Grid-Aware Evaluation of Regular Path Queries on Large Spatial Networks

by

Zhuo Miao
B.Sc. Honours, Bishop's University, 2005

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in the Department of Computer Science

© Zhuo Miao, 2007

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part by photocopy or other means, without the permission of the author.

Grid-Aware Evaluation of Regular Path Queries on Large Spatial Networks

by

Zhuo Miao

B.Sc. Honours, Bishop's University, 2005

Supervisory Committee

Dr. Alex Thomo, Supervisor (Department of Computer Science)

Dr. Jens H. Weber, Member (Department of Computer Science)

Dr. Venkatesh Srinivasan, Member (Department of Computer Science)

Dr. Lin Cai, Outside Member (Department of Electrical and Computer Engineering)

Supervisory Committee

Dr. Alex Thomo, Supervisor (Department of Computer Science)

Dr. Jens H. Weber, Member (Department of Computer Science)

Dr. Venkatesh Srinivasan, Member (Department of Computer Science)

Dr. Lin Cai, Outside Member (Department of Electrical and Computer Engineering)

Abstract

Regular path queries (RPQs), expressed as regular expressions over the alphabet of database edge-labels, are commonly used for guided navigation of graph databases. RPQs are the basic building block of almost all the query languages for graph databases, providing the user with a nice and simple way to express recursion. While convenient to use, RPQs are notorious for their high computational demand. Except for few theoretical works, there has been little work evaluating RPQs on databases of great practical interest, such as large spatial networks.

In this thesis, we present a grid-aware, fault tolerant distributed algorithm for answering RPQs on spatial networks. We engineer each part of the algorithm to account for the assumed computational-grid setting. We experimentally evaluate our algo-

rithm, and show that for typical user queries, our algorithm satisfies the desiderata for distributed computing in general, and computational-grids in particular.

Table of Contents

Supervisory Committee ii

Abstract iii

Table of Contents v

List of Figures vii

Acknowledgements ix

Dedication xi

1 Introduction 1

1.1 Spatial Networks 1

1.2 Preferential Path Queries 3

1.3 Distributed/Parallel Evaluation 5

1.4 Thesis Overview 7

2 Background and Challenges 9

2.1 Motivation 9

2.2 Related Works 11

2.3 Our Approach 12

3 Basic Definitions 16

4 Grid-Aware Distributed Evaluation of Regular Path Queries 20

4.1 Computation of Query Answers 23

4.2 Machine Loss and Termination Detection 31

4.3 Generating Paths 42

5 Queue Strategies 43

6 Data Partitioning 46

6.1 Background 46

6.2 R-Tree Partitioning 47

7 Experiments 50

8 Conclusions 54

List of Figures

1.1 A partial map of Ontario 2
1.2 A partial road network of Ontario 3

3.1 An example of a database. 17

4.1 A query automaton $\mathcal{A}$, and a labeled, directed and weighted graph database DB 25
4.2 A query automaton $\mathcal{A}$, and a labeled, directed, weighted and **partitioned** graph database DB 26
4.3 A sequence of snapshots of Algorithm 1 for only one machine 27
4.4 A sequence of snapshots of Algorithm 1 for three machines working in parallel - Part (1) 28
4.5 A sequence of snapshots of Algorithm 1 for three machines working in parallel - Part (2) 29
4.6 Termination spanning trees 39
4.7 Rebuilt termination spanning trees 41

6.1 An example of R-Tree indexing 48

7.1 Experimental result 1 (Computational stress). 51

7.2 Experimental result 2 (Communication messages) 52

7.3 Experimental result 3 (OSW updates and Queue size) 53

Acknowledgements

The work described in this thesis was accomplished while I was studying as a Master's student at the University of Victoria, and meanwhile working with Dr. Alex Thomo and Dr. Dan C. Stefanescu as a research assistant. It is the result of two years of work whereby I have been accompanied and supported by a number of people, and without whom this thesis might not have been completed. It is pleased that I have an opportunity here to express my gratitude to all of them.

I would like to gratefully acknowledge to my supervisor Alex for his constant guidance, sage advice, encouragement, and support through out the whole work. I have been in the project of Distributed Regular Path Queries since September 2005. During these two years, Alex always has been full of enthusiasm to our research and also his responsibility for supervising his students including Marina, Maryam, Manuel and I. Besides of being our supervisor, Alex was as close as our good friend.

I would like to thank Dr. Dan C. Stefanescu for his cooperation in this work, and especially for his guidance for presenting this work in the conference AINA-07.

I would like to thank my parents who supported me for finishing both Bachelor and Master degrees in Canada. I am forever indebted to them for their endless patience, encouragement and understanding.

My special gratitude is to my brother and his family for their loving support.

I am very grateful for my patient and loving fiancee, Bo, who has been a great source of strength during the Master's study.

Finally, I would like to thank all my office mates and colleagues, Marina, Maryam, and Manuel, who made me feel like family.

Great Thanks to All.

Dedication

This thesis is dedicated to the loving memory of my uncle, Miao Weizhen and my aunt, Wang Lumei; to my parents, who offered me unconditional love and support throughout the course of this thesis; and to my fiancee, who has supported me in all my endeavours.

Chapter 1

Introduction

1.1 Spatial Networks

Classically, a network is represented as a graph $G = (V, E)$, where V is the set of vertices and $E = \{(u, v) \mid u,\ v \in V\ \}$ is the set of edges between vertices. We say that two vertices u and v are directly connected, if and only if there exists an edge $e = (u, v) \in E$. A network can be directed, labeled, and/or weighted. In a directed network, each edge has a sense of direction from u to v and it can be written as $u \rightarrow v$. In a labeled network, each edge is associated with a label from some finite alphabet. In a weighted network, each edge is associated with a numerical weight.

A spatial network is an extension to a network such that additional spatial objects are associated with the elements (vertices and edges) of the graph. A road network is the most common case of a spatial network, and such a network can be viewed as a directed, labeled and weighted graph. Namely, each vertex represents a road intersection and each edge represents a road segment. The graph is directed as there can be one-way roads. The edge-labels are encodings of road designations like *provincial road*, *highway*, *bridge* etc. The edges are naturally weighted by the

Figure 1.1: A partial map of Ontario

kilometric length of the corresponding road segment.

Usually, the edge weights are not explicitly stored, but rather calculated from the vertex (end-point) coordinates. In general, the spatial position of each vertex with respect to a reference coordinate system is given in terms of geographical coordinates (*i.e.*, latitude and longitude).

In Figure 1.1 we show a partial map of Ontario, Canada that includes several cities such as Toronto, London, Hamilton, and so on. A road network graph corresponding to the map in Figure 1.1 is shown in Figure 1.2.

A path in a graph G is a sequence of vertices $\langle v_0, v_1, v_2, \ldots, v_n \rangle$ such that $\langle v_i, v_{i+1} \rangle$ for each $0 \leq i \leq n-1$ is an edge in G. The single-source shortest path problem is the problem of finding the "best" paths between a given source node and each other node in the graph, where "best" is defined depending on the application. Usually, in a road network we look for paths such that the sum of the weights of its constituent edges is minimized. The single-source shortest path problem has been widely studied for more than 40 years, and there are several famous shortest path algorithms such as

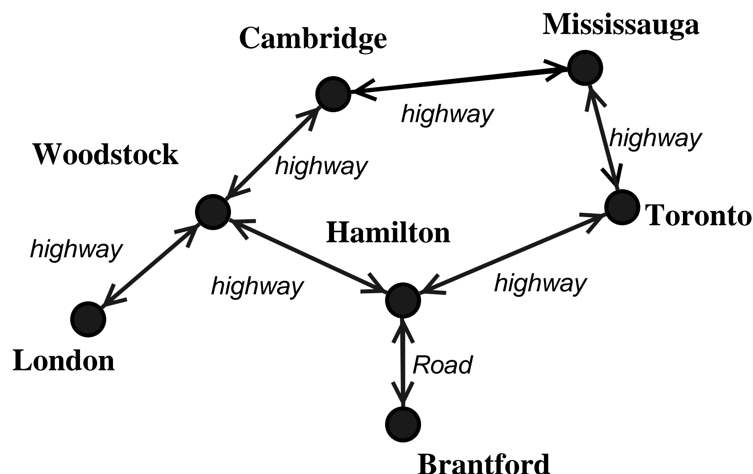

Figure 1.2: A partial road network of Ontario

the Bellman-Ford algorithm, Dijkstra's algorithm, etc. In Geographical Information Systems (GIS), single-source shortest path queries are one of the most frequently asked queries.

1.2 Preferential Path Queries

Often users want to ask queries not only for finding the shortest path between points in a map, but also for finding such a path conforming to their preferences. As an example from the map network shown in Figure 1.2, imagine a user who wants to find the shortest path from London to Toronto consisting of highway segments only.

How can the user express such natural preferences on database edge labels? If we consider the network graph to be stored in relational database tables, we could think to use SQL for expressing user preferences. For example, if we store the *highway* edges in a table *Highways(start, end)*, then in order to find all two-hops *highway* paths one could write the SQL query

SELECT H_1*.start,* H_2*.start,* H_2*.end*

FROM Highways H_1*, Highways* H_2

WHERE H_1*.end* = H_2*.start;*

In general, given a natural number n, in order to find all the n-hops *highway* paths, one could write the SQL query

SELECT H_1*.start,* H_2*.start,* $\ldots$*,* H_n*.start,* H_n*.end*

FROM Highways H_1*, Highways* H_2*,* $\ldots$*, Highways* H_n

WHERE H1.end = H2.start AND $\ldots$ *AND* H_{n-1}*.end* = H_n*.start;*

The problem is that we do not know in advance the number of hops of the best *highway* path. In GIS databases such as Tiger [17], the road segments are quite short and to go from a city to another city one needs to travel numerous road segments.

Clearly, for querying spatial network databases we ultimately need a recursive query language and SQL standard is not such a one.[1] Only by having a recursive query language, we can navigate arbitrarily long paths in spatial networks.

When it comes to navigating labeled graphs, *regular path queries* are the common choice (cf. [9, 1, 16, 11, 4]). Regular path queries are given by means of regular expressions over the database graph labels. For example, the user can express her *highway* preferences by giving the regular expression

$$\textit{highway}^*,$$

where the $*$ operator is the Kleene "star" standing for the transitive closure of the concatenation. It is this Kleene star operation that allows users to specify a limited form of recursion in their queries.

[1] A recursive variant of SQL is only implemented in DB2 from IBM.

Interestingly, by using regular path queries, the user can express a multitude of navigational preferences such as

$$highway^{*} \cdot (road + street) \cdot highway^{*},$$

which says that "I prefer *highways* and I am willing to tolerate up to one provincial *road* or city *street*!"

Now, the problem becomes to find the shortest of those paths, which spell words in the (regular) query language. Solving effectively this problem from a database perspective is very important as the spatial networks today are mainly stored in secondary storage rather than in main memory. Moreover the data might be distributed, and thus, one has to devise distributed strategies for evaluating regular path queries.

1.3 Distributed/Parallel Evaluation

Despite the easiness of writing regular path queries, it is quite expensive to evaluate such queries on real big data graphs. Based on our experiments on the Tiger spatial database ([17]), to evaluate regular path queries is usually in the order of minutes.

Hence, a distributed strategy of evaluating regular path queries is needed not only because the data might be distributed, but also to speed up the evaluation when multiple machines are available for use.

In general, there are two types of parallelism that database users can utilize inter-query parallelism and intra-query parallelism. Inter-query parallelism is the ability to use multiple processors to execute independent transactions simultaneously, and no transaction requires the output of other transactions to complete. Usually the inter-query parallelism works well for On-Line Transactional Processing (OLTP)

application support. It improves the throughput of the system by executing queries from multiple transactions in parallel, and such transaction are usually light-weight.

