Supporting Queries Spanning Across Phases of Evolving Artifacts using Steiner Forests

Siarhei Bykau
University of Trento
bykau@disi.unitn.eu

John Mylopoulos
University of Trento
jm@disi.unitn.eu

Flavio Rizzolo
Carleton University
flavio@scs.carleton.ca

Yannis Velegrakis
University of Trento
velgias@disi.unitn.eu

ABSTRACT

The problem of managing evolving data has attracted considerable research attention. Researchers have focused on the modeling and querying of schema/instance-level structural changes, such as, addition, deletion and modification of attributes. Databases with such a functionality are known as temporal databases. A limitation of the temporal databases is that they treat changes as independent events, while often the appearance (or elimination) of some structure in the database is the result of an evolution of some existing structure. We claim that maintaining the causal relationship between the two structures is of major importance since it allows additional reasoning to be performed and answers to be generated for queries that previously had no answers.

We present here a novel framework for exploiting the evolution relationships between the structures in the database. In particular, our system combines different structures that are associated through evolution relationships into virtual structures to be used during query answering. The virtual structures define “possible” database instances, in a fashion similar to the possible worlds in the probabilistic databases. The framework includes a query answering mechanism that allows queries to be answered over these possible databases without materializing them. Evaluation of such queries raises many interesting technical challenges, since it requires the discovery of Steiner forests on the evolution graphs. On this problem we have designed and implemented a new dynamic programming algorithm with exponential complexity in the size of the input query and polynomial complexity in terms of both the attribute and the evolution data sizes.

Categories and Subject Descriptors

H.m [**Information Systems**]: Miscellaneous

General Terms

Algorithms, Performance

Keywords

Data evolution, Probabilistic Databases; Graph Algorithms

1. INTRODUCTION

Advances in information and telecommunication technologies of the last two decades have allowed organizations and individuals alike to develop large scale data collections and make them available on-line. These collections are often about entities that persist over time and the changes that occur to their attributes/relationships. Considerable effort has gone toward the development of advanced solutions for managing the evolution of schemas[1, 22, 21], data [5, 23, 3, 19] and schema transformations [27, 26, 13]. Temporal databases are one of the outcomes of this effort, where the notion of versioning has been central [4, 6]. In temporal databases, users have the ability to access and query snapshots of the data at different points in time.

Unfortunately, such work fails to capture the full spectrum of evolutionary phenomena. Specifically, those approaches are founded on the assumption that the nature of each real world entity represented in the database persists over time, e.g., students are added, modified, and eventually deleted, but never become professors, with a direct link between the student tuple and the professor one. As such, evolution amounts only to temporal changes of attributes/relationships [1, 5, 21]. Evolution of an entity that spans different concepts (e.g., student to professor, research lab to independent corporate entity) are unaccounted for. And so are evolution phenomena where one entity is split into several (e.g., Germany splitting into East and West Germany at the end of WW II), or the other way around (e.g., East and West Germany amalgamating into one entity at the end of the Cold War). The result of this is that historical queries, such as "give me all the heads-of-state of Germany between 1800 and 2000" are hard to deal with, as they essentially require hand-coding of the history of Germany into several queries to be processed separately. Note that this may look similar to terminology evolution [25], i.e., using different terms to describe the same real world entity at different points in time, but it actually goes far beyond that.

This form of evolution finds a natural fit in Dataspace Systems [10, 16] that are anchored on the notion of an entity. Such entities may split/merge or otherwise evolve during their lifetime. Modeling and supporting evolution relationships for historical query processing finds many additional real-world applications. For instance, modern historians will be able to model and study the chains of human achievements and developments, e.g., how the concept of biotechnology evolved from its beginnings as an agricultural technology to the current notion that is coupled to genetics and molecular biology. Educators will better track how courses evolve and how the material and educational objectives for a course are ”transferred" and become associated with another. Finally, biologists will become more effective in studying the evolution of a

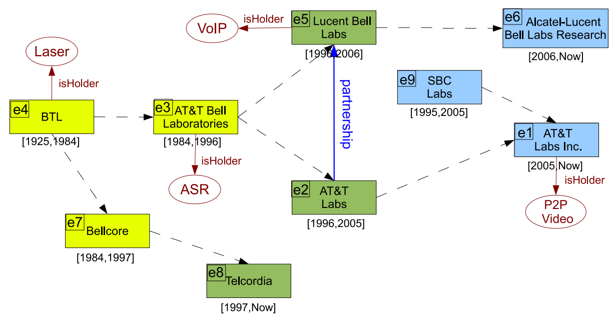


Figure 1: The history of the AT&T Labs.

species by querying the dependencies and changes that have taken place since the beginning of life on earth.

We have previously presented a formal foundation for modeling evolution [24]. This paper focuses on the query facilities required for our proposal and on algorithmic and performance issues. Specifically, our contributions in this paper are as follows: (i) we formally describe new semantics of query answering in which answers are returned from virtual instances constructed on-the-fly by merging series of entities representing different evolution phases of a real-world object into one; (ii) we introduce the concept of Steiner forest as a mean of finding the optimal mergings that need to be done to answer a query and devise a dynamic programming algorithm for solving it; (iii) we present an efficient indexing technique that facilitates the above queries, and we show effective evaluation methods; (iv) we experimentally evaluate our findings on some large publicly available dataset.

The remainder of the paper is structured as follows. Section 2 provides a motivating example and sketches our solution. Sections 3 presents our modeling framework and formally defines queries and evaluations. Section 4 shows the query evaluation strategy and Section 5 describes the Steiner forest algorithm. The optimization techniques are presented in Section 6. The experimental evaluation results are shown in Section 7. Finally, we conclude in Section 8.

2. MOTIVATING EXAMPLE

Consider AT&T, a company that over the years has gone through a large number of break-ups, merges and acquisitions. Its famous Bell Labs where many great innovations took place, had a similar fate. It was founded in 1925 under the name Bell Telecommunication Laboratories (BTL). In 1984, it was split into Bellcore (to become Telcordia in 1997) and AT&T Bell Laboratories. The latter existed until 1996 when it was split into Bell Labs, that was bought by Lucent, and to AT&T Labs. The Lucent Bell Labs became Alcatel-Lucent Bell Labs Research in 2006 due to the take-over of Lucent by Alcatel. Furthermore, due to the take-over of AT&T by SBC in 2005, the AT&T Labs were merged with the SBC Labs to form the new AT&T Inc. Labs. Despite being research labs of different legal entities, Lucent and AT&T Labs have actually maintained a special partnership relationship. All the different labs have produced a large number of inventions, as the respective patents can demonstrate. Examples of such inventions are the VoIP (Voice over Internet Protocol), the ASR (Automatic Speech Recognition), the P2P (Peer-to-Peer) Video and the laser. A graphical illustration of the above can be found in Figure 1 where the labs are modeled by rectangles and the patents by ovals.

Assume now a temporal database that models the above information as illustrated in Figure 1, and consider a user who is interested in finding the lab that invented the laser and the ASR patent. It is true that these two patents have been filed by two different labs, the AT&T Bell Labs and the AT&T Labs Inc. Thus, the query will return no results. However, it can be noticed that the latter entity is an evolution of the former. It may be the case that the user does not have the full knowledge of the way the labs have changed or in her own mind, the two labs are still considered the same. We argue that instead of expecting from the user to know all the details of the evolution granularity and the way the data has been stored, which means that the user's conceptual model should match the one of the database, we'd like the system to try to match the user's conceptual model. This means that the system should have the evolution relationships represented explicitly and take them into account when evaluating a query. In particular, we want the system to treat the AT&T Bell Labs, the AT&T Labs Inc, and the AT&T Labs as one unified (virtual) entity. That unified entity is the inventor of both the laser and the ASR, and should be the main element of the response to the user's query.

Of course, the query response is based on the assumption that the user did not indend to distinguish between the three aforementioned labs. Since this is an assumption, it should be associated with some degree of confidence. Such a degree can be based, for instance, on the number of labs that had to be merged in order to produce the answer. A response that involves 2 evolution-related entities should have higher confidence than one involving 4.

