, these work [4] [15] [19] adopted similar ranking strategy: the answers are ranked w.r.t the number of joins needed in the query answers, and the answers consisting more joins are ranked lower, since it is hard to understand the joining of many tables. BANKS [6] considered both the weight of tuple and the weight of edge in the tuple join tree to rank the answer. [13] used the IR-style ranking methods in the ranking scheme in a direct manner. Also, much work proposed to solve this problem by formulate SQL queries for the keyword queries and then rank these SQL queries accordingly [45] [38].

As for the searching algorithm, both DBXplorer [4] and DISCOVERY [15] both took advantages of the physical database design in order to answer the query efficiently. DBXplorer maintained a symbol table which mapped a keyword to the rows in the tables containing this keyword, and then based on the keyword matching rows, DBXplorer enumerated all the possible join trees which were further transferred to SQL statements. In addition, DISCOVERY [15] and [45] [38] proposed to find the optimal SQL statement generating plan.

Similar to the keyword search in XML data, the solution of relational databases cannot be directly applied to the keyword search on graphs, because the structure in the relational databases is restricted to the schema of tables but the free-schema graphs do not have such limitations.

2.1.3 Keyword Search on Graphs

Graphs formed from relational databases and XML data are confined by their schemas, which not only limit the search space of keyword query, but also help shape the query semantics [37]. So, in this subsection we discuss how previous graph keyword search work
[12] [21] [19] [32] [42] [5] addressed the three problems.

For the query answer semantics, some work including [12] [19] and [6] used a similar idea to the LCA semantics in the keyword search on XML data, and they defined the query answer to be the minimal rooted tree covering all the keywords embedded in the graph. Differently, ObjectRank [5] defined the query answer as a node that has high authority on all the keywords in the query.

For the ranking scheme, [19] and [6] evaluated the query answers in two aspects: edge weight and node importance, where the total path weight of the query answer can reflect the strength of connection between the root and the keyword matching nodes, and the total node score reflects its prestige. In addition, [12] required that the returned answers must have different roots, which can prevent the case that many query answers of similar semantics are rooted at a hub node. Differently, ObjectRank [5] scored the query answers based on its connections to the keyword matching nodes, and the connections are measured with different edge weights and node types.

As for the searching algorithm, there are mainly three kinds of graph exploration methods [37]. Firstly, BANKS [6] first proposed the Backward Graph Search. This method works as follows: it maintains a cluster for each keyword $k_i$, which contains the nodes which are reachable to $k_i$, and initially the cluster is consisted of all the keyword matchings of $k_i$; in each step of expansion, the method selects one cluster and one of this node to trace backwardly and the newly reached nodes are added to the cluster; When one node is contained in all the clusters, an answer root is found. BANKS [6] also proposed two expansion strategies: Equi-distance within cluster and Distance-balanced across clusters. [12] later proved that an optimal backward search algorithm must follow the strategy of Equi-distance expansion in each cluster. [12] also proved that Distance-balanced across clusters is not optimal and can lead to poor performance, instead, [12] proposed to choose the cluster with the smallest cardinality. Also, to address the problem
of Distance-balanced across clusters expansion in Backward Searching, [19] proposed a bidirectional search. This method enables forward tracing along the nodes based on the heuristic activation factors on each node, which can predict whether a node is reachable to the keywords. But the bidirectional search has no worst-case performance guarantee, and it can be as aimless as the backward search without additional connectivity information.

The third method is the index-based graph search proposed by BLINKS [12]. [12] proposed to use pre-computed indexes to help determine whether a node can reach a keyword without the step-by-step graph exploration. As the main concern of our work is the efficient query processing problem and we also use the idea of index-based search, we then compare in details our work with BLINKS.

Our search algorithm introduced in Chapter 6 uses the index-based search idea similar to the BLINKS index in [12]. BLINKS did the indexing in the following way: For each term in the system, it indexed all the nodes reachable to this term in the ascending order of their distances to the term. Also for every node in the graph, it indexed its distance to all the terms together with the next nodes along its path to this term. When the graph to be indexed is rather large and the above indexing method is inefficient to construct, BLINKS proposed to first partition the graph to smaller ones and then build the indexes on the partitioned graphs; In addition, extra indexes should be added based on the border points between the partitions in order to ensure the search validity in the whole graph. However, the BLINKS method cannot be used to solve our problem. The reasons are as follows. Firstly, BLINKS aimed to used the index to accelerate the search in the graph which is too large to fit in the memory, so it indexed much detailed information which makes the index quite large, but in our problem the graph can be processed within the memory so there is no need to build index to help the search within each machine. Secondly, since the BLINKS index has very high cost in the construction, updating and space consumption, it is not practical to build it in the distributed system, so our solution propose the index
optimization problem which is a compelling need of the distributed machines, we also address the problem of how to construct and update the indexes efficiently. Thirdly, the BLINKS did not concern about the communication cost of the search so its performance in terms of the data shipment is not ideal.

Besides the existing work mentioned above, [30] proposed to address the keyword search on data streams. In sum, the graph keyword searching is still facing the three challenges, and the improvement on the graph keyword search technique can also help the keyword search on XML data as well as on relational databases. Especially, seen from the review, there is still lacking a satisfactory efficient searching algorithm for the large scale graph keyword search.

### 2.2 Query Process in the Distributed System

Since the distributed system is getting more and more popular in processing large scale data, there has been much work about the query processing in the distributed system and they can be classified into three aspects: the first one is the structured query processing, second one is about the graph related queries such as shortest path queries and the graph pattern matching queries, and the last one is about the graph partition. We review them in the following three subsections.

#### 2.2.1 Structured Query Process in the Distributed System

In 2000, [20] has discussed the fundamental mechanisms to process structured queries on the structured data which involved several sites in the Internet. [41] proposed to add a merge phase to the MapReduce [9] to implement the join operations in the relational algebra expressions. [17] focused on how to process SPARQL queries on RDF data in a multi-node system; similarly, [46] managed to rapidly process SPARQL queries on large scale RDF data in the Trinity system [2]. [29] addressed the SQL queries processing in
the multi-core architecture which followed the idea of transfer the keyword queries in the relational databases into SQL queries.

### 2.2.2 Other Query Process in the Distributed System

On the other hand, there are other kinds of graph queries have been studied in the distributed system. Neo4j [3] has been focused on solving the human readable queries on large scale distributed graphs. InfiniteGraph [1] focused on efficiently finding the shortest path queries between two nodes in the distributed graphs. [10] presented a novel query language MRQL for large-scale XML data on a MapReduce environment. [44] focused on solving the structural similarity search on XML data in MapReduce framework. [24] proposed to address the pattern matching problem in distributed graphs. [33] proposed to compute subgraph matching on large scale graphs in the distributed system Trinity [2]. [47] proposed a distributed SLCA-based XML keyword search solution via MapReduce.

### 2.2.3 Graph Partition

When talking about the distributed graph, it is also important to discuss the graph partition methods, this is because the performance of the distributed system relates to the communications between the machines, and the partitioning of the graph has very great influence on this issue. But unfortunately there is no existing solution can provide a satisfactory method to optimally partition the graph so that the distributed system can achieve the best performance, since the graph partition problem is NP-Hard, and also the performance is relevant to the very kind of graph query it concerns. It is know that it is hard to partition the graph which can fulfill the goal of efficient processing of all kinds of graph queries.