However, the inter-query parallelism is not appropriate for heavy-weight queries (such as regular path queries), that need large amounts of data accesses and complex operations, since each query is still executed by only one processor. For such queries, one has to explore the possibility of intra-query parallelism.

For employing intra-query parallelism one has to intelligently break a single query into many subtasks and to execute those subtasks in parallel using different processors. Clearly, this is more challenging and not always possible. In this thesis, we show that regular path queries are good candidates for intra-query parallelism and we show how to do this effectively and efficiently. We remark here that "slicing up" a query is challenging because it has to satisfy (sometimes) conflicting desiderata, such as minimization of communications between computing machines, maximization of machine utility, progressive generation of reliable intermediate answers, and so on.

Finally, we would like to mention that our distributed strategy for evaluating regular path queries is viable for both paradigms of today's distributed computing: cluster and grid computing.

Usually, a *cluster* is a set of fast server-grade machines connected by high bandwidth links. Notably, in such systems, by devising an appropriate inter-query parallelism, we are able to achieve an "on-line" performance for evaluating regular path queries.

On the other hand, a *computational grid* is a community of machines, which are primarily destined for other tasks different from the tasks they can be assigned in a grid. Such machines are offered for community service by organizations and indi-

viduals during low intensity periods with respect to their main tasks. The machines are allowed to maintain a degree of freedom regarding the work they accept and reliability that they offer. In this thesis, we show how to adapt our distributed strategy to be resilient against machine losses. Also, we experimentally show that due to the good scaling of our method, the load imposed on participating grid machines becomes negligible when the number of machines grows considerably (as is the case in computational grids).

Adapting the computations to take into account the special computational and reliability restrictions in a grid has turned out to be a big challenge in many applications. There is much research being conducted into devising efficient grid-aware methods for various applications.

In this thesis, we will refer to our method as a “grid-aware” strategy because besides providing an “on-line” performance for a cluster setting, it also offers features appropriate for a grid setting.

1.4 Thesis Overview

The thesis consists of 8 Chapters, and it is organized as follows:

Chapter 1 is the introduction to this thesis. It provides the background of spatial networks, preferential path queries, and two types of query evaluation parallelism: inter-query and intra-query. The task description and thesis overview are also provided.

Chapter 2 discusses the challenges of distributed evaluation of RPQs by a “grid-aware” strategy. It presents an overview of the motivation and related works in the field, then the approach we use, and the related challenges we are facing.

Chapter 3 presents formal definition of spatial network databases, regular path queries (RPQs), and their semantics.

Chapter 4 presents our grid-aware distributed algorithm, which has two interwoven components: the RPQ evaluation and the resilient machine loss and termination detection, also their corresponding examples.

Chapter 5 discusses queue strategies for optimizing our algorithms.

Chapter 6 presents the background of database indexing technique, and the most common case, B-Tree, also introduces one of its extensions, R-Tree, which is utilized in our approach.

Chapter 7 provides our experiment settings and experimental results.

Chapter 8 states the conclusions drawn from our work.

Chapter 2

Background and Challenges

2.1 Motivation

As mentioned in the Introduction, regular path queries (RPQs) are used for (preferentially) navigating graph databases (or data-graphs in short). As their name suggests, these queries are described by means of regular expressions over the alphabet of database edge-labels. Computationally, RPQs are represented by finite state automata. The answer to an RPQ is the set of database objects reachable by paths spelling words in the corresponding regular language. For example, the answer to the query

$$\mathit{highway}^* || (\mathrm{road} + \epsilon)^k,$$

where || is the shuffle operator ([19]),[1] is the set of objects reachable by following highways interleaved by no more than k roads.

Over the last years, RPQs have been the focus of numerous works (see [1, 5, 18, 4, 8] etc). This is not surprising as RPQs are the basic building block of almost

[1] ϵ is the empty word.

all query languages for data-graphs, providing the user with a nice and simple way to express recursion (see for a discussion [18]). However, RPQs are notorious for their high computational demand. For this reason, there have been many attempts to find clever ways to evaluate the "nice" but "time consuming" RPQs. Most of such work is related to XML data-trees, and their methods seem to apply to trees only. There has been much less work regarding the evaluation of RPQs on general data-graphs or regarding the evaluation of RPQs on particular data-graphs of great practical interest, such as large spatial networks. As the seminal work [9] points out, RPQs are an integral part of an intelligent querying of spatial networks.

In this thesis, we focus on grid-aware evaluation of RPQs on spatial networks for which we present a distributed algorithm, which, experimentally, exhibits a desirable property for a grid setting: the computational stress on participating machines decreases proportionally with the increase of the number of machines. This is especially appropriate for grids, where the power comes from the large number of machines rather than from their individual power, and where machines can refuse to accept load above some predefined threshold.

The main characteristic of the spatial networks that distinguishes such databases from other graph databases studied in the literature, is the fact that they can be conveniently modeled by graphs, which besides being labeled are also weighted. Navigation of such databases implies more than "starting from object a we can reach object b by following some path spelling a word in a given RPQ." Rather, one is also interested in discovering the cheapest such path that connects a with b and its expense.

This additional requirement makes the evaluation task more difficult. Namely, we

cannot benefit anymore from special graph indexes, such as data-guides introduced in the literature (see for example [11]) because they ignore the cost of the database paths. The lack of index structures for this problem makes developing a distributed grid-aware approach even more important if one is to have RPQs feasible for practical use.

2.2 Related Works

Regular path queries are by now part of the folklore (cf. [9, 1, 16, 11, 4]). Despite the great attention on RPQs over the years, there has been no work in devising efficient algorithms for their evaluation on spatial networks which include, as a special case, road networks. As explained above, because of the special nature of the spatial network graphs, the known (efficient) methods for evaluating RPQs are not applicable in this case.

Few works have dealt with a distributed evaluation of path queries. The most important are [2], [16], and [14]. In [2] and [16], the data-graphs are unweighed, and their algorithms do not seem generalizable to our case. Moreover, the algorithm of [16], distributes the load unevenly among processors, which is an undesirable feature in a grid setting. The algorithm of [2] assumes that each processor services one object only, and this is one more reason for the inapplicability of [2] in our setting.

The distributed algorithm of [14] is a generalization of the parallel shortest path algorithm of [10], and works on weighted regular path queries. While the algorithm of [14] can be adapted to work for weighted databases, the approach lacks many important features that are needed in a grid setting, notably termination detection, fault tolerance and experimental considerations.

Hribar et. al. in [10] provide important insights on the factors affecting the performance of their parallel shortest path algorithm. However, they deal with graphs stored in main memory and not in secondary storage as in our setting. Also, we are not solving an unrestricted shortest path problem, but rather one guided by a query automaton. Furthermore, we engineer for stress reduction on the grid machines, significant suppression of inter-machine communication, and resilience against unexpected machine losses.

Finally, in [15], a distributed all-to-all algorithm is presented. The setting there is different, first because the query is assumed to start from each object (as opposed to starting from a designated object), and second because it is assumed that each object is served by a dedicated (for that object) process. Both these assumptions are not applicable to spatial networks with millions of objects.

2.3 Our Approach

While the general idea of [14] seems to work in a general distributed setting, it becomes "the devil is in details" when one tries it in a grid environment. In this thesis we tackle several challenges in turning the proposed method of [14] into a practical and resilient algorithm for a grid setting. Furthermore we present experimental results and discuss performance considerations.

First let us give a high level description of [14]. For this, assume a database is partitioned among a set of cooperating machines, and a given RPQ is to be evaluated starting from an object o residing in the partition belonging to some originating machine. Let s be the start state of an automaton $\mathcal{A}$ for the given RPQ. Now, associate o with s, and starting with this association, perform path "expansions,"

which generate other object-state associations. An expansion is possible if there is a match of a database edge with an automaton transition both originating from the object and state (respectively) of the association being expanded. The associations are labeled with the weight of the best path known so far. When the query evaluation needs to continue to other database partitions residing in other machines, the relevant information is packed into messages and sent to the corresponding machines. If an object a is associated with a final state, then a is produced as an answer to the given RPQ and is labeled with the weight of the path followed to reach it. As it is possible that the same object can be reached (later on) by some other path, the label of the corresponding answer can be "corrected" at a later stage.

The above simple procedure can become a viable practical algorithm (for a grid setting) if one further addresses issues related to termination detection, fault tolerance and performance optimization. These issues are the focus of this thesis.

Ensuring termination detection is an important, but challenging goal of any distributed algorithm. The task is more difficult if one allows for sudden machine losses during computation. Such losses can easily happen in a grid setting, and grid-aware algorithms must be resilient under these conditions. In this thesis we introduce an RPQ evaluation methodology that includes a resilient algorithm for termination detection, which smoothly adapts to machine losses. Our approach for fault tolerance allows for the computation to continue, on the fly, on mirroring machines, i.e. machines that handle portions of the database previously served by the defunct hardware. If such "back-up" machines are not available, we make provisions to provide at least the search results obtainable were the search to be run on the subset of the spatial network database served by the currently available machines. Furthermore,

we remark that, since at some point in time, some of the navigation used the whole database, we may get more results than those strictly available if we were to restart the RPQ evaluation once some machines failed.

We assess the performance of our RPQ evaluation approach by first investigating issues associated with computational and communicational costs. It is important to design for reduced computational stress on the participating machines since, in a grid setting, computational devices may not tolerate loads above certain threshold. One element to investigate in the quest for "balanced" computation is the choice of the data partitioning. It turns out that a good partitioning is interrelated with the database storage scheme for which our approach uses a clustering R-Tree (spatial) index. With such an index, we not only cluster together (in disk-blocks) segments that are spatially close to each other, but also partition the data among participating grid machines. Namely, each machine locally stores and works on a subset of the R-Tree leaves. After this R-Tree partitioning, each machine builds a local R-Tree index on the blocks assigned to it.

The investigation of the communication properties of our approach to RPQ evaluation revealed that the same structures that support computation recovery after machine losses also serve as "message suppressors," that significantly reduce communication, thus rendering the overall number of messages "negligible" for today's high-speed networks.

The other observation is that the total number of messages is almost independent of the number of participating grid machines. This is certainly very desirable in a grid environment because it allows for scaling to large number of machines without induced message penalties.

Further performance tuning requires investigating the details of the computation and, in particular, the choice of expanding object-state associations. These associations are inserted in a processing queue and then experimentation is done with various queue strategies having as objectives the reduction of stress on machines, the quality of intermediate query answers, and the minimization of the number of messages. It turns out that the choice of queueing strategy matters for all the above goals. Further improvements can be achieved by the processing order in an expansion, i.e. the order among three basic steps: dequeue (an object-state association), find "next" associations, and possibly relax weights in existing associations. This is a subtle point. While the dequeue step has to come first, the other two steps can have the order interchanged. In this thesis, we propose the order: dequeue, relax weights due to the dequeued association, and then find and enqueue next associations. This order provides better quality of intermediate query answers when using the best queue strategy.

Chapter 3

Basic Definitions

In this part, we give the definition of Regular Path Queries (RPQs), and the weighted answer of RPQs on spatial network databases. We consider a spatial network database (or just database in short) to be an edge-labeled graph with real non-negative values assigned to the edges. Intuitively, the nodes of the database graph represent objects and the edges represent spatial segments and their length.

Formally, let Δ be an alphabet. Elements of Δ will be denoted $R, S, \ldots$. In practice such symbols are for example road, highway, freeway, bridge, and so on. As usual, Δ^* denotes the set of all finite words over Δ. Words will be denoted by $u, w, \ldots$. Objects will be denoted $a, b, c, \ldots, o, \ldots$. A *database DB* is then a weighted graph (V, E), where V is a set of objects, and $E \subseteq V \times \Delta \times \mathbb{R}^+ \times V$ is a set of directed edges labeled with symbols from Δ and weighted with numbers from $\mathbb{R}^+$.