As a similar example, consider a query asking for all the partners of AT&T Labs Inc. Apart from those explicitly stated in the data (in the specific case, none), a traversal of the history of the labs can produce additional items in the answer, consisting of the partners of its predecessors. The further this traversal goes, the less the likely it is that this is what the user wanted; thus, the confidence of the answer that includes the partners of its predecessors should be reduced. Furthermore, if the evolution relationships have also an associated degree of confidence, i.e., less than 100% certainty, the confidence computation of the answers should take this into consideration as well.

3. DATA MODEL

We adopt a *concept model* [7] that is gaining popularity in many areas including dataspaces [10]. Its fundamental component is the *entity* which is used to model a real world object. An entity is a data structure consisting of a unique identifier and a set of attributes. Each attribute has a name and a value. The value of an attribute can be an atomic value or an entity identifier. More formally, assume the existence of an infinite set of entity identifiers $\mathcal{O}$, an infinite set of names $\mathcal{N}$ and an infinite set of atomic values $\mathcal{V}$.

DEFINITION 3.1. *An* attribute *is a pair* $\langle n,v \rangle$*, with* $n \in \mathcal{N}$ *and* $v \in \mathcal{V} \cup \mathcal{O}$*. Attributes for which* $v \in \mathcal{O}$ *are specifically referred to as* associations*. Let* $\mathcal{A} = \mathcal{N} \times \{\mathcal{V} \cup \mathcal{O}\}$ *be the set of all the possible attributes. An* entity *is a tuple* $\langle id,\ A \rangle$ *where* $A \subseteq \mathcal{A}$*, is finite, and* $id \in \mathcal{O}$*. The id is referred to as the* entity identifier *while the set* A *as the set of* attributes *of the entity.* ∎

We will use the symbol $\mathcal{E}$ to denote the set of all possible entities that exist and we will also assume the existence of a Skolem function Sk [15]. Recall that a Skolem function is a function that provides a unique different value for two different arguments. Each entity is uniquely identified by its identifier, thus, we will often use the terms *entity* and *entity identifier* interchangingly if there is no risk of confusion. A *database* is a collection of entities, that is closed in terms of associations between the entities.

DEFINITION 3.2. *A* database *is a finite set of entities $E \subseteq \mathcal{E}$ such that for each association $\langle n, e' \rangle$ of an entity $e \in E$: $e' \in E$.* ∎

As a query language we adopt a datalog style language. A query consists of a head and a body. The body is a conjunction of atoms. An atom is an expression of the form $e(n_1{:}v_1,\ n_2{:}v_2, \ldots, n_k{:}v_k)$ or an arithmetic condition such as $=$, $\leq$, etc. The head is always a non-arithmetic atom. Given a database, the body of the query is said to be true if all its atoms are true. A non-arithmetic atom $e(n_1{:}v_1,\ n_2{:}v_2,\ \ldots,\ n_k{:}v_k)$ is true if there is an entity with an identifier e and attributes $\langle n_i, v_i \rangle$ for every $i{=}1..k$. When the body of a query is true, the head is also said to be true. If a head $e(n_1{:}v_1,\ n_2{:}v_2, \ldots, n_k{:}v_k)$ is true, the answer to the query is an entity with identifier e and attributes $\langle n_1{:}v_1 \rangle, \langle n_2{:}v_2 \rangle, \ldots, \langle n_k{:}v_k \rangle$.

The components e, n_i and v_i, for $i{=}1..k$ of any atom in a query can be either a constant or a variable. Variables used in the head or in arithmetic atoms must also be used in some non-arithmetic atom in the body. If a variable is used at the beginning of an atom, it is bound to entity identifiers. If the variable is used inside the parenthesis but before the ":" symbol, it is bound to attribute names, and if the variable is in the parenthesis after the ":" symbol, it is bound to attribute values. A variable assignment in a query is an assignment of its variables to constants. A *true assignment* is an assignment that makes the body of the query true. The answer set of a query involving variables is the union of the answers produced by the query for each true assignment.

EXAMPLE 3.3. *Consider the query:*
$\$x(isHolder{:}\$y){:}\text{-}\$x(name{:}'AT\&TLabsInc.', isHolder{:}\$y)$
that looks for entities called "AT&T Labs Inc." and are holders of a patent. For every such entity that is found, an entity with the same identifier is produced in the answer set and has an attribute isHolder with the patent as a value. ∎

In order to model evolution we need to model the lifespan of the real world objects that the entities represent and the evolution relationship between them. For the former, we assume that we have a temporal database, i.e., each entity is associated to a time period; however, this is not critical for this work so we will omit that part from the following discussions. To model the evolution relationship, on the other hand, we consider a special association that we elevate into a first-class citizen in the database. We call this association an *evolution relationship*. Intuitively, an evolution relationship from one entity to another is an association indicating that the real world object modeled by the latter is the result of some form of evolution of the object modeled by the former. In Figure 1, the dotted lines between the entities illustrate evolution relationships. A database with evolution relationships is an *evolution database*.

DEFINITION 3.4. *An* evolution database *is a tuple $\langle E,\Omega \rangle$, such that $\langle E \rangle$ is a database and Ω is a partial order relation over E. An* evolution relationship *is every association $(e_1, e_2) \in \Omega$.* ∎

Given an evolution database, one can construct a directed acyclic graph by considering as nodes the entities and as edges its evolution relationships. We refer to this graph as the *evolution graph* of the database.

Our proposal is that entities representing different evolution phases of the same real world object can be considered as one for query answering purposes. To formally describe this idea we introduce the notion of *coalescence*. Coalescence is defined only on entities that are connected through a series of evolution relationships; the coalescence of those entities is a new entity that replaces them and has as attributes the union of their attributes (including associations).

DEFINITION 3.5. *Given an evolution database $\langle E,\Omega \rangle$, The* coalescence *of two entities $e_1{:}\langle id_1,A_1 \rangle$, $e_2{:}\langle id_2,A_2 \rangle \in E$, connected through an evolution relationship ev is a new evolution database $\langle E',\Omega' \rangle$ such that $\Omega'{=}\Omega{-}ev$ and $E'{=}(E - \{e_1, e_2\}) \cup \{e_{new}\}$, where $e_{new}{:}\langle id_{new},A_{new} \rangle$ is a new entity with a fresh identifier $id_{new}{=}Sk(id_1, id_2)$ and $A_{new}{=}A_1{\cup}A_2$. Furthermore, each association $\langle n,id_1 \rangle$ or $\langle n,id_2 \rangle$ of an entity $e{\in}E$, is replaced by $\langle n,id_{new} \rangle$. The relationship between the two databases is denoted as $\langle E,\Omega \rangle \xrightarrow{ev} \langle E',\Omega' \rangle$* ∎

The Skolem function that we have mentioned earlier defines a partial order among the identifiers, and this partial order extends naturally to entities. We call that order *subsumption*.

DEFINITION 3.6. *An identifier id_1 is said to be* subsumed *by an identifier id, denoted as $id_1 \dot{\prec} id$ if there is some identifier $id_x{\neq}id$ and $id_x{\neq}id_1$ such that $id{=}Sk(id_1, id_x)$. An entity $e_1{=}\langle id_1,\ A_1 \rangle$ is said to be* subsummed *by an entity $e_2{=}\langle id_2,\ A_2 \rangle$, denoted as $e_1 \dot{\prec} e_2$, if $id_1 \dot{\prec} id_2$ and for every attribute $\langle n,v_1 \rangle {\in} A_1$ there is attribute $\langle n,v_2 \rangle {\in} A_2$ such that $v_1{=}v_2$ or, assuming that the attribute is an association, $v_1 \dot{\prec} v_2$.* ∎

Given an evolution database $\langle E,\Omega \rangle$, and a set $\Omega_s {\subseteq} \Omega$ one can perform a series of consecutive coalescence operations, each one coalescing the two entities that an evolution relationship in the Ω_s associates.

