Fault-Tolerant Computation of Distributed Regular Path Queries

Maryam Shoaran and Alex Thomo

University of Victoria, Canada
{maryam,thomo}@cs.uvic.ca

Abstract. Regular path queries are the building block of almost any mechanism for querying semistructured data. Despite the fact that the main applications of such data are distributed, there are only few works dealing with distributed evaluation of regular path queries. In this paper we present a message-efficient and truly distributed algorithm for computing the answer to regular path queries in a multi-source semistructured database setting. Our algorithm is general as it works for the larger class of weighted regular path queries on weighted semistructured databases.

Also, we show how to make our algorithm fault-tolerant to smoothly work in environments prone to process (or machine) failures. This is very desirable in a grid setting, which is today's new paradigm of distributed computing, and where one does not have full control over machines that can unexpectedly leave in the middle of computation.

1 Introduction

Semistructured data is the foundation for a multitude of applications in many important areas such as information integration, Web and communication networks, biological data management, etc. The data in these applications is conceptualized as edge-labeled graphs, and there is an inherent need to navigate these graphs by means of a recursive query language. As pointed out by seminal works in the field (cf. [10, 19, 6–8]), regular path queries (RPQ's) are the "winner" when it comes to expressing navigational recursion over semistructured data. These queries are in essence regular expressions over the database edge symbols, and in general, one is interested in finding query-matching database paths, which spell words in the (regular) query language.

Taking an example from spatial network databases (such as [27]), suppose that the user wants to find database paths consisting mainly of highway segments and tolerating up to k provincial roads or city streets. Clearly, such paths can easily be captured by the regular path query

$$Q = \mathit{highway}^* \parallel (\mathit{road} + \mathit{street} + \epsilon)^k,$$

where $\parallel$ is the shuffle operator (see e.g. [16]).

In this paper, we consider generalized RPQ's with weights as in [11, 12, 24, 13, 14]. For example, the user can write

$$Q = (\mathit{highway} : 1)^* \parallel (\mathit{road} : 2 + \mathit{street} : 3 + \epsilon)^k,$$

to express that she ideally prefers highways, then roads, which she prefers less, and finally she can tolerate streets, but with an even lesser preference.

Moreover, inherent database edge weights (or importance) can be naturally incorporated to scale up or down query preferences. Thus, in our spatial example, the edge importance could simply be the edge-length, and so, traversing a 100 kms highway would be less preferable than traversing a 49 kms provincial road, even though in general provincial roads are less preferable than highways.

Based on query-matching paths, there are two ways of defining the answer to an RPQ. The first is the single-source variant [1, 3], where the answer is defined to be the set of objects reachable from a given source by following some query-matching path. The second is the multi-source variant [19, 6–8, 14], where the answer is defined to be the set of *pairs* of objects that are connected by some query-matching path.

For generalized RPQ's, in the single-source variant, the answer is the set of (b, w) pairs, where w is the weight of the cheapest query-matching path connecting the database source object with object b.

On the other hand, in the multi-source variant, the answer is the set of (a, b, w) triples, where w is the weight of the cheapest query-matching path connecting database objects a and b.

In this paper, we focus on the second variant of generalized RPQ's. As the main applications based on semistructured data are distributed, we look at RPQ's from a distributed strategy angle.

Computing the answer to a generalized RPQ in the multi-source variant amounts to computing the "all-pairs shortest paths" in the subgraph of database paths spelling words in the query language. However, for each user query, there would be a new subgraph on which to compute all-pairs shortest paths, and such a subgraph cannot be known in advance, but rather only after the query evaluation finishes. This is "too late" for applying algorithms, which need global knowledge of the whole graph. With such algorithms, the user cannot see partial answers while waiting for the query to finish, and there is extra computation and communication overhead incurring after the subgraph [relevant to the query] is determined. Thus, the well-known Floyd-Warshall algorithm and its distributed variants are not appropriate to our database setting.

Regarding work on distributed shortest path computation, we remark here Haldar's algorithm in [15], which computes all-pairs shortest paths with the best known number of messages. In this paper, we adapt and extend Haldar's algorithm to compute instead answers to regular path queries and to work in an environment where the relevant part of the database graph is not known beforehand, but rather incrementally computed on the fly.

Our algorithm works under the assumption that the nodes of the relevant graph are computed on demand and they have local [neighbor] knowledge only. The central idea of our algorithm is to overlap computations starting from different database objects. We achieve this overlap in a careful way in order to guarantee the expansion of the best path first, in a similar spirit with the Dijkstra's methodology. However, at the same time we allow multiple expansions at different processes, which is what makes the algorithm truly distributed.[1]

Next, we extend our algorithm to account for process failures.[2] Having a fault-tolerant algorithm is very important especially in today's new paradigm of grid computing. Notably, in a grid setting the power comes from the synergy of many participating machines, whose main purpose might be completely different from the "grid-community service" performed during their low intensity periods. As such, grid machines are quite "unreliable" because they can withdraw at any time from a grid computation in order to perform their main "duties" they primarily are intended for.

Our fault-tolerant algorithm can smoothly adapt and be resilient to any number of process failures. Furthermore, it guarantees finding *at least* all the query answers obtainable if the computation were to be started from the scratch on the remaining live processes. Furthermore, we remark that, since some of the computation used supersets of these remaining processes, in general, we get more results than those strictly available if we were to restart the computation on the remaining processes only.

Finally, we note that our fault-tolerant algorithm does not require additional messages apart from the "ping"-like messages of the infrastructure for detecting process failures. We require for the processes to monitor the health of their neighbors only.

Notably, all the above are important and desirable properties for distributed fault-tolerant algorithms.

---

[1] A short version of this algorithm is described in [22]. However, the description there is quite partial, with a coarse-grained complexity analysis, and without proofs and useful observations.

[2] This is not approached at all in [22].

Related Work. To the best of our knowledge, only very few works present a distributed evaluation of regular path queries. In [24], a distributed algorithm is presented, which works based on local knowledge only. However, it has a message complexity which is quadratically worse than the complexity in this paper.

Besides [24], other works that have dealt with distributed RPQ's are [3, 25, 23, 20]. All four consider the single-source variant of RPQ's.

Finally, two recent works, [5] and [9], have presented distributed methods for the XPath query evaluation over XML trees using partial evaluation techniques. Their methods are not applicable to our case due to the following reasons. First, the methods of [5] and [9] work on a tree structure of XML documents, whereas databases in our context are general graphs and there are no "leaf" designated nodes. Second, they consider *unweighted* tree databases, and thus, the problem they deal with is in fact about reachability rather than shortest paths, which in turn is the case for our algorithm.

Organization. The rest of the paper is organized as follows. In Section 2, we give the definitions we are based on. In Section 3, we present our distributed algorithm. Next, in Section 4 and 5, we discuss its termination and complexity, respectively. In Section 6, we show the soundness and completeness of our algorithm. In Section 7, we extend our algorithm to be resilient against process failures. Finally, Section 8 concludes the paper.

2 Databases and Weighted RPQ's

We consider a database to be an edge-labeled graph with positive real values assigned to its edges. Intuitively, the nodes of the database graph represent objects and the edges represent relationships (and their importance) between the objects.

Formally, let Δ be an alphabet. Elements of Δ will be denoted $R, S, \ldots$. As usual, Δ^* denotes the set of all finite words over Δ. We also assume that we have a universe of objects, and objects will be denoted $a, b, c, \ldots$. A *database* DB is then a weighted graph (V, E), where V is a finite 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 $\mathbb{R}^+$.

Before talking about weighted preference path queries, it will help to first review the classical path queries.

A *regular path query* (RPQ) is a regular language over Δ. 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, $\tau \subseteq P \times \Delta \times P$ 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 RPQ's and FSA's that represent them.

Let $\mathcal{A}$ be a query FSA and $DB = (V, E)$ a database. Then, the *answer* to $\mathcal{A}$ on DB is defined as

$$Ans(\mathcal{A}, DB) = \{(a, b) \in V \times V : a \xrightarrow{w} b \text{ in } DB \text{ and } w \text{ is accepted by } \mathcal{A}\},$$

where $a \xrightarrow{w} b$ denotes a path from a to b spelling w in the database.

Now, let $\mathbb{N} = \{1, 2, \ldots\}$. A *weighted finite state automaton* (WFSA) $\mathcal{A}$ is a quintuple $(P, \Delta, \tau, p_0, F)$, where P, p_0, and F are similarly defined as for a classical FSA, while the transition relation τ is now a subset of $P \times \Delta \times \mathbb{N} \times P$. Query WFSA's are given by means of weighted regular expressions (WRE's). The reader is referred to [2] for efficient algorithms translating WRE's into WFSA's.

Given a weighted database $DB = (V, E)$, and a query WFSA $\mathcal{A} = (P, \Delta, \tau, p_0, F)$, the preferentially *scaled weighted answer* (SWAns) of $\mathcal{A}$ on DB is

$$SWAns(\mathcal{A}, DB) = \{(a, b, r) \in V \times V \times \mathbb{R}^+ :$$
$$r = inf\left\{\sum_{i=1}^{n} r_i k_i : \; n \in \mathbb{N}, (c_{i-1}, R_i, r_i, c_i) \in E, (p_{i-1}, R_i, k_i, p_i) \in \tau \right.$$
$$\left. c_0 = a, \; c_n = b, \text{ and } p_n \in F\right\}\}.$$

Observe that, according to this definition, if $(a, b, r) \in SWAns(\mathcal{A}, DB)$, then there exists a path (possibly a set of paths) from a to b in DB spelling some word(s) in the query language. Furthermore r is the weight of the cheapest sequence of edge-transition matches corresponding to such paths. Number $n \in \mathbb{N}$ denotes the length of a path and is (possibly) different for different paths.

As an example, consider the database DB and query automaton $\mathcal{A}$ in Fig. 1. There are three paths going from object a to object c. The shortest path consisting of a single edge T of weight 1, is not the cheapest path according to the query. Rather, the cheapest path is the one spelling RS. The other path, spelling RT, does not match any query automaton path, so it is not considered at all. Hence, we have that $(a, c, 3)$ is the answer with respect to a and c.

Similarly, we find the other query answers and finally have $SWAns(\mathcal{A}, DB) = \{(a, b, 1)$, $(a, c, 3)$, $(a, d, 6)$, $(a, a, 7)$, $(b, c, 5)$, $(b, d, 8)$, $(b, a, 9)\}$.