A recent work, [31] analyzed nine kinds of important graph queries’ behaviors when they are processed in the distributed system, such as PageRank, BFS, Minimal Spanning
Tree, etc. Based on this analysis, they tried to predict the window of the active nodes, which can be simply described as the set of nodes to be involved in the next steps of processing w.r.t the current node. Then, the work proposed to dynamically shift the graph fragments from the current machine to its neighbors based on the windows, so less inter-machine communication is introduced. This is quite an interesting idea and also seen from their experiment results, it works well with some kinds of queries but its performance for the BFS queries is not ideal, which makes sense because the window of this kind of queries is also very large compared to the other kinds of queries. And, unlike other kinds of queries, BFS query needs to scan the whole graph. From this comparison we can know that, this work cannot solve our problem well because 1) The keyword query processing is similar to the BFS queries to some extent and this method is not effective enough from the experiment results; 2) The keyword query searching procedure is much more complicated compared to the BFS queries, and it added quite much difficulties to predict its active windows which is a crucial factor to the performance of this method.

To sum up this chapter, though there has been much work about the graph keyword search and about the distributed query processing, none of the existing solution can solve the keyword search problem in the distributed graphs in a satisfactory manner (In Chapter 7 we show the limitations of the previous work). Therefore, in this thesis, we propose to address the keyword search in large scale graphs via the distributed system.
CHAPTER 3

PROBLEM DEFINITION

In this chapter, we introduce the basic concepts used in this thesis and then give the formal definition of the problem.

Definition 3.1 (Graphs in Distributed System) For a graph $G(V, E, L)$ where $G.V$ denotes the vertices in $G$, $G.E$ denotes the edges and $G.L$ denotes the labels, there can be a partition on $G\{G_1, G_2, \ldots, G_k\}$ where $\bigcup_{j=1}^{k} G_j.V = G.V$ and for $\forall i \neq j \in [1, k]$, $G_i.V \cap G_j.V = \emptyset$. Based on this partition, graph $G$ can be deployed into the machines of a distributed system $S$ where each partition $G_i$ is in machine $M_i \in S$.

Please note that our work can be applied to both directed and undirected graphs, but for the sake of brevity we use directed graph in this thesis, and it is trivial to extend our work to the undirected graph. In Definition 3.1, any partition method can be used to compute the graph partition. As shown in Figure 1.1, the graphs in different dashed ellipses are different partitions: $a_1$ and $a_2$ are partitioned together and then stored on the machine $M_1$, and the entire graph is deployed onto three machines $M_1$, $M_2$ and $M_3$.

Definition 3.2 (Border Point) For the graph $G_i$, if a vertex $b \in G_i.V$ has incoming edges from other machines $M_j \neq M_i$, $b$ is a border point. The set of all the border points in $M_i$ is denoted as $M_i.B$. $b.nM$ denotes the set of machines containing the nodes linking to $b$ except $M_i$.

Similar definition has been proposed in the work [12] w.r.t the partitioned graphs.
Take the Figure 1.1 for example, the border points in machine $M_3$ are $p_1$ and $c_1$, $M_2.B = \{p_2\}$, and there are no border points in $M_1$. Also, $p_1.nM = p_2.nM = c_1.nM = \{M_1\}$.

**Definition 3.3 (Query Answer)** Given a graph $G$ and a keyword query $Q = (k_1, k_2, \ldots, k_m)$, an answer to $Q$ is a pair $(r, (x_1, x_2, \ldots, x_m))$ where $r, x_i \in G.V$ and

1. Every $x_i$ must contain at least one $k_i$ and $(x_1, x_2, \ldots, x_m)$ must cover all the keywords in $Q$;
2. Every $x_i$ is connected to $r$;
3. Along each path from $r$ to $x_i$, there is no such node $r'$ that also connects to the nodes $(x_1, x_2, \ldots, x_m)$.

In the keyword search literature, the definition of query answer is not unique, we choose to use the above definition since it is widely used in the previous work about graph keyword search [35] [12] [32] [42] [19].

Please note that, it is possible that there is no query answer in the given graph, so usually during the search procedure we need to limit the size of the query answer, such as put a upper bound to the distance between the answer root and the keyword matchings. With this method, the search does not need to scan the entire graph for the queries which has no answer at all.

**Definition 3.4 (Problem Definition)** The problem of effective keyword search on the graphs in the distributed system is: given a keyword query $Q$ and a graph $G$, compute the query answers using the distributed system $S$.

Please note that there are several ways to present the keyword query answers, but it is not within the scope of this work. Since we aim to compute all the keyword query answers in the graph, any query answer presentation method can be plugged in.
CHAPTER 4

NAIVE SEARCH SOLUTION

In this chapter we introduce a naive algorithm to answer the keyword queries in the distributed system: we first introduce the general framework of this algorithm, then elaborate the algorithm in detail, and finally we analyze its shortcomings.

4.1 General Framework

Generally speaking, in order to explore the graph $G$ deployed in the distributed system $S$, we follow the Backward Search strategy which is widely used in many previous graph keyword search approaches [6] [35] [12] [32] [42] [19]. We have two basic assumptions as follows:

- No specific graph partition is assumed, which means the query answers have bad locality in the system $S$.

- No difference between the machines in $S$ is assumed, which means all machines in $S$ play the same role.

Seen from a brief scenario of the query processing shown in Example 1.2, besides the search within each machine, it also requires to search between the machines. Therefore, we divide the search procedure into two parts: local search and inter-machine search, which are as follows.

**Local Search.** Local search is performed within each machine $M_i \in S$, and it follows the BWS to find the query answers. During the exploration some query answers are
produced, but if the exploration encounters the border point in $M_i$, we need to continue the search to its neighbor machines to determine whether there is a query answer covering several machines. So when the local search encounters the border point, it enters the inter-machine search part.

**Inter-machine Search.** This part is in fact to do the communication between the machines in $S$. When one machine has explored to a border point, it would wrap up its exploration involving this border point (which is defined as partial answer in the next section) as a message and send it backwardly to its neighbor machines. Then, when one of the neighbors receives this message, it enters local search part to continue the local exploration.

Please note that these two parts are not sequential but interweaved, and the query processing terminated when there is no message exchanged in $S$. In the next section we discuss in detail about these two parts.

### 4.2 Naive Algorithm

Now we dive into the details of the naive algorithm which is shown in Algorithm 1.

**Local Search.** Since local search is basically the backward search, here we briefly introduce the BWS strategy which is non-distributed: Firstly, for each keyword $k_i \in Q$ we compute all the nodes in $M_i$ containing the keyword $k_i$, which is called $k_i$’s cluster and denoted as $c_i$, and the nodes in $c_i$ are referred as keyword origins. The backward search starts from the nodes in the clusters. Secondly, for each step of backward expanding, a node in $c_i$ is selected to be expanded to all its neighbors, and they are labeled as reachable to $k_i$ and added to $c_i$. Finally, when the node $r$ is in all the clusters, namely, a node $r$ is reachable to all the keywords in the $Q$, then $r$ is an answer root and hence a query answer of $Q$ is found (Line 3).
As for the expansion, we need to 1) choose a proper cluster \( c_i \) to continue; 2) choose a proper node in \( c_i \) to expand, in order to reach the answer root node earlier. We follow the optimal expanding strategies in [12]: We always choose the cluster with the smallest size to expand next, which is called the Cost Balanced expansion strategy; And we always choose the nodes with the shortest distance to the keyword origins to expand next, which is called the Equi-distance strategy.

**Algorithm 1: Naive Search Algorithm naive**

**Input**: a keyword query \( Q \)

**Output**: the query answers of \( Q \)

```plaintext
At Machine \( M_i \in S \) while the backward expanding continues do
  if node \( n \in M_i.G.V \) reaches all keywords in \( Q \) then
    \( M_i.qa.add(n.qa) \);
  if node \( n \in M_i.B \) then
    \( M_i.pa.add(n, kset) \);
    Send(WrapMeg(pa, pa.b.nM));
  if \( pa=ReceivedMeg() \) then
    AddtoCluster(pa.kset);
```

During the local search, some query answers can be determined without going out of \( M_i \), and when the expansion reaches a border point which only connects to part of the keywords (Line 5), the expansion needs to be continued in other machines. Therefore we define the partial answer in such case as follows.