DEFINITION 3.7. *Given an evolution database $D{:}\langle E,\Omega \rangle$ and a set $\Omega_s{=}\{m_1, m_2, \ldots, m_m\}$ such that $\Omega_s {\subseteq} \Omega$, let D_m be the evolution database generated by the sequence of coalescence operations $D \xrightarrow{m_1} D_1 \xrightarrow{m_2}, \ldots, \xrightarrow{m_m} D_m$. The* possible world *of D according to Ω_s is the database D_{Ω_s} generated by simply omitting from D_m all its evolution relationships.* ∎

Intuitively, a set of evolution relationships specifies sets of entities in a database that should be considered as one, while the possible world represents the database in which these entities have actually been coalesced. Our notion of a possible world is similar to the notion of a possible worlds in probabilistic databases [8]. Direct consequence of the definition of a possible world is the following theorem:

THEOREM 3.8. *The possible world of an evolution database $D{:}\langle E,\Omega \rangle$ for a set $\Omega_s {\subseteq} \Omega$ is unique.* ∎

Due to this uniqueness, a set Ω_s of evolution relationships of a database can be used to refer to the possible world.

According to the definition of a possible world, an evolution database can be seen as a shorthand of a set of databases, i.e., its possible worlds. Thus, a query on an evolution database can be seen as a shorthand for a query on its possible worlds. Based on this observation we define the semantics of query answering on an evolution database.

DEFINITION 3.9. *The evaluation of a query q on an evolution database D is the union of the results of the evaluation of the query on every possible world D_c of D.* ∎

For a given query, there may be multiple possible worlds that generate the same results. To eliminate this redundancy we require every coalescence to be well-justified. In particular, our principle is that no possible world or variable assignment will be considered, unless it generates some new results in the answer set. Furthermore, among the different possible worlds that generate the same results in the answer set, only the one that requires the smaller number of coalescences will be considered. To support this, we define a subsumption relationship among the variable assignments across different possible worlds and we redefine the semantics of the evaluation of a query.

$x	$y	Possible World	Answer	Cost
e1	P2P Video	$\emptyset$	e1(isHolder:"P2P Video")	0
Sk(e1,e2)	**P2P Video**	**e1,e2**	**Not generated**	**1**
Sk(e1,e2,e3)	**P2P Video**	**e1,e2,e3**	**Not generated**	**2**
Sk(e1,e2,e3)	ASR	e1,e2,e3	Sk(e1,e2,e3)(isHolder:"ASR")	2
Sk(e1,e2,e3,e4)	Laser	e1,e2,e3,e4	Sk(e1,e2,e3,e4)(isHolder:"Laser")	3
…	…	…	…	…

Table 1: A fraction of variable assignments for Example 3.11.

DEFINITION 3.10. *Let h and h' be two variable assignment for a set of variables X. h' is said to be* subsumed *by h, denoted as $h' \subseteq h$ if $\forall x \in X$: $h(x) = h'(x) = constant$, or $h(x) = e$ and $h'(x) = e'$, with e and e' being entities for which $e' \dot{\prec} e$ or $e = e'$. Given an evolution database D, let $\mathcal{W}$ be the set of its possible worlds. The answer to a query q is the union of the results of evaluating q on every database in $\mathcal{W}$. During the evaluation of q on a database in $\mathcal{W}$, true variable assignments that are subsumed by some other true variable assignment, even in other possible worlds, are not considered.* ∎

It is natural to assume that not all possible worlds are equally likely to describe the database the user has in mind when she was formulating the query. We assume that the more a possible world differs from the original evolution database, the less likely it is to represent what the user had in mind. This is also in line with the minimality and well-justification principle described previously. We reflect this as a reduced confidence to the answers generated by the specific possible world and quantify it as a score assigned to each answer. One way to measure that confidence is to count how many evolution relationships have to be coalesced for the possible world to be constructed. The evolution relationships may also be assigned a weight reflecting the confidence to the fact that its second entity is actually an evolution of the first.

EXAMPLE 3.11. *Consider again the query of Example 3.3. and assume that it is to be evaluated on the database of Figure 1. Table 1 illustrate a set of true variable assignments for some of the possible worlds of the database. The possible world on which each assignment is defined is expressed through its respective set Ω_s. The fourth column contains the result generated in the answer set from the specific assignment and the last column contains its respective cost, measured in number of coalesces that are required for the respective possible world to be generated from the evolution database. Note that the second and the third assignment (highlighted in bold), are redundant since they are subsumed by the first.* ∎

The existence of a score for the different solutions, allows us to rank the query results and even implement a top-k query answering. The challenging task though is how to identify in an efficient way the possible worlds and more specifically the true variable assignments that lead into correct results.

4. QUERY EVALUATION TECHNIQUES

4.1 The naive approach

The straight forward approach in evaluating a query is to generate all the possible worlds and evaluate the query on each one individually. In the sequel, generate the union of all the individually produced results, eliminate duplication and remove answers subsumed by others. Finally, associate to each of the remaining answers a cost based on the coalescences that were performed in order to generate the possible world from which the answer was produced, and rank the answers according to that score. The generation of all possible worlds is a time consuming task. For an evolution database with an evolution graph of N edges, there are 2^N possible worlds. This is clearly a brut force solution, not desirable for online query answering.

4.2 Materializing all the possible worlds

Since the possible worlds do not depend on the query that needs to be evaluated, they can be pre-computed and stored in advance so that they are available at query time. Of course, as it is the case of any materialization technique, the materialized data need to be kept in sync with the evolution database when its data is modified. Despite the fact that this will require some effort, there are already well-known techniques for change propagation [3] that can be used. The major drawback, however, is the space overhead. A possible world contains all the attributes of the evolution database, but in fewer entities. Given that the number of attributes are typically larger than the number of entities, and that entities associated with evolution relationships are far fewer than the total number of entities in the database, we can safely assume that the size of a possible world will be similar to the one of the evolution database. Thus, the total space required will be 2^n times the size of the evolution database. The query answering time, on the other hand, will be 2^n times the average evaluation time of the query on a possible world.

4.3 Materializing only the maximum world

An alternative solution is to generate and materialize the possible world D_{max} generated by performing all possible coalescences. For a given evolution database $\langle E, \Omega \rangle$, this world is the one constructed according to the set of all evolution relationships in Ω. Any query that has an answer in some possible world of the evolution database will also have an answer in this maximal possible world D_{max}. This solution work has two main limitations. First it does not follow our minimalistic principle and performs coalescences that are not needed, i.e., they do not lead to any additional results in the result set. Second, the generated world fails to include results that distinguish difference phases of the lifespan of an entity (phases that may have to be considered individual entities) but the approach coalesces them in one just because they are connected through evolution relationships.

4.4 On-the-fly coalescence computations

To avoid any form of materialization, we propose an alternative technique that computes the answers on the fly by performing coalescences on a need-to-do basis. In particular, we identify the attributes that satisfy the different query conditions and from them the respective entities to which they belong. If all the attributes satisfying the conditions are on the same entity, then the entity is added in the answer set. However, different query conditions may be satisfied by attributes in different entities. In these cases we identify sets of entities for each one of which the union of the attributes of its entities satisfy all the query conditions. For each such a set, we coalesce all its entities into one if they belong to the same connected component of the evolution graph. Doing the coalescence it is basically like creating the respective possible world; however, we generate only the part of that world that is necessary to produce an answer to the query. In more details, the steps of the algorithm are the following.

[Step 1: Query Normalization] We decompose every non-arithmetic atom in the body of the query that has more than one condition into a series of single-condition atoms. More specifically, any atom of the form $x(n_1{:}v_1, n_2{:}v_2, \ldots, n_k{:}v_k)$ is decomposed into a conjunction of atoms $x(n_1{:}v_1), x(n_2{:}v_2), \ldots, x(n_k{:}v_k)$.

[Step 2: Individual Variable Assignments Generation] For each non-arithmetic atom in the decomposed query, a list is constructed that contains assignments of the variables in the respective atom to constants that make the atom true. Assuming a total of N non-arithmetic atoms after the decomposition, let L_1, L_2, ..., L_N be the generated lists. Each variable assignment actually specifies the part of the evolution database that satisfies the condition described in the atom.

