Grid-Aware Evaluation of Regular Path Queries on Spatial Networks

Zhuo Miao
University of Victoria
zhuomiao@cs.uvic.ca

Dan Stefanescu
Suffolk University
dan@mcs.suffolk.edu

Alex Thomo
University of Victoria
thomo@cs.uvic.ca

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

1 Introduction

Motivation. Regular path queries (RPQs) are used for 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. 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

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

where $||$ is the shuffle operator, 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, 7, 9, 14, 10, 20] etc). This is not surprising as RPQs are the basic building block of almost all query languages for data-graphs, providing the user with a nice and simple way to express recursion (see for a discussion [20]). 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 [15] points out, RPQs are an integral part of an intelligent querying of spatial networks.

In this paper, 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 databases studied in the literature, is the fact that they can be conveniently modeled by *weighted* graphs. 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 (cf. for example [16]) because they ignore the cost of the database paths. The lack of index structures for this problem makes developing a grid-aware approach even more important if one is to have RPQs feasible for practical use.

Related Work. Regular path queries are by now part of the folklore (cf. [1, 8, 11, 12, 13, 15, 16, 19]). 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], [19], and [18]. In [2] and [19], the data-graphs are unweighed, and their algorithms do not seem generalizable to our case. Moreover, the algorithm of [19], 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 methodology described in [18] works on weighted regular path queries and, while its algorithm 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.

Finally, in [17], 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.

Our Approach. While the general idea of [18] 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 paper we tackle several challenges in turning the proposed method of [18] 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 [18]. 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 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 evaluation needs to continue to other partitions, the relevant information is packed into messages and sent to the corresponding machines. If an object a is associated with 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 paper.

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 paper 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 RTree (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 RTree leaves. After this RTree partitioning, each machine builds a local RTree 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 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 paper, 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.

The rest of the paper is organized as follows. In Section 2, we formally define spatial network databases, regular path queries (RPQs), and their semantics. In Section 3, we present our grid-aware distributed algorithm, which has two interwoven components: the RPQ evaluation and the resilient machine loss and termination detection. In Section 4, we thoroughly discuss queue strategies for optimizing our algorithm. In Section 5, we present our experimental results. Finally, Section 6 concludes the paper.

2 Spatial Network Databases and Regular Path Queries

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. 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 between RPQs and FSA's that represent them. Let us denote with $\mathit{length}(\pi)$ 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

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

As an example consider the database DB in Figure 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 $(c, 3) \in \mathit{WAns}(\mathcal{A}_Q, o, DB)$.

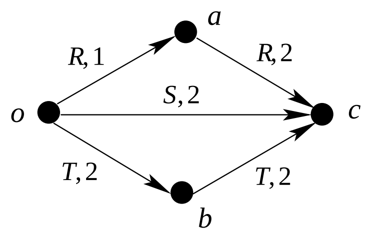

Figure 1. *An example database.*

3 Grid-Aware Distributed Evaluation of Path Queries

Before evaluating any query, we spatially cluster the database into disk blocks by using a clustering RTree 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 RTree 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.

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. OSW tables have 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 section.

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 tables 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 from object a and state p an edge 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.

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.
```

Observe that in step 3a of the above algorithm, we incrementally build $WAns(\mathcal{A}, o, DB)$ 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 Section 4.

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 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 Section 5).

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. We also assume that once sent, messages are guaranteed to be delivered, although, perhaps with some delay. Furthermore, messages exchanged between any two processors 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 nonroot 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 1–11 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. 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).
10. 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 $M_k$).
11. If the local processing queue is empty and $\forall j \; tasks_i^j = 0$,
    then query was answered and global termination is declared.
```

4 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.

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.

5 Experiments

We conducted extensive experiments in a large GIS road network, provided by US Census Bureau ([4]). Namely, we chose the New York city map, which is one of the most dense areas, with approximately 435000 edges (roads, highways, etc). We store the map partitions in edge organized tables, whose storage is structured using a clustering RTree index (see Section 3), which is available in MySQL 5.0.

We ran our experiments on a network of Pentium IV 2.4 Ghz machines, running Fedora Core Linux, Java 1.5, MySQL 5.0, and MySQL Connector/J 5.0, connected via a Cisco 1G switch.

Initial experiments used a database partitioning based on the original TIGER point organization([4])) 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