A *regular path query* (RPQ) is a regular language over Δ. As such, computationally, an RPQ is a finite state automaton (FSA) $\mathcal{A} = (P, \Delta, \tau, p_0, F)$, where P is the set of states, Δ is the alphabet, τ is the transition relation, p_0 is the initial state, and F is the set of final states. For the ease of notation, we will blur the distinction

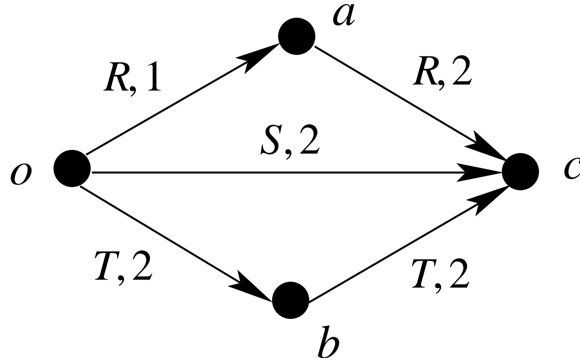

Figure 3.1: An example of a database.

between RPQs and FSA's that represent them. Let us denote with $length(\pi)$ the usual length of a path π in the database, *i.e.* the sum of the edge-weights along the path. We define the *weighted answer* (WAns) to $\mathcal{A}$ as follows

$$
\begin{aligned}
WAns(\mathcal{A}, o, DB) = \{(a, r) \in V \times \mathbb{R} \ : \ & \text{there exists a path } \pi \text{ from } o \text{ to } a \text{ in } DB, \\
& \text{which matches an accepting path } \rho \text{ in } \mathcal{A}, \\
& \text{and } r = min\{length(\pi) : \pi \text{ as above}\}\}.
\end{aligned}
$$

As an example consider the database DB in Figure 3.1 and the query $Q = RR + TT$. There are three paths going from object o to object c. The shortest path consisting of a single edge of weight 2 spells the word S which does not belong to query Q. The two other paths spell words in Q (RR and TT respectively). The shorter of these two is the path spelling RR with a length of 3, and thus we have that $(c, 3) \in WAns(\mathcal{A}_Q, o, DB)$.

Before presenting the distributed (weighted) query evaluation, it will help to

shortly review the evaluation in the classical case. In essence, the evaluation (in the classical case) proceeds by creating object-state pairs from the database and query automaton. For this, let $\mathcal{A}$ be an automaton that accepts a query Q. Starting from a root object o of a database DB, we first create the pair (o, p_0), where p_0 is the initial state in $\mathcal{A}$. Then, we create all the pairs (a, p) such that there exist an edge from o to a in DB and a transition from p_0 to p in $\mathcal{A}$, and, furthermore the labels of the edge and transition match. In the same way, we continue to create new pairs from the existing ones, until we are not able anymore to do so. At that point, we produce as the answer to the query the set of objects, which have been associated with some final state of the query automaton $\mathcal{A}$.

It is worth mentioning here that the state-object pairs induce an (implicit) edge labeled graph with these pairs as its nodes. Regarding the edges, let (b, q) be obtained by another pair, say (a, p), through a database edge and automaton transition both labeled by R. Then, we consider an R-labeled edge from (a, p) to (b, q) in the induced graph.

Now, when having a weighted database, we can modify the classical matching algorithm to build instead a weighted object-state graph. This can be achieved by assigning to the edges of this graph the corresponding database edge weights.

It is not difficult to see that, in order to find the weighted answers to the query, we have to find, in the object-state graph, the shortest paths from the “source” (o, p_0) to all the nodes (a, p), where p is a final state in the query automaton $\mathcal{A}$. The complexity of *regular path query* (RPQ) is proportional to the Cartesian product of the database with the query automaton in the worst case.

However, the challenge is that when the database is large and distributed, we

cannot afford to construct the above graph, and then use some centralized shortest path algorithm on it.

In the next chapter we present an effective method for distributively computing the weighted answers to a user query.

Chapter 4

Grid-Aware Distributed Evaluation of Regular Path Queries

Before evaluating any query, we spatially cluster the database into disk blocks by using clustering R-Tree index described in Chapter 6. Then the database blocks are distributed to the participating machines.

Since in a grid setting, the machines may leave in the middle of a computation, we take into account the possibility of replicating the data partitions to several machines. During the evaluation, only one of the machines that store a partition is selected. If later on, the machine decides to leave the evaluation, then the algorithm smoothly "switches" to another machine storing the same data partition(s), and recovers the lost part of the computation.

We denote the participating machines with $\ldots, M_i, \ldots, M_j, \ldots$ We organize the database as an edge relation. The edges of the database are categorized as: a) local edges connecting objects stored in the same machine or b) cross-boundary edges connecting objects stored in different machines. In an edge $(a, R, r, b) \in D \times \Delta \times \mathbb{R}^+ \times D$ we also store a flag indicating whether the edge is local or not. If the edge is

a cross-boundary one, we also store an id-list of the machines where the (next) edges $(b, _, _, _)$ are stored.

Each machine maintains three structures:

1. A table of object-state-weight (OSW) triples. We call them OSW tables, and denote with OSW_i the table of a machine M_i. The OSW table has a hash organization in our implementation. Initially, these tables are empty.

2. A processing queue (Q) of object-state-weight triples. We denote with Q_i the queue of a machine M_i. Initially, all queues are empty except for the queue of the initiating machine, which stores the triple $(o, s, 0)$, where o is the designated origin object and s is the starting state of the query automaton. The queue strategies and their significance for grid settings are discussed in the next chapter.

3. A message log table (Log) of object-state-weight-machine quadruples. We denote with Log_i the message log of a machine M_i. The Log table have a hash organization in our implementation. These tables are the key structure for recovering the lost computation when machines (suddenly) leave the query evaluation. Also, the Log tables are important in significantly reducing the messages to an almost negligible number. Initially, these tables are empty.

We can visualize the answering of an RPQ query as (implicitly) building on the fly a weighted graph of object-state associations. With this visualization, the weights in the OSW triples correspond to the weights that a classical single-source shortest path algorithm maintains for the processed nodes of the graph.

Each machine M_i works in parallel with other machines as follows. First a triple, say (a, p, r), is removed from queue Q_i, and checked against OSW_i to see whether there is already a triple $(a, p, _)$. If not, then (a, p, r) is inserted into OSW_i. Otherwise, when a triple, say (a, p, s) is found in OSW_i, we update (or relax) its weight by setting $s \leftarrow min\{r, s\}$.

If a dequeued triple was "useful," *i.e.* if it caused an insertion or update in OSW_i, then such a triple might trigger further insertions or updates. Thus, for a useful triple we have a second step in its processing. During this step, we "expand" it by generating the "next" OSW triples. The expansion is done by trying to find an edge from object a and state p and a transition (respectively) that match. Each expansion creates new OSW triples, which are inserted in Q_i. It might happen that some new triple, say (b, q, s), is created by following a cross-boundary database edge. In such a case the triple is "not local," *i.e.* object b is not stored in M_i, but in another machine M_j. [When we say that "object b is stored in M_j," actually what we mean is that the edge tuples $(b, _, _, _)$ are stored in M_j.] When such triples will be dequeued and processed, they will be packed into messages and sent to the corresponding machines. Each message sent by machine M_i is also logged in table Log_i.

A machine M_i can also receive messages of the type $\langle M_j, gone\rangle$, which informs M_i about the loss of a grid machine M_j from the query evaluation. As we mentioned above, we can assume the existence of partition replicas, and so, the data stored in machine M_j might also exists in some other machine, say M_k. In such a case, machine M_i retrieves from Log_i all the messages (if any) sent earlier to M_j, and resends them to M_k. Machine M_k, receiving work through these messages (sent earlier to M_j), will be able to redo the work of M_j, and continue further. It should be clear that our

algorithm continues to work fine when there are machine losses even without having data replication. The algorithm will not get "stuck," but continue to produce all the answers, which are obtainable from the part of the database graph stored in the remaining "live" machines. The fact that our algorithm can take advantage of data replication makes it more general, and able to produce the "perfect" query answer set when replication is in place.

The algorithm terminates when the processing queues of each machine get empty, and when there are no messages, which are sent but not yet received.

Our algorithm has two interwoven components: the computation of query answers (as described above), and its machine loss and termination detection. To simplify the presentation and to improve readability, we present the components separately. The two components can be easily merged in an unified algorithm.

4.1 Computation of Query Answers

The computation of query answers is formally as follows.

Algorithm 1

Input: A query automaton $\mathcal{A} = (P, \Delta, \tau, p_0, F)$, a database DB, and a start object o of the DB. Automaton $\mathcal{A}$ is sent first to all participating machines.

Output: $WAns(\mathcal{A}, o, DB)$.

Method:

Suppose that object o is in machine M_0.

1. Insert $(p_0, o, 0)$ in queue Q_0 (of M_0).
2. Repeat 3 and 4, at each machine M_i in *parallel*, until termination is detected.

3. Remove from queue Q_i (according to its policy) a triple (a, p, r).

   (a) **if** $p \in F$ (*i.e.* p is a final state in query automaton $\mathcal{A}$)

   then

   insert (a, r) in $WAns(\mathcal{A}, o, DB)$ or update some existing

   $(a, s) \in WAns(\mathcal{A}, o, DB)$ by setting

   $s \leftarrow min\{r, s\}$

   (b) **if** a is a local object

   then

   insert (a, p, r) in OSW_i if there does not exist a triple (a, p, s), or otherwise (possibly) update (a, p, s) by setting $s \leftarrow min\{r, s\}$.

   else (object a belongs to a remote machine)

   if $(a, p, r) \notin Log_i$ or

   $(a, p, r') \in Log_i$ and $r' > r$

   then

   pack (a, p, r) in a message and send it to the machine responsible for a. Insert (a, p, r) in Log_i table.

   (c) **if** an insertion or update happened in OSW_i

   then

   For each edge $a \xrightarrow{R,c} b$ in DB and each transition $(p, R, q) \in \tau$, insert the triple $(q, b, r + c)$ into Q_i.

4. Upon receipt of a message $\langle a, p, r \rangle$ insert the triple (a, p, r) into Q_i.

5. Upon receipt of a message $\langle M_j, \text{gone} \rangle$, resend all the messages $\langle a, p, r \rangle \in Log_i$, where object a belongs to M_j, to a live machine M_k, which replicates the M_i data partition.

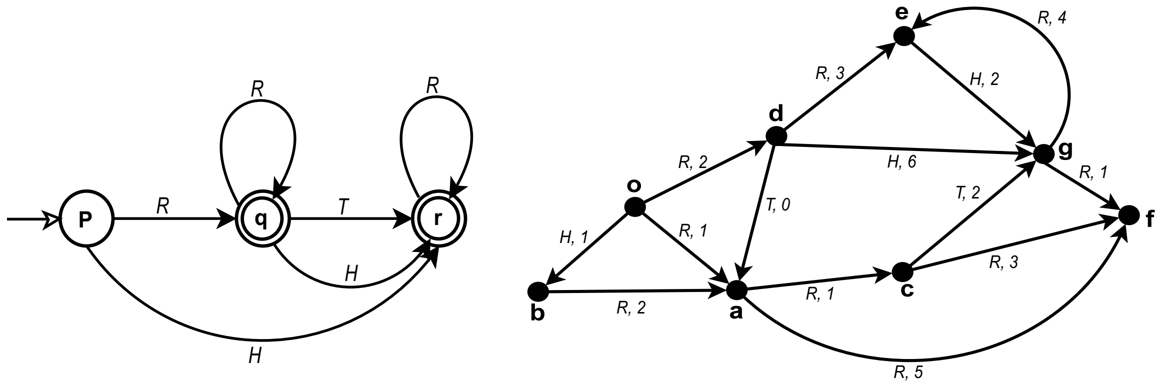

Figure 4.1: A query automaton $\mathcal{A}$, and a labeled, directed and weighted graph database DB

Observe that in step 3 (a) of the above algorithm, we build $WAns(\mathcal{A}, o, DB)$ incrementally each time that an object is associated with a final state. In practice, such answers are directly sent by the machines to the user as soon as they are discovered. The question is what is the quality of the produced answers? This question arises because the weight of the answer objects might be lowered later due to discovery of new cheaper paths from the origin. At the termination of the algorithm the weights will be optimal, but the question is what can be done to have almost optimal intermediate answer weights. Interestingly, the quality of intermediate answers is significantly influenced by the processing queue strategy that we discuss in detail in Chapter 5.