[Step 3: Candidate Assignment Generation] The elements of the lists generated in the previous step are combined together to form complete variable assignments, i.e., assignments that involve every variable in the body of the query. In particular, the cartesian product of the lists is created. Each element in the cartesian product is a tuple of assignments. By construction, each such tuple will contain at least one assignment for every variable that appears in the body of the query. If there are two assignments of the same attribute bound variable to different values, the whole tuple is rejected. Any repetitive assignments that appear within each non-rejected tuple is removed to reduce redundancy. The result is a set of variable assignments, one from each of the tuples that have remained.

[Step 4: Arithmetic Atom Satisfaction Verification] Each assignment generated in the previous step for which there is at least one arithmetic atom not evaluating to true, is eliminated from the list.

[Step 5: Candidate Coalescence Identification] Within each of the remaining assignments we identify entity-bound variables that have been assigned to more than one values. Normally this kind of assignment evaluates always to false. However, we treat them as suggestions for coalescences, so that the assignment will become a *true assignment* (ref. previous Section). For each assignment h in the list provided by Step 4, the set $\mathcal{V}_h$=$\{V_{x_1}, V_{x_2}, \ldots, V_{x_k}\}$ is generated, where V_x is the set of different entities that variable x has been assigned in assignment h. In order for the assignments of variable x to evaluate to true, we need to be able to coalesce the entities in V_x. To do so, these entities have to belong to the same connected component in the evolution graph of the database. If this is not the case, the assignment h is ignored.

[Step 6: Coalescence Realization & Cost Computation] Given a set $\mathcal{V}_h$=$\{V_{x_1}, V_{x_2}, \ldots, V_{x_k}\}$ for an assignment h among those provided by Step 5, we need to find the minimum cost coalescences that need to be done such that all the entities in a set V_i, for i=$1..k$, are coalesced to the same entity. This will make the assignment h a *true* assignment, in which case the head of the query can be computed and an answer generated in the answer set. The cost of the answer will be the cost of the respective possible world, which is measured in terms of the number of coalescences that need to be performed. Finding the required coalescences for the set $\mathcal{V}_h$ that minimizes the cost boils down to the problem of finding a Steiner forest [12],

EXAMPLE 4.1. *Let us consider again the query of Example 3.3. In Step 1, its body will be decomposed into two parts:* $\$x(name{:}'AT\&TLabsInc.')$ *and* $\$x(isHolder{:}\$y)$. *For those two parts, during Step 2, the lists* L_1=$\{\{\$x$=$e1\}\}$ *and* L_2=$\{\{\$x$=$e1$, $\$y$=$'P2PVideo'\}$, $\{\$x$=$e3$, $\$y$=$'ASR'\}$, $\{\$x$=$e4$, $\$y$=$'Laser'\}$, $\{\$x$=$e5$, $\$y$=$'VoIP'\}\}$ *will be created. Step 3 creates their cartesian product* L=$\{\{\$x$=$e1$, $\$x$=$e1$, $\$y$=$'P2PVideo'\}$, $\{\$x$=$e1$, $\$x$=$e3$, $\$y$=$'ASR'\}$, $\{\$x$=$e1$, $\$x$=$e4$, $\$y$=$'Laser'\}$, $\{\$x$=$e1$, $\$x$=$e5$, $\$y$=$'VoIP'\}\}$. *The only attribute bound variable is* $\$y$ *but this is never assigned to more than one different value at the same time so nothing is eliminated. Since there are no arithmetic atoms, Step 4 makes no change either to the list L. If for instance, the query had an atom* $\$y{\neq}'VOIP'$ *in its body, then the last element of the list would have been eliminated. Step 5 identifies that the last three elements in* L *have the entity bound variable* $\$x$ *assigned to two different values; thus, it generates the candidate coalesences:* V_1=$\{e1, e3\}$, V_2=$\{e1, e4\}$ *and* V_3=$\{e1, e5\}$. *Step 6 determines that all three coalescences are possible. Entities* $e1$, $e2$ *and* $e3$ *will be coalesced for* V_1, $e1$, $e2$, $e3$ *and* $e4$ *for* V_2, *and the* $e1$, $e2$, $e3$ *and* $e5$ *for* V_3. ∎

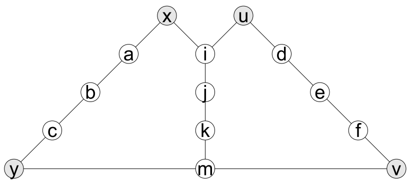


Figure 2: An illustration of the Steiner forest problem.

5. STEINER FOREST ALGORITHM

The last step of the evaluation algorithm presented in the previous section takes as input a set of entity sets and needs to perform a series of coalesce operations such that all the entities within each set will become one. To do so, it needs to find an interconnect on the evolution graph among all the entities within each set. Note that the interconnect may involve additional entities not in the set that unavoidably will also have to be coalesced with those in the set. Thus, it is important to find an interconnect that minimizes the total cost of the coalescences. The cost of a coalescence operation is the weight of the evolution relationship that connects the two entities that are coalesced. Typically, that cost is equal to one, meaning that the total cost is actually the total number of coalescence operations that need to be performed. For a given set of entities, this is known as the problem of finding the Steiner tree [11]. However, given a set of sets of entities, it turns out that finding the optimal solution, i.e., the minimum cost interconnect of all the entities, is not always the same as finding the Steiner tree for each of the sets individually. The specific problem is found in the literature as the Steiner forest problem [12].

The difference in the case of the Steiner forest is that edges can be "used" by more than one interconnect. More specifically, the Steiner tree problem aims at finding a tree on an undirected weighted graph that connects all the nodes in a set and has the minimum cost. In contrast to the minimum spanning tree, a Steiner tree is allowed to contain intermediate nodes in order to reduce its total cost. The Steiner forest problem takes as input sets of sets of nodes and needs to find a set of non-connected trees (branches) that make all the nodes in each individual set connected and the total cost is minimal, even if the cost of the individual trees are not always the minimal. We refer to these individual trees with the term *branches*. Figure 2 illustrates the difference through an example. Assuming that we have the graph shown in the figure and the two sets of nodes $\{x,y\}$ and $\{u,v\}$. Clearly, the minimum cost branch that connects nodes x and y is the one that goes through the nodes a, b and c. Similarly the minimum cost branch that connects u and v is the one that goes through the nodes e, f and g. Each of the two branches has cost 4 (the number of edges in the branch), thus, the total cost will be 8. However, if instead we connect all the four nodes x, y, u and v though the tree that uses the nodes i, j, k and m, then, although the two nodes in each set are connected with a path of 5 edges, the total cost is 7.

Algorithm 1 Steiner tree algorithm

```
Input: graph $G$,$f : E \rightarrow \mathbb{R}^+$, groups $\mathcal{V}= V_1, \ldots, V_L$
Output: ST for each element in $flat(\mathcal{V})$
 1: $Q_T$: priority queue sorted in the increasing order
 2: $Q_T \Leftarrow \emptyset$
 3: for all $s_i \in maxflat(\mathcal{V})$ do
 4:     enqueue $T(s_i, \{s_i\})$ into $Q_T$;
 5: end for
 6: while $Q_T \neq \emptyset$ do
 7:     dequeue $Q_T$ to $T(v, \mathbf{p})$;
 8:     if $\mathbf{p} \in flat(\mathcal{V})$ then
 9:         $ST(\mathbf{p}) = T(v, \mathbf{p})$
10:     end if
11:     if $ST$ has all values then
12:         return $ST$
13:     end if
14:     for all $u \in N(v)$ do
15:         if $T(v, \mathbf{p}) \oplus (v, u) < T(u, \mathbf{p})$ then
16:             $T(u, \mathbf{p}) \Leftarrow T(v, \mathbf{p}) \oplus (v, u)$;
17:             update $Q_T$ with the new $T(u, \mathbf{p})$;
18:         end if
19:     end for
20:     $\mathbf{p}_1 \Leftarrow \mathbf{p}$;
21:     for all $\mathbf{p}_2$ s.t. $\mathbf{p}_1 \cap \mathbf{p}_2 = \emptyset$ do
22:         if $T(v, \mathbf{p}_1) \oplus T(v, \mathbf{p}_2) < T(v, \mathbf{p}_1 \cup \mathbf{p}_2)$ then
23:             $T(u, \mathbf{p}_1 \cup \mathbf{p}_2) \Leftarrow T(v, \mathbf{p}_1) \oplus T(v, \mathbf{p}_2)$;
24:             update $Q_T$ with the new $T(u, \mathbf{p}_1 \cup \mathbf{p}_2)$;
25:         end if
26:     end for
27: end while
```

Formally, the *Steiner forest* problem is defined as follows. Given a graph $G = \langle N, E \rangle$ and a cost function $f : E \rightarrow \mathbb{R}^+$, alongside a set of groups of nodes $\mathcal{V}= V_1, \ldots, V_L$, where $V_i \subseteq N$, find a set $C \subseteq E$ such that C forms a connected component that involves all the nodes of every group V_i and the $\sum_i f(c_i) \mid c_i \in C$ is minimal.