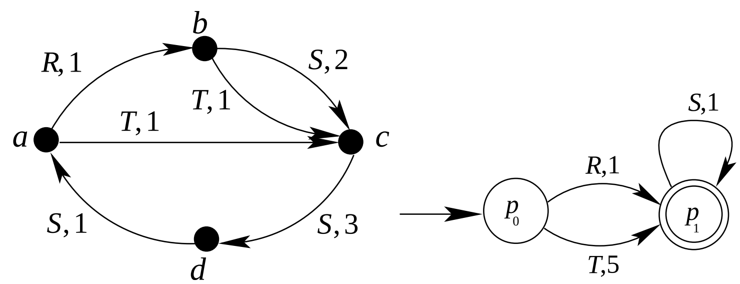

Fig. 1. A database DB and a query automaton $\mathcal{A}$

In order to help understanding of our distributed algorithm, we will first review the well-known method for the evaluation of classical RPQ's (cf. [1]). The evaluation proceeds by creating object-state pairs from the query automaton and the database. For this, let $\mathcal{A}$ be a query FSA. Starting from an object a of a database DB, we first create the pair (a, p_0), where p_0 is the initial state in $\mathcal{A}$. Then, we create all the pairs (b, p) such that there exists an edge from a to b in DB and a transition from p_0 to p in $\mathcal{A}$, and furthermore the labels of the edge and the transition match. In the same way, we continue to create new pairs from existing ones, until we are not anymore able to do so. In essence, what is happening is a *lazy* construction of a Cartesian product graph of the database with the query automaton. Of course, only a small (hopefully) part of the Cartesian product is really constructed. This ultimately depends on the selectivity of the query.

After obtaining the above Cartesian product graph, producing query answers becomes a question of computing reachability of nodes (b, p), where p is a final state, from (a, p_0), where p_0 is the initial state. Namely, if (b, p) is reachable from (a, p_0), then (a, b) is a tuple in the query answer.

Now, when having instead a weighted query automaton and database, one can build a weighted Cartesian product graph. We show that in order to compute weighted answers, we have to find, in the Cartesian product graph, the cheapest paths from all (a, p_0) to all (b, p), where p is a final state in the query automaton $\mathcal{A}$.

As we mentioned in the Introduction, in general there is a different Cartesian product graph for each query. Thus, a useful distributed algorithm must not rely on having global knowledge about this graph, since it will only be known after the completion of the query evaluation.

We formally define the Cartesian product $\mathcal{C}$ of a database $DB = (V, E)$ and a query automaton $\mathcal{A} = (P, \Delta, \tau, p_0, F)$ as the graph with

- nodes (b, p), where b is an object in V and p is a state in P, and
- edges $((b, p), R, rk, (c, q))$, such that there exists an edge (b, R, r, c) in E and a transition (p, R, k, q) in τ.

Based on this definition, we have that

Theorem 1 *$(a, b, r) \in SWAns(\mathcal{A}, DB)$ if and only if there exists some path from (a, p_0) to (b, p_y) in $\mathcal{C}$, with p_y being a final state in $\mathcal{A}$ and r the weight of a cheapest of such paths.*

Proof. By the construction of $\mathcal{C}$, we have that:

1. For every path π_1 in DB matching some weighted transition path π_2 in $\mathcal{A}$, there exists some path π in $\mathcal{C}$ spelling the same word as π_1 (and π_2) and annotated by the product of the weights of the edges and transitions in π_1 and π_2, respectively.
2. For every path π in $\mathcal{C}$ there exist paths π_1 in DB and π_2 in $\mathcal{A}$, which match and spell the same word as π, and furthermore, the corresponding edges and transitions of π_1 and π_2, respectively, have weights whose products give the weights of the edges in π.

Now, our claim is a direct consequence of the above, and the definition of $SWAns(\mathcal{A}, DB)$. □

3 Distributed Algorithm

The key feature of our algorithm is the overlapping of computations starting from different database objects. We assume that each database object has only local knowledge about the database graph, that is, it only knows the identities of its neighbors and the labels and weights of its outgoing edges. Further, we assume that each object a, is being serviced by a dedicated process for that object P_a. Our algorithm can be easily modified for the case when subgraphs of the database (as opposed to single objects) are being serviced by the processes. In such a case, many of the basic computation messages are sent and received locally by the processes from and to themselves.

First, the query automaton is sent to each process. Such a service is commonly achieved by distributively creating a minimum spanning tree (MST) of the processes before any query starts to be evaluated (cf. [4] for a message optimal MST algorithm).

We can note here that such an MST can be used by the processes to transmit their id's and get so to know each other. However, we do not require this coordination step. Even if such a step is undertaken, the real challenge [which remains] is that the relevant subgraph of the [query–database] Cartesian product cannot be known in advance for a new query. In other words, a shortest path algorithm has to work with a target graph not known beforehand.

Continuing the description of our algorithm, a process, say P_a (which serves object a), starts by creating an initial task for itself. The tasks are "keyed" (uniquely identified) by the automaton states, with the initial tasks being keyed by the initial state p_0. Each task has three components:

1. an automaton state,
2. a status flag that can switch between *active*, *passive*, and *completed* values, and
3. a table (or set) of tuples representing knowledge about "objects reached so far" along with additional information (to be precisely described soon).

A typical task will be written as $\langle p_x, status, \{\dots\}\rangle$. We will refer to the table $\{\dots\}$ as $P_a.p_x.T$ or $p_x.T$ when P_a is clear from the context. The tuples in this table have four components, and will be written as $[(c, p_z), (b, p_y), weight, status]$, where

1. (c, p_z) states that the algorithm, starting from object a and state p_x, has reached (possibly through multiple hops) object c and state p_z,
2. (b, p_y) states that the best path (known so far) to reach (c, p_z) is by passing via object b and state p_y, where b and p_y are neighbors of a and p_x in the database and query automaton, respectively,
3. *weight* is the weight of this best path (determined as in Section 2), and
4. *status* is a flag switching from *prov* to *opt* values telling whether *weight* is provisional and would possibly be improved or optimal and permanently stay as is.

Initially, when a p_x-task is created, process P_a tries to find all the outgoing edges from a, which match (w.r.t. the symbol label) outgoing transitions from p_x. Let (a, R, r, b) be such an edge which matches transition (p_x, R, k, p_y). Then, P_a inserts tuple $[(b, p_y), (b, p_y), k \cdot r, prov]$ in table $P_a.p_x.T$. If there are multiple $(a, \_, \_, b)$ - $(p_x, \_, \_, p_y)$ edge-transition matches, then only the match with the cheapest weight product is considered.

Each process P_a starts by creating and initializing a *passive* p_0-task, which is possibly selected next for processing. We say "possibly" because a process might receive new tasks from neighboring processes.

When a task is selected for processing, its *provisional*-status tuples (or *provisional* tuples in short) will be "expanded" in a best-first order with respect to their weights. If there are no more *provisional* tuples in the table of the p_0-task, then the task attains a *completed* status, and the process reports its *local termination*.

All (working) processes run in parallel exactly the same algorithm, which consists of four concurrent threads. These threads are as follows:

Expansion: A process P_a selects a *passive* task, say p_x–task, which still has provisional tuples in its table.
Then, P_a makes the p_x–task *active*, and selects for expansion the cheapest *provisional* tuple in its table $P_a.p_x.T$.
The *active* status for the p_x–task prevents the expansion of other *provisional* tuples in $P_a.p_x.T$.
Next, P_a sends a request message to its neighbor P_b asking it to: (1) create a task p_y, and (2) send its "knowledge" regarding the $[(c, p_z), \_, \_, \_]$ tuple.

Task Creation: When a process P_b receives a request message from P_a (w.r.t p_x) for the creation of a task, say p_y, it creates a p_y-keyed task (if such does not exist) and properly initializes it. Next, P_b establishes a virtual communication channel between its p_y-task and the p_x-task of P_a. This communication channel is specialized for the relevant tuple (keyed by (c, p_z)), whose expansion caused the request message. The weight of the channel will be equal to the cost of going from (a, p_x) to (b, p_y), which is in fact the weight of the (b, p_y)–keyed tuple in $P_a.p_x.T$.
Notably, overlapping of computations happens when process P_b receives another request message for the same task from a different neighboring process. In such a case, the receiving process P_b only establishes a communication channel with the sending process.

Reply: After creating the communication channel, process P_b will send table $P_b.p_y.T$ backward to task $P_a.p_x$. This backward message will be sent only when the (c, p_z)-keyed tuple in $P_b.p_y.T$ attains an *optimal* status. The weight of the communication channel is added to the weights of the tuples as they are bundled together to be sent. We refer to this modified (message) table as $P_b.p_y.T^*$.

Update: When a process P_a receives from some process P_b a backward reply message, which is related to a tuple $[(c, p_z), \_, \_, prov]$ of task $P_a.p_x$, and contains the table $P_b.p_y.T^*$, it will: (1) update (relax) the *provisional* tuples in $P_a.p_x.T$ as appropriate (if there are tuples with the same keys in $P_b.p_y.T^*$), (2) add to table $P_a.p_x.T$ all tuples of $P_b.p_y.T^*$, which do not have any "peer" (tuple with the same key) in $P_a.p_x.T$, and (3) change the status of the p_x-task to *passive*.

Figure 2 illustrates the different possible statuses of a task during the execution of the algorithm. As described above, at the moment of creation, each task has *passive* status. If a *passive*–status task

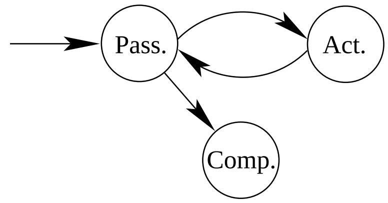

Fig. 2. Task Status Diagram.

does not have any *provisional* tuple in its table, the status is changed to *completed*. Otherwise, the process can start the expansion of *provisional* tuples in the task table. Starting the expansion of a tuple, the task status is changed to *active* which, as mentioned in the Expansion thread, prevents the expansion of other *provisional* tuples until receiving the reply to the last request message. When an *active*–status task receives a reply message for the recent expansion, it starts the Update thread, at the end of which the task status is changed to *passive* making the task ready for another expansion. So, the *passive* and *active* statuses can interleave several times during the execution of the algorithm, but the *completed* status does not change once it has been reached.

Formally our algorithm is as follows.

Algorithm 1

Input:

1. A database DB. For simplicity we assume that each database object, say a, is being serviced by a dedicated process for that object P_a.
2. A query WFSA $\mathcal{A} = (P, \Delta, \tau, p_0, F)$.