**Definition 4.1 (Partial Answer)** In machine \( M_i \in S \), a partial answer \( pa_{b,M_i} \) is consisted of a border point \( b \) which connects to the keywords \( b.kset \), where \( b.kset \subseteq Q \).

Once the local search in \( M_i \) produces any partial answers, it is sent from \( M_i \) to the related neighbors to continue the search (Line 6) to check whether this partial answer is a part of a query answer, which enters the inter-machine search part.

**Inter-machine Search.** For a partial answer \( pa_{b,M_i} \), we send \( pa_{b,M_i} \) from \( M_i \) to all the machines in \( b.nM \) (Line 6). After a machine \( M_j \in b.nM \) receives the message, \( M_j \)
starts the local search parts. Specially, at this point the local backward search starts from
the nodes which connect to $b$, and these nodes are already reachable to the keywords in
$b.kset$ (Line 8).

When all the machines in the distributed system $S$ produce no more partial answers,
both the local search and the inter-machine search terminate, and all the query answers
of $Q$ are computed. The following example shows how the naive algorithm works.

**Example 4.1 (Naive Algorithm Processing)** Search the query “John, Sigmod” in
Figure 1.1, in $M_3$ the search starts from $c_1$ which contains “Sigmod” and it expands
to the border point $p_1$, and a partial answer $pa_{p_1,M_3}$ is produced during the local search
in $M_3$. For $p_1.nM = \{M_1\}$, $M_3$ sends $pa_{p_1,M_3}$ to $M_1$, which is the inter-machine search.
Then, in $M_1$ the local search starts from $a_1$, which contains “Sigmod” from $pa_{p_1,M_3}$ and
also contains “John” in $M_1$, so the answer $(a_1, writeBy, p_1, publishedBy, c_1)$ is yield in
$M_1$.

The computation complexity of the naive algorithm is $O(m \times |G.V| + m \times |G.B|)$
where $S.B$ is the set of border points in the system $S$. Since the backward search has no
effective guidance, so in the worst case the entire graph is scanned for $m$ times, hence the
exploration complexity is $O(m \times |G.V|)$, and the communication cost of sending partial
answers among machines is $O(m \times |G.B|)$. It is obvious that naive algorithm is prohibitive,
in the next section we give a detailed analysis about why the cost is so high and how to
enhance the performance.

### 4.3 Analysis

In the naive algorithm we would encounter the following two cases, through which we can
clearly see the computation waste and its reason.
• **Case 1.** The answer root is in the different machine from the keyword matchings and the answer root contains no keywords. For example in Figure 4.1(a), $b_1$ and $b_2$ are the keyword matchings and $n_1$ is the answer root. The partial answers $pa_{b_1,M_1}$ and $pa_{b_2,M_2}$ are sent to $M_3$, then the answer $(n_1, \langle b_1, b_2 \rangle)$ can be discovered. This takes one round of message exchange because $n_1$ is in the 1-hop neighborhood of $b_1$ and $b_2$. If the answer root is $n_1$ which is $k$ machines away, then there would be $k$ rounds of message exchange.

• **Case 2.** The keyword matchings are in different machines, and the answer root is a keyword matching. For example in Figure 4.1(b), $b_1$, $b_2$, $n_1$ are the keyword matchings and $n_2$ is the root, and $n_1$ can be several machines away. So, first $pa_{b_1,M_1}$ and $pa_{b_1,M_1}$ are sent to $M_3$ and reach $n_2$, and then the partial answer derived from $n_2$, $pa_{b_1,M_1}$ and $pa_{b_1,M_1}$ in $M_3$ is sent to $M_4$, and finally the answer is found in $M_4$.

From these two cases, we can clearly discover the waste of *makespan* and *data shipment* in the naive solution. Please note that besides these two measures, there are other measures to evaluate the performance of distributes algorithms, like *visit time*. In our work, we use these two measures since we focus on the time efficiency and system’s traffic load, and they are also used in the previous work such as [24] [8].
Waste of Makespan. The waste of makespan would undermine the system parallelism. As shown in the two cases, if the root is $k$ machines away from the keyword matchings, then it takes $k$ rounds to complete the inter-machine search, which is in fact sequential. For example in the Case 2, above when $k = 1$, other than the naive method, it is better to make $M_1$ send $pa_{b_1,M_1}$, $M_2$ send $pa_{b_1,M_1}$ and $M_4$ send $pa_{n_1,M_4}$ to $M_3$, which can get the query answer in only one round of message exchange.

Waste of Data shipment. Since the messages are sent backwardly, in many cases the system would produce huge data shipment. For example, when the answer is $(n_1, (n_1, b_2))$ in Figure 4.1(a), in the naive solution, $pa_{b_1,M_1}$ and $pa_{b_2,M_2}$ are sent to $M_3$. However, if the total size of the partial answers of $n_1$ and $b_2$ is smaller than that of $pa_{b_1,M_1}$ and $pa_{b_2,M_2}$, it is better to send partial answers from $M_2$ and $M_3$ to $M_1$. Also in Figure 4.1(a), when the answer is $(b_1, (n_2, b_1, b_2))$, according to the naive solution, $M_2$ sends $pa_{b_2,M_2}$ to $M_3$ and $M_3$ sends out $pa_{n_2,M_3}$ to $M_1$, then at $M_3$ the a new partial answer can be formed given $n_2$ and $b_2$ and this new partial answer is sent from $M_3$ to $M_1$ where the query answer can be formed, so the message containing $pa_{n_2,M_3}$ sent to $M_1$ is unnecessary. In this case, it is better to send $pa_{b_1,M_1}$ to $M_3$, which eliminates unnecessary traffic.

Seen from the above analysis, we can discover that the naive solution wastes considerable amount of makespan and data shipment because the inter-machine search is not effectively guided. It would be more efficient if the partial answers are exchanged smartly both backwardly and forwardly, which is the first difficulty discussed in Chapter 1. In the next two chapters we introduce an advanced solution which can overcome the shortcomings of naive algorithm.
CHAPTER 5

INDEX CONSTRUCTION

We propose the advanced solution including the following two parts: firstly, we build the indexes in order to provide effective information to guide the inter-machine search, which is introduced in this chapter; secondly, we propose an advanced searching algorithm aiming to obtain the minimal makespan and data shipment of the system, which is introduced in Chapter 6.

Observed from the problems in the naive solution, its incapacity originates from the fact that the machines only have their local knowledge and know nothing about their neighbors. Therefore, it is called for that we have a coordinator machine \( M_o \in S \) which holds a global picture of the entire system, so it can make optimal decisions about the message forwarding between the machines, which updates the Assumption 2 in Chapter 4. Naturally, since the coordinator machine and other machines play different roles in the system, we need to design different indexing schemes for them i.e. the local index and the global index. As explained in Chapter 1, there is limited space in the machines for the index materialization, therefore, we proposal the index optimization problem to ensure that the indexes can provide the most information within storage constraints. Both local and global indexes are tested correct in Chapter 7. Specifically, the first two sections in this chapter introduce the local index and the global index, resp., and then we discuss how to optimize, construct and maintain the indexes.
5.1 Local Index

The local index is built on the non-coordinator machine $M_i \in S$, and it is designed to provide guidance for local search in such case: when all the keyword matchings are in $M_i$ but the answer root is on another machine. We want the local index to provide information so that when such case happens, the search does not need to go beyond $M_i$, and hence makespan and data shipment are saved. To achieve this goal, we build the local index as follows:

On machine $M_i \neq M_o$, for any pair $(b_i, b_j)$ where $b_i, b_j \in M_i.B$ and $b_i \neq b_j$, we add the following two entries to the index:

1. the shortest paths connecting $b_i$ to $b_j$ in the graph $G$;
2. the set of nodes that connect to both $b_i$ and $b_j$ in graph $G$, and along any such paths to both $b_i$ and $b_j$, there is no more node connecting to $b_i$ and $b_j$.