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

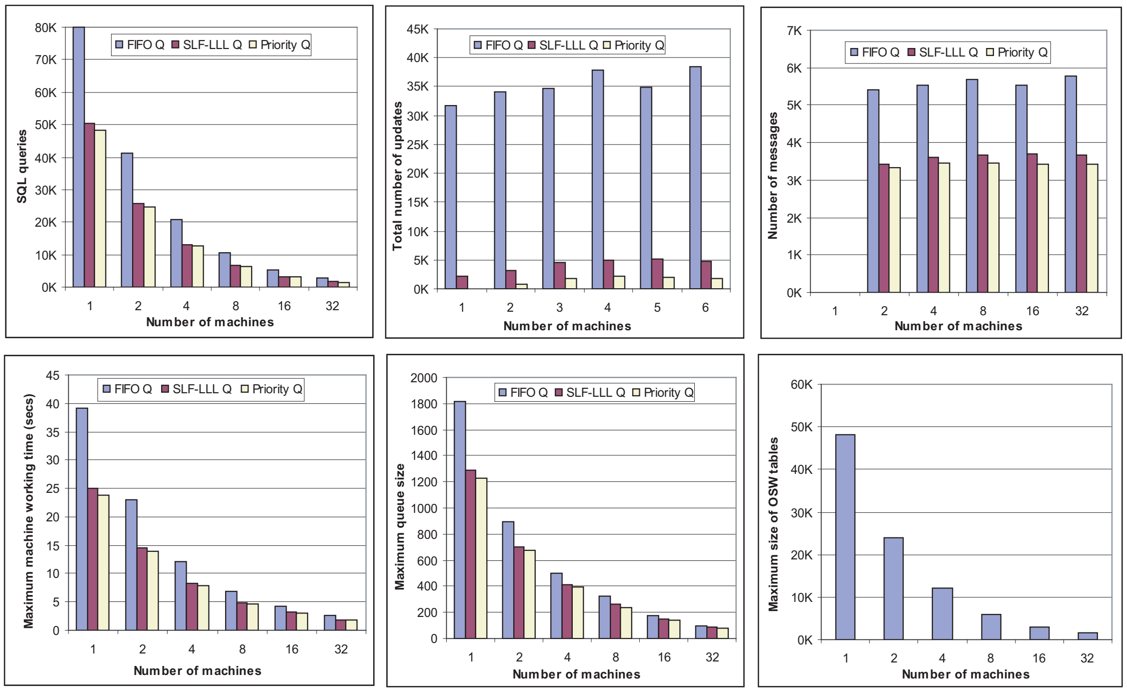

Figure 2. *Experimental results.*

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

We show here only 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 2. All the results are given with respect to the number of participating machines.

The top-left and bottom-left graphs are similar in shape. The top-left graph shows the maximum number (among machines) of SQL queries executed (for fetching database edges), while the bottom-left 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 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 top-middle graph, 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 top-right graph in Fig. 1 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 *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 bottom-middle and bottom-right graphs, 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 half as the number of machines doubles. These structures fit very well in main memory especially as the number of machines becomes larger.

6 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.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: from relations to semistructured data and XML*. Morgan Kaufmann, 2000.

[2] S. Abiteboul and V. Vianu. Regular path queries with constraints. *J. Comput. Syst. Sci.*, 58(3):428–452, 1999.

[3] D. P. Bertsekas, F. Guerriero, and R. Musmanno. Parallel asynchronous label-correcting methods for shortest paths. *Journal of Optimization Theory and Applications*, 88(2):297–320, 1996.

[4] U. Bureau. Tiger: Topologically integrated geographic encoding and referencing system.

[5] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Answering regular path queries using views. In *ICDE*. IEEE, 2000.

[6] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[7] G. Grahne and A. Thomo. An optimization technique for answering regular path queries. In *The World Wide Web and Databases*. Springer, 2000.

[8] G. Grahne and A. Thomo. Algebraic rewritings for optimizing regular path queries. *Theoretical Computer Science*, 296(3):453–471, 2003.

[9] G. Grahne and A. Thomo. New rewritings and optimizations for regular path queries. In *ICDT*. Springer, 2003.

[10] G. Grahne and A. Thomo. Query containment and rewriting using views for regular path queries under constraints. In *PODS*. ACM, 2003.

[11] G. Grahne and A. Thomo. Query answering and containment for regular path queries under distortions. In *FoIKS*. Springer, 2004.

[12] G. Grahne and A. Thomo. Regular path queries under approximate semantics. *Annals of Mathematics and Artificial Intelligence*, 46(1-2):165–190, 2006.

[13] G. Grahne and A. Thomo. Boundedness of regular path queries in data integration systems. In *IDEAS*. IEEE, 2007.

[14] G. Grahne, A. Thomo, and W. Wadge. Preferentially annotated regular path queries. In *ICDT*. Springer, 2007.

[15] R. H. Güting. Graphdb: Modeling and querying graphs in databases. In *VLDB*, volume 94, pages 12–15. Citeseer, 1994.

[16] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*. IEEE, 2002.

[17] D. Stefanescu and A. Thomo. Enhanced regular path queries on semistructured databases. In *Current Trends in Database Technology–EDBT 2006*. Springer, 2006.

[18] D. C. Stefanescu, A. Thomo, and L. Thomo. Distributed evaluation of generalized path queries. In *SAC*. ACM, 2005.

[19] D. Suciu. Distributed query evaluation on semistructured data. *ACM Transactions on Database Systems (TODS)*, 27(1):1–62, 2002.

[20] M. Y. Vardi. A call to regularity. In *Principles of computing & knowledge: Paris C. Kanellakis memorial workshop*. ACM, 2003.