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

Method:

1. **Initialization**: Each process P_a creates a task $\langle p_0, \textit{passive}, \{\ldots\}\rangle$ for itself. The table $\{\ldots\}$ (referred to as $P_a.p_0.T$) is initialized as follows:
   (a) insert tuple $[(a, p_0), (a, p_0), 0, \textit{opt}]$, and
   (b) For each edge-transition match,
   (a, R, r, b) in DB and
   (p_0, R, k, p) in $\mathcal{A}$,
   insert tuple $[(b, p), (b, p), k \cdot r, \textit{prov}]$
   (if there are multiple $(a, \_, \_, b) - (p_0, \_, \_, p)$ edge-transition matches, then the cheapest weight product is considered.)

   If at point (b) there is no edge-transition match, then make the status of the p_0-task *completed*.

2. Concurrently execute all the four following threads at each process in parallel until termination is detected. [For clarity, we describe the threads at two processes, P_a and P_b.]

3. **Expansion**: [At process P_a]
   (a) Select a *passive* p_x-task for processing. Make the status of the task *active*.
   (b) Select the cheapest *provisional*-status tuple, say $[(c, p_z), (b, p_y), w, \textit{prov}]$ from table $P_a.p_x.T$.
   (c) Request P_b, with respect to state p_y, to provide information about (c, p_z).
   For this, send a message $\langle p_y, [p_x, (c, p_z), w_{ab}]\rangle$ to P_b, where w_{ab} is the cost of going from (a, p_x) to (b, p_y), which is equal to the weight of the (b, p_y)–keyed tuple in $P_a.p_x.T$.
   (d) Sleep, with regard to p_x-task, until the reply message for (c, p_z) comes from P_b.

4. **Task Creation**: [At process P_b]
   Upon receiving a message $\langle p_y, [p_x, (c, p_z), w_{ab}]\rangle$ from P_a:

if there is not yet a p_y-task
then create a task $\langle p_y, passive, \{\ldots\}\rangle$ and initialize its table similarly as in the first phase. That is,

(a) insert tuple $[(b, p_y), (b, p_y), 0, opt]$, and

(b) For each edge-transition match,
(b, R, r, d) in DB and
(p_y, R, k, p_u) in $\mathcal{A}$,
insert tuple $[(d, p_u), (d, p_u), k \cdot r, prov]$
(if there are multiple $(b, \_, \_, d)$–$(p_y, \_, \_, p_u)$ edge-transition matches, then the cheapest weight product is considered.)

Also, establish a virtual communication channel with P_a. This channel relates the p_y-task of P_b with the p_x-task of P_a. Further, it is indexed by (c, p_z) and is weighted by w_{ab} (the weight included in the received message).

else [P_b has already a p_y-task.] Do not create a new task, but only establish a communication channel with P_a as described above.

5. **Reply**: [At process P_b]
When in the p_y-task, the tuple $[(c, p_z), (\_, \_), \_, \_]$ is or becomes optimally weighted, *reply back* to all the neighbor processes, which had sent a task requesting message $\langle p_y, [\_, (c, p_z), \_]\rangle$ to P_b.
For example, P_b sends to such a neighbor, say P_a, through the corresponding communication channel, the message $\langle P_b.p_y.T^*\rangle$, which is table $P_b.p_y.T$ after adding the channel weight to the weight of each tuple.

6. **Update**: [At process P_a]
Upon receiving a reply message $\langle P_b.p_y.T^*\rangle$ from a neighbor P_b w.r.t. the expansion of a (c, p_z)-keyed tuple in table $P_a.p_x.T$ do:
   (a) Change the status of (c, p_z)-keyed tuple to the status of the same keyed tuple in $P_b.p_y.T^*$ [3].
   (b) For each tuple $[(d, p_u), (\_, \_), v, prov]$ in $P_b.p_y.T^*$, which has a smaller weight (v) than a same-key tuple $[(d, p_u), (\_, \_), \_, prov]$ in $P_a.p_x.T$, replace the latter by $[(d, p_u), (b, p_y), v, prov]$.
   (c) Add to $P_a.p_x.T$ all the rest of the $P_b.p_y.T^*$ tuples, i.e., those which do not have corresponding same-key tuples in $P_a.p_x.T$.
   Also, change the via component of these tuples to be (b, p_y).
   (d) **if** the p_x-task does not have anymore *provisional* tuples,
   then make its status *completed*.
   If $p_x = p_0$, then report that all query answers from P_a have been computed.
   else make the status of the p_x-task *passive*.

Finally upon termination, which happens when all the tasks in every process have attained *completed* status, set

$$eval(\mathcal{A}, DB) = \{(a, b, r) : [(b, p_y), (\_, \_), r, opt)] \in P_a.p_0.T \text{ and } p_y \in F\}.$$

In the next section, we show the soundness and completeness of our algorithm. Based on them, the following theorem can be stated.

Theorem 2 *Upon termination of the above algorithm, we have that*

$$eval(\mathcal{A}, DB) = SWAns(\mathcal{A}, DB).$$

[3] This status is *optimal*.

The algorithm can report answers as soon as their corresponding tuples become *optimal*.

We define a *partial answer set* to be a *subset* of $SWAns(\mathcal{A}, DB)$.

Now, instead of creating $eval(\mathcal{A},\ DB)$ upon termination of the algorithm, we can incrementally grow it each time that a tuple becomes optimal. Because the weight of an optimal tuple does not change any further, any snapshot of $eval(\mathcal{A},\ DB)$ at any time during the execution of the above algorithm is a partial answer set. Upon termination, all the answers would have been reported. While the user waits for the query evaluation to finish, new answers will eventually arrive. However, the ones already reported preserve their weights, which are optimal. This is in contrast to [24] in which the user might see the already reported answers to possibly get their weights lowered.

Now, we illustrate Algorithm 1 by the following example. Consider the database and query automaton in Fig. 3, *left* and *right* respectively.

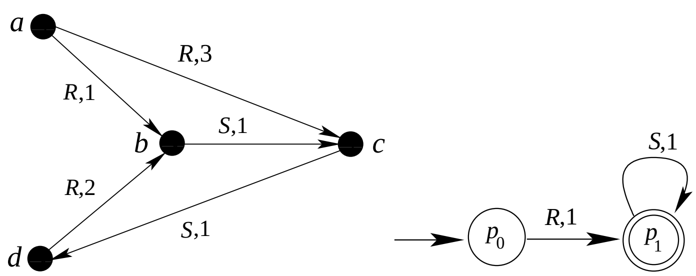

Fig. 3. A database and a query automaton

A possible sequence of actions for Algorithm 1 is given in Table 4. In the first column labeled "T" we number the hypothetical time points in which we observe the system. An explanation of the actions at each time point follows.

1. All processes create a task $\langle p_0, \mathit{passive}, \{\ldots\}\rangle$ for themselves and initialize their tables.

2. (a) P_a and P_d do have *provisional* tuples in the tables of their p_0–tasks, and thus, make their p_0-tasks *active* and expand their cheapest *provisional* tuples.
   For this, they send a request message to P_b for the creation of a p_1–task.
   On the other hand, processes P_b and P_c do not have *provisional* tuples in their p_0-tasks. Hence, they make their p_0–tasks *completed*. That is, there are no $(b, \_, \_)$ and $(c, \_, \_)$ query answers to be expected.

   (b) P_b receives the request messages from P_a and P_d, and creates the p_1–task. Also, P_b initializes this task as described in the algorithm. Of course, P_b creates only one such task to serve both P_a and P_d, and thus, we see here an effective computation overlap.
   Then, P_b establishes the appropriate communication channels between its p_1–task and the p_0–tasks in P_a and P_d.
   P_b is not only asked to create the p_1–task, but also to provide information about the (b, p_1)-keyed tuple. Since the status of this tuple in the p_1–task of P_b is *optimal*, P_b sends its $p_1.T$ knowledge to $P_a.p_0$ and $P_d.p_0$ adding along the way the *weights* of the related channels.

3. Upon receiving the reply message from P_b, processes P_a and P_d update the tables of their p_0–tasks. Note that the statuses of the (b, p_1)–keyed tuples in $P_a.p_0.T$ and $P_d.p_0.T$ become *optimal*.
   P_a relaxes the (c, p_1)-keyed tuple in $p_0.T$ and changes its via to (b, p_1).
   P_d adds to $P_d.p_0.T$ the rest of the $P_b.p_1.T^*$ tuples setting their via component to (b, p_1).
   Then, P_a and P_d change the status of their p_0–tasks to *passive* becoming thus ready for the next expansion.