As for the first entry, $b_i.c$ contains $b_i$’s child points in the other machines in $S$ and $b_j.p$ contains $b_j$’s parent points in the other machines in $S$. To compute the shortest path that connects from $b_i$ to $b_j$, we check for any combination of $n_1 \in b_i.c$ and $n_2 \in b_j.p$ to see if there is a shortest path connecting $n_1$ to $n_2$ in graph $G$ by running the shortest path algorithm such as Dijkstra. So, for the pair $(b_i, b_j)$, there can be no path to be indexed, or one path, or many shortest paths of the same length. The computation complexity of this entry is $O(|b_i.c| \times |b_j.p| \times |G.E| \times \log|G.V|)$.

As for the second entry, similar to the first one, we need to find the common ancestors of $n_1$ and $n_2$ in $G$. So, for the pair $(b_i, b_j)$, there can be many common ancestors which are indexed. The computation complexity of this entry is $O(|b_i.c| \times |b_j.p| \times |G.V|)$.

In addition, the local index has the storage cost as $O((\frac{x}{2})^2)$ where $x = |M_i.B|$, so the index is light-weighted since it only concerns the border points, which are of a much
Some of $M_i$’s Border Points

local index structure

Figure 5.1: Local Index Construction

smaller size compared to entire $G_i$. The following example illustrates the construction of local index.

Example 5.1 (Local Index) As shown in Figure 5.1(a), to build the local index on $M_1$ we find two border points on $M_1$: $b_1$ and $b_2$, where $b_1.c = \{n_3\}$, $b_1.p = \{n_4\}$ and $b_2.p = \{n_1, n_2\}$. For the pair $(n_2, n_3)$ there is a shortest path so we can index the path $p$ in its first entry; also for $(n_2, n_4)$ we find the node $r$ connecting to both $n_2$ and $n_4$, so we can index the path from $r$ to $n_2$ and $n_4$ in its second entry. Meanwhile, though the node $r'$ can also connect to both nodes, but since $r$ is already indexed, $r'$ is ruled out. Figure 5.1(b) shows the local index structure for the pair $(b_1, b_2)$.

Comparison. The following example shows how the local index can provide useful guidance so that both makespan and data shipment are saved. For example in Figure 5.1(a), in $M_1$ we have the partial answers $pa_{b_1, M_1}$ and $pa_{b_2, M_1}$, by checking the local index, we can now that $r$ can be their answer root without going out of $M_1$ and issuing any message. Also, for the answer containing $b_1$, $b_2$ and $p$, we can get it without inter-machine search. Without the index’s help, the answer is computed with high cost: first send $pa_{b_2, M_1}$ to $p$’s machine, and then send the partial answer back to $M_1$.

To sum up, with the help of the local index, when all the keyword matchings are in the same machine $M_i$ and the answer root is in other machine, there is no need of
inter-machine search which can reduce much time and traffic.

5.2 Global Index

Different from the local index, we want to build a global index on the coordinator machine $M_o$, which can provide useful guidance for the inter-machine search to enhance the system performance when local index can offer no help.

For the coordinator machine $M_o$, we build the global index as follows. For the any pair border points $(b_i, b_j)$ where $b_i \in M_i.B$ and $b_j \in M_j.B$ and $M_i \neq M_j \in S$, we index the following two entries:

1. If there exists a shortest path connecting from $b_i$ to $b_j$, we index the first machine and the last machine, $M_f$ and $M_l$, on this shortest path.

2. If there exists a node $r$ connecting to both $b_i$ and $b_j$, we index the two last machines, $M^l_i$ and $M^l_j$, on the two paths; and long the paths to both $b_i$ and $b_j$, there is no more node connecting to $b_i$ and $b_j$.

The idea of the entries in global index is similar to those of the local index, so the computation of global index is alike. Therefore, the computation complexity of the first entry is $O(|b_i.c| \times |b_j.p| \times |E| \times \log|V|)$ where $E$ (V, resp.) denotes the edges (nodes, resp.) in the graph $G$. As for the second entry, the computation complexity is $O(|b_i.c| \times |b_j.p| \times |V|)$. Also, the storage cost of the global index is $O((y^2))$ where $y = \sum_{M_i\in S} |M_i.B|$. The following example illustrates the global index.

**Example 5.2 (Global Index)** As shown in the Figure 5.2(a), for the border point $b_1$ in $M_1$ and $b_2$ in $M_2$, we find that in $M_3$ and $M_4$ there is a shortest path $p$ connecting from $b_1$ to $b_2$, so we can index the first and last machines i.e. $M_3, M_4$ as the first entry for the pair $(b_1, b_2)$; and node $r$ connects to both $b_1$ and $b_2$, so we can index $M_3, M_4$ for the second entry for the pair $(b_1, b_2)$. Figure 5.2(b) shows the index structure of the pair $(b_1, b_2)$.
Comparison. With the help of the global index, we can avoid forwarding the inter-machine messages in an aimless manner. Take the Figure 4.1(b) for example, according to the naive solution, the partial answers $pa_{b_1,M_1}$ and $pa_{b_2,M_2}$ are sent to all the neighbors until they both reach $M_3$ and connect together via $n_2$, then the partial answer $pa_{n_2,M_3}$ is sent to all its neighbors until it reaches $M_4$. Given the information in the global index, when $M_o$ has the partial answers from $M_1$, $M_2$ and $M_4$, by checking the global index we can know that $M_3$ contains the node which connects the partial answers from $M_1$, $M_2$ and $M_4$, then $M_o$ can schedule the partial answers to be sent forwardly from $M_4$ to $M_3$, and backwardly from $M_1$ and $M_2$ to $M_3$.

5.3 Index Optimization

However, usually there is limited space to fully materialize the index, we therefore propose the INDEXOPTIMIZATION problem for both the local and global indexes as follows.

Definition 5.1 The INDEXOPTIMIZATION problem is to build the index which can provide the most information for the search within the space budget of the machine.

In order to solve the INDEXOPTIMIZATION problem, we formalize it to the following cost model.
\[
\max_{b_i \in M_i, B; b_j \in M_j, B} \sum \text{gain}(a_{i,j})a_{i,j} + \text{gain}(b_{i,j})b_{i,j}
\]

\[
\begin{cases}
\sum_{b_i \in M_i, B; b_j \in M_j, B} \text{cost}(a_{i,j})a_{i,j} + \text{cost}(b_{i,j})b_{i,j} \leq bg \\
\quad a_{i,j}, b_{i,j} \in \{0, 1\}
\end{cases}
\]

(5.1)

(5.2)

(5.3)

In this cost model, as restrained in Formula 5.3, \(a_{i,j}\) (\(b_{i,j}\), resp.) denotes the first (second, resp.) entry for the pair \((b_i, b_j)\) in both local and global indexes, and \(a_{i,j} = 0\) (\(b_{i,j} = 0\), resp.) means the first (second, resp.) entry is not materialized, otherwise it is materialized. Formula 5.2 ensures that the space cost of the entries would not exceed the budget \(bg\) of the machine, and the \(\text{cost}(a_{i,j})\) (\(\text{cost}(b_{i,j}),\) resp) is the space cost of the entry presented by \(a_{i,j}\) (\(b_{i,j}\), resp.). In the Formula 5.1, \(\text{gain}(a_{i,j})\) (\(\text{gain}(b_{i,j}),\) resp.) denotes the performance enhanced by adding the entry presented by \(a_{i,j}\) (\(b_{i,j}\), resp.). Given this cost model above, we can build the optimal index within the space budget \(bg\) for each machine by materializing the selected entries.