We would like to mention here that the above algorithm can be easily enhanced to also produce the cheapest paths corresponding to the query answers.

Another feature that we also want to stress is about the *Log* tables. Namely, they not only make possible the recovery of the computation in case of machine losses, but also serve as *message suppressors*, which significantly reduce the number of messages in the system and the computational stress on the grid-machines. To see this, let us

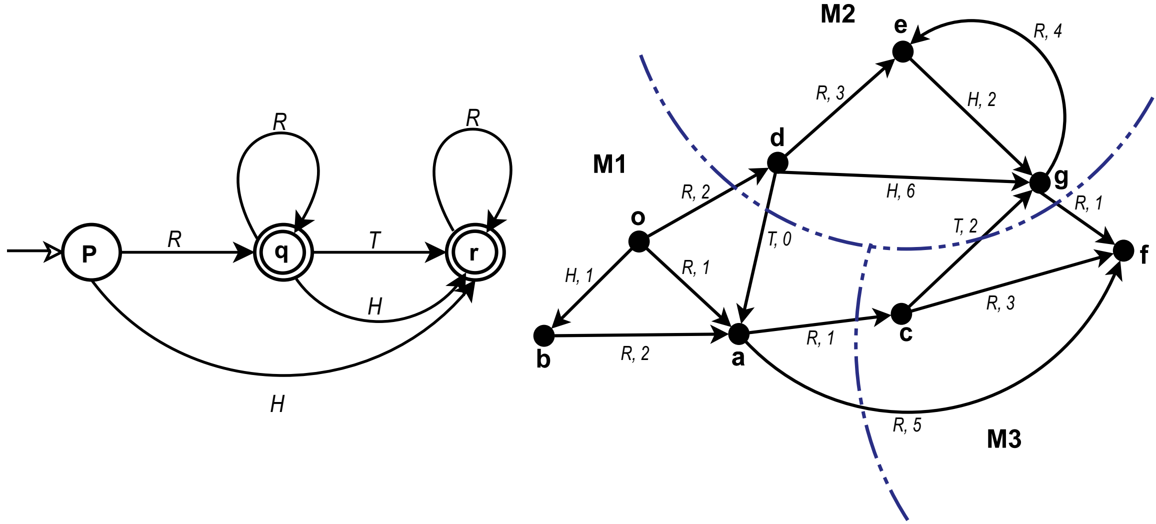

Figure 4.2: A query automaton $\mathcal{A}$, and a labeled, directed, weighted and **partitioned** graph database DB

consider an object a in some machine M_j, which has several incoming cross-boundary edges from objects in some other machine M_i. Object a can be reached by several paths, going through M_i objects, and thus, there might be an attempt (by M_i) to send many $\langle a, p, _ \rangle$ messages, for some state p in $\mathcal{A}$. However, it makes sense to send $\langle a, p, r \rangle$ only if it is the first $\langle a, p, _ \rangle$ message, or if the last such message, say $\langle a, p, r' \rangle$, has $r' > r$. Notably, *Log* tables give us the ability to perform this check, and suppress many useless messages. Experimentally, we found that *Log* tables are smaller than *OSW* tables, and both were small enough to be kept in main memory. For typical queries on (big) real spatial databases, these tables were in the order of only few thousand elements (see Chapter 7).

Before illustrating Algorithm 1 in distributed evaluation of RPQs, we first would like to show how this algorithm works when having a single machine only. Suppose we have a machine M, which maintains only two data structures: an object-state-

Step No.	Processing Queue	Object-State-Weight (OSW) table
1	(o, p, 0)	
2	*(d, q, 2) (b, r, 1) (a, q, 1)*	*(o, p, 0)*
3	(b, r, 1) (a, q, 1) *(a, r, 2) (e, q, 5) (g, r, 8)*	(o, p, 0) *(d, q, 2)*
4	(a, q, 1) (a, r, 2) (e, q, 5) (g, r, 8) *(a, r, 3)*	(o, p, 0) (d, q, 2) *(b, r, 1)*
5	(a, r, 2) (e, q, 5) (g, r, 8) (a, r, 3) *(c, q, 2) (f, q, 6)*	(o, p, 0) (d, q, 2) (b, r, 1) *(a, q, 1)*
6	(e, q, 5) (g, r, 8) (a, r, 3) (c, q, 2) (f, q, 6) *(f, r, 7) (c, r, 3)*	(o, p, 0) (d, q, 2) (b, r, 1) (a, q, 1) *(a, r, 2)*
7	(g, r, 8) (a, r, 3) (c, q, 2) (f, q, 6) (f, r, 7) (c, r, 3) *(g, r, 7)*	(o, p, 0) (d, q, 2) (b, r, 1) (a, q, 1) (a, r, 2) *(e, q, 5)*
8	**(a, r, 3)<ignored>** (c, q, 2) (f, q, 6) (f, r, 7) (c, r, 3) (g, r, 7) *(f, r, 9) (e, r, 12)*	(o, p, 0) (d, q, 2) (b, r, 1) (a, q, 1) **(a, r, 2)** (e, q, 5) *(g, r, 8)*
9	(f, q, 6) (f, r, 7) (c, r, 3) (g, r, 7) (f, r, 9) (e, r, 12) *(f, q, 5) (g, r, 4)*	(o, p, 0) (d, q, 2) (b, r, 1) (a, q, 1) (a, r, 2) (e, q, 5) (g, r, 8) *(c, q, 2)*
10	(f, r, 7) (c, r, 3) (g, r, 7) (f, r, 9) (e, r, 12) (f, q, 5) (g, r, 4)	(o, p, 0) (d, q, 2) (b, r, 1) (a, q, 1) (a, r, 2) (e, q, 5) (g, r, 8) (c, q, 2) *(f, q, 6)*
11	(c, r, 3) (g, r, 7) (f, r, 9) (e, r, 12) (f, q, 5) (g, r, 4)	(o, p, 0) (b, r, 1) (d, q, 2) (a, q, 1) (a, r, 2) (e, q, 5) (g, r, 8) (c, q, 2) (f, q, 6) *(f, r, 7)*
12	(g, r, 7) (f, r, 9)(e, r, 12) (f, q, 5) (g, r, 4) *(f, r, 6)*	(o, p, 0) (b, r, 1) (d, q, 2) (a, q, 1) (a, r, 2) (e, q, 5) **(g, r, 8)** (c, q, 2) (f, q, 6) (f, r, 7) *(c, r, 3)*
13	**(f, r, 9)<ignored>** (e, r, 12) (f, q, 5) (g, r, 4) (f, r, 6) *(e, r, 11) (f, r, 8)*	(o, p, 0) (b, r, 1) (d, q, 2) (a, q, 1) (a, r, 2) (e, q, 5) ***(g, r, 7)<update>*** (c, q, 2) (f, q, 6) **(f, r, 7)** (c, r, 3)
14	(f, q, 5) (g, r, 4) (f, r, 6) (e, r, 11) (f, r, 8)	(o, p, 0) (b, r, 1) (d, q, 2) (a, q, 1) (a, r, 2) (e, q, 5) (g, r, 7) (c, q, 2) **(f, q, 6)** (f, r, 7) (c, r, 3) *(e, r, 12)*
15	(g, r, 4) (f, r, 6) (e, r, 11) (f, r, 8)	(o, p, 0) (b, r, 1) (d, q, 2) (a, q, 1) (a, r, 2) (e, q, 5) **(g, r, 7)** (c, q, 2) ***(f, q, 5)<update>*** (f, r, 7) (c, r, 3) (e, r, 12)
16	(f, r, 6) (e, r, 11) (f, r, 8) *(e, r, 8) (f, r, 5)*	(o, p, 0) (b, r, 1) (d, q, 2) (a, q, 1) (a, r, 2) (e, q, 5) ***(g, r, 4)<update>*** (c, q, 2) (f, q, 5) (f, r, 7) (c, r, 3) (e, r, 12)
17	(e, r, 11) ***(f, r, 8)<ignored>*** (e, r, 8) (f, r, 5)	(o, p, 0) (b, r, 1) (d, q, 2) (a, q, 1) (a, r, 2) (e, q, 5) (g, r, 4) (c, q, 2) (f, q, 5) (c, r, 3) ***(f, r, 6)<update>*** **(e, r, 12)**
18	(e, r, 8) (f, r, 5)	(o, p, 0) (b, r, 1) (d, q, 2) (a, q, 1) (a, r, 2) (e, q, 5) (g, r, 4) (c, q, 2) (f, q, 5) (c, r, 3) (f, r, 6) ***(e, r, 11)<update>***
19	*EMPTY*	(o, p, 0) (b, r, 1) (d, q, 2) (a, q, 1) (a, r, 2) (e, q, 5) (g, r, 4) (c, q, 2) (f, q, 5) (c, r, 3) ***(f, r, 5)<update> (e, r, 8)<update>***

Figure 4.3: A sequence of snapshots of Algorithm 1 for only one machine

Step No.		Machine 1 *(o, a, b)*	Machine 2 *(e, d, g)*	Machine 3 *(c, f)*
1	**Queue**	(o, p, 0)	*Empty*	*Empty*
	OSW			
	Log			
2	**Queue**	*(d, q, 2) (b, r, 1) (a, q, 1)*	*Empty*	*Empty*
	OSW	*(o, p, 0)*		
	Log			
3	**Queue**	(b, r, 1) (a, q, 1)	*(d, q, 2)*	*Empty*
	OSW	(o, p, 0)		
	Log	(d, q, 2, 2)		
4	**Queue**	(a, q, 1) *(a, r, 3)*	*(a, r, 2)(e, q, 5)(g, r, 8)*	*Empty*
	OSW	(o, p, 0) *(b, r, 1)*	*(d, q, 2)*	
	Log	(d, q, 2, 2)		
5	**Queue**	(a, r, 3) *(a, r, 2)* *(c, q, 2) (f, q, 6)*	(e, q, 5)(g, r, 8)	*Empty*
	OSW	(o, p, 0) (b, r, 1) *(a, q, 1)*	(d, q, 2)	
	Log	(d, q, 2, 2)	(a, r, 2, 1)	
6	**Queue**	(a, r, 2) (c, q, 2) (f, q, 6) *(f, r, 8) (c, r, 4)*	(g, r, 8) *(g, r, 7)*	*Empty*
	OSW	(o, p, 0) (b, r, 1) (a, q, 1) *(a, r, 3)*	(d, q, 2) *(e, q, 5)*	
	Log	(d, q, 2, 2)	(a, r, 2, 1)	
7	**Queue**	(c, q, 2) (f, q, 6) (f, r, 8) (c, r, 4) *(f, r, 7) (c, r, 3)*	(g, r, 7) *(f, r, 9)* *(e, r, 12)*	*Empty*
	OSW	(o, p, 0) (b, r, 1) (a, q, 1) ***(a, r, 2)**<update>*	(d, q, 2) (e, q, 5) *(g, r, 8)*	
	Log	(d, q, 2, 2)	(a, r, 2, 1)	
8	**Queue**	(f, r, 8) (c, r, 4) (f, r, 7) (c, r, 3)	(f, r, 9) (e, r, 12) *(f, r, 8)* *(e, r, 11)*	*(c, q, 2) (f, q, 6)*
	OSW	(o, p, 0) (b, r, 1) (a, q, 1) (a, r, 2)	(d, q, 2) (e, q, 5) ***(g, r, 7)**<update>*	
	Log	(d, q, 2, 2) (c, q, 2, 3) (f, q, 6, 3)	(a, r, 2, 1)	
9	**Queue**	(f, r, 7) (c, r, 3)	(e, r, 12) (f, r, 8) (e, r, 11)	(f, q, 6) *(f, r, 9) (f, q, 5) (g, r, 4) (f, r, 8) (c, r, 4)*
	OSW	(o, p, 0) (b, r, 1) (a, q, 1) (a, r, 2)	(d, q, 2) (e, q, 5) (g, r, 7)	*(c, q, 2)*
	Log	(d, q, 2, 2) (c, q, 2, 3) (f, q, 6, 3) (f, r, 8, 3) (c, r, 4, 3)	(a, r, 2, 1) (f, r, 9, 3)	