T	P_a	P_b	P_c	P_d
1	$\langle p_0$, *passive*, $\{[(a, p_0), (a, p_0), 0, o]$, $[(b, p_1), (b, p_1), 1, p]$, $[(c, p_1), (c, p_1), 3, p]\}\rangle$	$\langle p_0$, *passive*, $\{[(b, p_0), (b, p_0), 0, o]\}\rangle$	$\langle p_0$, *passive*, $\{[(c, p_0), (c, p_0), 0, o]\}\rangle$	$\langle p_0$, *passive*, $\{[(d, p_0), (d, p_0), 0, o]$, $[(b, p_1), (b, p_1), 2, p]\}\rangle$
2	$\langle p_0$, *active*, $\{[(a, p_0), (a, p_0), 0, o]$, $\mathbf{[(b, p_1), (b, p_1), 1, p]}$, $[(c, p_1), (c, p_1), 3, p]\}\rangle$	$\langle p_0$, *completed*, $\{[(b, p_0), (b, p_0), 0, o]\}\rangle$ $\langle p_1$, *passive*, $\{[(b, p_1), (b, p_1), 0, o]$, $[(c, p_1), (c, p_1), 1, p]\}\rangle$	$\langle p_0$, *completed*, $\{[(c, p_0), (c, p_0), 0, o]\}\rangle$	$\langle p_0$, *active*, $\{[(d, p_0), (d, p_0), 0, o]$, $\mathbf{[(b, p_1), (b, p_1), 2, p]}\}\rangle$
3	$\langle p_0$, *passive*, $\{[(a, p_0), (a, p_0), 0, o]$, $[(b, p_1), (b, p_1), 1, o]$, $[(c, p_1), (b, p_1), 2, p]\}\rangle$	$\langle p_0$, *completed*, $\{[(b, p_0), (b, p_0), 0, o]\}\rangle$ $\langle p_1$, *passive*, $\{[(b, p_1), (b, p_1), 0, o]$, $[(c, p_1), (c, p_1), 1, p]\}\rangle$	$\langle p_0$, *completed*, $\{[(c, p_0), (c, p_0), 0, o]\}\rangle$	$\langle p_0$, *passive*, $\{[(d, p_0), (d, p_0), 0, o]$, $[(b, p_1), (b, p_1), 2, o]$, $[(c, p_1), (b, p_1), 3, p]\}\rangle$
4	$\langle p_0$, *active*, $\{[(a, p_0), (a, p_0), 0, o]$, $[(b, p_1), (b, p_1), 1, o]$, $\mathbf{[(c, p_1), (b, p_1), 2, p]}\}\rangle$	$\langle p_0$, *completed*, $\{[(b, p_0), (b, p_0), 0, o]\}\rangle$ $\langle p_1$, *active*, $\{[(b, p_1), (b, p_1), 0, o]$, $\mathbf{[(c, p_1), (c, p_1), 1, p]}\}\rangle$	$\langle p_0$, *completed*, $\{[(c, p_0), (c, p_0), 0, o]\}\rangle$ $\langle p_1$, *passive*, $\{[(c, p_1), (c, p_1), 0, o]$, $[(d, p_1), (d, p_1), 1, p]\}\rangle$	$\langle p_0$, *active*, $\{[(d, p_0), (d, p_0), 0, o]$, $[(b, p_1), (b, p_1), 2, o]$, $\mathbf{[(c, p_1), (b, p_1), 3, p]}\}\rangle$
5	$\langle p_0$, *passive*, $\{[(a, p_0), (a, p_0), 0, o]$, $[(b, p_1), (b, p_1), 1, o]$, $[(c, p_1), (b, p_1), 2, o]$, $[(d, p_1), (b, p_1), 3, p]\}\rangle$	$\langle p_0$, *completed*, $\{[(b, p_0), (b, p_0), 0, o]\}\rangle$ $\langle p_1$, *passive*, $\{[(b, p_1), (b, p_1), 0, o]$, $[(c, p_1), (c, p_1), 1, o]$, $[(d, p_1), (c, p_1), 2, p]\}\rangle$	$\langle p_0$, *completed*, $\{[(c, p_0), (c, p_0), 0, o]\}\rangle$ $\langle p_1$, *passive*, $\{[(c, p_1), (c, p_1), 0, o]$, $[(d, p_1), (d, p_1), 1, p]\}\rangle$	$\langle p_0$, *passive*, $\{[(d, p_0), (d, p_0), 0, o]$, $[(b, p_1), (b, p_1), 2, o]$, $[(c, p_1), (b, p_1), 3, o]$, $[(d, p_1), (b, p_1), 4, p]\}\rangle$
6	$\langle p_0$, *active*, $\{[(a, p_0), (a, p_0), 0, o]$, $[(b, p_1), (b, p_1), 1, o]$, $[(c, p_1), (b, p_1), 2, o]$, $\mathbf{[(d, p_1), (b, p_1), 3, p]}\}\rangle$	$\langle p_0$, *completed*, $\{[(b, p_0), (b, p_0), 0, o]\}\rangle$ $\langle p_1$, *active*, $\{[(b, p_1), (b, p_1), 0, o]$, $[(c, p_1), (c, p_1), 1, o]$, $\mathbf{[(d, p_1), (c, p_1), 2, p]}\}\rangle$	$\langle p_0$, *completed*, $\{[(c, p_0), (c, p_0), 0, o]\}\rangle$ $\langle p_1$, *active*, $\{[(c, p_1), (c, p_1), 0, o]$, $\mathbf{[(d, p_1), (d, p_1), 1, p]}\}\rangle$	$\langle p_0$, *active*, $\{[(d, p_0), (d, p_0), 0, o]$, $[(b, p_1), (b, p_1), 2, o]$, $[(c, p_1), (b, p_1), 3, o]$, $\mathbf{[(d, p_1), (b, p_1), 4, p]}\}\rangle$ $\langle p_1$, *completed*, $\{[(d, p_1), (d, p_1), 0, o]\}\rangle$
7	$\langle p_0$, *passive*, $\{[(a, p_0), (a, p_0), 0, o]$, $[(b, p_1), (b, p_1), 1, o]$, $[(c, p_1), (b, p_1), 2, o]$, $[(d, p_1), (b, p_1), 3, o]\}\rangle$	$\langle p_0$, *completed*, $\{[(b, p_0), (b, p_0), 0, o]\}\rangle$ $\langle p_1$, *passive*, $\{[(b, p_1), (b, p_1), 0, o]$, $[(c, p_1), (c, p_1), 1, o]$, $[(d, p_1), (c, p_1), 2, o]\}\rangle$	$\langle p_0$, *completed*, $\{[(c, p_0), (c, p_0), 0, o]\}\rangle$ $\langle p_1$, *passive*, $\{[(c, p_1), (c, p_1), 0, o]$, $[(d, p_1), (d, p_1), 1, o]\}\rangle$	$\langle p_0$, *passive*, $\{[(d, p_0), (d, p_0), 0, o]$, $[(b, p_1), (b, p_1), 2, o]$, $[(c, p_1), (b, p_1), 3, o]$, $[(d, p_1), (b, p_1), 4, o]\}\rangle$ $\langle p_1$, *completed*, $\{[(d, p_1), (d, p_1), 0, o]\}\rangle$
8	$\langle p_0$, *completed*, $\{[(a, p_0), (a, p_0), 0, o]$, $[(b, p_1), (b, p_1), 1, o]$, $[(c, p_1), (b, p_1), 2, o]$, $[(d, p_1), (b, p_1), 3, o]\}\rangle$	$\langle p_0$, *completed*, $\{[(b, p_0), (b, p_0), 0, o]\}\rangle$ $\langle p_1$, *completed*, $\{[(b, p_1), (b, p_1), 0, o]$, $[(c, p_1), (c, p_1), 1, o]$, $[(d, p_1), (c, p_1), 2, o]\}\rangle$	$\langle p_0$, *completed*, $\{[(c, p_0), (c, p_0), 0, o]\}\rangle$ $\langle p_1$, *completed*, $\{[(c, p_1), (c, p_1), 0, o]$, $[(d, p_1), (d, p_1), 1, o]\}\rangle$	$\langle p_0$, *completed*, $\{[(d, p_0), (d, p_0), 0, o]$, $[(b, p_1), (b, p_1), 2, o]$, $[(c, p_1), (b, p_1), 3, o]$, $[(d, p_1), (b, p_1), 4, o]\}\rangle$ $\langle p_1$, *completed*, $\{[(d, p_1), (d, p_1), 0, o]\}\rangle$

Fig. 4. A possible execution of Algorithm 1. Due to space constraints, we have abbreviated *prov* by p, and *opt* by o. We show in **bold** the tuples under expansion.

4. (a) P_a and P_d make the status of their p_0–tasks *active*, and expand the tuples $[(c, p_1), (b, p_1), 2, prov]$ and $[(c, p_1), (b, p_1), 3, prov]$ respectively by sending request messages to process P_b.
   (b) P_b has already a p_1–task, and thus, it just establishes communication channels with P_a and P_d specialized for (c, p_1).
   As the status of the (c, p_1)-keyed tuple in $P_b.p_1.T$ is *provisional*, P_b cannot yet reply back to P_a or P_d.
   Instead, P_b makes the status of task p_1 *active* and starts its processing. That is, P_b selects the cheapest *provisional* tuple, i.e., the tuple $[(c, p_1), (c, p_1), 1, prov]$, and sends a request message to P_c to create task p_1.
   (c) Upon receiving the request message from P_b, process P_c creates and initializes a p_1–task. Also, P_c establishes a communication channel with P_b, which is specialized for (c, p_1). Since the status of the (c, p_1)-keyed tuple is *optimal*, P_c replies back to P_b with the message $\langle P_c.p_1.T^* \rangle$.

   [The rest of the steps will be described more briefly.]

5. (a) Upon receiving the reply message from P_c, P_b updates its $p_1.T$ table as appropriate.
   (b) Now, P_b has an *optimal* status for the (c, p_1)–keyed tuple in $p_1.T$, and thus, replies back to P_a and P_d with the message $\langle P_b.p_1.T^* \rangle$.
   (c) Upon receiving the reply message from P_b, P_a and P_d update their $p_0.T$ tables as appropriate.

6. (a) P_a and P_d expand the tuples $[(d, p_1), (b, p_1), 3, prov]$ and $[(d, p_1), (b, p_1), 4, prov]$ respectively.
   (b) In effect, P_b expands $[(d, p_1), (c, p_1), 2, prov]$, and then P_c expands $[(d, p_1), (d, p_1), 1, prov]$. P_c requests from P_d to create a p_1–task and provide information about (d, p_1).
   (c) P_d creates and initializes the p_1–task and replies back to P_c with the message $\langle P_d.p_1.T^* \rangle$.

7. (a) Upon receiving the reply message from P_d, P_c updates its $p_1.T$ table as appropriate.
   Then, P_c replies back to P_b with the message $\langle P_c.p_1.T^* \rangle$.
   (b) Upon receiving the reply message from P_c, P_b updates its $p_1.T$ table as appropriate.
   Then, P_b replies back to P_a and P_d.
   (c) Upon receiving the reply message from P_b, P_a and P_d update their $p_0.T$ tables as appropriate.
8. Finally, as there are no more provisional tuples in any of the tasks, they attain a *completed* status.

Observe that we can terminate as soon as the p_0-tasks become *completed* in all the processes. There is no need to continue with the completion of the rest of the tasks. Their completion would not bring any new query answers, thus we can safely abort them.

Note that, we can incrementally report the query answers as soon as their corresponding tuple appears in the table of a p_0–task in some process. For example, $(a, b, 1)$ and $(d, b, 2)$ can be reported at time point 3, $(a, c, 2)$ and $(d, c, 3)$ can be reported at time point 5, and so on.

At time point 4, when P_a and P_d expand the (c, p_1)-keyed tuples requesting P_b to provide information about such a tuple in $P_b.p_1.T$, it happens that this tuple is the cheapest *provisional* tuple in $P_b.p_1.T$. Another instance of such a situation is at time point 6, in which again, the requested information is about a tuple that is the cheapest *provisional* tuple in $P_b.p_1.T$. These are not coincidental and by the following theorem, we show that this is indeed a property of the algorithm which guarantees the soundness (in Section 6). Of course, the request might be for an *optimal* tuple, and there is no need for further expansion in order to reply back. Note, that the following theorem is about the case when the request is for a *provisional* tuple.