The literature contains a number of approximate solutions [2][18][14] as well as a number of exact solution using Dynamic Programming [11][9][20] for the discovery of Steiner trees. However, for the Steiner forest problem (which is known to be NP-hard [12]) although there are approximate solutions [12], no optimal algorithm has been proposed so far. In the current work we are making a first attempt toward that direction by describing a solution that is based on dynamic programming and is constructed by extending an existing Steiner tree discovery algorithm.

To describe our solution it is necessary to introduce the set $flat(\mathcal{V})$. Each element in $\flat((\mathcal{V}))$ is a set of nodes created by taking the union of the nodes in a subset of $\mathcal{V}$. More specifically, $flat(\mathcal{V})= \{U \mid U{=}\bigcup_{V_i \in S} V_i \text{ with } S \subseteq \mathcal{V}\}$. Clearly $flat(\mathcal{V})$ has 2^L members. We denote by $maxflat(\mathcal{V})$ the maximal element in $flat(\mathcal{V})$ which is the set of all possible nodes that can be found in all the sets in $\mathcal{V}$, i.e., $maxflat(\mathcal{V}){=}\{n \mid n{\in}V_1{\cup}\ldots{\cup}V_L\}$.

Our solution for the computation of the Steiner forest consists of two parts. In the first part, we compute the Steiner trees for every member of the $flat(\mathcal{V})$ set, and in the second part we use the computed Steiner trees to generate the Steiner forest on $\mathcal{V}$.

The state-of-the-art optimal (i.e., no approximation) algorithm for the Steiner tree problem is a dynamic programming solution developed in the context of keyword searching in relational data [9]. The algorithm is called the Dynamic Programming Best First (DPBF) algorithm and is exponential in the number of input nodes and polynomial with respect to the size of graph. We extend DPBF in order to find a *set* of Steiner trees, in particular a Steiner tree for every element in $flat(\mathcal{V})$. The intuition behind the extension is that we initially solve the Steiner tree problem for the $maxflat(\mathcal{V})$ and continue iteratively until the Steiner trees for every element in $flat(\mathcal{V})$ has been computed. We present next a brief description of DPBF alongside our extension.

Let $T(v, \mathbf{p})$ denote the minimum cost tree rooted at v that includes the set of nodes $\mathbf{p}{\subseteq}maxflat(\mathcal{V})$ Note that by definition, the cost of the tree $T(s, maxflat(\mathcal{V}))$ is 0, for every $s{\in}maxflat(\mathcal{V})$.

Trees can be iteratively merged in order to generate larger trees by using the following three rules.

$$T(v, \mathbf{p}) = min(T_g(v, \mathbf{p}), T_m(v, \mathbf{p})) \quad (1)$$

$$T_g(v, \mathbf{p}) = min_{u \in N(v)}((v, u) \oplus T(u, \mathbf{p})) \quad (2)$$

$$T_m(v, \mathbf{p}_1 \cup \mathbf{p}_2) = min_{\mathbf{p}_1 \cap \mathbf{p}_2 = \emptyset}(T(v, \mathbf{p}_1) \oplus T(v, \mathbf{p}_2)) \quad (3)$$

where $\oplus$ is an operator that merges two trees into a new one and $N(v)$ is the set of neighbour nodes of node v. In [9] it was proved that these equations are dynamic programming equations leading to the optimal Steiner tree solution for $maxflat(\mathcal{V})$ set of nodes. To find it, the DPBF algorithm employs the Dijkstra's shortest path search algorithm in the space of $T(v, \mathbf{p})$. The steps of the Steiner tree computation are shown in Algorithm 1. In particular, we maintain a priority queue Q_T that keeps in an ascending order the minimum cost trees that have been found at any given point in time. Naturally, a *dequeue* operation retrieves the tree with the minimal cost. Using the greedy strategy we look for the next minimal tree which can be obtained from the current minimal. In contrast to DPBF, we do not stop when the best tree has been found, i.e. when the solution for $maxflat(\mathcal{V})$ has been reached, but we keep collecting minimal trees (lines 7-10) until all elements in $flat(\mathcal{V})$ have been computed (lines 11-13). To prove that all the elements of $flat(\mathcal{V})$ are found during that procedure, it suffices to show that our extension corresponds to the finding of all the shortest paths for a single source in the Dijkstra's algorithm. The time and space complexity for finding the Steiner trees is $O(3^{\sum l_i}n + 2^{\sum l_i}((\sum l_i + logn)n + m))$ and $O(2^{\sum l_i}n)$, respectively, where n and m are the number of nodes and edges of graph G, and l_i is the size of the ith set V_i in the input of set $\mathcal{V}$ of the algorithm.

Once all the Steiner trees for $flat(\mathcal{V})$ have been computed, we use them to find the Steiner forest for $\mathcal{V}$. The Steiner forest problem has an optimal substructure and its subproblems overlap. This means that we can find a dynamic programming solution to it. To show this, we first we consider the case for $L{=}1$, i.e., the case in which we have only one group of nodes. In that case finding the Steiner forest is equivalent to finding the Steiner tree for the single set of nodes that we have. Assume now that $L{>}1$, i.e., the input set $\mathcal{V}$ is $\{V_1, \ldots, V_L\}$, and that we have already computed all the Steiner forests for every set $\mathcal{V}' {\subset} \mathcal{V}$. Let $SF(\mathcal{V})$ denote the Steiner forest for an input set $\mathcal{V}$. We do not know the exact structure of $SF(\mathcal{V})$, i.e. how many branches it has and what elements of $\mathcal{V}$ are included in each. Therefore, we need to test all possible hypotheses of the forest structure, which are 2^L, and pick the one that has minimal cost. For instance, we assume that the forest has a branch that includes all nodes in V_1. The total cost of the forest with that assumption is the sum of the Steiner forest on V_1 and the Steiner forest for $\{V_2, \ldots, V_L\}$ which is a subset of $\mathcal{V}$, hence it is considered known. The Steiner forest on V_1 is actually a Steiner tree. This is based on the following lemma.

LEMMA 5.1. *Each branch of a Steiner forest is a Steiner tree.*

Algorithm 2 Steiner forest algorithm

```
Input: $G = \langle N, E\rangle$, $\mathcal{V}= \{V_1, \ldots, V_L\}$,$ST(s)\ \forall s{\in}flat(\mathcal{V})$
Output: $SF(\mathcal{V})$
 1: for all $V_i \in \mathcal{V}$ do
 2:     $SF(V_i) = ST(V_i)$
 3: end for
 4: for $i = 2$ to $L - 1$ do
 5:     for all $H \subset \mathcal{V}$ and $\mid H \mid= i$ do
 6:         $u \Leftarrow \infty$
 7:         for all $E \subseteq H$ and $E \neq \emptyset$ do
 8:             $u \Leftarrow min(u, ST(maxflat(E)) \oplus SF(H \setminus E))$
 9:         end for
10:         $SF(H) \Leftarrow u$
11:     end for
12: end for
13: $u \Leftarrow \infty$
14: for all $H \subseteq \mathcal{V}$ and $H \neq \emptyset$ do
15:     $u \Leftarrow min(u, ST(maxflat(H)) \oplus SF(\mathcal{V} \setminus \mathcal{H}))$
16: end for
17: $SF(\mathcal{V}) \Leftarrow u$
```

PROOF. This proof is done by contradiction. Assuming that a branch of the forest is not a Steiner tree, it can be replaced with a Steiner tree and reduce the overall cost of the Steiner forest. This means that the initial forest was not minimal. □

We can formally express the above reasoning as:

$$SF(\mathcal{V}) = \min_{H \subseteq V}(ST(maxflat(H)) \oplus SF(\mathcal{V} \setminus H)) \qquad (4)$$

Using the above equation in conjunction with the fact that $SF(\mathcal{V}){=}ST(V_1)$, if $\mathcal{V}{=}\{V_1\}$, we construct an algorithm (Algorithm 2) that finds the Steiner forest in a bottom-up fashion. The time and space requirements of the specific algorithm are $O(3^L - 2^L(L/2 - 1) - 1)$ and $O(2^L)$, respectively. Summing this with the complexities of the first part, it gives a total time complexity $O(3^{\sum l_i}n{+}2^{\sum l_i}((\sum l_i{+}logn)n{+}m)){+}3^L{-}2^L(L/2{-}1){-}1)$ with space requirement $O(2^{\sum l_i}n + 2^L)$.