Figure 4.4: A sequence of snapshots of Algorithm 1 for three machines working in parallel - Part (1)

Step No.		**Machine 1 *(o, a, b)***	**Machine 2 *(e, d, g)***	**Machine 3 *(c, f)***
10	**Queue**	*Empty*	(e, r, 11)	(f, r, 9) (f, q, 5) (g, r, 4) (f, r, 8) (c, r, 4) *(f, r, 7)* *(c, r, 3)* *(f, r, 8)*
	OSW	(o, p, 0) (b, r, 1) (a, q, 1) (a, r, 2)	(d, q, 2) (e, q, 5) (g, r, 7) *(e, r, 12)*	(c, q, 2) (f, q, 6)
	Log	(d, q, 2, 2) (c, q, 2, 3) (f, q, 6, 3) **(f, r, 7, 3)<update>** **(c, r, 3, 3)<update>**	(a, r, 2, 1) **(f, r, 8, 3)<update>**	
11	**Queue**	*Empty*	*Empty*	(g, r, 4) (f, r, 8) (c, r, 4) (f, r, 7) (c, r, 3) (f, r, 8)
	OSW	(o, p, 0) (b, r, 1) (a, q, 1) (a, r, 2)	(d, q, 2) (e, q, 5) (g, r, 7) ***(e, r,11)<update>***	(c, q, 2) ***(f, q, 5)<update>*** *(f, r, 9)*
	Log	(d, q, 2, 2) (c, q, 2, 3) (f, q, 6, 3) (f, r, 7, 3) (c, r, 3, 3)	(a, r, 2, 1) (f, r, 8, 3)	
12	**Queue**	*Empty*	*(g, r, 4)*	*Empty*
	OSW	(o, p, 0) (b, r, 1) (a, q, 1) (a, r, 2)	(d, q, 2) (e, q, 5) (g, r, 7) (e, r,11)	(c, q, 2) (f, q, 5) ***(f, r, 7)<update>*** ***(c, r, 3)<update>***
	Log	(d, q, 2, 2) (c, q, 2, 3) (f, q, 6, 3) (f, r, 7, 3) (c, r, 3, 3)	(a, r, 2, 1) (f, r, 8, 3)	(g, r, 4, 2)
13	**Queue**	*Empty*	(e, r, 8) (f, r, 5)	*Empty*
	OSW	(o, p, 0) (b, r, 1) (a, q, 1) (a, r, 2)	(d, q, 2) (e, q, 5) ***(g, r, 4)<update>*** (e, r,11)	(c, q, 2) (f, q, 5) (f, r, 7) (c, r, 3)
	Log	(d, q, 2, 2) (c, q, 2, 3) (f, q, 6, 3) (f, r, 7, 3) (c, r, 3, 3)	(a, r, 2, 1) (f, r, 8, 3)→(f, r, 5, 3)	(g, r, 4, 2)
14	**Queue**	*Empty*	*Empty*	*(f, r, 5)*
	OSW	(o, p, 0) (b, r, 1) (a, q, 1) (a, r, 2)	(d, q, 2) (e, q, 5) (g, r, 4) ***(e, r, 8)<update>***	(c, q, 2) (f, q, 5) (f, r, 7) (c, r, 3)
	Log	(d, q, 2, 2) (c, q, 2, 3) (f, q, 6, 3) (f, r, 7, 3) (c, r, 3, 3)	(a, r, 2, 1) **(f, r, 5, 3)<update>**	(g, r, 4, 2)
15	**Queue**	*Empty*	*Empty*	*Empty*
	OSW	(o, p, 0) (b, r, 1) (a, q, 1) (a, r, 2)	(d, q, 2) (e, q, 5) (g, r, 4) (e, r, 8)	(c, q, 2) (f, q, 5) ***(f, r, 5)<update>*** (c, r, 3)
	Log	(d, q, 2, 2) (c, q, 2, 3) (f, q, 6, 3) (f, r, 7, 3) (c, r, 3, 3)	(a, r, 2, 1) (f, r, 5, 3)	(g, r, 4, 2)

Figure 4.5: A sequence of snapshots of Algorithm 1 for three machines working in parallel - Part (2)

weight OSW table and a processing queue Q, M does not need a message log table (Log), since the whole process will run locally and there will not be any message communication.

We consider a query automaton $\mathcal{A}$ and a graph database DB as shown in Figure 4.1.

Initially, the OSW table and processing queue of machine M are both empty. The algorithm proceeds by initially creating the object-state-weight triple $(o, p, 0)$ where o is the DB start object, and p is the initial state of $\mathcal{A}$. Triple $(o, p, 0)$ is inserted in the processing queue. When dequeued, triple $(o, p, 0)$ is checked to see whether it is a useful triple, which means it might cause further OSW insertions or updates. Obviously $(o, p, 0)$ is useful because table OSW is empty, and so, there is no triple having the same object-state key (o, p). Triple $(o, p, 0)$ is inserted into OSW, and then it will be checked to see if it is expandable. In Figure 4.1, we can easily find that the following edge-transition matches:

1. the $(R, 2)$-edge from o to d in DB and the R-transition from p to q in $\mathcal{A}$,
2. the $(H, 1)$-edge from o to b in DB and the H-transition from p to r in $\mathcal{A}$, and
3. the $(R, 1)$-edge from o to a in DB and the R-transition from p to q in $\mathcal{A}$.

There are three triples which are obtained by expanding $(o, p, 0)$, namely $(d, q, 2)$, $(b, r, 1)$, and $(a, q, 1)$ which are inserted into the processing queue. When dequeuing, let us assume that $(d, q, 2)$ is removed from the queue[1], and it is checked if it is useful and further expandable. If it is useful and expandable, then it will be expanded some other triples will be obtained which will be inserted into the processing queue.

[1]See Chapter 5 for a discussion of queue policies that we use.

Tuple $(d, q, 2)$ is in fact inserted into *OSW* and further expanded. A sequence of steps of tracing Algorithm 1 are shown in Figure 4.3. Now, we are ready to illustrate Algorithm 1 with multiple machines working in parallel. We assume that we have three machines and that the database is partitioned into three parts as shown in Figure 4.2. Now, at some point, a new triple (p, q, w) is created by following a cross-boundary database edge, so (p, q, w) is not a local triple and it must be packed and then sent to the corresponding machine. Also (p, q, w) is inserted into the *Log* table of the local machine. For the example in Figure 4.3, consider time point 3. We dequeue triple $(d, q, 2)$. Because node d resides in M_2 (rather than in M_1), triple $(d, q, 2)$ must be packed and sent to M_2. Also a quadruple $(d, q, 2, 2)$ is inserted into the *Log* table of M_1. A sequence of steps of tracing Algorithm 1 for three machines working in parallel is shown in Figure 4.4 and Figure 4.5.

Due to the limited space, we could not list every single time point of tracing the algorithm 1 for three machines working in parallel. For example, from step 7 to time step 8, we assumed that the M_1 packed the two triples $(c, q, 2)$ and $(f, q, 6)$ in the processing queue together, and sent them to M_3 once. Another example is shown from step 11 to step 12, where more than one updates or ignorings happening in the *OSW* table while dequeuing. We just show the final updates happened in the *OSW* for saving spaces. However, such assumption and step saves won't impact the whole process of distributed evaluation.

4.2 Machine Loss and Termination Detection

Now, we turn our attention to the machine loss and termination detection component of our algorithm. For this, we adapt the Dijkstra-Scholten termination detection

algorithm ([6]). The original DS algorithm assumes that the machines are alive through all the computation, which is an assumption we cannot make in a realistic grid setting.

The idea of [6] is to organize the active machines, i.e., those currently processing the query, in a spanning tree rooted at the query originator, which, by definition starts as an active machine. We assume a (previously passive) machine joins the tree upon the receipt of its first message/task from an active machine which becomes its *parent*. Active machines send messages/tasks to other machines which, in turn, acknowledge them back as appropriate. Each active machine uses a local variable *tasks* to keep count of its unacknowledged messages/tasks. Messages are, in general, acknowledged immediately unless they are the "engaging messages," i.e. messages that result in passive machines becoming active. A non-originating active machine can become passive, and attain "local termination" if its local processing queue is empty and it has no unacknowledged messages. At that time, the respective machine acknowledges its parent and severs its connection from the spanning tree. The processing of the query ends when the originating machine has no unacknowledged messages left.

We extend the above procedure to account for machine losses by monitoring each node of the Dijkstra-Scholten spanning tree except for its root since we make the practical assumption that M_0, the originating machine, does not fault during the search – otherwise the user can restart the computation. In practice, it is natural for M_0 to be the machine of the user who gives the query, and thus, it makes sense to assume that at least M_0 does not leave before the full computation of query answers.

We also assume that once sent, messages are guaranteed to be delivered, although, perhaps with some delay. Furthermore, messages exchanged between any two pro-

cessors are guaranteed to be delivered in the order they are sent.

At any time, each node in the spanning tree is responsible for monitoring the health of its parent. In turn, the parent maintains the records necessary to ensure termination detection. For completeness, each leaf node is monitored by a "dummy offspring leaf." We will let the root machine M_0 to play this (additional) role!

All monitoring is done using a loss detection service which can be as simple as a ping command and which reports to a child the demise of its parent.

As machines fault, the termination spanning tree needs to be rebuilt on the fly by the remaining machines involved in the search. As such two issues need to be resolved: live spanning tree orphans need to acquire new parents and all nodes need to readjust their termination bookkeeping in order to account for faulty machines. Upon detecting its parent loss, a non-root node makes, as the new parent, the closest alive ancestor in the spanning tree. A node determines its new parent by traversing its path to the root, a piece of information that is given to each node by its (old)parent upon "engagement."

This traversal is also used by the live nodes to adjust the termination bookkeeping: the new parent, as determined above, erases any bookkeeping associated with its offspring on this particular path.

Monitoring by the originating machine M_0, in its other role as a dummy offspring of the leaves, relies on other nodes to communicate changes in their leaf-status, i.e. when they change from "passive" to "active" and vice versa. This information is then used to maintain the list of the monitored leaves.

The details of the algorithm are as follows:

Algorithm 2 (Machine loss and termination detection)

Each non-originating machine M_i initializes the local variables $parent_i = null$ and $pathToRoot_i = \emptyset$.