Theorem 3 *If a process, through a task request message, is asked to provide information about a* provisional *tuple, then this tuple is the cheapest one among such tuples in the requested task.*

Proof. Suppose process P_a asks process P_b for a tuple in its p_y-task. Let the expanded tuple in P_a be $[(c, p_z), (b, p_y), w_{ac}, prov]$. This expansion will ask from P_b to provide information about the

(c, p_z)-keyed tuple in its p_y-task. Let this tuple be $[(c, p_z), (\_, \_), w_{bc}, prov]$. We want to show that this tuple is the cheapest among the *provisional* tuples in $P_b.p_y.T$.

Since (b, p_y) is the via component of the (c, p_z)-keyed tuple in $P_a.p_x.T$, we conclude that this tuple has got its weight, during an update phase, from the tuple $[(c, p_z), (\_, \_), w_{bc}, prov]$ in $P_b.p_y.T$ after adding the weight of the corresponding communication channel.

Along with the $[(c, p_z), (\_, \_), w_{bc}, prov]$ tuple, P_a got from P_b all the other tuples in $P_b.p_y.T$, on whose weights the same channel weight w_{ab} was added. Now, since $[(c, p_z), (b, p_y), w_{ac}, prov]$ is the cheapest *provisional* tuple in $P_a.p_x.T$, and its weight w_{ac} is in fact equal to $w_{ab} + w_{bc}$, we have that $[(c, p_z), (\_, \_), w_{bc}, prov]$ is the cheapest tuple in $P_b.p_y.T$. □

Based on the above, we show now the following theorem which is needed in the proofs for the soundness and completeness of our algorithm (Section 6).

Theorem 4 *Let $[(c, p_z), (b, p_y), w, prov]$ be a tuple in $P_a.p_x.T$ selected for expansion, and $[(c, p_z), (b, p_y), w', opt]$ be this tuple with optimal status after the expansion. Then, $w = w'$.*

Proof. When $[(c, p_z), (b, p_y), w, prov]$ gets expanded, a request message asking information about (c, p_z) is propagated through a path π with nodes $(a, p_x), (b, p_y), \ldots, (c, p_z)$ until reaching a process with an optimal (c, p_z)-keyed tuple ($P_c.p_z.T$, at least, will have such an optimal tuple). Let π' be the subpath of π (starting from (a, p_x)) that is in fact traversed. Of course, π' might be the whole π when the only optimal (c, p_z)-keyed tuple along π is the one in $P_c.p_z.T$ (which is surely optimal due to the initialization).

According to Theorem 3, in the task tables of the processes along π' there is no provisional tuple with a weight less than the weight of the (c, p_z)-keyed tuple. Thus, all the processes along π' expand in turn their (c, p_z)-keyed tuples. Since there is no other expansion during the processing of the (c, p_z)-keyed tuples along π', there is no change in the weight of these (c, p_z)-keyed tuples including the weight of tuple $[(c, p_z), (b, p_y), w, prov]$ in $P_a.p_x.T$. Thus, we have $w = w'$. □

4 Termination

In the following theorem we show that the algorithm terminates and it does not enter an infinite loop. That is, eventually there will be no more *provisional* tuples in the tables of the p_0 tasks, which is the condition for termination of the algorithm at each process.

Theorem 5 *Algorithm 1 (positively) terminates.*

Proof. Suppose there is a deadlock. Without loss of generality and for better clarity, assume there are only three processes involved in a deadlock.

Such deadlock can assumedly be created in the following scenario.

1. Process P_a expands tuple $[(d, p_u), (b, p_y), w_{ad}, prov]$ in its p_x-task. Thus, it sends a corresponding message to P_b requesting a p_y-task and asking information about the (d, p_u)-keyed tuple in this p_y-task.
2. Process P_b already has a p_y-task, but cannot reply back at the moment since there is some tuple $[(e, p_v), (c, p_z), w_{be}, prov]$ in the p_y-task, whose w_{be} weight is smaller than the weight of the (d, p_u)-keyed tuple.
   Thus, P_b sends a message to P_c requesting a p_z-task and asking information about the (e, p_v)-keyed tuple in this p_z-task.
3. Process P_c already has a p_z-task, but cannot reply back at the moment since there is some tuple $[(f, p_w), (a, p_x), w_{cf}, prov]$ in the p_z-task, whose w_{cf} weight is smaller than the weight of the (e, p_v)-keyed tuple.
   Thus, P_c sends a message to P_a requesting a p_x-task and asking information about the (f, p_w)-keyed tuple in this p_x-task.

4. Process P_a has an (f, p_w)-keyed tuple in the table of its p_x-task, and this tuple has a *provisional* status. Note that an (f, p_w)-keyed tuple certainly exists in the p_x-task of P_a. This is because otherwise, the via object-state pair of the (f, p_w)-keyed tuple in $P_c.p_z$ would not be (a, p_x).
   On the other hand, process P_a has the p_x-task in *active* status waiting for a reply to the expansion of the (d, p_u)-keyed tuple. This prevents P_a to expand any other tuple including the (f, p_w)-keyed tuple. Hence, it cannot reply back to P_c and the deadlock assumedly occurs.

Now, we show that such a situation cannot happen during the execution of our algorithm.

Since P_a expands tuple $[(d, p_u), (b, p_y), w_{ad}, prov]$ (in the table of the p_x-task), we have that w_{ad} is the smallest weight among the *provisional* tuples of the p_x-task. In particular, $w_{af} \geq w_{ad}$, where w_{af} is the weight of the (f, p_w)-keyed tuple in $P_a.p_x.T$.

Process P_a has to get information about the (d, p_u)-keyed tuple through its neighbor process P_b, which is the via process for that tuple.

By the **Update** thread, we have that
$w_{ad} = w[(a, p_x), (b, p_y)] + w_{bd}$, where $w[(a, p_x), (b, p_y)]$ is the cheapest weight product of a matching automaton transition from p_x to p_y with a database edge from a to b, and w_{bd} is the weight of the (d, p_u)-keyed tuple in $P_b.p_y.T$.

Hence, $w_{af} \geq w_{ad} = w[(a, p_x), (b, p_y)] + w_{bd}$. As P_b selects the tuple keyed by (e, p_v) to expand, we have $w_{bd} \geq w_{be}$. Therefore, it can be concluded that $w_{af} \geq w[(a, p_x), (b, p_y)] + w_{be} = w[(a, p_x), (b, p_y)] + w[(b, p_y), (c, p_z)] + w_{ce}$.

According to the deadlock scenario outlined in the beginning of this proof, P_c tries to expand tuple $[(f, p_w)$, (a, p_x), w_{cf}, $prov]$ of the p_z-task when it is asked for information on the (e, p_v)-keyed tuple. So, $w_{ce} \geq w_{cf}$, and hence,

$$\begin{aligned} w_{af} &\geq w[(a, p_x), (b, p_y)] + w[(b, p_y), (c, p_z)] + w_{cf} \\ &= w[(a, p_x), (b, p_y)] + w[(b, p_y), (c, p_z)] \\ &\quad + w[(c, p_z), (a, p_x)] + w_{af}. \end{aligned}$$

However, recall from Section 2 that the edge weights are positive numbers, and thus the above cannot happen, reaching so a contradiction. □

As mentioned earlier, the algorithm should terminate when each process has a *completed* p_0-task. However, there is the question of how to detect the global termination of our algorithm. This can be done using an algorithm for distributed termination detection. There are many of such algorithms (see [18] for a thorough review) and they can be superimposed into any other distributed algorithm.

5 Complexity

Theorem 6 *The number of messages required for a query evaluation is $2|E|$, where E is the set of edges in the lazy database-query Cartesian product graph.*

Proof. We base our claim on the following facts:

1. Each (traversed) edge in the Cartesian product graph indicates a communication channel between two tasks of two processes which also is indexed by an object-state pair.
2. Only one forward message is needed to cause the creation of a communication channel.
3. Each communication channel is traversed only once, which happens when the tuple keyed by the object-state pair of the channel becomes optimally weighted.

□

The real number of messages ultimately depends on the query selectivity, and in practice one hopes that the lazy Cartesian product size is much smaller than the size of the database (cf. [1]).

Note that if a set of database objects is serviced by a process as opposed to having only one object serviced by a process, then the message complexity will be $2|E'|$, where E' $(E' \subset E)$ is the set of inter–process edges of the lazy Cartesian product.

We note that the above upper bound coincides with the message lower bound of Ramarao and Venkatesan in [26] for the distributed computation of *single-source* shortest paths. However, the messages in [26] have a constant size, while our messages have an $O(|V|)$ size, where V is the set of object-state pairs in the lazy Cartesian product graph. Thus, in terms of $O(1)$ size messages, our algorithm can be considered as having $O(|E| \cdot |V|)$ such messages. On the other hand, our problem is more difficult than the classical *single-source* shortest paths problem of [26].

Remark. One might be tempted to apply instead of our fully distributed algorithm the following semi-distributed approach.

First, collect the whole database in one process only. Then, apply a centralized shortest path algorithm on the Cartesian product of the database and query automaton.

This semi-distributed approach has several shortcomings. First, depending on the selectivity of the queries, large parts of the transmitted database might not be used at all during evaluation, thus resulting in unnecessary communication traffic.

Second, this solution asks from a single process to perform a huge computation which needs also to store the complete database. In other words, the memory requirement for the process performing the computation is at least $|E_{DB}|$, where E_{DB} is the set of edges in database DB.

On the other hand, the memory requirement for each process in our fully distributed algorithm is only $O(|V|)$, where V is the set of object-state pairs in the lazy Cartesian product graph.

6 Soundness and Completeness

In this section, we show the soundness and completeness of Algorithm 1. For the former, we show that each reported query answer is optimally weighted. For the latter, we show that all the query answers are indeed reported. In the following, we present two lemmas and then the main theorem of the section.

Lemma 1. *If there exists a path from (a, p_0) to (c, p_z) in $\mathcal{C}$, then there will be some (c, p_z)-keyed tuple that will be eventually inserted into $P_a.p_0.T$.*

Proof. Suppose that there exists a path π from (a, p_0) to (c, p_z) in $\mathcal{C}$, but the algorithm, during its execution, never inserts some (c, p_z)-keyed tuple into $P_a.p_0.T$. Let π be the sequence $(c_0, p_0), (c_1, p_1), \ldots, (c_n, p_n)$, where $n \geq 1$, $c_0 = a$, $c_n = c$, and $p_n = p_z$. Clearly, $[(c_0, p_0), (c_0, p_0), 0, \mathit{opt}]$ will be inserted into $P_a.p_0.T$ by the Initialization thread.