6. QUERY EVALUATION OPTIMIZATION

In the case of top-k query processing there is no need to actually compute all possible Steiner forests to only reject some of them later. It is important to prune as early as possible cases which are expected not to lead to any of the top-k answers. We have developed a technique that achieves this. It is based on the following lemma.

LEMMA 6.1. *Given two sets of sets of nodes $\mathcal{V}'$ and $\mathcal{V}''$ on a graph G for which $\mathcal{V}'{\subseteq}\mathcal{V}''$: $cost(SF(\mathcal{V}')){\leq}cost(SF(\mathcal{V}''))$.*

PROOF. The proof is based on the minimality of a Steiner forest. Let SF(V') and SF(V'') be Steiner forests for V' and V'', with costs w_1 and w_2, respectively. If $cost(SF(\mathcal{V}'')){\leq}cost(SF(\mathcal{V}'))$, then we can remove $V''{\setminus}V'$ from V'' and cover V' with a smaller cost forest than $SF(V')$, which contradicts the fact that $SF(V')$ is a Steiner forest. □

To compute the top-k answers to a query we do the following steps. Assume that $B = \{\mathcal{V}_1, \ldots, \mathcal{V}_n\}$ is a set of inputs for the Steiner forest algorithm. First, we find the $B_{min}{\subseteq}B$ such that for each $\mathcal{V}'{\in}B_{min}$ there is no $\mathcal{V}''{\in}B$ such that $\mathcal{V}'{\subset}\mathcal{V}''$. Then, we compute the Steiner forest for each element in B_{min}. According to Lemma 6.1 and the construction procedure of B_{min} we ensure that the the Top-1 is among the computed Steiner forests. We remove the input which corresponds to that Top-1 answer from B and then we continue with the computation of Steiner forests to update B_{min}. The above steps are repeated until k answers have been found.

7. EXPERIMENTS

To evaluate the efficiency and effectiveness of our approach we performed two kinds of experiments. First we studied the behavior of the Steiner forest discovery algorithm in isolation, and then we evaluated the performance of the query answering mechanism we have developed and which uses internally the Steiner forest algorithm. We also studied the effectiveness of our optimization technique for top-k query answering.

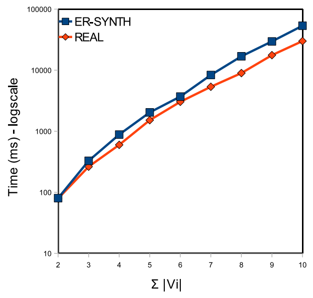
In the experiments we used both synthetic and real data. We used the term *non-evolution* data to refer to entities and attributes, and the term *evolution* data to refer to the the evolution relationships and more generally to the evolution graph. We noticed that in the real datasets, the non-evolution data were much larger than the evolution data and we maintained a similar analogy during our synthetic data generation.

For the synthetic data generation we used the Erdös-Rényi graph generator [17] which can produce random graphs for which the probability to have an edge between two nodes is constant and independent of other edges. Since many real world data follow a power law distribution, for the non-evolution synthetic data we used the Zipf's distribution. In our own implementation of the Zipfian distribution, as a rank we considered the number of occurrences of an attribute-value pair in the entire dataset (e.g., if the attribute $\langle State, CA\rangle$ appeared 15 time, its rank was 15). This allowed us to model the fact that there are few frequent attribute-value pairs and the majority are rare attributes. The real corpora that we used had similar properties. We will refer to the synthetic dataset generated using this method as *ER-SYNTH*.

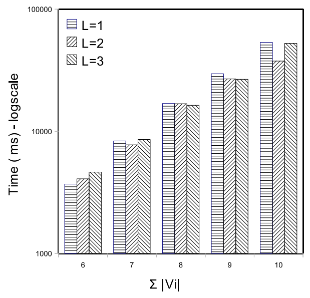
For real dataset we used an extract from the trademark corpora which is available from the United States Patent and Trademark Office[1]. The trademarks are a kind of intellectual property which may belong to an individual or a company. If the owner of some trademark changes, the trademark has to be re-registered accordingly. Analyzing the trademark owner lists we could extract sequences of companies that have registered the same trademark, and we used this as an indication for evolution. We constructed the evolution graphs by considering each such a pair as an evolution relationship. The dataset we generated that way from the UPTSO files contained approximately 16K unique companies, 200K attributes (the information about companies such as name, place where it is registered and so on), and an evolution graph with 573 components of sizes between 5 and 373. To make the dataset extracted from real data even richer, i.e., with components of higher complexity, we used two graph merging strategies. In the first, a new evolution component is constructed by connecting two components through an artificial evolution relationship edge between two random nodes from the two components. We refer to this kind of merge as *CHAIN*, because it creates a chain of source graphs. In the second strategy, two components are merged, by choosing an arbitrary node from one component and then adding evolution relationship edges to some random node of every other component. We refer to this method as *STAR*. Datasets generated using these methods will be denoted as *REAL-CHAIN* and *REAL-STAR*, respectively.

The naive evaluation strategies that were described in Sec-

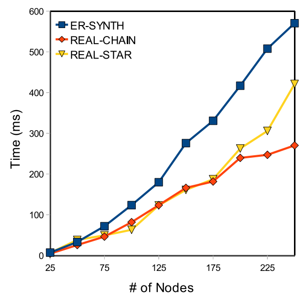
[1]http://www.uspto.gov/



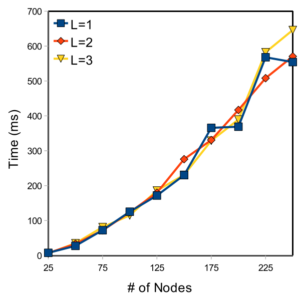
(a)



(b)



(c)



(d)

Figure 3: Steiner forest discovery performance

tions 4.1 and 4.2 are omitted from the discussion since their high complexity makes them practically infeasible to implement. In some sense, the naive case corresponds to the brute force way of finding a minimal Steiner forest, which is exponential in the size of evolution graph.

The experiments were all carried out on a 2.4GHz CPU and 4G memory PC running MS Windows Vista.

7.1 Steiner Forest

To study in detail the Steiner forest algorithm we performed two kinds of experiments. First, we studied the scalability properties of the algorithm for varying inputs, and then the behavior of the algorithm for graphs with different characteristics.

Scaling the Input. Recall that the input to the Steiner forest algorithm is the set $\mathcal{V}=\{V_1,\ldots,V_L\}$. In this experiment we studied the query evaluation time with respect to the size of the input. By size we considered two parameters: (i) the total number of elements in the sets of $\mathcal{V}$, i.e., the $\sum_{i=1}^{L}|V_i|$; and (ii) the number of the groups, i.e., the value L.