The originating (root) machine M_0, in order to serve its (other) role as a dummy offspring of the leaves, initializes a list of leaves, $L = \emptyset$, and a map $pathToRootMap = \emptyset$. The elements of $pathToRootMap$ (which is implemented in practice as a hash table) will be keyed by machine ids. For example if there exists an element keyed by machine id i, then it will be the sequence of machine ids starting with i and continuing with the ids of its ancestors (in order) all the way to the root. Such an element (sequence of machine ids) is denoted with $pathToRootMap[i]$, and will exist only if machine M_i becomes a leaf in the termination detection tree.

Repeat steps 1–14 in *parallel*, until global termination is detected.

At each non-root machine M_i:

1. When a basic message $\langle a, p, r \rangle$ is about to be sent (in step 3 of Algorithm 1) to machine M_j

   (a) create and initialize a variable $tasks_i^j = 0$ if it did not exist before

   (b) **if** $tasks_i^j = 0$ (*i.e.* $\langle a, p, r \rangle$ is a potentially engagement message for M_j)

   then send message $\langle a, p, r, i + pathToRoot_i \rangle$ to M_j

2. Upon receipt of a message $\langle a, p, r, path \rangle$ from some other machine M_k

   if $i \neq 0$ and $parent_i = null$ (i.e. M_i is passive)

   then (M_i becomes active and joins the termination detection tree as a leaf)

   (a) set $parent_i = k$ and $pathToRoot = path$

   (b) start, with respect to M_k, a loss detection service S_i^k.

(c) send a monitoring requesting message $\langle active, k, path\rangle$ to M_0

(This is done because M_i is a leaf now, and root M_0 also serves the (other) role as a dummy offspring of the leaves.)

(d) send a confirmation message to parent M_k

else send acknowledgment *ack* to machine M_k

3. Upon the receipt of a confirmation message from some machine M_j set $tasks_i^j = tasks_i^j + 1$.

4. Upon receipt of an acknowledgment message from some machine M_j, set $tasks_i^j = tasks_i^j - 1$.

5. Upon receipt of a message notifying the loss of machine M_k:

   (a) scan $pathToRoot_i$ for the first live ancestor l (and its dead offspring q) and send $\langle M_q, \text{gone}\rangle$ to M_l.

   (b) Furthermore, change the parent: set $parent_i = l$ and send a *newOffspring* message to machine M_l.

6. Upon receipt of a *newOffspring* message from machine M_j set $task_i^j = task_i^j + 1$ if such variable exists, or otherwise create and initialize $task_i^j = 1$.

7. Upon receipt of a $\langle M_q, \text{gone}\rangle$ message, delete $task_i^q$ if such variable exists.

8. **If** the local processing queue is empty and $\forall j\ tasks_i^j = 0$ **then** send
   $\langle passive, parent_i, tail(pathToRoot)\rangle$ to M_0,
   send acknowledgment to $parent_i$,
   set $parent_i = null$ and
   declare local termination (become passive).

At root machine M_0:

9. When a basic message $\langle a, p, r \rangle$ is about to be sent (in step 3 of Algorithm 1) to machine M_i

   (a) create and initialize a variable $tasks_0^i = 0$ if it did not exist before

   (b) **if** $tasks_0^i = 0$ ($i.e.$ $\langle a, p, r \rangle$ is a potentially engagement message for M_i)

   then send message $\langle a, p, r, \emptyset \rangle$ to M_i

10. Upon the receipt of a confirmation message from some machine M_i set $tasks_0^i = tasks_0^i + 1$.

11. Upon receipt of an acknowledgment message from some machine M_i, set $tasks_0^i = tasks_0^i - 1$.

12. Upon receipt of a message $\langle active, k, path \rangle$ from M_i (see Step 2), start monitoring of M_i and stop monitoring of M_k (if appropriate) as follows:

    (a) set $L = L \cup \{i\}$, $pathToRootMap[i] = path$ and start S_0^i (a loss detection service for M_i).

    (b) **if** $k \in L$ **then** (now M_k is not anymore a leaf, so we delete the information about it)

    set $L = L - \{k\}$,

    remove $pathToRootMap[k]$, and

    stop S_0^k (the loss detection service for M_k).

13. Upon receipt of a message $\langle passive, k, path \rangle$ from machine M_i (which finished computation, see Step 8) do:

    (a) $L = L - \{i\}$,

    remove $pathToRootMap[i]$, and

    stop S_0^i (the loss detection service for M_i)

(b) **if** M_0 monitors no other offspring of M_k

(i.e. the parent of M_i, which is M_k, has become a leaf)

then set $L = L \cup \{k\}$,

$pathToRootMap[k] = path$ and

start S_r^k (a loss detection service for M_k).

14. If the local processing queue is empty and $\forall j\ tasks_i^j = 0$, then query was answered and global termination is declared.

Now, we illustrate Algorithm 2 by the following examples. Assume that there are five participating machines M_0, M_1, ..., M_4 in our system. Algorithm 2 turns to solve the problems of (1) how to construct a termination spanning tree when a query is originated by the root machine M_0; (2) how to rebuild the spanning tree when one vertex of the graph is lost; and (3) how to gradually remove the spanning tree as the query is answered.

First, we give an example for the construction of the termination detection spanning tree.

1. Suppose M_0 is the root machine which starts a query by inserting the triple $\langle o, p_0, 0\rangle$ in its processing queue. Initially, both L and $pathToRootMap$ are empty. All other four machines are passive, and $parent_i = null$ and $pathToRoot_i = \emptyset$, for $i = 1 \ldots 4$. Suppose that M_0 continues to work a while, and at some point comes across a boundary database edge connecting with the database partition stored at machine M_1. M_1 creates and initializes a variable $task_0^1 = 0$. Then, M_0 sends an engaging message $\langle o, p_0, 0, \emptyset\rangle$, where $\emptyset$ indicates the sender is the root machine, to M_1 with the latter becoming subsequently active.

2. Upon becoming active, M_1 joins the termination detection spanning tree as a leaf and its two variables are updated (i.e., $parent_1 = 0$; $pathToRoot_1 = \emptyset$). Then, M_1 starts a loss detection service S_1^0. Since M_1 becomes a leaf and root M_0 needs to monitor the health of leaves, M_1 sends a message of $\langle active, 0, \emptyset \rangle$ to root M_0. Also M_1 sends a confirmation message to M_0.

3. Upon receiving the monitor-requesting message from M_1, machine M_0 adds M_1 to its list of leaves and updates $pathToRootMap[1]$ to be the $pathToRoot$ received from M_1. Thus, now M_0 has $L = 1$ and $pathToRootMap[1] = \emptyset$. Also, M_0 starts the loss detection service S_0^1. Currently, the termination spanning tree has two vertices M_0 and M_1. Notice that during the query processing, M_0 and M_1 can send query processing triples $\langle a, p, r \rangle$ to each other. However, such messages are not engaging messages and they will not impact the structure of the termination detection spanning tree that is constructed so far except for initializing the variable $task_1^0$ if it did not exit before.

4. Suppose that M_1 dequeues an object-state-weight triple $\langle a_2, p_2, r_2 \rangle$ and a_2 is stored in M_2. In this case, M_1 sends an engaging message $\langle a_2, p_2, r_2, (1) \rangle$ to M_2, in which (1) indicates the $pathToRoot$ of M_1. Since this is an engaging message, M_1 sets $task_1^2 = 0$.

5. Upon receiving the message $\langle a_2, p_2, r_2, (1) \rangle$, machine M_2 sets its parent to be M_1, updates its $pathToRoot$ variable (i.e., $parent_2 = 1$; $pathToRoot_2 = 1$), and starts its loss detection service S_2^1 with respect to M_1. Since M_2 is a new leaf, in order to make the root M_0 to monitor M_2, M_2 sends a message $\langle active, 1, (1) \rangle$ to M_0, in which the first 1 indicates its parent M_1 and the second (1) indicates

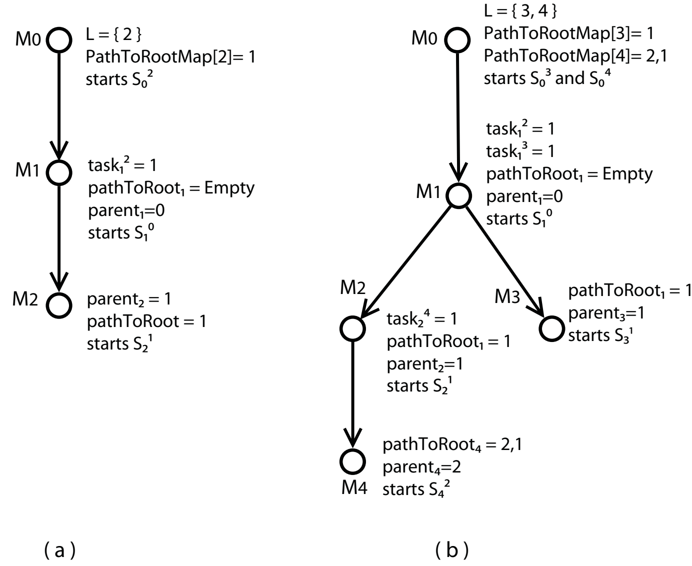

Figure 4.6: Termination spanning trees

its *pathToRoot*. Finally, M_2 also sends a confirmation message to M_1.

6. Upon receiving the confirmation message from M_2, machine M_1 increases its variable $task_1^2$ by 1.

   Upon receiving the monitor-requesting message $\langle active, 1, 1\rangle$ from M_2, machine M_0 deletes M_1 from its list of leaves and records M_2 as a leaf. M_0 also updates its *pathToRootMap* entry corresponding to M_2 and removes the other one corresponding to M_1 (i.e., $L = \{2\}$; $pathToRootMap[1] = \emptyset$, $pathToRootMap[2] = 1$). The termination spanning tree becomes as shown in Figure 4.6 (a).

7. Moreover, suppose that eventually M_2 sends to M_4 and M_1 sends to M_3 engaging messages. Finally the termination spanning tree would be as shown in Figure 4.6 (b).

It is interesting to note that the termination detection tree is essential in prop-

agating the messages about loss of machines, which are helpful in Algorithm 1 for redoing the lost computation on mirroring machines (if such machines exist). Next, we illustrate Algorithm 2 by an example to show the termination tree rebuilding while one machine is lost.

1. Consider the spanning tree in Figure 4.6 (b). Suppose that M_2 is lost during the query processing. Upon receiving a message notifying the loss of machine M_2 by the service S_4^2, M_4 scans its variable $pathToRoot_4 = \{2, 1\}$ for the first live ancestor, which is M_1. Then M_4 sends a message $\langle M_2, gone \rangle$ to M_1. Moreover, M_4 changes its ancestor to M_1 and sends a *newOffspring* message to machine M_1.

2. Upon receiving the $\langle M_2, gone \rangle$ message from M_4, M_1 deletes the variable $task_1^2$.

3. Upon receiving the *newOffspring* message from M_4, M_1 creates a variable $task_1^4$ and initializes it to 1. Finally, a new termination spanning tree is shown in Figure 4.7 (a) corresponding to the loss of M_2.

Now, we turn to solve the problem of how to detect the "global termination" of query processing and gradually remove the spanning tree when the "local termination" is detected by an active machine. Consider the termination detection spanning tree shown in Figure 4.7 (b).

1. M_4 is a leaf node in the tree, so $\forall j \ tasks_4^j = 0$. Suppose the local processing queue of M_4 is empty. M_4 becomes passive and declares local termination status. M_4 sends a message of $\langle passive, 2, 1 \rangle$ to root M_0, where 2 indicates that the parent of M_4 is M_2 and 1 indicates the tail of $pathToRoot_4$ of M_4. M_4 also sends an acknowledgement message to its parent M_2 and sets $parent_4 = null$.