Let $k \in [1, n-1]$ be the number for which we have that for all $h \in [0, k-1]$ there is some (c_h, p_h)-keyed tuple inserted at some point into $P_a.p_0.T$, but there is never a (c_k, p_k)-keyed tuple inserted into $P_a.p_0.T$.

Clearly, there will be some expansion (in fact only one) of tuple $[(c_{k-1}, p_{k-1}), (\_, \_), \_, \mathit{prov}]$. Now, as (c_{k-1}, p_{k-1}) and (c_k, p_k) are consecutive nodes in π, there exists at least one edge connecting them; (at least) the edge in π.

The expansion of $[(c_{k-1}, p_{k-1}), (\_, \_), \_, \mathit{prov}]$ will trigger a series of request messages all the way to process $P_{c_{k-1}}$ for task p_{k-1}. Process $P_{c_{k-1}}$ will in turn create (if it has not already done so) a p_{k-1}-task and insert an optimal (c_{k-1}, p_{k-1})-keyed tuple into $P_{c_{k-1}}.p_{k-1}.T$. Also, by the task creation, since (c_{k-1}, p_{k-1}) and (c_k, p_k) are connected in $\mathcal{C}$, we have that a $[(c_k, p_k), (\_, \_), \_, \mathit{prov}]$ tuple is as well inserted into $P_{c_{k-1}}.p_{k-1}.T$ (see step 4.b in Algorithm 1). Now, through the back-reply messages, tuple $[(c_k, p_k), (\_, \_), \_, \mathit{prov}]$ will travel and reach process P_a where it is inserted into $P_a.p_0.T$. But this, contradicts our initial supposition.

Thus, for all the nodes (c_i, p_i) in π, where $i \in [0, n]$, we have that some (c_i, p_i)-keyed tuple will be certainly inserted (at some point) in $P_a.p_0.T$. This applies to $(c_n, p_n) = (c, p_z)$ as well, and so, some tuple keyed by (c, p_z) will be eventually inserted into $P_a.p_0.T$. □

From the above lemma and the specification of the Expansion and Update threads, we have that

Corollary 1. *If there exists a path from (a, p_0) to (c, p_z) in $\mathcal{C}$, then there will be eventually a tuple $[(c, p_z), (\_, \_), \_, opt]$ in $P_a.p_0.T$.*

Clearly, there is only one such tuple in $P_a.p_0.T$. Now we show that

Lemma 2. *Let $[(c, p_z), (\_, \_), w, opt]$ be a tuple in $P_a.p_0.T$. Then, w is the weight of a cheapest path going from (a, p_0) to (c, p_z) in $\mathcal{C}$.*

Proof. Let $[(c, p_z), (\_, \_), w, prov]$ be the (c, p_z)-keyed tuple in $P_a.p_x.T$ that gets expanded. By the specification of the Update thread, after receiving the back-reply message corresponding to the expansion, the (c, p_z)-keyed tuple gets an optimal status and by Theorem 4 its weight is w.

Now, let π, with a weight z, be a cheapest path from (a, p_0) to (c, p_z) in $\mathcal{C}$. Then, we claim that $[(c, p_z), (\_, \_), z, prov]$ will exist at some point in $P_a.p_0.T$, eventually expanded, and finally attain an optimal status. From this, our claim will follow as there can be only one (c, p_z)-keyed tuple in $P_a.p_0$, i.e. w will have to be equal to z.

Suppose π has the following nodes: $(c_0, p_0), (c_1, p_1), \ldots, (c_n, p_n)$, where $n \geq 1$, $c_0 = a$, $c_n = c$, $p_n = p_z$. Let w_h, where $h \in [1, n]$, be the weight of the subpath of π from (c_0, p_0) to (c_h, p_h).

Clearly, $[(c_0, p_0), (c_0, p_0), 0, opt]$ will be inserted into $P_a.p_0.T$ by the Initialization thread. Suppose now that $[(c_{h-1}, p_{h-1}), (\_, \_), w_{h-1}, opt]$, for some $h \in [1, n]$, is in $P_a.p_0.T$. By the specification of the Expansion and Update threads, and Theorem 4, we have that (the provisional variant) $[(c_{h-1}, p_{h-1}), (\_, \_), w_{h-1}, prov]$ has been in $P_a.p_0.T$ and at some point has been expanded.

Since (c_{h-1}, p_{h-1}) and (c_h, p_h) are neighbors, reasoning similarly as in the proof of Lemma 1, we have that the expansion of the (c_{h-1}, p_{h-1})-keyed tuple causes, through the corresponding back-reply message, the arrival (in $P_a.p_0$) of tuple $[(c_h, p_h), (\_, \_), w_h, prov]$.

If there is no (c_h, p_h)-keyed tuple in $P_a.p_0.T$, then $[(c_h, p_h), (\_, \_), w_h, prov]$ will be inserted in this table, and preserve weight w_h till the end (becoming eventually an optimal tuple with weight w_h). This is because π and its subpaths are cheapest paths, and thus, there does not exist a cheaper way going from (c_0, p_0) to (c_h, p_h) in $\mathcal{C}$.

On the other hand, if there is already a (c_h, p_h)-keyed tuple in $P_a.p_0.T$, then its weight cannot be less than w_h because, otherwise, we could go from (c_0, p_0) to (c_h, p_h) through a cheaper way than the subpath of π between these two nodes, and this would imply that π is not a cheapest path. Thus, in this case, the arriving tuple $[(c_h, p_h), (\_, \_), w_h, prov]$ will lower the weight of the (c_h, p_h)-keyed tuple in $P_a.p_0.T$ to w_h (if it is not already so).

Concluding, in both cases, $P_a.p_0.T$ will have at some point a $[(c_h, p_h), (\_, \_), w_h, prov]$ tuple, whose weight cannot be lowered any further. Since the algorithm continues until there is no provisional tuple in $P_a.p_0.T$, there will be a moment when the (c_h, p_h)-keyed tuple will get expanded and then attain an optimal status while preserving weight w_h (by Theorem 4).

Inductively, $P_a.p_0.T$ will have at some point a $[(c_n, p_n), (\_, \_), w_n, prov] = [(c, p_z), (\_, \_), z, prov]$ tuple, whose weight cannot be lowered any further. Upon expansion, based on Theorem 4, we will have $[(c, p_z), (\_, \_), z, opt]$ in $P_a.p_0.T$, and this completes our proof. □

Based on all the above, we have that

Theorem 7 *Algorithm 1 is sound and complete.*

Proof.
"*soundness.*" From the definition of $eval(\mathcal{A}, DB)$ we have that the output produced by the algorithm is

$$\{(a, b, r) : [(b, p_y), (\_, \_), r, opt] \in P_a.p_0.T \text{ and } p_y \in F\}.$$

Now, let $[(b, p_y), (\_, \_), r, opt]$ be a tuple in $P_a.p_0.T$ and (a, b, r) be the corresponding produced answer to the given query. From Lemma 2, we have that r is the weight of a cheapest path in $\mathcal{C}$ connecting (a, p_0) to (b, p_y). From Theorem 1, (a, b, r) is an answer to the given query.

"*completeness.*" Let (a, b, r) be an answer to the given query. From Theorem 1, there exists some path from (a, p_0) to (b, p_y) in $\mathcal{C}$, and r is the weight of a cheapest of such paths. From Corollary 1, the existence of some path from (a, p_0) to (b, p_y) in $\mathcal{C}$ means that a tuple $[(b, p_y), (\_, \_), \_, opt]$ will be eventually inserted into $P_a.p_0.T$. From Lemma 2, the exact weight of this tuple will be equal to the weight of a cheapest path from (a, p_0) to (b, p_y) in $\mathcal{C}$, i.e. r. Thus, (a, b, r) will be produced as an answer by the algorithm. □

7 Fault Tolerance

Having a fault-tolerant algorithm is very important in distributed settings that are prone to process failures. Although defunct hardware is rare, fault-tolerance is very prevalent today due to the popular geographically distributed grid systems (see PlanetLab [21]). In such systems, extreme power comes from the participation of numerous machines, whose service in a grid is usually offered during their low intensity periods. As such, grid machines are quite "unreliable" because they can withdraw at any time from a grid computation in order to perform other "duties" they are primarily intended for.

In this section, we show how to extend Algorithm 1 in order to be resilient against process failures. We assume that even if a process fails, the corresponding database object still exists. This assumption is the norm in database applications, where the data lives longer than the processes that access it.

Let DB' be the subset of database DB serviced by the remaining alive processes at the end of the query evaluation. After each failure, we will have a smaller database being serviced by the alive processes. Since the query evaluation is not started from the scratch on DB', but is continually evaluated on a series of databases which are supersets of DB', we can obtain more and better-weighted answers than what we would get on DB' only.

To make formal the description of the query answers returned by our fault-tolerant algorithm, we first present the following definition.

Let A and B be sets of object-object-weight triples, i.e. $A, B \subseteq V \times V \times \mathbb{R}^+$. Then, we say that A is *superior* to B, denoted by $A \sqsupseteq B$, if $(a, b, r) \in B$ implies that $(a, b, r') \in A$, and $r' \leq r$.

Now, our fault-tolerant algorithm will compute a set $eval(\mathcal{A}, DB)$ of triples. After the description of the algorithm, we will show that

$$SWAns(\mathcal{A}, DB) \sqsupseteq eval(\mathcal{A}, DB) \sqsupseteq SWAns(\mathcal{A}, DB').$$

Thus, our algorithm produces better answers than restarting the computation from the scratch on DB', while saving time by not wasting the computation done so far.

Furthemore, we are able to clearly identify the answers which happen to be optimal with respect to DB, i.e. belong to $SWAns(\mathcal{A}, DB)$.

In the following we provide a description of our fault-tolerant algorithm.

We assume that the network infrastructure provides a fault-detection service, in which any process can subscribe in order to be informed of the failure of the processes of interest. Such fault-detection service might be as simple as a ping command, and its existence is the common assumption in constructing fault-tolerant algorithms (cf. [17]). We make each process subscribe to the fault-detection service and be informed of the health of its neighbors only.