For the former, we started with L=1 and we scaled the $\sum|V_i|$ (which in this case is actually equal to $|V_1|$) from 2 to 10. For each size the average evaluation time of 25 random queries was recorded. The queries were evaluated both on synthetic and on real data. The synthetic graph was obtained using the Erdös-Rényi method and had $n = 57$ nodes and $m = 65$ edges. The real dataset graph was the one described previously. The results of this experiment are presented in Figure 3(a). The exponential grows in time (note that the time is presented on a logarithmic scale) with respect to a query size is consistent with the theoretical complexity of the Steiner forest algorithm.

To study how the parameter L affects the query execution time we kept the $\Sigma_{i=1}^{L}|V_i|$ constant but modified the number of the sets L from 1 to 3, and then we repeated the experiment for values of $\sum_{i=1}^{L}|V_i|$ from 6 to 10 (we assumed that a minimal set size was 2). The results of the average of 25 random query execution times are reported in Figure 3(b). The characteristics of the graph were the same as those in the previous experiment. The current experiment showed that the execution time depends fully on the $\sum_{i=1}^{L}|V_i|$ and not on L itself. This means that within a reasonable range of query sizes the number of forest branches does not have any influence on the performance.

Scaling the Graph. In this experiment we studied the Steiner forest discovery time with respect to the size of the graph. We used three kinds of graph data: *ER-SYNTH*, *REAL-CHAIN* and *REAL-STAR*, with sizes from 25 to 250 nodes with a step of 25.

For the synthetic dataset the number of edges and nodes was almost the same. We generated 25 random inputs to the Steiner forest problem with $L = 2$ and $|V_1|$=3, and $|V_2|$=3. The results of this experiment are presented in Figure 3(c). The query evaluation time has a linear trend as expected, and it was interesting that the execution time was always less than a second..

We also studied the scalability of the algorithm in terms of the parameter L. For three queries with $\sum|V_i| = 6$ and L=1, 2 and 3 we varied the evolution graph size from 25 to 250 with step 25. The graph we used was the *ER-SYNTH* with the same number of nodes and edges as before. The results are shown in Figure 3(d), where it can be observed that the scalability of the algorithm depends on the total number of elements in the sets in the input set $\mathcal{V}$, i.e., the $\sum_{i=1}^{L}|V_i|$, and not on the number of forest branches, i.e., the number L, at least for values of $\sum_{i=1}^{L}|V_i|$ up to 10.

7.2 Query Evaluation

Apart from experimenting with the Steiner forest algorithm in isolation, we run a number of experiments to evaluate the query answering mechanism we have developed for evolution databases. The query evaluation time depends not only on the evolution graph size and structure but also on the size and structure of the whole evolution database. First, we analyzed the behaviour of the system with respect to the query size. The query size is determined by the number of distinct variables, and their number of occurrences in the query. We started with a 1-variable query and we observe its behavior as size increases.. Then, we tested the scalability of the query evaluation mechanism as a function of the evolution graph only. Finally, we studied the scalability as a function of the data (i.e., attributes and associations) and we found that their distribution (but not their size) can dramatically affect the performance of the system.

In the synthetic data generation, we generated values that were following the Zipfian distribution for the attributes/associations. We controlled the generation of the *ER-SYNTH* dataset through four parameters, and in particular, the *pool of entities*, the *exponent* that is used to adjust the "steepness" of the Zipfian distribution, the *number of elements* that describes the maximum frequency of an attribute or association, and the *number of attributes*.The values of the parameters for the synthetic data are chosen to coincide with those of the real corpora.

Scaling the Query. We considered a number of 1-variable queries with a body of the form:

$$\$x(attr_1{:}value_1),\ldots,\$x(attr_N{:}value_N)$$

and we performed a number of experiments for different values of N, i.e., the number of atoms in the query. For every atom we randomly chose an attribute-value pair from a pool of available distinct attribute name/value pairs. The *ER-SYNTH* graph that was generated had 57 nodes and 65 edges. The results are shown in

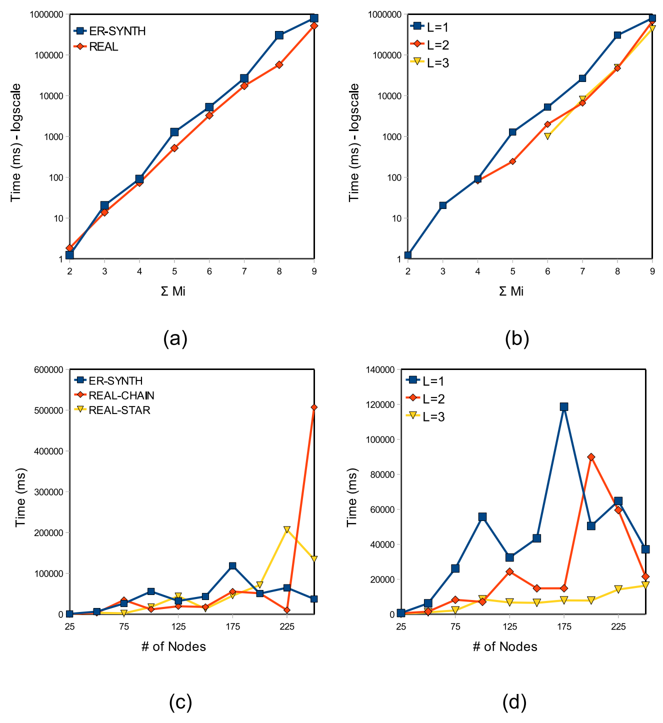


Figure 4: Query evaluation performance

Figure 4(a). The same figure includes the results of the query evaluation on the real dataset that had a size similar to the synthetic. For the generation of their non-evolution data we had the exponent set to 3, the number of elements parameter set to 15 and their total number was 537. The results of the Figure 4(a) are in a logarithmic scale and confirm the expectation that the query evaluation time is growing exponentially as the number of variables in the query grows. If we compare these results with those of the Steiner forest algorithm for the respective case, it follows that the integrated system adds a notable overhead on top of the Steiner forest algorithm execution time. This is was due to the number of coalescence candidates and the number of Steiner forests that needed to be computed in order to obtain the cost of the elements in the answer set. Although the parameters for the generation of the synthetic and real data coincided, their trends were different, as Figure 4(c) illustrates.

We further tested how the number of entity variables in the query affect the performance. Note that we are mainly interested in the entity-bound variables. Let M represent the number of distinct entity-bound variables in the query, and M_i the number of appearances of the i-th variable in the query. Note that the number of distinct variables will require to solve a Steiner forest problem in which the input $\mathcal{V}$=$\{V_1, \ldots, V_L\}$ will have L=M and $|V_i|$=M_i, for each i=1..M. The total number of variable appearances in the query will naturally be $\sum_{i=1}^{M} M_i$.

In the experiment, we chose a constant value for the $\sum_{i=1}^{M} M_i$ and we run queries for M=1, 2 or 3. As a dataset we used the *ER-SYNTH* with 57 nodes and 65 edges. 537 attributes were generated with the exponent parameter having the value 3 and the number of elements parameter to have the value 15.A total of 53 synthetic associations were also generated with the exponent parameter having the value 3, and the number of elements parameter to have the value 10. We used 25 randomly generated queries for each of the 3 M values, and took their average execution time. We did multiple runs of the above experiments for different values of $\sum_{i=1}^{M} M_i$ between 4 and 10. The outcome of the experiments is shown in Figure 4(b) in a logarithmic scale. Clearly, the number of branches in a forest did not affect the query evaluation time, i.e., queries with many variables showed the same increase in time as the 1-variable query for the same $\sum_{i=1}^{M} M_i$.

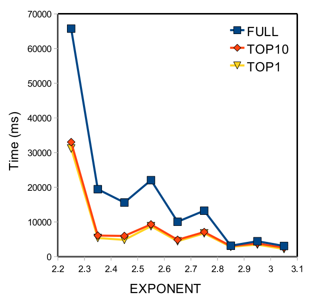
Scaling the Data. In this experiment we examined the query evaluation time with respect to the size of the evolution graph. As evolution data, we used both real and synthetic sources. Regarding the real data, we used a series of graphs (and their attributes as non-evolution data) with sizes from 25 to 250 with step 25. The number of edges was 110% of the number of nodes for all graphs. For the real dataset we used both the *REAL-CHAIN* and the *REAL-STAR* data. For each graph we generated 25 random queries with 3 distinct variables, i.e., M=3, and each variable had M_1=2, M_2=2 and M_3=3 appearances in the query, and we measured the average time required to evaluate them. As a synthetic dataset the *ER-SYNTH* was used, generated to be the same size as before but with the following Zipfian distribution parameters: exponent 3, number of elements 15 and number attributes 10 times more than the number of nodes. Note that we did not generate associations because the trademark dataset did not have any association that we could use as a guide. Figure 4(c) depicts the results of this experiment. It shows that there is a linear growth of time which is is accompanied with an increasing oscillations which can be explained by the growing exponent of non-evolution data, i.e. the number of coalescence candidates may become too large for evolution graphs with considerable size.