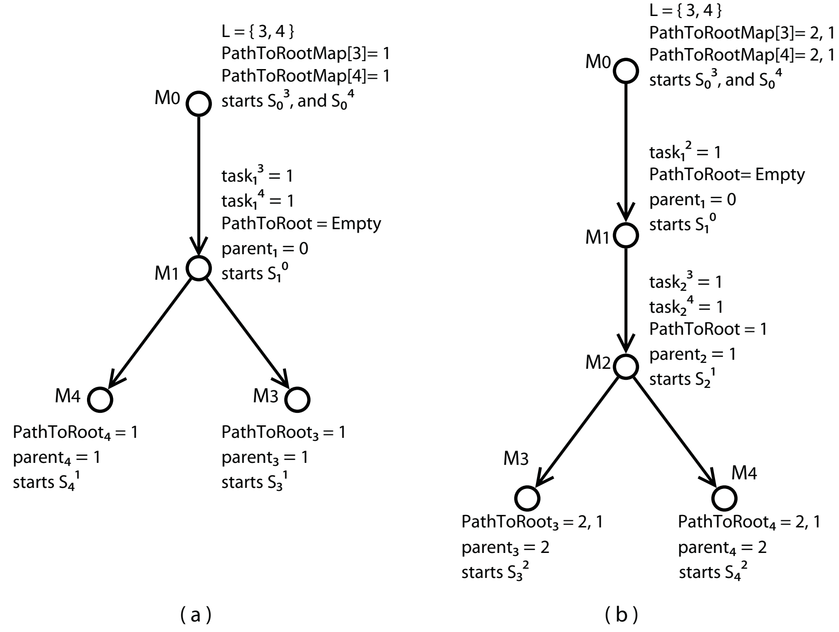

Figure 4.7: Rebuilt termination spanning trees

2. Upon receiving the acknowledgement from M_4, M_2 knows that M_4 declared its "local termination" status, and so, M_2 decrements its $task_2^4$ variable by 1 (i.e., $task_2^4 = 0$).

3. Upon receiving the message of $\langle passive, 2, 1\rangle$ from M_4, M_0 knows the "termination status" of M_4. M_0 removes M_4 from its list of leaves and stops the S_0^4 service. M_0 also detects that M_4 does not have any other child, so it adds M_2 to its list of leaves (i.e., $L = \{2, 3\}$; $pathToRootMap[2] = 1$) and starts a service S_0^2. Now M_4 is removed from the spanning tree. M_2 and M_3 are leaves in the spanning tree, so the termination would be only detected if the processing queues of M_2 and M_3 are empty.

4. Furthermore, suppose that the processing queue of M_3 becomes empty, and M_3

declares local termination status. It sends an acknowledgement to its parent and a *passive* message to the root. This results in M_3 being removed from the spanning tree. Now the spanning tree changes to the one shown in the Figure 4.6 (a). Finally, M_2, M_1 and M_0 terminate in turn, and the global termination is declared. The termination detection exactly follows the reverse path of which machines are engaged and spanning tree is constructed.

4.3 Generating Paths

Here, we would also like to note that it is easy to modify our algorithm to produce as well the "best" paths that were used to finally obtain the optimal weights of query answers. For this, we can augment each object-state-weight triple (a, p, r) with a "back-pointer," which points to the state-weight entry (in some OSW table) corresponding to the "previous triple," which was used by the algorithm to obtain the triple's current weight r. Such a back-pointer could simply be the object-state[2] pair of the previous triple. The back-pointers are recorded in the corresponding entries of the OSW tables. It is easy to see that the back-pointers enable a distributed backward computation of the best paths. Starting from the OSW entries, which correspond to the query answers, we follow the back-pointers. When a back-pointer points to a non-local entry, the partial path computed so far is packed into a message and sent to the corresponding processor. We continue this procedure till we reach the initial $(o, p_0, _)$ entry in OSW_0.

[2]Recall that we work in fact with the id's of the objects rather than with the objects themselves.

Chapter 5

Queue Strategies

Our basic algorithm running at each processor is in fact an extension of one-to-all label-correcting shortest path algorithms. In such algorithms, the "labels" refer to the numeric value recording the length of the shortest paths (discovered so far) to the nodes of the graph.[1]

The label-correcting (or better phrased "weight-correcting") algorithms maintain a processing queue, similar to our processing queues, into which the candidate nodes for update are inserted. Notably, there has been an extensive research on finding the best strategy of inserting and removing from the queue. For the single processor case, by using a priority queue, we obtain the Dijkstra's classical algorithm. In this case, each node will enter and exit the processing queue exactly once. For this reason, Dijkstra's algorithm is not in fact a label (weight) correcting method but rather a label (weight) setting one. In contrast, the label correcting methods avoid the overhead associated with a priority queue, with the tradeoff of more processing queue node insertions.

[1] The numeric labels in label-correcting algorithms should not be confused with the symbol labels of the database edges.

The simplest label correcting method, the Bellman-Ford method, uses a FIFO processing queue; nodes are removed from the top of the queue and are inserted at the bottom. More sophisticated label correcting methods maintain the processing queue in one or in two queues and use a more complex removal and insertion strategies. The objective is to reduce the number of node re-entries in the processing queue. The general principle behind the rationale of each algorithm is that the number of node re-entries is reduced if nodes with relatively small weight are removed first from the processing queue.

The most well known queue strategy for label correcting algorithms is the Smallest-Labels-First-Large-Labels-Last (SLF-LLL) strategy of [3]. In this strategy, the processing queue is maintained as a double ended list. At each iteration, the node removed is the top node of the list. However, when the top node has a larger weight than the average node weight in the queue (defined as the sum of weights of nodes in the queue divided by the cardinality of the queue), this node is not removed but rather it is repositioned to the bottom of the queue. Regarding the insertion of a new node in the processing queue, we compare first its weight with the weight of the node at top of the list. If the weight of the new node is smaller, then it is inserted at the top of the list. Otherwise it is inserted at the bottom of the list.

What Bertsekas et. al. observed experimentally, is that in the single processor case, shortest path algorithms which employ the SLF-LLL method are faster than the classic Dijkstra's algorithm. This was argued to happen due to the computational overhead for maintaining the priority queue in Dijkstra's algorithm.

Moreover, label-correcting algorithms have been the main class of algorithms that have been successfully parallelized, achieving an impressive speed-up in computation.

Parallelization of these algorithms using a shared memory approach is presented in [3]. Notably, in their parallel approach, Bertsekas et. al. obtain even better results when using SLF-LLL queues than when using priority queues.

We note here, that although using a priority queue in the single processor case gives us the Dijsktra's algorithm, which is in fact a label setting algorithm, in the multiprocessor case, even if we use local priority queues at each processor, we still have a label correcting algorithm. This is because the weight of a node might need to be updated due to paths that "cross over" to different processors during the evaluation of the query.

We want to stress here that experiments of Bertsekas et. al. were designed for graphs stored in main memory and not in secondary storage, as in our setting. Furthermore, we are not solving an unrestricted shortest path problem, but rather one guided by a query automaton. Consequently, Bertsekas' observations needed to be checked anew. We discuss these and other performance issues next.

Chapter 6

Data Partitioning

6.1 Background

In databases, indexing is a technique to locate, access, and possibly organize data within a database. In general, there are two kinds of indexes: clustering and non-clustering. A clustering index is a type of index that is built on the same key by which the records are physically ordered on disk. Viewed from a different angle, a clustering index "dictates" the placement of the data, and it is often called a "data organizing index". On the other hand, a non-clustering index is a type of index that is built on any key in which the records can be logically ordered. Clearly, there can be at most one clustering index on a given set of data, while there can many non-clustering indexes on the same set of data.

Indexes can be implemented using a variety of data structures. The most common case is B-Tree and it is widely utilized by many databases such as Oracle, MySQL, Microsoft SQL Server, etc. Each node in a B-Tree is of one disk block size. If a B-Tree is used for a clustering index, then the leaves of the tree contain the data records themselves (rather than pointers to data records); hence the "data organizing"

naming for such indexes.

It is important to keep in mind that by using B-Trees for implementing clustering indexes, we only achieve organizing the data with respect to one attribute only. On the other hand, in spatial network databases we have to typically deal with two dimensional objects. If we use B-Trees for clustering two dimensional data, then we get a line ordering of data, which is north-south or east-west depending on the coordinate we choose as the key for the clustering index. Using the leaves of such B-Tree for partitioning the data is not good for evaluating RPQs. This is because the evaluation of RPQs can navigate the spatial network in any direction, i.e. not only in a north-south or east-west order. Hence, B-Tree clustering and partitioning is not appropriate for spatial networks, as the query evaluation would have to "jump" back-and-forth between partitions many times. In order to deal with this problem, we cluster and partition the spatial data by using clustering R-Trees.

6.2 R-Tree Partitioning

6.2.1 R-Trees

R-Trees and their variants are popular approaches for accessing spatial data due to their efficiency and simplicity. R-Trees can be viewed as a multi-dimensional extension of B-Trees.

An R-Tree is a tree indexing structure that divides the space into hierarchically nested and possibly overlapping minimum bounding rectangles (MBR). Each leaf node in a *clustering* R-Tree contains 2-dimensional data entries[1]. Each non-leaf node contains entries of the form: $(I, ChildPointer)$, where $ChildPointer$ is the address

[1] In fact an R-Tree can be used for n-dimensional data.

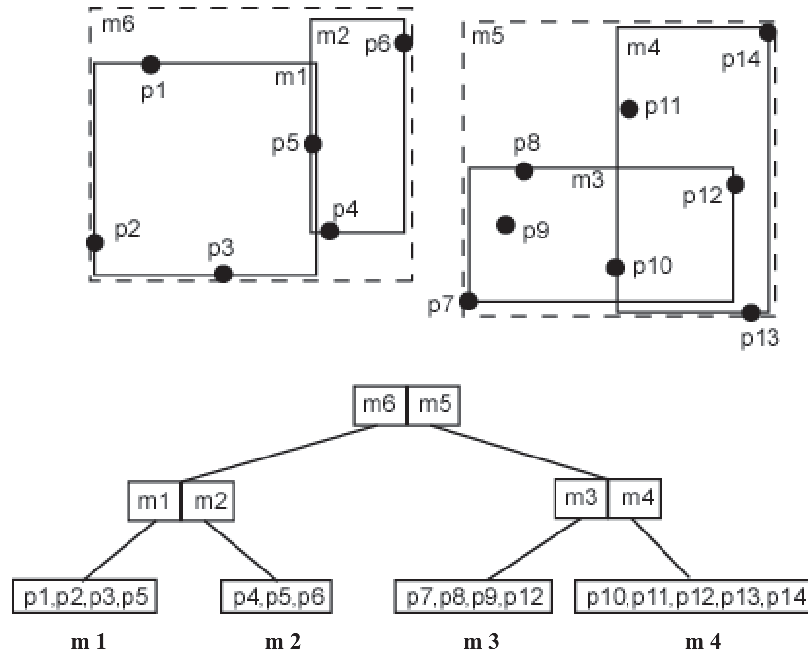

Figure 6.1: An example of R-Tree indexing

of a lower node in the R-Tree index and I is an MBR generated by (geometrically) enclosing the objects in the lower node entries.

Figure 6.1 shows an example of an R-Tree for a set of objects (points) $\{p_1, p_2, p_3, \ldots, p_{14}\}$. The points in a node are grouped based on their locations: for example, p_4, p_5 and p_6 are clustered in the same leaf node m_2. Nodes are then recursively grouped together following the same principle until reaching to the root node.

In reality, the size of each node is, as in the B-Tree case, equal to one disk block. However, differently from a clustering B-Tree, a clustering R-Tree groups (into blocks) spatial objects that are close to each other with respect to their distance, rather than to their closeness regarding one coordinate only.