Now, we are ready to present our fault-tolerant algorithm. First, we introduce an additional status value for the tuples. This new value is *gone*, and is given to a tuple when the process of its key or via component has failed. Thus, the algorithm deals now with tuples whose status can be *optimal*,

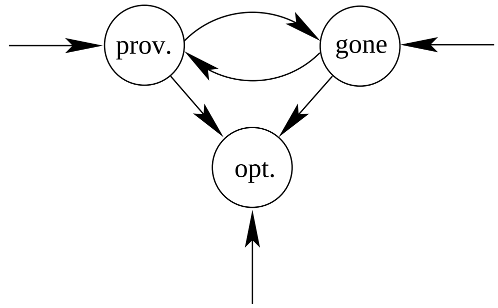

Fig. 5. Tuple Status Diagram.

provisional, or *gone*. Figure 5 illustrates these three different status values and the transitions among them.

A tuple might start with one of the three possible status values. If a tuple is (or becomes) *optimal* it preserves this status till the end of the algorithm. On the other hand, a tuple with a *provisional* status may later have a status change to *optimal* or *gone*. Similarly, a tuple with a *gone* status may later have a status change to *optimal* or *provisional*. In the end, only tuples with an *optimal* or *gone* status will be in the tables of the p_0-tasks across processes.

Each process keeps track of its failed neighbors in a list. Suppose that a process P_a detects a failed neighbor, say P_b. Thus, P_a first adds a failure record for process P_b in its failed-neighbor list. Then, P_a changes the status of all *provisional* tuples having b in their key or via component to *gone* in all of its tasks. In our fault tolerant algorithm, we assign these jobs to a new thread called **Failure Detection**.

Regarding the other threads, they change as follows.

In the **Initialization** thread, we set to a *gone* status all the tuples having their key component refer to a failed process. Since in this thread, the process of the key is a neighbor process, we can easily determine its health by consulting the list of failed neighbors. The same is also done in the **Task Creation** thread when the table of a new task is being initialized.

The **Expansion** thread remains unchanged and continues to expand only *provisional* tuples.

In the **Reply** thread, we make the process send replies also in the case when it is asked to provide information about tuples with a *gone* status.

In the **Update** thread, the tuple under expansion might get an *optimal* or *gone* status. Also in this thread, a *provisional* or *gone* tuple carried in the reply message can relax a *provisional* or *gone* tuple with the same key in the table of the receiver task. We note that, the incoming *provisional* tuples can relax *provisional* or *gone* tuples. Similarly, the incoming *gone* tuples can relax *provisional* or *gone* tuples. Thus, as shown in Figure 5, we have transitions from a *provisional* status to a *gone* status and vice-versa.

In the modified **Reply** and **Update** threads, the *gone*-status tuples are treated as being *optimal* ones. These tuples are backpropagated in a similar way in reply messages causing along the way, through the **Update** threads, other tuples (in other processes) to attain a *gone* status. For example, suppose as above that P_a detects the failure of its neighbor P_b. The neighbor processes of P_a, having a $(b, \_)$-keyed *provisional* tuple with an $(a, \_)$ as their via component, will eventually assign a *gone* status to this tuple. Specifically, this will happen when such tuples are expanded and P_a is asked for its $(b, \_)$-keyed tuple.

We emphasize that a *gone* status prevents tuples from being expanded by the process. Nevertheless, the weight and via of a *gone* status tuple might be updated to some lower values as an effect of the expansion of some *provisional* tuple in the same task. Such an update might also change the status of the tuple, as we explained above, from *gone* to *provisional*, thus making the expansion of the tuple possible again. Also, through such updates, a *gone* status tuple can even attain an *optimal* status.

Finally, we note that the message complexity of our extended algorithm is the same as that of Algorithm 1.[4]

Formally, our fault tolerant algorithm is given in the following, where we emphasize only the changes and extensions to Algorithm 1. The parts that remain unchanged are shown in gray.

Algorithm 2

Input:

1. A database DB. For simplicity we assume that each database object, say a, is being serviced by a dedicated process for that object P_a.
2. A query WFSA $\mathcal{A} = (P, \Delta, \tau, p_0, F)$.

Output: Set $eval(\mathcal{A}, DB)$ which will be characterized in Theorem 8.
Method:

1. Each process subscribes to the fault-detection service.

2. Each process P_a creates a list, called $FailList_a$, and initializes it to $\emptyset$.

3. **Initialization**: Each process P_a creates a task $\langle p_0, \mathit{passive}, \{\ldots\}\rangle$ for itself. The table $\{\ldots\}$ (referred to as $P_a.p_0.T$) is initialized as follows:
   (a) insert tuple $[(a, p_0), (a, p_0), 0, \mathit{opt}]$, and
   (b) For each edge-transition match,
      (a, R, r, b) in DB and
      (p_0, R, k, p) in $\mathcal{A}$,
      insert tuple $[(b, p), (b, p), k \cdot r, \mathit{prov}]$
      (if there are multiple $(a, \_, \_, b) - (p_0, \_, \_, p)$ edge-transition matches, then the cheapest weight product is considered.)
   (c) For each tuple in the task,
      if the process of its key component is found in $FailList_a$,
      then change the status of the tuple to *gone*.

   If at point (b) there is no edge-transition match, then make the status of the p_0-task *completed*.

4. Concurrently execute all the five following threads at each process in parallel until termination is detected. [For clarity, we describe the threads at two processes, P_a and P_b.]

5. **Expansion**: [At process P_a]
   (a) Select a *passive* p_x-task for processing. Make the status of the task *active*.
   (b) Select the cheapest *provisional*-status tuple, say $[(c, p_z), (b, p_y), w, \mathit{prov}]$ from table $P_a.p_x.T$.
   (c) Request P_b, with respect to state p_y, to provide information about (c, p_z).
      For this, send a message $\langle p_y, [p_x, (c, p_z), w_{ab}]\rangle$ to P_b, where w_{ab} is the cost of going from (a, p_x) to (b, p_y), which is equal to the weight of the (b, p_y)–keyed tuple in $P_a.p_x.T$.
   (d) Sleep, with regard to p_x-task, until the reply message for (c, p_z) comes from P_b.

6. **Task Creation**: [At process P_b]
   Upon receiving a message $\langle p_y, [p_x, (c, p_z), w_{ab}]\rangle$ from P_a:
   if there is not yet a p_y-task,
   then create a task $\langle p_y, \mathit{passive}, \{\ldots\}\rangle$ and initialize its table similarly as in the first phase.
      That is,
      (a) insert tuple $[(b, p_y), (b, p_y), 0, \mathit{opt}]$, and

[4] We do not consider the elementary messages of the fault-detection infrastructure.

(b) For each edge-transition match,
(b, R, r, d) in DB and
(p_y, R, k, p_u) in $\mathcal{A}$,
insert tuple $[(d, p_u), (d, p_u), k \cdot r, prov]$
(if there are multiple $(b, \_, \_, d)$–$(p_y, \_, \_, p_u)$ edge-transition matches, then the cheapest weight product is considered.)

(c) For each tuple in the task,
if the process of the key component is in $\mathit{FailList}_b$,
then change the status of the tuple to *gone*.

Also, establish a virtual communication channel with P_a. This channel relates the p_y-task of P_b with the p_x-task of P_a. Further, it is indexed by (c, p_z) and is weighted by w_{ab} (the weight included in the received message).

else [P_b has already a p_y-task.] Do not create a new task, but only establish a communication channel with P_a as described above.

7. **Reply**: [At process P_b]
When in the p_y-task, the tuple $[(c, p_z), (\_, \_), \_, \_]$ is or becomes optimally weighted, or if it has *gone* status, *reply back* to all the neighbor processes, which have sent a task requesting message $\langle p_y, [\_, (c, p_z), \_] \rangle$ to P_b.
For example, P_b sends to such a neighbor, say P_a, through the corresponding communication channel, the message $\langle P_b.p_y.T^* \rangle$, which is table $P_b.p_y.T$ after adding the channel weight to the weight of each tuple.

8. **Update**: [At process P_a] Upon receiving a reply message $\langle P_b.p_y.T^* \rangle$ from a neighbor P_b w.r.t. the expansion of a (c, p_z)-keyed tuple in table $P_a.p_x.T$ do:
   (a) Change the status of (c, p_z)-keyed tuple to the status of the same keyed tuple in $P_b.p_y.T^*$ [5].
   (b) For each tuple $[(d, p_u), (\_, \_), v, s]$[6] in $P_b.p_y.T^*$, which has a smaller weight (v) than a same keyed tuple $[(d, p_u), (\_, \_), \_, s']$[7] in $P_a.p_x.T$, replace the latter by $[(d, p_u), (b, p_y), v, s]$.
   (c) Add to $P_a.p_x.T$ all the rest of the $P_b.p_y.T^*$ tuples, i.e., those which do not have corresponding same-key tuples in $P_a.p_x.T$.
   Also, change the via component of these tuples to be (b, p_y).
   (d) **if** the p_x-task does not have anymore *provisional* tuples,
   then make its status *completed*.
   If $p_x = p_0$, then report that all query answers from P_a have been computed.
   else make the status of the p_x-task *passive*.

9. **Failure Detection**: [At process P_a upon detecting failure of a neighbor process P_b]
   (a) Add an entry in $\mathit{FailList}_a$ for process P_b.
   (b) For each *provisional* tuple in each task of P_a,
   if the key or via component is $(b, \_)$,
   then change the status of the tuple to *gone*.

□

As for Algorithm 1, the termination happens when each process has a *completed* p_0-task. Detecting this can be done by using an algorithm for fault-tolerant distributed termination detection (cf. [17]). Finally upon termination, we set

$$\mathit{eval}(\mathcal{A}, DB) = \{(a, b, r) : [(b, p_y), (\_, \_), r, s)] \in P_a.p_0.T, p_y \in F \text{ and } s \in \{opt, gone\}\}.$$

[5] This status is either *optimal* or *gone*.
[6] s can be *prov* or *gone*.
[7] s' can be *prov* or *gone*.

Let $\mathcal{C}$ and $\mathcal{C}'$ be the Cartesian products of databases DB and DB' with query automaton $\mathcal{A}$. We show the following two lemmas.

Lemma 3. *If there exists $(a, b, r) \in eval(\mathcal{A}, DB)$ then there exists a path, in $\mathcal{C}$, from (a, p_0) to (b, p_y), where p_y is a final state of $\mathcal{A}$.*

Proof. By the definition of $eval(\mathcal{A}, DB)$, if $(a, b, r) \in eval(\mathcal{A}, DB)$, then there will exist a tuple $[(b, p_y), (\_, \_), r, s]$, where $s \in \{opt, gone\}$, in $P_a.p_0.T$.