Please note that, although the indexing requires the a coordinator machine \(M_o\) in the system, \(M_o\) is not specially chosen and can be dynamically adjusted. We can separate one of the machines from the system as the coordinator, so it has no data but an optimized global index on it. Also, if there is no such empty machine with no data, we can also use one machine with data as the coordinator; In such case this machine plays two roles: it has the data, the optimized local index and the global index, and our indexing scheme is adaptive with it because we only need to properly adjust the space budget parameters for the index implementation without any more modifications.
5.4 Index Implementation and Maintenance

In the last section we have discussed the models of building optimal local and global indexes, but one may notice that we did not reveal the details of how to measure the gain and cost functions. It is quite important to choose the proper measurement for gain and cost since it not only determines the complexity of index implementation, but also influences the effectiveness of the models. In this section, we discuss two manners to measure the gain and cost: a heavy measurement and a swift measurement, where the heavy measurement is more accurate than the swift measurement and heavy measurement introduces a more complex and costly procedure to implement the indexes than the swift measurement.

The Heavy Measurement. In the local index of $M_i$, $a_{i;j}$ and $b_{i;j}$ are computed based on the graph $G$ in the entire distributed system $S$, so the gain can be measured as follows:

1. $\text{gain}(a_{i;j}) = 2k$ w.r.t makespan and $\text{gain}(a_{i;j}) = 3k$ w.r.t data shipment, since adding entry $a_{i;j}$ saves makespan of $2k$ and data shipment as $3k$, if $k$ is set to be the number of machines involved in the path related to the entry $a_{i;j}$ excluding $M_i$ itself.

2. $\text{gain}(b_{i;j}) = k$ w.r.t makespan and $\text{gain}(b_{i;j}) = 2k$ w.r.t data shipment, since adding entry $b_{i;j}$ saves makespan of $k$ and data shipment as $2k$, if $k$ is set to be the number of involved in the path in the entry $b_{i;j}$ excluding $M_i$ itself.

Correspondingly, in the global index of $M_o$, $\text{gain}(a_{i;j})$ and $\text{gain}(b_{i;j})$ can be measured as follows:

1. $\text{gain}(a_{i;j}) = k$ w.r.t the makespan and $\text{gain}(a_{i;j}) = |M_j.n.M_k| - k$ w.r.t the data shipment, since adding entry $a_{i;j}$ saves the makespan of $k$ and data shipment as
\[ \text{gain}(a_{i,j}) = |M_j.nM_k| - k, \] if \( k \) is set to be the number of machines involved in the path related to the entry \( a_{i,j} \) excluding \( M_i \) and \( M_j \).

2. \( \text{gain}(b_{i,j}) = 0 \) w.r.t the makespan and \( \text{gain}(b_{i,j}) = |M_i.nM_{k_1}| + |M_j.nM_{k_2}| - k \) w.r.t the data shipment, since adding entry \( b_{i,j} \) saves no makespan and saves the data shipment as \( |M_i.nM_{k_1}| + |M_j.nM_{k_2}| - k \), if \( k \) is set to be the number of machines involved in the path related to the entry \( b_{i,j} \) excluding \( M_i \) and \( M_j \), and \( k_1 \) (\( k_2 \), resp.) denotes the number of machines involved in the path from the node \( r \) to \( M_i \) (\( M_j \), resp.) so \( k_1 + k_2 = k \).

Seen from the \textit{gain} measurement from both local and global indexes, it is clear that we need to scan the entire graph \( G \) for each local index entry and simultaneously we can calculate the \textit{cost} for each entry. This measurement introduces great cost because even for those index entries which are not to be materialized in the optimization problem, we need to do the graph scan for it.

\textbf{The Swift Measurement}. In order to reduce the cost of heavy measurement, we want to avoid the graph scan for every index entry in when measuring the \textit{gain} and \textit{cost}. Therefore, instead of the entire graph \( G \), we restrain the graph scan within a radius \( K_0 \). Consequently, in the local index, the \textit{gain} is measured as follows and all the paths mentioned below are within the \( K_0 \)-hop neighborhood of \( M_i \) so \( k < K_0 \). Also the \textit{cost} is simultaneously calculated.

1. \( \text{gain}(a_{i,j}) = 2k \) w.r.t makespan and \( \text{gain}(a_{i,j}) = 3k \) w.r.t data shipment.

2. \( \text{gain}(b_{i,j}) = k \) w.r.t makespan and \( \text{gain}(b_{i,j}) = 2k \) w.r.t data shipment.

Similarly, in the global index the \textit{gain} is measured as follows and all the paths mentioned below is within the \( K_0 \)-hop neighborhood of \( M_i \) and \( M_j \) i.e. \( M_i.nK_{K_0} \cup M_j.nK_{K_0} \) so \( k < K_0 \):
1. \( \text{gain}(a_{i,j}) = k \) w.r.t the makespan and \( \text{gain}(a_{i,j}) = |M_j.nM_k| - k \) w.r.t the data shipment.

2. \( \text{gain}(b_{i,j}) = 0 \) w.r.t the makespan and \( \text{gain}(b_{i,j}) = |M_i.nM_{k_1}| + |M_j.nM_{k_2}| - k \) w.r.t the data shipment where \( k_1 + k_2 = k \).

It is reasonable to use the result from scanning the partial graph as the measurements of \( \text{gain} \) and \( \text{cost} \) because, although we do not scan the entire graph but most of the entries are correctly measured from the partial scan when \( K_0 \) is properly set.

After choosing the entries to materialize based on the swift measurement of \( \text{gain} \) and \( \text{cost} \), we run the complete graph scan to get the correct contents for the index entries. Please note that, for the local indexes we can trace along the paths to see if there exists any path shorter in the non-scanned part of graph, which ensures the correctness of the local index. However, for the global index, we can skip this step without undermining the effectiveness of the global index. Since the \( K_0 \) neighborhood may not cover \( G \), some of the index entries computed can be globally invalid, as shown in the Figure 5.3. In this case, the shortest path containing \( p \) from \( b_1 \) to \( b_2 \) within the range of \( M_{K_0} \) can be globally invalid, due to the shortest path containing \( p' \) within the range of \( M_{K_0+1} \). Fortunately, such invalid cases does not undermine the correctness of the search guidance provided by global index. This is because, if there exists a valid shortest path in a larger range, it must share the begin and end parts of the invalid one, like the shortest path containing \( p' \).
shares the nodes $n_2$ and $n_3$ with the one containing $p$ in Figure 5.3. Also similar argument can be made for the second entry of global index. In light of this, since we only index the endpoint machines, the guidance of the invalid entries still work correctly.

**Comparison.** We compare the *heavy measurement* and *swift measurement* in the following aspects:

- In terms of the index implementation, the swift measurement is not as correct as the heavy one, but it is much faster than the heavy one.

- In terms of the index maintenance, when a new node is added or deleted from $M_i$, the heavy measurement is inferior to the swift one: the heavy measurement requires a rather prohibitive updating procedure of the local and global indexes which is in fact to do the indexing all from scratch; However, the swift one needs a much smaller cost since it only needs to refresh the local indexes for the machines in the $M_i.nM_{K_0}$ and the entries related to the nodes in $M_i.nM_{K_0}$ for the global index.