Furthermore, we studied how the query evaluation time scales for different values of M, i.e. for different distinct variables but with the same total number of variable appearances in the query (i.e., the $\sum_{i=1}^{M} M_i$). We used the *ER-SYNTH* dataset again with sizes from 25 to 250, using a step 25. The number of evolution relationships was 110% of the number of entities. For each case, we generated 25 random queries with M=1 having M_1=6, M=2 having M_1=3, and M_2=3 and finally, M=3 having M_1=3, M_2=2, and M_3=2. We executed these queries and measured the average evaluation time. The non-evolution data was following the Zipfian distribution with exponent 3, the number of elements was 15 and the total number of attributes was 10 times more that the number of nodes (entities). For the associations, the exponent was 3, the number of elements was 10 and their total number was 5 times more that the respective number for nodes. The results are presented in Figure 4(d). Similarly to the previous experiment, we observed a linear growth with increasing oscillations.

Evolution scalability for different forest structures. We further examined how the number of evolution graph components influence the query evaluation time. For this purpose, we generated data using *ER-SYNTH*, and in particular 5 datasets of evolution graphs with a total size of 300 nodes and 330 edges. The sets had 1, 2, 3, 4 and 5 evolution graphs respectively. For each set we run 25 random queries with two distinct variables (L = 2) that were appearing in the query 3 times each, i.e., M_1=3 and M_2=3 and measured their average execution time. As non-evolution data, we generated attributes and associations with varying exponent parameter, 2.5, 3 and 3.5. The total number of elements and attributes/associations were 15 and 1000 in one case, while it was 10 and 100 in the other. Figure 5(a) contains a table with the query evaluation time for each number of branches and exponent values. From the result, we could observe the dramatic decrease in time with respect to the number of evolution graph components. This can be explained by the fact that the query evaluation ”smeared” across a number of evolution

# of Branches	s=2.5	s=3.0	s=3.0
1	93,845	44,098	35,382
2	2,400	1,414	4,686
3	374	637	485
4	485	20	91
5	124	260	16

(a)



(b)

Figure 5: Query evaluation time for graphs with different numbers of connected components and for varying exponent of data distribution

graph components where each evaluation time becomes considerably small.

Data distribution dependency. Finally, we studied the properties of the system in relation to the data distribution parameter, namely the exponent of Zip's distribution. The query optimizer described in Section 6 was taken into consideration here and we analyzed how the non-evolution data were affecting the top-k query answering. For this experiment we used the following input parameters: 25 random queries with M=2 distinct variables, and M_1=3 and M_2=3 respective appearances of each distinct variable in the query. We used an *ER-SYNTH* dataset, the evolution graph of which had n = 57 nodes and m = 65 evolution edges. We also had 10000 attributes distributed over 30 entities, and 1000 associations distributed over 15 entities. The exponent we used varied from 2.25 to 3.05 with a step of 0.1. The results of the specific experiment are presented in Figure 5(b). For small exponents the difference between regular query answering and the top-10 or top-1 was significant. To justify this, recall that the number of pruned candidates depends on how different are the input sets in the Steiner forest algorithm input (ref. Section 6), thus, when the exponent is small the input sets share many entities.

8. CONCLUSION

In this work we presented a novel framework for dealing with evolution of entities at different granularity levels. We have made first class citizens of the system associations between entities indicating that they represent the same real world object but in different evolution phases. We have designed and implemented a technique that allows query answering over such databases even if the evolution model that the user has in mind is of different granularity than the one used in the database. The solution required the computation of a Steiner forest. For the later we have presented a novel algorithm for computing its optimal solution. Finally, we have performed a number of extensive experimental evaluation to determine the efficiency of our technique.

References

[1] J. Banerjee, W. Kim, H. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *SIGMOD*, pages 311–322, May 1987.

[2] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.

[3] J. Blakeley, P. A. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *SIGMOD*, pages 61–71, 1986.

[4] P. Buneman, S. Khanna, K. Tajima, and W. Tan. Archiving scientific data. In *SIGMOD*, pages 1–12, 2002.

[5] S. Chawathe, S. Abiteboul, and J. Widom. Representing and Querying Changes in Semistructured Data. In *ICDE*, pages 4–19, 1998.

[6] S. Chien, V. J. Tsotras, and C. Zaniolo. Efficient Management of Multiversion Documents by Object Referencing. In *VLDB*, pages 291–300, 2001.

[7] N. N. Dalvi, R. Kumar, B. Pang, R. Ramakrishnan, A. Tomkins, P. Bohannon, S. Keerthi, and S. Merugu. A web of concepts. In *PODS*, pages 1–12, 2009.

[8] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. volume 16, pages 523–544, Secaucus, NJ, USA, 2007. Springer-Verlag New York, Inc.

[9] Bolin Ding, J Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. Finding Top-k Min-Cost Connected Trees in Databases. *ICDE*, pages 836–845, 2007.

[10] X. Dong and A. Y. Halevy. Indexing dataspaces. In *SIGMOD*, pages 43–54, 2007.

[11] S E Dreyfus and R A Wagner. The Steiner problem in graphs. *Networks*, 1(3):195–207, 1972.

[12] Elisabeth Gassner. The Steiner Forest Problem revisited. *Journal of Discrete Algorithms*, 8(2):154–163, June 2010.

[13] A. Gupta, I. S. Mumick, and K. A. Ross. Adapting Materialized Views after Redefinitions. In *SIGMOD*, pages 211–222, 1995.

[14] Hao He, Haixun Wang, Jun Yang 0001, and Philip S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD Conference*, pages 305–316, 2007.

[15] R. Hull and M. Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. In *VLDB*, pages 455–468, 1990.

[16] E. Ioannou, W. Nejdl, C. Niederee, and Y. Velegrakis. Onthe-Fly Entity-Aware Query Processing in the Presence of Linkage. *PVLDB*, 3(1):429–438, 2010.

[17] Richard Johnsonbaugh and Martin Kalin. A graph generation software package. In *SIGCSE*, pages 151–154, 1991.

[18] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[19] A. M. Keller. Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins. *SIGMOD*, March 1985.

[20] B. Kimelfeld and Y. Sagiv. New algorithms for computing steiner trees for a fixed number of terminals. http://www.cs.huji.ac.il/ bennyk/papers/steiner06.pdf.

[21] B. S. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. *ACM TODS*, 25(1):83–127, March 2000.

[22] P. McBrien and A. Poulovassilis. Schema Evolution in Heterogeneous Database Architectures, A Schema Transformation Approach. In *CAiSE*, pages 484–499, 2002.

[23] F. Rizzolo and A. A. Vaisman. Temporal XML: modeling, indexing, and query processing. *VLDB Journal*, 17(5):1179–1212, 2008.

[24] F. Rizzolo, Y. Velegrakis, J. Mylopoulos, and S. Bykau. Modeling Concept Evolution: A Historical Perspective. In *ER*, pages 331–345, 2009.

[25] N. Tahmasebi, T. Iofciu, T. Risse, C. Niederée, and W. Siberski. Terminology Evolution in Web Archiving: Open Issues. In *IWAW*, Sep 2008.

[26] Y. Velegrakis, R. J. Miller, and J. Mylopoulos. Representing and Querying Data Transformations. In *ICDE*, pages 81–92, 2005.

[27] C. Yu and L. Popa. Semantic Adaptation of Schema Mappings when Schemas Evolve. In *VLDB*, pages 1006–1017, 2005.