Now, by the specification of the Intialization, Expansion and Update threads, it is clear that if there is no path from (a, p_0) to (b, p_y) in $\mathcal{C}$, then a $[(b, p_y), (\_, \_), r, s]$ tuple would never be in $P_a.p_0.T$, and this would be a contradiction. □

Lemma 4. *Let (a, p_0) and (b, p_y) be connected in $\mathcal{C}'$, and let r be the weight of a cheapest path between these two nodes (in $\mathcal{C}'$). Then, there exists a triple (a, b, r') in $eval(\mathcal{A}, DB)$, and $r' \leq r$.*

Proof. Since (a, p_0) and (b, p_y) are connected in $\mathcal{C}'$, they were never disconnected during the execution of the algorithm, and thus, Lemma 1 holds (with respect to Algorithm 2). Similar to Corollary 1, we have that eventually there will exist tuple $[(b, p_y), (\_, \_), r', s]$ in $P_a.p_0.T$, where $s \in \{opt, gone\}$. Value r' is the weight of the cheapest paths that Algorithm 2 could explore, and clearly this set of paths is a superset of paths from (a, p_0) to (b, p_y) in $\mathcal{C}'$. Hence, $r' \leq r$. □

Now, we show that

Theorem 8 *$SWAns(\mathcal{A}, DB) \sqsupseteq eval(\mathcal{A}, DB) \sqsupseteq SWAns(\mathcal{A}, DB')$.*

Proof.

"$SWAns(\mathcal{A}, DB) \sqsupseteq eval(\mathcal{A}, DB)$." Let $(a, b, r) \in eval(\mathcal{A}, DB)$. By Lemma 3, this means that there exists a path π (in $\mathcal{C}$) from (a, p_0) to (b, p_y), where p_y is a final state in $\mathcal{A}$. Since there are process failures, path π might not be a cheapest one in $\mathcal{C}$ going from (a, p_0) to (b, p_y). Let π' be a cheapest path in $\mathcal{C}$ with a weight of r'. Clearly, $r' \leq r$, and by the definition of $SWAns(\mathcal{A}, DB)$, $(a, b, r') \in SWAns(\mathcal{A}, DB)$.

"$eval(\mathcal{A}, DB) \sqsupseteq SWAns(\mathcal{A}, DB')$." Let $(a, b, r) \in SWAns(\mathcal{A}, DB')$. By Theorem 1, (a, p_0) is connected to (b, p_y) in $\mathcal{C}'$, and the weight of the cheapest paths between these two nodes is r.

By Lemma 4, there exists a tuple (a, b, r') in $eval(\mathcal{A}, DB)$, and $r' \leq r$, and this concludes our proof. □

Now, we further characterize the produced query answers. Suppose that upon termination, in the table of $P_a.p_0$, we have some tuples with a *gone* status. Let $[(c, p_z), (\_, \_), w_a, gone]$ be the cheapest of those tuples.

We classify the tuples in $P_a.p_0.T$ as

1. tuples with smaller (or equal) weight than w_a, and
2. tuples with greater weight than w_a.

We can show that the tuples in the first set have weights which are optimal with respect to the original database DB, i.e. they belong to $SWAns(\mathcal{A}, DB)$.

For this, observe that at the end of the algorithm, since tuple $[(c, p_z), (\_, \_), w_a, gone]$ is the cheapest of the tuples with a *gone* status, the tuples with weight less than w_a are all *optimal*. They have obtained this status by the expansion of *provisional* tuples, which at the time of expansion have been the cheapest ones among *provisional* and *gone* tuples. Since there is no gone status tuple with a weight smaller than the weight of these tuples, all the cheaper paths possibly reaching the nodes corresponding to these tuples have been already explored. Thus, reasoning similarly as in Section 6, these tuples attain the cheapest weight obtainable in the original DB.

We can also observe that weight w_a of the cheapest *gone* status tuple in $P_a.p_0.T$ is optimal considering the original database DB. This is because a *gone* status tuple has been a *provisional* one earlier, and thus, the cheapest *gone* tuple would have been the next tuple to be expanded if there had been no failure in the corresponding path. By Theorem 4, the weight of this tuple is optimal with respect to the original database.

Clearly, the w_a values, for $a \in V$, can be produced as additional output in order to characterize the query answers as the above discussion suggests.

7.1 Intermittent Process Failures

Here, we discuss how Algorithm 2 can be extended to handle a dynamic scenario, where the failed processes can come back to the computation.

Let us assume that when a failed process, say P_b, comes back to the computation it has no information from the past. Therefore, it creates task p_0, initializes its failed-neighbor list, and starts expanding tuples in its p_0-task.

Each neighbor of process P_b, say P_a, realizing that P_b is back, continues processing as follows:

1. P_a deletes the P_b-entry in $FailList_a$ and then changes, in all its tasks, the *gone* status of the tuples having b in their key or via component to *provisional*.
2. P_a will possibly cancel the current expansion should some smaller weighted *provisional* tuple become available due to a status change from *gone* to *provisional*. The eventual back-reply message corresponding to the cancelled expansion is ignored.
3. Upon receival of some back-reply message, due to expansion of a tuple, say $[(c,p),(\_,\_),w_{ac},prov]$, P_a will not only update *provisional* tuples as before, but it will also update (if applicable) the *optimal* tuples having weights greater than w_{ac}. This is because these *optimal* tuples have an optimal weight in a subset of original DB. By having P_b back to the computation, we can (possibly) lower the weight of such *optimal* tuples.
4. P_a propagates the news about P_b becoming alive again through neighboring relationships. In turn, all processes receiving the news about P_b behave exactly as P_a.

Finally, we remark that the behavior of the P_b's neighbors remains the same as described above even if P_b does have information from the past. The only difference is that, in this case, P_b will continue processing using its stored information.

8 Conclusions

We have presented a fully distributed algorithm for answering generalized regular path queries on database graphs. We have discussed in detail the complexity of our algorithm and shown that the number of messages is proportional to the number of inter-process edges in the lazy Cartesian product graph of the database and query.

Then, we presented a resilient algorithm against process failures. This algorithm can tolerate any number of process losses and possesses two desirable properties:

1. It produces answers which are at least as good as those obtainable on the remaining live processes.
2. It does not need additional, algorithm-specific, messages to achieve resilience against process losses.

Given that RPQs are an important building block of virtually all the query languages for semistructured graph-data, we believe that our work is an important step towards effective and efficient solutions for distributed and resilient computation of queries on semistructured data.

Acknowledgment. We would like to thank an anonymous reviewer for providing very constructive comments.

References

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. Allauzen C., M. Mohri. A Unified Construction of the Glushkov, Follow, and Antimirov Automata. *Proc. of MFCS'06*.
3. Abiteboul S., V. Vianu. Regular Path Queries with Constraints. *J. of Computing and System Sciences* 58 (3), pp. 428–452, 1999.
4. Awerbuch B. Optimal distributed algorithms for minimum-weight spanning tree, counting, leader election and related problems. *Proc. of STOC'87*.
5. Buneman P., G. Cong, W. Fan, and A. kementsietsidis. Using partial evaluation in distributed query evaluation. *VLDB'06*.
6. Calvanese D., G. Giacomo, M. Lenzerini, and M. Y. Vardi. Answering Regular Path Queries Using Views. *Proc. of ICDE'00.*
7. Calvanese D., G. Giacomo, M. Lenzerini, and M. Y. Vardi. Reasoning on Regular Path Queries. *SIGMOD Record* 32 (4), pp. 83–92, 2003.
8. Calvanese D., G. Giacomo, M. Lenzerini, and M. Y. Vardi. View-based Query Processing: On the Relationship between Rewriting, Answering and Losslessness. *Proc. of ICDT '05*.
9. Cong G., W. Fan, and A. Kementsietsidis. Distributed query evaluation with performance guarantees. *Proc. of SIGMOD'07*.
10. Consens M. P, A. O. Mendelzon. GraphLog: A Visual Formalism for Real Life Recursion. *Proc of PODS'90*.
11. Flesca S., F. Furfaro, and S. Greco. Weighted Path Queries on Web Data. *Proc. of WebDB '01*.
12. Flesca S., F. Furfaro, and S. Greco. Weighted Path Queries on Semistructured Databases. *Inf. Comput.* 204 (5), pp. 679–696, 2006.
13. Grahne G., and A. Thomo. Regular Path Queries Under Approximate Semantics. *Ann. Math. Artif. Intell.* 46 (1–2), pp. 165–190, 2006.
14. Grahne G., A. Thomo, and W. Wadge. Preferentially Annotated Regular Path Queries. *Proc. of ICDT'07*.
15. Haldar S. An "All Pairs Shortest Paths" Distributed Algorithm Using $2n^2$ Messages. *J. of Algorithms,* 24 (1), pp. 20–36, 1997.
16. Hopcroft J. E., and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.
17. Lai H. T., and F. L. Wu. An (N-1)-Resilient Algorithm for Distributed Termination Detection. *IEEE Trans. Parallel Distrib. Syst.,* 6 (1), pp. 63–78, 1995.
18. Matocha J., and T. Camp. A Taxonomy of Distributed Termination Detection Algorithms. *J. of Systems and Software,* 43 (3), pp. 207–221, 1998.
19. Mendelzon A. O., and P. T. Wood, Finding Regular Simple Paths in Graph Databases. *SIAM J. Comp.* 24 (6), pp. 1235–1258, 1995.
20. Miao Z., D. Stefanescu, A. Thomo. Grid-Aware Evaluation of Regular Path Queries on Spatial Networks. *Proc. of AINA'07.*
21. Planet–Lab: www.planet-lab.org
22. Shoaran M., A. Thomo. Distributed Multi-source Regular Path Queries *Proc. of ISPA'07 Workshops.*
23. Stefanescu D., A. Thomo, and L. Thomo. Distributed Evaluation of Generalized Path Queries *Proc. of SAC'05*.
24. Stefanescu D., A. Thomo. Enhanced Regular Path Queries on Semistructured Databases. *Proc. of QLQP'06*.
25. Suciu D., Distributed Query Evaluation on Semistructured Data. *ACM Trans. on Database Systems,* 27 (1), pp. 1–62, 2002.
26. Ramarao S. V. K., S. Venkatesan. The Lower Bounds on Distributed Shortest Paths. *Inf. Process. Lett.*, 48 (3), pp. 145–149, 1993.
27. TIGER: Topologically Integrated Geographic Encoding and Referencing system, US Census Bureau http://www.census.gov/geo/www/tiger
28. Vardi M. Y. A Call to Regularity. *Proc. PCK50 - Principles of Computing & Knowledge, Paris C. Kanellakis Memorial Workshop '03*, pp. 11.