Seen from the comparison, the swift measurement is much better than the heavy measurement in the implementation efficiency and index maintenance, and also we can demonstrate that the correctness of the swift measurement is very close to the heavy one by the experiments in Chapter 7. In the next section, we will show the hardness of the INDEXOPTIMIZATION problem, then propose an algorithm to solve it.

## 5.5 Hardness of the Index Optimization

In this section, we first prove the hardness of the INDEXOPTIMIZATION problem to be NP-Hard, and then we propose a greedy algorithm to select the entries which achieves a factor-2 approximation to the optimal solution.

**Theorem 5.1** The INDEXOPTIMIZATION problem is a NP-Hard problem.
Algorithm 2: Greedy Algorithm for INDEXOPTIMIZATION

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initially, $E_m = \emptyset$;</td>
</tr>
<tr>
<td>2</td>
<td>while $\text{space}(E_m) \leq bg$ do</td>
</tr>
<tr>
<td>3</td>
<td>$e_0 = \arg\max_{e_0 \in E_m} \left( \text{val}(e_0) = \frac{\text{gain}(E_m+e_0)-\text{gain}(E_m)}{\text{cost}(e_0)} \right)$;</td>
</tr>
<tr>
<td>4</td>
<td>$E_m = E_m \cup e_0$;</td>
</tr>
</tbody>
</table>

Proof 5.1 (Sketch) We reduce Knapsack to INDEXOPTIMIZATION. An instance of Knapsack is: given $n$ items with weights $\{w_i\}$ and values $\{v_i\}$, we want to select a subset of items with total weight $\leq B_0$, so that the total value is maximized. We construct an instance of INDEXOPTIMIZATION: There are $n$ entries in the local index $E = \{e_1, e_2, \ldots, e_n\}$; For each entry $e_i$, $w_i = \text{cost}_i$ and $v_i = \text{gain}_i$. We want to select a subset of $E$ with total cost $\leq bg_i$ so that the total gain is maximized. It is straightforward that, with the same values for $B_0$ and $bg_i$ in these two problems, the optimal solution to Knapsack is OPT if and only if the optimal solution to INDEXOPTIMIZATION is OPT.

In light of the hardness of index optimization problem, we use the greedy approximation algorithms to solve it, namely, we always pick up the entry which has the largest gain per cost unit (Line 3). The Algorithm 2 provides factor-2 approximation to the optimal solution and can be accomplished polynomially. The experiments in Chapter 7 verify the effectiveness of Algorithm 2.

So far we have constructed the optimal local and global index in order to provide useful information as much as possible. In the next chapter, we introduce how the distributed system answers the queries using the constructed indexes in a nice way.
CHAPTER 6

ADVANCED SEARCH SOLUTION

Based on the indexes, we propose the advanced search solution in this chapter: we first introduce the general framework and then elaborate on the algorithm details. Also, the experiments in Chapter 7 show the better performance of the advanced solution over the naive one.

6.1 General Framework

Different from the naive solution, advanced solution requires a coordinator machine $M_o$ in the distributed system $S$, which can provide smart guidance for the inter-machine search. Specifically, we show the intuitive idea of the advanced search procedure w.r.t. the different roles of machines. Also please note that the processions below are interweaved during the query processing.

**Machine** $M_i \neq M_o$. The $M_i$ does the following work: 1) After receiving query $Q$ and partial answers from other machines, $M_i$ does the local search with the help of its local index: when partial answers are produced, $M_i$ check the entries in its local index to see if any query answer can be assembled from the partial answers. 2) For the left-out partial answers, $M_i$ wraps up messages about them and sends to $M_o$. 3) After receiving orders from $M_o$, $M_i$ sends related partial answers to the required machine if it is a forwarding task, or $M_i$ assembles the received partial answers if it is a assembly task.

**Coordinator Machine** $M_o$. $M_o$ does the following work: 1) After receiving the messages from other machines, by checking global entries, $M_o$ discovers those partial answers which can be directly assembled together as query answers, and $M_o$ would arrange
such partial answers to be assembled on a proper machine $M_i$, which is called the Assembly Task $Task_{asm}$. 2) For the left-out partial answers, $M_o$ decides how to forward them based on the information in the global index in order to find the query answers as fast as possible. This is the Forwarding Task $Task_{fwd}$. 3) To achieve the best overall performance of $S$, $M_o$ should schedule all the tasks optimally.

### 6.2 Advanced Algorithm

**Details.** The framework details are as in Algorithm 3.

**Machine** $M_i \neq M_o$. Similar to the local search in the naive search, $M_i$ does graph exploration, and the exploration can either be initiated from keyword origins or partial answers sent from other machines. During the exploration, query answers are put into the final result (Line 5); the partial answers (Line 7) are examined for the minimal partial answer set (Line 10), which is defined as follows.

**Definition 6.1 (Minimal Partial Answer Set)** A set of partial answers forms a minimal partial answer set if and only if 1) the partial answers cover all the keywords in $Q$; 2) when removing any partial answer from the set, the rest of them cannot cover all the keywords in $Q$.

We compute the minimal partial answer set because, if any query answer can be formed from the partial answers in $M_i$ by checking its local index, these partial answers must forms a minimal partial answer set. For example, if two partial answers $pa_{b_1,M_i}$ and $pa_{b_2,M_i}$ form a minimal partial answer set, and $M_i$’s local index has materialized the entries for $(b_1,b_2)$ and $(b_2,b_1)$, we can directly assemble the query answers and put them into the final result without further graph exploration. (Line 12).

As for the left-out partial answers, $M_i$ wraps up messages for each of them and send the messages to $M_o$, and each message contains: 1) the border point $b$; 2) the reachable
keywords $kset$; 3) the size of the partial answer; 4) the machine $M_i$. (Line 14).

We will continue discussing the $M_i$’s work after receiving $M_o$’s orders after introducing $M_o$’s work.

**Coordinator Machine** $M_o$. After $M_o$ has collected all the messages from the other machines, $M_o$ first computes the minimal partial answer sets to see whether any of them can be directly assembled according to the materialized entries in the global index. For example, in the Figure 5.2(a) the partial answers $pa_{n_1,M_4}$ and $pa_{b_2,M_2}$ can form a query answer. Furthermore, such minimal partial answer sets are added by $M_o$ as assembly tasks (Line 20). Please note that, since the received messages do not contain complete details of the partial answers, $M_o$ needs to order the related machines to send involved partial answers to the target machine for assembly, where the target machine is assigned by $M_o$.

For the left-out minimal partial answer sets, $M_o$ needs to check with the materialized global entries to see if any exploration guidance is provided. For example in the Figure 5.2(a), if $M_o$ has the minimal partial answer set including $pa_{b_2,M_2}$, $pa_{b_1,M_1}$ and $pa_{n_2,M_4}$, by checking the materialized global entries about $(b_1,n_2)$ and $(n_2,b_2)$, $M_o$ discovers that it is promising to send $pa_{b_2,M_2}$ to $M_4$ and $pa_{b_1,M_1}$ to $M_3$ in order to find the query answer fast. But if there is no such helpful materialized entry, the partial answer is ordered by $M_o$ to be sent to all its neighbors. Therefore, after consulting the global index, the $M_o$ makes decisions about how each partial answers should be forwarded, which is added to the system’s forwarding tasks (Line 22).

At this point, there are two kinds of tasks, assembly task $Task_{asm}$ and forwarding task $Task_{fwd}$, for $M_o$ to schedule for the entire system $S$, so that there should not be cases where some machines are heavily loaded while others have extra work force. Namely, $S$ should finish the query processing with minimal time and traffic. Obviously, $Task_{asm}$ can be operated on any machine, so choosing the less busy destination machines can save...
makespan; \( Task_{fwd} \) must operate on the specific destination machine otherwise the graph exploration cannot be performed. Therefore, we formally define this scheduling problem as follows.