6.2.2 Partitioning

We spatially cluster the database into disk blocks by using a clustering R-Tree index. Then, in a round-robin fashion, we assign each block to a participating machine. The preference of this partitioning method over a hierarchical one is motivated by the grid setting for the query evaluation. If we assign machines big partitions of contiguous areas, then machines work for long continuous intervals under heavy stress. On the other hand, if we assign machines small partitions of contiguous areas, such as block-sized leaves of a clustering R-Tree index, then machines alternate shorter work intervals with idle intervals. This method is desirable in a grid setting, where machines(servers) allow only a limited quantum of running time for each served task. Furthermore, as described in the results section, this partitioning approach pays a negligible price in terms of increased volume of communication.

Chapter 7

Experiments

We conducted extensive experiments in a large GIS road network, provided by US Census Bureau ([17]). Namely, we chose the New York city map, which is one of the most dense areas, with approximately 435,000 edges (roads, highways, etc). We stored the map partitions in edge organized tables, whose storage is structured using a clustering R-Tree index (see Chapter 4), which is available in MySQL 5.0.

We ran our experiments on a network of Xeon 3.4 GHz machines, running Fedora Core Linux, Java 1.5, MySQL 5.0, and MySQL Connector/J 5.0, connected via a Gigabit Ethernet.

Initial experiments used a database partitioning based on the original TIGER point organization([17]) which orders points in a line-based order from north to south. It was soon clear that this organization was causing significant number of disk accesses and, thus, slowdowns due to its one-dimensional spatial locality. We then switched to an R-Tree clustering based partition which halved the execution time. This partitioning method was used in all subsequent experiments as it seems to be quite appropriate for grid environments in which, at any time, computations

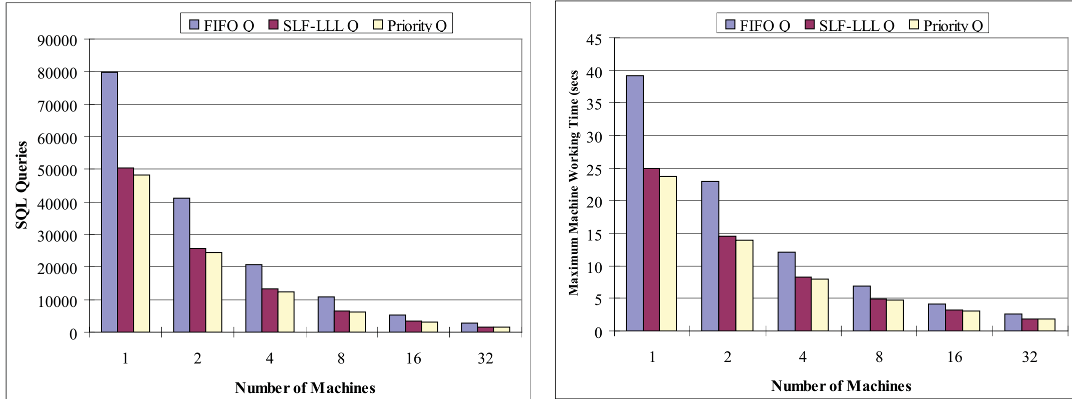

Figure 7.1: Experimental result 1 (Computational stress).

are provided with large disk space, but limited memory footprint and execution time.

We show here results for a typical query, which is $highway^*||(\text{road} + \epsilon)^k$, where $||$ is the shuffle operator. This query asks for finding the objects reachable by following highways interleaved by no more than k roads. The results we show here are for $k = 10$. We want to mention here, that for other queries, we got results that were similar to the ones that we show. We experimented with 2, 4, 8, 16, and 32 machines. Let us discuss our results shown in Figure 7.1, Figure 7.2 and Figure 7.3. All the results are given with respect to the number of participating machines.

The two graphs in Figure 7.1 are similar in shape. The left graph shows the maximum number (among machines) of SQL queries executed (for fetching database edges), while the right graph shows the maximum working time (in seconds) of machines. These two graphs are good indicators of the stress reduction on participating machines as the number of machines increases. Namely, we observe reduction of stress in (almost) half as the number of machines doubles at each point. Also, we

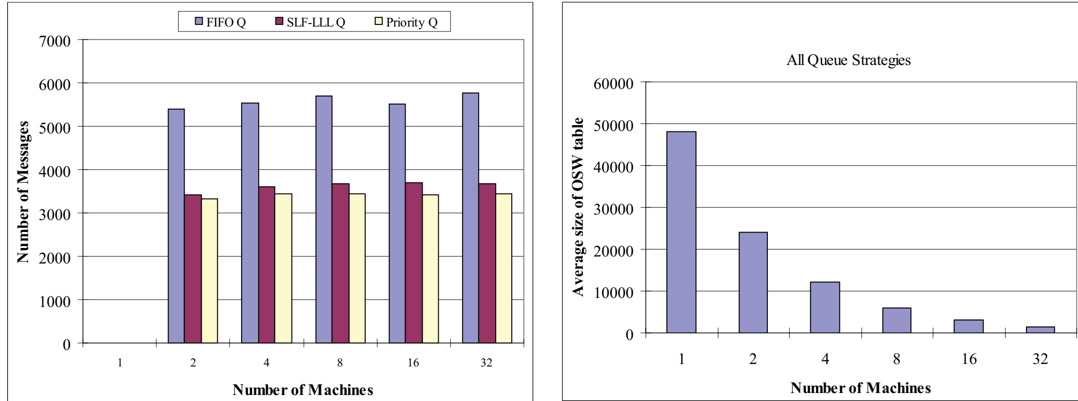

Figure 7.2: Experimental result 2 (Communication messages)

observe that the use of SLF-LLL or just FIFO queues, in the name of reducing the computational overhead of maintaining a priority queue, is in fact not justified. This is also amplified by the left graph in Figure 7.3, which shows that the number of total updates in OSW tables is minimal when using a priority queue. Recall (from steps 3a and 3b of Algorithm 1) that the number of updates translates directly to the quality of (intermediate) query answers that the user receives while the computation is still going on. Clearly, less updates means that there are less inconsistent query answers that get their weight corrected. By considering the number of updates, we conclude that when machines use priority queues, the quality of answers is twice better than when using SLF-LLL queues, and an order of magnitude better than when using FIFO queues.

The left graph in Figure 7.2 shows the total number of messages sent during the query evaluation. Clearly, when the machines use priority queues the number of messages is smaller. However, more important is the observation that our algorithm is

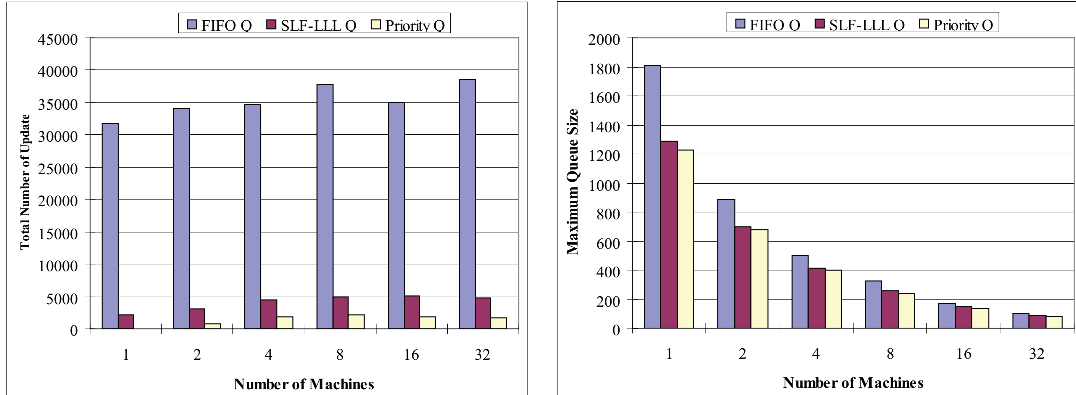

Figure 7.3: Experimental result 3 (OSW updates and Queue size)

not message intensive. Notably, when using priority queues, the number of messages is approximately 3500 and this is quite negligible for today's high speed networks. In general, the communication is quite balanced as there is little variance between the number of messages employed by different machines. Also, we remark that our algorithm is scalable as the number of messages grows very slowly with the number of machines. Finally, in the two graphs of Figure 7.3, we show the maximum size (number of triples) of processing queues and OSW tables (respectively) among participating machines. [The size of OSW tables does not depend on the queue being used] The sizes reduce in almost half as the number of machines doubles. These structures fit very well in main memory especially as the number of machines becomes larger.

Chapter 8

Conclusions

We have identified the major problems faced in a grid-aware evaluation of regular path queries on spatial network databases. We have provided a complete distributed solution, which reduces the computational stress proportionally to the number of participating grid machines. Also, our solution is resilient with respect to machine losses and termination detection, which are common phenomena in a grid setting. Experimental evidence shows that our algorithm, under normal conditions, is *not* message intensive, with the total number of messages being negligible for today's networks. Finally, experimental evidence shows that our grid-aware algorithm provides an on-line evaluation performance for the notoriously hard regular path queries.

Bibliography

[1] Abiteboul, S., P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, San Francisco, CA, 1999.

[2] Abiteboul, S., and V. Vianu. Regular Path Queries with Constraints. *JCSS* 58(3) 1999, pp. 428-452.

[3] Bertsekas D. P., F. Guerriero, and R. Musmanno Parallel asynchronous label-correcting methods for shortest paths *J. Opt. Theory and App.* 88 (2), 1996, pp. 297–320.

[4] Bleiholder J., S. Khuller, F. Naumann, L. Raschid, and Y. Wu. Query Planning in the Presence of Overlapping Sources *EDBT'06.*

[5] Calvanese, D., G. Giacomo, M. Lenzerini, and M. Y. Vardi. Answering Regular Path Queries Using Views. *ICDE'00.*

[6] Dijkstra, W., and C. S. Scholten. Termination Detection for Diffusing Computations. *Inf. Proc. Let.*, 11-1 (26), 1980, pp. 1–4.

[7] Gallo, G., and S. Pallottino. Shortest path methods in transportation models. In: (M. Florian, ed.) *Transportation Planning Models.* Elsevier, North-Holland, 1984.

[8] Grahne G., and A. Thomo. Query Containment and Rewriting Using Views for Regular Path Queries under Constraints *PODS'03*.

[9] Guting, R., H. GraphDB: Modeling and Querying Graphs in Databases. *VLDB'94*.

[10] Hribar, M., R., Taylor, V., E., and Boyce, D., E. Implementing parallel shortest path for parallel transportation applications. Parallel Computing, 27 (12), 2001, pp. 1537–1568.

[11] Kaushik R., P. Shenoy, P. Bohannon, and E. Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. *ICDE'02*.

[12] Miao Z., D. Stefanescu, A. Thomo. Grid-Aware Evaluation of Regular Path Queries on Spatial Networks. *Proc. of AINA'07*.

[13] MySQL Documentation, *http://www.mysql.com*

[14] Stefanescu D. C., A. Thomo, and L. Thomo. Distributed evaluation of generalized path queries *SAC'05*.

[15] Stefanescu D. C., A. Thomo. Enhanced Regular Path Queries on Semistructured Databases *QLQP'05*.

[16] Suciu D., Distributed query evaluation on semistructured data. ACM TODS 27(1), 2002, pp. 1–62.

[17] TIGER: Topologically Integrated Geographic Encoding and Referencing system, US Census Bureau *http://www.census.gov/geo/www/tiger*

[18] Vardi. M. Y. A Call to Regularity. *Proc. PCK50 - Principles of Computing & Knowledge, Paris C. Kanellakis Memorial Workshop '03.*

[19] Hopcroft J. E., and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.