**Algorithm 3: Advanced Searching Algorithm advanced**

```
Input: a keyword query \( Q \)
Output: the query answers of \( Q \)

At Machine \( M_i \neq M_o \)

begin
    while the backward expanding continues do
        if node \( n \in M_i.G.V \) reaches all keywords in \( Q \) then
            \( M_i.qa.add(n.qa); \)
        if node \( n \in M_i.B \) then
            \( M_i.pa.add(n,kset); \)
        CheckLocalIndex(\( M_i.pa \));

CheckLocalIndex(\( M_i.pa \)) begin
    for each \( s \in FindMPAS(\( M_i.pa \)) \) do
        if checkInLocalIndex(\( s \)) then
            \( M_i.pa.Add(s.qa); \)
        else
            SendtoCoordinator(WrapMessage(\( s \)));

end

At Machine \( M_o \)

begin
    \( M_o.pa = M_1.msg \cup M_2.msg \cup \ldots M_k.msg; \)
    for each \( s \in FindMPAS(\( M_i.pa \)) \) do
        if checkInGlobalIndex(\( s \)) then
            \( Task_{asm}.Add(s.qa); \)
        else
            \( Task_{fwd}.Add(s.qa); \)
    Schedule(\( Task_{asm}, Task_{fwd} \));
```

**Definition 6.2 (Scheduling)** \( M_o \) arranges \( Task_{asm} \) and \( Task_{fwd} \) for the machines in \( S \) so that all the tasks be accomplished with minimal makespan and data shipment.

**The Hardness of the Scheduling Problem.** According to [36], the minimum makespan problem in the distributed system i.e. schedule all the tasks among the machines so that the tasks can be finished with minimal time, is NP-Complete. Our scheduling
problem aims to achieve not only the minimal makespan but also minimal data shipment. As proved in [24], such problem is not approximatable within \((\alpha, \max(k - 1, 2))\) for any \(\alpha > 1\). (The scheduling problem is approximatable within \((\alpha, \beta)\) if there exists a polynomial algorithm such that given any instance of the problem, the algorithm produces a scheduling solution where the data shipment is bounded by \(\alpha\) times of the optimal data shipment and the makespan is bounded by \(\beta\) times of the optimal makespan.)

Considering the hardness of the scheduling problem, we use the greedy methods to achieve a factor-2 approximation, and we use several heuristics to ensure the minimal data shipment.

According to the greedy approximation in [36] of minimal makespan problem, the system should always choose the least busy machine to assign the next task. Based on this strategy, we additionally propose the following heuristics to minimize the data shipment.

After the \(M_o\) has forwarded the \(Task_{fwd}\) based on the global index, we use the following rules to schedule the \(Task_{asm}\): a query answer \(qa\) with its partial answers in \(M_1, M_2, \ldots, M_k\) denoted as \(qa_1, qa_2, \ldots, qa_k\), and we pick the max size among \(qa_1, qa_2, \ldots, qa_k\) denoted as \(qa_{max}\), and the machine having \(qa_{max}\) is denoted as \(M_{max}\).

1. If the \((qa_1 + qa_2 + \cdots + qa_k - qa_{max}).size + M_{max}.load > (qa_1 + qa_2 + \cdots + qa_k).size + M_i.Load\), \(qa_1, qa_2, \ldots, qa_k\) are sent to \(M_i\), where \(M_i\) is the machine with the least working load currently;

2. Otherwise, \(qa_1, qa_2, \ldots, qa_k\) except \(qa_{max}\) are sent to \(M_{max}\).

Now we continue to discuss the \(M_i\)’s work. If \(M_i\) has been chosen by \(M_o\) as the machine to do one job in \(Task_{asm}\), \(M_i\) starts assembling after all the involved partial answers arrive; If it is a forwarding task, \(M_o\) just sends out the partial answers as \(M_o\) required.
The whole system terminates when no partial answer is sent to $M_o$. Please note that it is unlikely to cause the bottleneck at $M_o$ because, 1) the communication between $M_o$ and other machines uses brief messages instead of complete partial answers; 2) the scheduling can reduce the unnecessary communications as much as possible. The extensive experiments in Chapter 7 verifies the effectiveness of the advanced solution.
CHAPTER 7

EXPERIMENTS

In this chapter we present the experiments about our indexing scheme and the search solutions. The experiments were run on a cluster of 15 machines, all with Intel Xeon E5620 CPUs and 64GB memory.

We use two datasets: 1) DBLP dataset \(^1\), which contains 26M RDF triples about computer science publications that has been commonly used for keyword search evaluation \([12] [19] [32] [35]\); it is duplicated 10 times in the experiments. 2) Yago dataset \(^2\), which contains about 20 million RDF triples of 3.4 GB which is used in \([32]\). It is duplicated 5 times in the experiments. Both datasets are uniformed partitioned into the distributed system.

7.1 Index Construction and Maintenance

We design three groups of experiments to compare the performance of the heavy measurement and the swift measurement in terms of the correctness, construction efficiency and the updating cost.

7.1.1 Correctness

Firstly, we test the two measurements in Section 5.4. We first build the local indexes and the global index in the system using the heavy measurement, then we use the swift measurement to construct the index. In order to test the swift measurement can achieve

\(^1\)http://sw.deri.org/~aharth/2004/07/dblp/
\(^2\)http://www.mpi-inf.mpg.de/yago-naga/yago
close effect as the heavy measurement, we compute the how many entries which are chosen to be materialized with the heavy measurement are also chosen by the swift one with the varied value of $K_0$.

As shown in the Figure 7.1, we can see that in both the datasets, the correctness percentage is very close to 1, which means that the swift measurement can get a almost equally good optimization of the index as the heavy one. Also, as the $K_0$ gets larger, the correctness percentage gets higher, this is reasonable since larger $K_0$ can cover more machines in the system and its measurement gets closer to the heavy measurement; and we can see that when $K_0 = 2$ and 3, the correctness is already good and it does not increase greatly when $K_0$ gets larger, this is reasonable because when $K_0 = 2$ and 3 most of the machines has been covered and there is a few machines left uncovered. This verifies our statement in the Section 5.4 that, most of the entries can get the correct measurement from the partial scan when the radius $K_0$ is properly set. Consider the fact that the DBLP dataset is more dense than YAGO, it is reasonable that when $K_0 > 1$, the correctness percentage on DBLP gets higher than that of YAGO.

7.1.2 Construction Efficiency

Secondly, we test the building time cost of these two measurements. We compute the percentage of the time cost of building indexes with varied $K_0$ using the swift measurement over using the heavy one.
From the result in Figure 7.2(a), we can see that as the $K_0$ gets larger, it takes longer time to build with the swift measurement, this is reasonable since more machines are scanned with larger $K_0$. And, the building time cost using the swift measurement is smaller than the heavy one. Seen from the building time cost and the correctness from last experiment, we can see that when $K_0 = 2$, we can achieve a good optimization with fast speed.

7.1.3 Updating Efficiency

Thirdly, we test the maintenance of the index with the two measurements. We randomly insert or delete 10, 100, 1000 and 10000 nodes to the system, and then compute the percentage of updating using the swift measurement. Based on the experiments about the correctness and the building time cost, we now set $K_0 = 2$.

As shown in the Figure 7.2(b), when the number of nodes being updated grows, the updating time cost also grows, but it still smaller than using the heavy measurement. Specifically, when the updating amount is 10, the time cost is rather low using the swift measurement because the nodes are so few that only a few machines are affected to do the update. When the amount gets larger, the time grows because most of the machines are affected but using the swift measurement is still faster since it only need to scan the partial graph within the radius of $K_0$. When the amount gets over 1000, the time cost
increase slows down because the update amount is large enough so basically all the indexes need to be updated. So using the swift measurement scales well with data updates.

Therefore, from the experiments about the correctness, building time cost and updating time cost, we can demonstrate that using the swift measurement in the index optimization problem performs better in terms of the index effectiveness, construction efficiency and updating efficiency. In the following experiments, we use the optimized local index and global index of the swift measurement with $K_0 = 2$ when there is no additional specification.

7.2 Search Efficiency

We design four groups of experiments to test the search performance in terms of the time cost. 1) We test to see if the distributed methods run faster than the non-distributed one; 2) We test the effectiveness of advanced against the naive as well as the methods in the previous work which is [12] and [31]; 3) Then we test the advanced’s performance with different indexes; 4) At last we test scalability of advanced with varied size of large graphs.

7.2.1 Non-distributed VS. Distributed Search

Firstly, we want to show that our search solutions runs faster than the non-distributed algorithm. As shown in Figure 7.3, we compare three search algorithms: nondis, naive and advanced. We use the keyword search algorithm in [6] as nondis, because it is a classical search framework for keyword query process and almost all its later work about keyword search has followed it [35] [12] [32] [42] [19]. For the nondis algorithm, we put all the data in one machine while the data is distributed for naive and advanced, and we test the same set of queries with the query size ranging from 2 to 10.
As shown in Figure 7.3(a) and Figure 7.3(b), nondis runs slowest and advanced runs fastest in both datasets with all the queries. This is reasonable because, the advanced and naive runs in parallel with each machine loaded with partial data while nondis load all the data in one machine; and it is natural that it gets slower as the query gets longer because query processing grows polynomially w.r.t the query size. Through this test result of naive and advanced performing better than nondis, we can see the advantages of parallelism in the distributed system over non-distributed processing, which validates the motivation of us proposing distributed keyword search.

7.2.2 advanced VS. Other Solutions

Secondly, we verify that advanced processes the queries with the best performance. Five different methods are tested: advanced, naive, advanced⋆, blinks and wind. advanced⋆ is the advanced algorithm without the scheduling strategies, blinks refers to the searching algorithm which has no local index and the global index is built according to BLINKS’s indexing scheme about the border points [12], and wind refers to the searching process which uses the active windows to dynamically adjust the graph partitions [31]. From such design we can clearly test our indexes’ performance compared with the BLINKS work and the Wind work and the overall performance of advanced. Also, different from the above efficiency test, we again duplicate the datasets over 64GB so they cannot be held in one
machine. We run the same set of queries by the five algorithms with varied query size.

As shown in Figure 7.4(a) and Figure 7.4(b), the makespan increases smoothly when the query gets longer, and advanced is the fastest. This is reasonable because 1) the computation complexity of the backward search and the index lookup grows polynomially with the query size; 2) advanced processes the query with the least time cost due to the help of the indexes, which saves much meaningless exploration.

We can see that the advanced⋆ is slower than the advanced because the latter uses the optimal scheduling strategies to arrange the tasks; And from the difference in performance we can see the indexes are more effective in saving the time cost.

Also we can see that blinks is slower than the advanced⋆ because blinks cannot assemble the partial answers as soon as possible compared to the advanced⋆, so all the query answers which cover more than one machine need to be computed via inter-machine communication and hence much slower.

And, wind is slower than blinks. This makes sense as it is difficult to accurately predict the active windows, and much time is used to adjust the graph partitions within the system based on the inaccurate active windows.
7.2.3 \textit{advanced} with Varied $K_0$

We also test algorithm \textit{advanced}'s efficiency with different optimized index w.r.t $K_0$, and the efficiency is the average time cost of processing the keyword queries from the length 2 to 10.

Figure 7.5 shows that when $K_0$ gets larger, \textit{advanced} saves more time cost compared to \textit{naive}, which is coincident with the experiment results from Chapter 5, that when $K_0$ gets larger, the optimized indexes perform better. So, the efficiency gain grows with larger $K_0$.

![Figure 7.5: \textit{advanced}'s Efficiency with Varied $K_0$](image)

7.2.4 \textit{advanced} with Varied Graph Size

At last we test the scalability of \textit{advanced}. We increase the graph size from 64GB to 128GB and the makespan is the average time cost of processing the keyword queries from the length 2 to 10.

From the result Figure 7.6, we can see that as the graph size increases, the processing time cost of \textit{advanced} increases smoothly, which makes sense because there are more query answers in larger graphs so longer time is needed during the search. From the experiment we can conclude that \textit{advanced} has good scalability of large scale graphs.

Therefore, in terms of makespan, the above test results about efficiency as show in Figure 7.3, Figure 7.4, Figure 7.5 and Figure 7.6 demonstrate that \textit{advanced} performs
and scales well.

### 7.3 Data Shipment

We design three sets of experiments to test the performance of the advanced in terms of the data shipment. 1) We test the effectiveness of advanced against the naive as well as the methods in the previous work which are [12] and [31]; 2) Then we test the advanced’s performance with different indexes; 3) At last we test scalability of advanced with varied size of large graphs.

#### 7.3.1 advanced VS. Other Solutions

As shown in Figure 7.7 and Figure 7.8, we can clearly test our indexes’ performance compared with the BLINKS work and the Wind work, our scheduling strategies’ performance and the overall performance of advanced. The makespan increases smoothly as the query gets longer, and the advanced has the least data shipment. This is reasonable since the query length increases the graph exploration and hence the inter-machine search increases, and naive has to do much useless inter-machine message exchange which can be successfully avoided by the indexes in advanced.

We can see that the advanced* introduces more traffic than the advanced because the latter uses the optimal scheduling strategies to arrange the tasks; And from the difference
in performance we can see the indexes are more effective in saving the data shipment.

Also the blinks introduces more data shipment then the advanced\textdagger. This makes sense because blinks index cannot help with the fast assembly of the query answers as the advanced\textdagger does, and all the query answers which cover more than one machine should be exchanged within the system to accomplish the search.

And, wind introduces more amount of message exchange in the system than blinks. It is probably because the predicted active windows in wind is less effective than the correct search guidance provided by the BLINKS index.

7.3.2 advanced with Varied $K_0$

Figure 7.9 shows that when $K_0$ gets larger, advanced saves more data shipment compared to naive, which is also coincident with the experiment results from last section, that when
$K_0$ gets larger, the optimized indexes perform better.

### 7.3.3 *advanced* with Varied Graph Size

At last we test the scalability of *advanced*. From the result Figure 7.10, we can see that as the graph size increases, the communication cost of *advanced* increases smoothly, which makes sense because there are more query answers in larger graphs so more partial answers are exchanged. From the experiment we can conclude that *advanced* has good scalability of large scale graphs.

Therefore, in terms of data shipment, the above experiment results shown in Figure 7.7, Figure 7.8, Figure 7.9 and Figure 7.10 demonstrate that *advanced* performs well and scales well.
CHAPTER 8

CONCLUSION

In this work we address the problem of keyword query search on large scale graph in the distributed system. First, we discuss the naive distributed search algorithm and it results in much makespan and data shipment waste. Then, we propose the advanced search algorithm with two contributions: one is to provide the search guidance using pre-compute indexes which are optimized to save the most make-span and data shipment given the limited space on machines; the other contribution is to schedule the tasks in the system so that all the tasks can be accomplished with minimal time cost. Finally, the experiments verify the effectiveness and efficiency of our work.
REFERENCES


LIST OF PUBLICATIONS